INFO 2124

JavaScript

```
const module    = "Module 6";
const title     = "Working with Data & Exception Handling";

document.addEventListener("DOMContentLoaded", ()=> {
        const message =
                `
                <h1>${module}</h1>
                <h2>${title}<h2>
                `;

        document.write(message);
});
```

Speaker notes

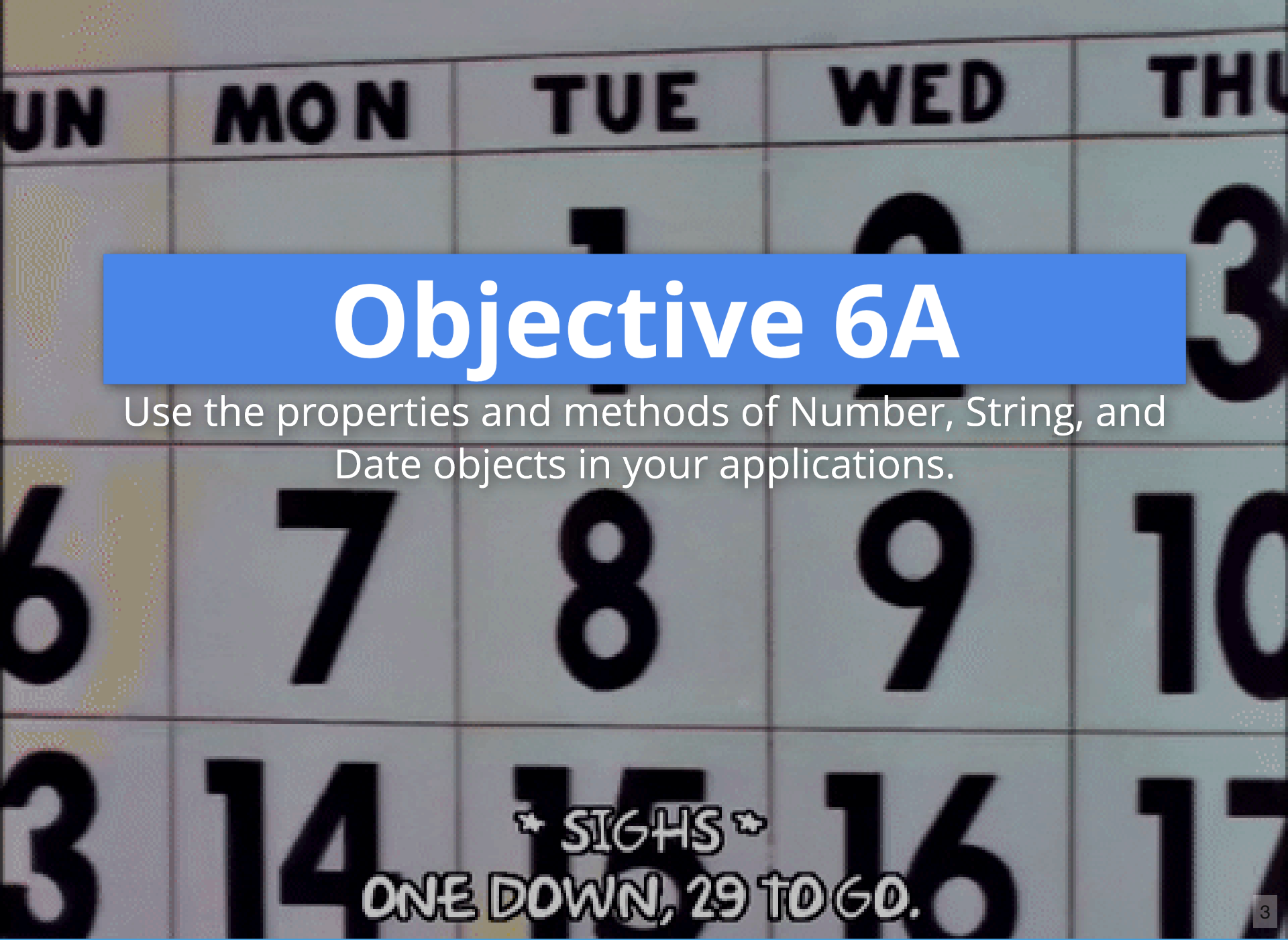
speaker notes

Module Objectives

A. Use the properties and methods of Number, String, and Date objects in your applications.
B. Use the equality (i.e, == and !=) and identity (i.e., === and !==) operators in your control structures.
C. Use switch statements, including those that use fall through and default cases.
D. Use the conditional (ternary) operator for simple logic requirements.
E. Use try-catch statements to catch errors.
F. Create and throw Error objects.

Objective 6A

Use the properties and methods of Number, String, and Date objects in your applications.



How to Work With Numbers

- Any numerical operation that results in a number greater than **Number.MAX_VALUE** will return the value **Infinity**
  - Any numerical operation that results in a number less than **Number.MIN_VALUE** will return the value **-Infinity**
- Any numerical operation with a non-numeric operand will return **NaN**
- You can't test for equality with the **NaN** property.
  - Instead, you must use the **isNaN()** or **Number.isNaN()** method.
- Division of zero by zero results in **NaN**.
- Division of a non-zero number by zero results in either **Infinity** or **-Infinity**.
- To work with integers larger than **Number.MAX_SAFE_INTEGER** or integers smaller than **Number.MIN_SAFE_INTEGER**, you can use the **BigInt** data type.
  - **BigInt** is a feature defined in the ES2020 version of JavaScript. You will want to verify your target browser(s) support(s) **BigInt** before you attempt to use it:
    - "BigInt" | Can I use... Support tables for HTML5, CSS3, etc

Static Properties of the Number Object

Property	Shortcut
`Number.MAX_VALUE`	
`Number.MIN_VALUE`	
`Number.MAX_SAFE_INTEGER`	
`Number.MIN_SAFE_INTEGER`	
`Number.POSITIVE_INFINITY`	`Infinity`
`Number.NEGATIVE_INFINITY`	`-Infinity`
`Number.EPSILON`	
`Number.NaN`	`NaN`

Testing for Infinity, -Infinity, and NaN

```
1 if ( result == Infinity ) {
2     alert( "The result is greater than " + Number.MAX_VALUE );
3 } else if ( result == -Infinity ) {
4     alert( "The result is less than " + Number.MIN_VALUE );
5 } else if ( isNaN(result) ) {
6     alert( "The result is not a number" );
7 } else {
8     alert( "The result is " + result );
9 }
```

Division by zero

```
1 alert(  0 / 0 );      // displays NaN
2 alert( 10 / 0 );      // displays Infinity
3 alert( -1 / 0 );      // displays -Infinity
```

Safe and unsafe integers

```
1 const safe1 = Number.MAX_SAFE_INTEGER - 1;
2 const safe2 = Number.MAX_SAFE_INTEGER - 2;
3 const tooBig1 = Number.MAX_SAFE_INTEGER + 1;
4 const tooBig2 = Number.MAX_SAFE_INTEGER + 2;
5 alert( safe1 == safe2 );          // displays false
6 alert( tooBig1 == tooBig2 );      // displays true;
7                                   // ints not stored accurately
```

Methods of the Number Object

- The following are **instance** methods of the Number object. **Instance** methods are methods that can be called on individual constants or variables that store numeric values
- **toFixed(*digits*)**
  - Returns a string with the number rounded to the given decimal places.
- **toString(*[base]*)**
  - Returns a string with the number in the given base, or base 10.

```
1 //Example 1: Using the toFixed() method
2 const subtotal = 19.99, rate = 0.075;
3 const tax = subtotal * rate;            // tax is 1.49925
4 alert( tax.toFixed(2) );                // displays 1.50
5
6 //Example 2: Implicit use of the toString() method for base 10 conversions
7 const age = parseInt( prompt("Please enter your age.") );
8 alert( "Your age is " + age );
```

Methods of the Number Object

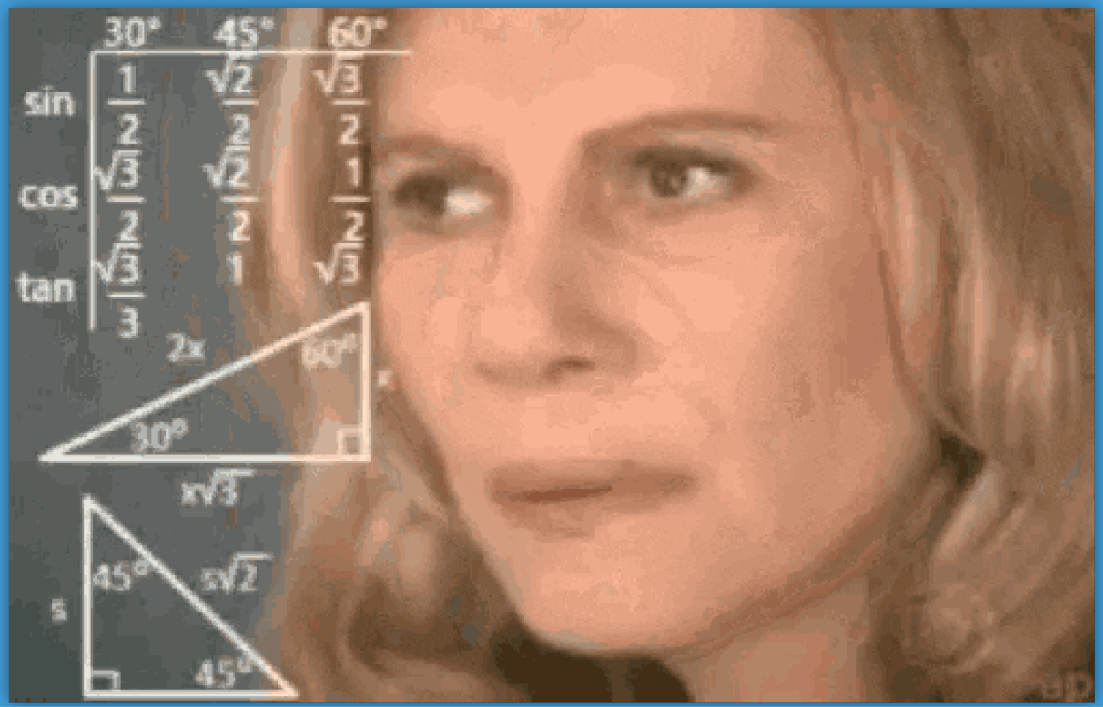
- The following are **static** methods of the Number object. **Static** methods are methods that require you include the name of the **Number** object when you call them.
- **Number.isNaN(*value*)**
  - Returns a Boolean that indicates whether a value is NaN and has the Number type.
- **Number.isFinite(*value*)**
  - Returns a Boolean that indicates whether a value is finite.
- **Number.isInteger(*value*)**
  - Returns a Boolean that indicates whether a value is an integer.
- **Number.isSafeInteger(*value*)**
  - Returns a Boolean that indicates whether a value is a "safe" integer.
    - "A safe integer is an integer value that can be exactly represented as an IEEE-754 double precision number without rounding. A safe integer can range in value from -9007199254740991 to 9007199254740991 (inclusive) which can be represented as -(2^53 - 1) to 2^53 - 1" (source).

Methods of the Number Object

```
1  //Example 1: isNaN() vs Number.isNaN()
2  alert( isNaN("four") );                 // displays true
3  alert( isNaN([1,2,3]) );                // displays true
4  alert( isNaN(NaN) );                    // displays true
5
6  alert( Number.isNaN("four") );          // displays false
7  alert( Number.isNaN([1,2,3]) );         // displays false
8  alert( Number.isNaN(NaN) );             // displays true
9
10 //Example 2: Using the isFinite() method
11 alert( Number.isFinite( 10 / 0 ) );     // displays false
12 alert( Number.isFinite( -1 / 0 ) );     // displays false
13 alert( Number.isFinite(200) );          // displays true
14
15 //Example 3: Using the isInteger() and isSafeInteger() methods
16 const tooBig = Number.MAX_SAFE_INTEGER + 1;
17
18 alert( Number.isInteger(tooBig) );       // displays true
19 alert( Number.isSafeInteger(tooBig) );   // displays false
20 alert( Number.isSafeInteger(tooBig - 1) );
21                                          // displays true
```

The JavaScript Math Object

- JavaScript's **Math** object provides properties and methods for mathematical constants and functions.
- The **Math** object is static.
  - This means you do not have to create a new Math() object in your code. Rather, you reference the **Math** objects properties and functions by referencing **Math.*property*** or **Math.*method***



The JavaScript Math Object

```
1  //Example 1: The abs() method
2  const result_1a = Math.abs(-3.4);      // result_1a is 3.4
3
4  ////Example 2: The round() method
5  const result_2a = Math.round(12.5);    // result_2a is 13
6  const result_2b = Math.round(-3.4);    // result_2b is -3
7  const result_2c = Math.round(-3.5);    // result_2c is -3
8  const result_2d = Math.round(-3.51);   // result_2d is -4
9
10 ////Example 3: The floor(), ceil(), and trunc() methods
11 const result_3a = Math.floor(12.5);    // result_3a is 12
12 const result_3b = Math.ceil(12.5);     // result_3b is 13
13 const result_3c = Math.trunc(12.5);    // result_3c is 12
14 const result_3d = Math.floor(-3.4);    // result_3d is -4
15 const result_3e = Math.ceil(-3.4);     // result_3e is -3
16 const result_3f = Math.trunc(-3.4);    // result_3f is -3
```

The JavaScript Math Object

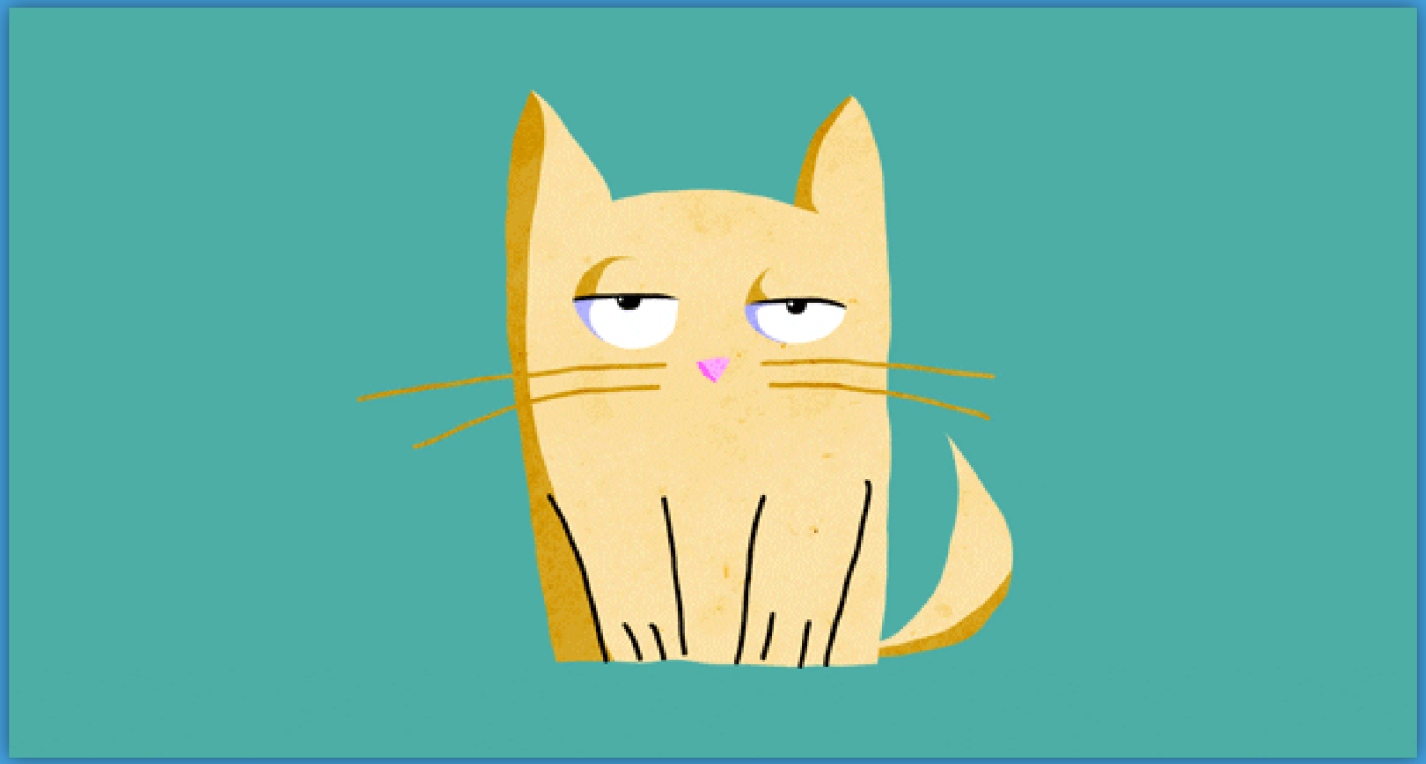
```
1  //Example 4: The pow() and sqrt() methods
2  const result_4a = Math.pow(2,3);         // result_4a is 8
3  const result_4b = Math.pow(125, 1/3);    // result_4b is 5
4  const result_4c = Math.sqrt(16);         // result_4c is 4
5
6  //Example 5: The min() and max() methods
7  const result_5a = Math.max(12.5, -3.4);
8                                           // result_5a is 12.5
9  const result_5b = Math.min(12.5, -3.4);
10                                          // result_5b is -3.4
```

The JavaScript Math Object

```
// A function that generates a random number
const getRandomNumber = max => {
    let random = null;

    if (!isNaN(max)) {
        // value >= 0.0 and < 1.0
        random = Math.random();

        // value is an integer between 0 and max - 1
        random = Math.floor(random * max);

        // value is an integer between 1 and max
        random = random + 1;
    }
    // if max is not a number, will return null
    return random;
};
// A statement that calls the getRandomNumber() function
// returns an integer that ranges from 1 through 100
const randomNumber = getRandomNumber(100);
```

How to Work With Strings

- We have already worked with strings. For example, we know we can use the **trim()** method of the String object to remove white space from the beginning and the end of a string.
- In this section, we'll look at some of the properties and methods of the String object.



String Properties

- String exposes one property, which is **length**. The **length** property returns the number of characters in a string, this includes any spaces in the character, whether the spaces are in the begging of the string, at the end of the string, or in the middle of the string.

```
1 //Example of the length property
2 const message = "JavaScript";
3 const result_1 = message.length;         // result_1 is 10
4
5 const message2 = "    this is another string    ";
6 const result_2 = message2.length;                // result_2 is 30
7                                                  // since spaces are included in
8                                                  // the length
```

Select String Methods

- Here's a small listing of methods exposed by string. A more exhaustive list can be viewed here: JavaScript String Methods (w3schools.com)
  - **charAt(***position*)
    - Returns the character at the specified position in the string.
  - **concat(***string1, string2, ...*)
    - Returns a new string that is the concatenation of each of the strings in the parameter list.
  - **indexOf**(*search[, start])*
    - Searches the string for the *first* occurrence of the search string at the specified starting position (or the beginning of the string if *start* is omitted). If the search string is found, it returns the position of the string. If the search string is not found, it returns -1.
  - **substr**(*start, length*)
    - Returns the substring from the start position to the end of the string.

Select String Methods

- continues . . .
  - **substring(***start, end***)**
    - Returns the substring from the start position to, but not including, the end position.
  - **toLowerCase()**
    - Returns the string with all letters converted to lowercase.
  - **toUpperCase()**
    - Returns the string with all letters converted to uppercase.

Examples of Methods of String Objects

```
1  // Constant used by the following examples
2  const message = "JavaScript";
3
4  // Example 1: The charAt() method
5  const letter = message.charAt(4);     // letter is "S"
6
7  // Example 2: The concat() method
8  const result_3 = message.concat(" rules");
9                      // result_3 is "JavaScript rules"
10
11 // Example 3: The indexOf() method
12 const result_4a = message.indexOf("a");          // result_4a is 1
13 const result_4b = message.indexOf("a", 2);       // result_4b is 3
14 const result_4c = message.indexOf("s");          // result_4c is -1
15
```

Examples of Methods of String Objects

```
// Example 4: The substr() and substring() methods
const result_5a = message.substr(4, 5);          // result_5a is "Scrip"
const result_5b = message.substring(4);          // result_5b is "Script"
const result_5c = message.substring(0, 4);       // result_5c is "Java"

// Example 5: The toLowerCase() and toUpperCase() methods
const result_6a = message.toLowerCase();         // result_6a is "javascript"
const result_6b = message.toUpperCase();         // result_6b is "JAVASCRIPT"

// compare two strings ignoring case
alert( result_6a.toLowerCase() ==
    result_6b.toLowerCase() )                    // displays true

```

More Methods of the String Object

- **startsWith**(*string[, start]*)
  - Returns a Boolean that indicates whether the string start with the specified string. Searches from the specified position or the start of the string if the position isn't specified.
- **endsWith**(*string[, length]*)
  - Returns a Boolean value that indicates whether the string contains the specified string. Searches from the start of the string for the specified length or the entire string if the length isn't specified.
- **includes**(*string[, start]*)
  - Returns a Boolean that indicates whether the string contains the specified string. Searches from the specified position or the start of the string if a position isn't specified.
- **trimStart()**
  - Returns a string with whitespace removed from the beginning.

More Methods of the String Object

- **trimEnd()**
  - Returns a string with whitespace removed from the end.
- **trim()**
  - Returns a string with whitespace removed from the beginning and the end.
- **padStart**(*length[, string]*)
  - Returns the string with the specified string added to the beginning until the string is the specified length. If a string isn't specified, a space is used.
- **padEnd**(*length[, string]*)
  - Returns the string with the specified string added to the end until the string is the specified length. If a string isn't specified, a space is used.
- **repeat**(*times*)
  - Returns the string repeated the specified number of times.

More Methods of the String Object

```
// Constants used by the following examples
const str = "jQuery";
const len = str.length;

//Example 1: The startsWith(), endsWith(), and includes() methods
const result_1a = str.startsWith("j");      // result_1a is true
const result_1b = str.startsWith("j", 2);   // result_1b is false
const result_1c = str.endsWith("ry");       // result_1c is true
const result_1d = str.endsWith("ry", 2);    // result_1d is false
const result_1e = str.includes("Que");      // result_1e is true
const result_1f = str.includes("Que", 2);   // result_1f is false
```

More Methods of the String Object

```
// Example 2: The padStart() and padEnd() methods
const result_2a = str.padStart(len + 3);
                                // result_2a is "   jQuery"
const result_2b = str.padEnd(len + 3, ".");              // result_2b is "jQuery..."

// Example 3: The trimStart(), trimEnd(), and trim() methods
const str_2 = str.padStart(len + 3).padEnd(len + 6);     // str_2 is "   jQuery   "
const result_3a = str_2.trimStart();                     // result_3a is "jQuery   "
const result_3b = str_2.trimEnd();                       // result_3b is "   jQuery"
const result_3c = str_2.trim();                          // result_3c is "jQuery"

// Example 4: The repeat() method
const result_4a = "Hey ".repeat(3);                      // result_4a is "Hey Hey Hey "

```

More Methods of the String Object

- The **split()** method of a String object breaks a string into *substrings* and returns those substrings as elements in an array.
- If a string doesn't include the separator that's specified in the parameter of the **split()** method, the entire string is returned as the first element in a one-element array.
- If the separator that's specified by the parameter is an empty string, each character in the string becomes an element in the array that's returned by the method.
- If the string includes the separator at the beginning or end of the string, an empty string is included as an element at the beginning or end of the array.

More Methods of the String Object

```
1  // Two constants that are used by the following examples
2  const fullName = "Grace M Hopper";
3  const date = "7-4-2021";
4
5  // How to split a string that's separated by spaces into an array
6  const words = fullName.split(" ");      // words is ["Grace", "M", "Hopper"]
7  const len = words.length;               // len is 3
8  const lastName = words[len - 1];        // lastName is "Hopper"
9
10 //How to split a string that's separated by hyphens into an array
11 const dateParts = date.split("-");      // dateParts is ["7", "4", "2021"]
12 const month = dateParts[0];             // month is "7"
13 const year = dateParts[2];              // year is "2021"
14
```

More Methods of the String Object

```
1  // How to split a string into an array of characters
2  const characters = fullName.split("");
3
4  // How to get just one element from a string
5  const firstName = fullName.split(" ", 1);
6                                                  // firstName is ["Grace"]
7  const len = firstName.length;                   // len is 1
8
9  // A statement that uses the firstName constant like a regular string
10 const greeting = `Hello, ${firstName}!`;          // greeting is "Hello, Grace!"
11
```

More Methods of the String Object

```
1  // How it works if the string doesn't contain the separator
2  const dateParts = date.split("/");
3                              // dateParts is ["7-4-2021"]
4  len = dateParts.length;     // len is 1
5
6  // How it works if the string contains the separator at the beginning or end
7  const path = "/directory/";
8  const pathParts = path.split("/");
9                             // pathParts is ["", "directory", ""]
10
```

How to Work with Dates and Time

- In JavaScript, dates are represented by the number of milliseconds since midnight, January 1, 1970.
  - This date is also known as *Unix time, Epoch time, etc.:* Unix time - Wikipedia
- When you create a **Date** object, the date and times are specified as local time.
  - Local time is what time is in the time zone specified on the computer that's running the user's web browser.
- Month numbers begin with **zero** (0), so January is **0** and December is **11**.
  - This allows values to be used with arrays (remember, arrays start at zero).
- You can use the **instanceof** operator to test whether an object is a Date object.
- To work with a **Date()** object, you need to create a new *instance* of the **Date()** object using the **new** keyword.

```
1 // How to create a Date object that represents
2 // the current date and time
3
4 const now = new Date();
```

Summary of How the Date Constructor Works

Parameters	Description
None	Creates a new Date object set to the current date and time.
String value	Creates a new Date object set to the date and time of the string.
Numeric values	Creates a new Date object set to the year, month, day, hours, minutes, seconds, and milliseconds of the numbers. Year and month are required.
Date object	Creates a new Date object that's a copy of the Date object it received.
Invalid values	Creates a new Date object that contains "Invalid Date".

Working with the Date Constructor

```
// How to create a Date object by specifying
// a date string

const electionDay = new Date("11/3/2020");
const grandOpening = new Date("2/16/2021 8:00");
const departureTime = new Date("4/6/2021 18:30:00");
```

Working with the Date Constructor

```
// How to create a Date object
// by specifying numeric values

// Syntax of the constructor for the Date object
new Date( year, month, day, hours, minutes, seconds,
          milliseconds )

////Examples
const electionDay = new Date(2020, 10, 3);              // 10 is November
const grandOpening = new Date(2021, 1, 16, 8);          // 1 is February
const departureTime = new Date(2021, 3, 6, 18, 30);     // 3 is April

```

Working with the Date Constructor

```
// How to create a Date object
// by copying another Date object

const invoiceDate = new Date("8/8/2021");
const dueDate = new Date( invoiceDate );
// You can then add a number of days to due_date.

```

Working with the Date Constructor

```
1  // What happens when an invalid date is passed
2  // to the Date Constructor
3  const leapDate = new Date("2/29/2021");  // Invalid Date
4
5  // Date strings that can lead to unexpected
6  // results in some browsers
7  const electionDay = new Date("11-3-2020");      // Invalid Date
8  const electionDay = new Date("11/3/20");        // 11/3/1920
9
10 // How to check whether an object is a Date object
11 if (electionDay instanceof Date) { ... }
12
```

How to Use the Methods of the Date Object

- Formatting methods of a Date object:
  - **toString()**
    - Returns a string containing the date and time in local time using the client's time zone.
  - **toDateString()**
    - Returns a string representing just the date in local time.
  - **toTimeString()**
    - Returns a string representing just the time in local time.

```
1 // Examples of the formatting methods
2 const birthday = new Date( 2001, 0, 7, 8, 25);
3
4 alert( birthday.toString() );
5                   // "Sun Jan 07 2001 08:25:00 GMT-0800"
6
7 alert( birthday.toDateString() );  // "Sun Jan 07 2001"
8
9 alert( birthday.toTimeString() );  // "08:25:00 GMT-0800"
```

How to Use the Methods of the Date Object

- The get methods of a Date object
  - **getTime()**
    - Returns the number of milliseconds since midnight, January 1, 1970 in universal time (GMT).
  - **getFullYear()**
    - Returns the four-digit year in local time.
  - **getMonth()**
    - Returns the month in local time, starting with 0 for January.
  - **getDate()**
    - Returns the day of the month in local time.
  - **getDay()**
    - Returns the day of the week (0 = Sunday, 1 = Monday, etc.).
      - See speaker's notes for important note.

Speaker notes

Page. 381 of the text states that getDay() returns the day of the week, which is correct. The book states, though, that a return value of 1 corresponds to Sunday. This is incorrect. getDay() will return, "An integer number, between 0 and 6, corresponding to the day of the week for the given date, according to local time: 0 for Sunday, 1 for Monday, 2 for Tuesday, and so on." This statement comes from the Mozilla Developer Network's documentation for the behavior of the Date object's getDay() method (source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date/getDay) -- a zero-based value is congruent with other date behavior (such as the getMonth() method), which is zero-based as well, and corresponds to array indices.

How to Use the Methods of the Date Object

- Continues . . .
  - **getHour()**
    - Returns the hour in 24-hour format in local time.
  - **getMinutes()**
    - Returns the minutes in local time.
  - **getSeconds()**
    - Returns the seconds in local time.
  - **getMilliseconds()**
    - Returns the milliseconds in local time.

How to Use the Methods of the Date Object

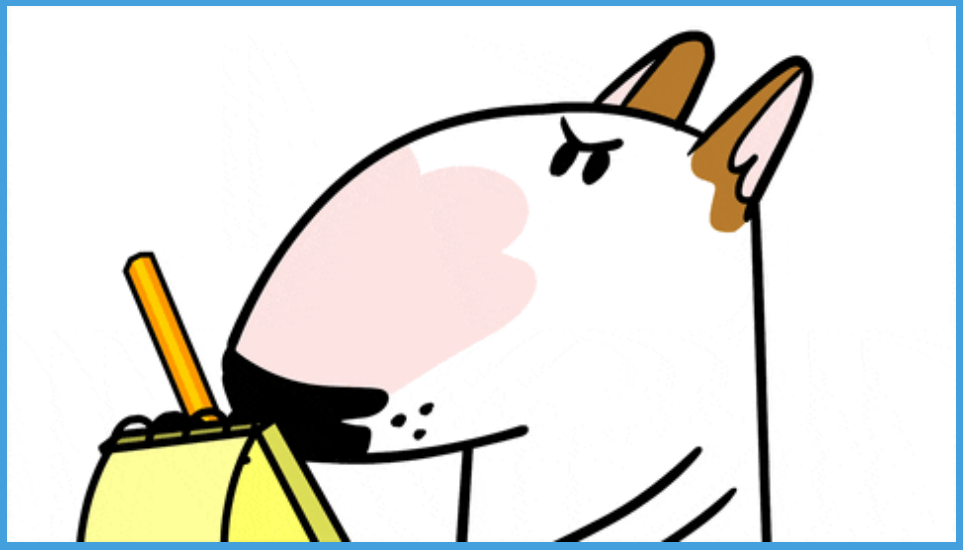
- The set methods of the Date object
  - **setFullYear(*year*)**
    - Sets the four-digit year in local time.
    - See parameters: Date.prototype.setFullYear() - JavaScript | MDN (mozilla.org)
  - **setMonth(*month*)**
    - Sets the month in local time.
    - See parameters: Date.prototype.setMonth() - JavaScript | MDN (mozilla.org)
  - **setDate(*day*)**
    - Sets the day of the month in local time.
    - See parameters: Date.prototype.setDate() - JavaScript | MDN (mozilla.org)
  - setHours(*hour*)
    - Sets the hour in 24-hour format in local time.
    - See parameters: Date.prototype.setHours() - JavaScript | MDN (mozilla.org)

How to Use the Methods of the Date Object

- Continues . . .
  - **setMinutes(***minute***)**
    - Sets the minutes in local time.
    - See parameters: Date.prototype.setMinutes() - JavaScript | MDN (mozilla.org)
  - **setSeconds(***seconds***)**
    - Sets the seconds in local time.
    - See parameters: Date.prototype.setSeconds() - JavaScript | MDN (mozilla.org)
  - **setMilliseconds(***ms***)**
    - Sets the milliseconds in local time.
    - See parameters: Date.prototype.setMilliseconds() - JavaScript | MDN (mozilla.org)

How to Use the Methods of the Date Object

- Additional notes . . .
  - Except for the **getTime()** method, the **get** and **set** methods of the Date object use the time zone specified on the user's computer to work with local time.
  - There are complimentary **get** and **set** methods that start with **getUTC** and **setUTC** that work with the Date object in universal time (GMT).
    - For example, the **getUTCHours()** method will return the hour in 24-hour format in universal time (GMT).



Examples of Working with Dates

```
// How to display the date in your own format

const departTime = new Date(2021, 3, 16, 18, 30);
// // April 16, 2021 6:30pm

const year = departTime.getFullYear();

// add 1 since months start at 0
const month = departTime.getMonth() + 1;
const day = departTime.getDate();

let dateText = year + "-";
// pad month if 1 digit
dateText += month.toString().padStart(2, "0") + "-";
// pad day if 1 digit
dateText += day.toString().padStart(2, "0");

// final dateText is "2021-04-16"
```

Examples of Working with Dates

```
// How to calculate the days until the New Year

const now = new Date();             // get current date and time
const newYear = new Date(now);      // copy current date and time
newYear.setMonth(0);                // set month to January
newYear.setDate(1);                 // set day to the 1st
newYear.setFullYear( newYear.getFullYear() + 1 );

// time in milliseconds
const timeLeft = newYear.getTime() - now.getTime();
// milliseconds in a day: hrs * mins * secs * milliseconds
const msInOneDay = 24 * 60 * 60 * 1000;
// convert milliseconds to days
const daysLeft = Math.ceil( timeLeft / msInOneDay );

let message = "There ";
if (daysLeft == 1) {
    message += "is one day";
}
else {
    message += "are " + daysLeft + " days";
}
message += " left until the New Year.";

// If today is November 3, 2020, message is
// "There are 59 days left until the New Year."

```

Examples of Working with Dates

```
1  // How to calculate a due date
2
3  const invoiceDate = new Date();
4  const dueDate = new Date( invoiceDate );
5  dueDate.setDate( dueDate.getDate() + 21 );
6  // due date is 3 weeks later
7
8  // How to find the end of the month
9
10 const endOfMonth = new Date();
11
12 // Set the month to next month
13 endOfMonth.setMonth( endOfMonth.getMonth() + 1 );
14
15 // Set the date to one day before the start of the month
16 endOfMonth.setDate( 0 );
17
```

Objective 6B

Use the equality (i.e, == and !=) and identity (i.e., === and !==) operators in your control structures.

Let's Talk Control Structures . . .

- In previous chapters, we've learned about the equality operator, which is comprised of two equals symbols (==).
- The equality operator (==) performs *type conversion* whenever necessary. This means that the equality operator will automatically convert data from one data type to another to perform the comparison. For example, they often convert strings to numbers before comparisons, allowing us to write something like:
  if ("42" == 42)
- However, the type conversion performed by the equality operators is different from the type conversion performed by the parseInt() and parseFloat() methods.
- We can avoid the problem of automatic type conversion by using the *identity operators*
  - These operators are sometimes called "strict" equality and "strict" inequality.

Let's Talk Control Structures . . .

Operator	Description	Example
==	Equal	`lastName == "Hopper"`
!=	Not equal	`months != 0`

These are the *equality* operators we have worked with up to this point. Remember, these operators help us to determine if the value on the *left* is equal to the value on the *right,* or if the value on the *left* is not equal to the value on the *right*.

Let's Talk Control Structures . . .

Operator	Description	Example
`===`	Equal	`lastName === "Hopper"`
`!==`	Not equal	`months !== 0`

These are the *identity* operators (or "strict" equality/inequality). They also compare whether the value on the *left* is equal to the value on the *right*, or if the value on the *left* is not equal to the value on the *right*.

However, *identity* operators compare values and data types to determine if values and types of data are/aren't the same.

Remember!

- The *equality operators* perform *type coercion/conversion* when necessary:
  - So they will automatically convert a string to a number or vice versa before comparing the data.
    - So, *if ("42" == 42)* would be true.
  - The *identity operators* do not perform type coercion/conversion.
    - If two operands are not the same type, the result will be false.
    - So, *if ("42" === 42)* would be false.

Other Control Structures

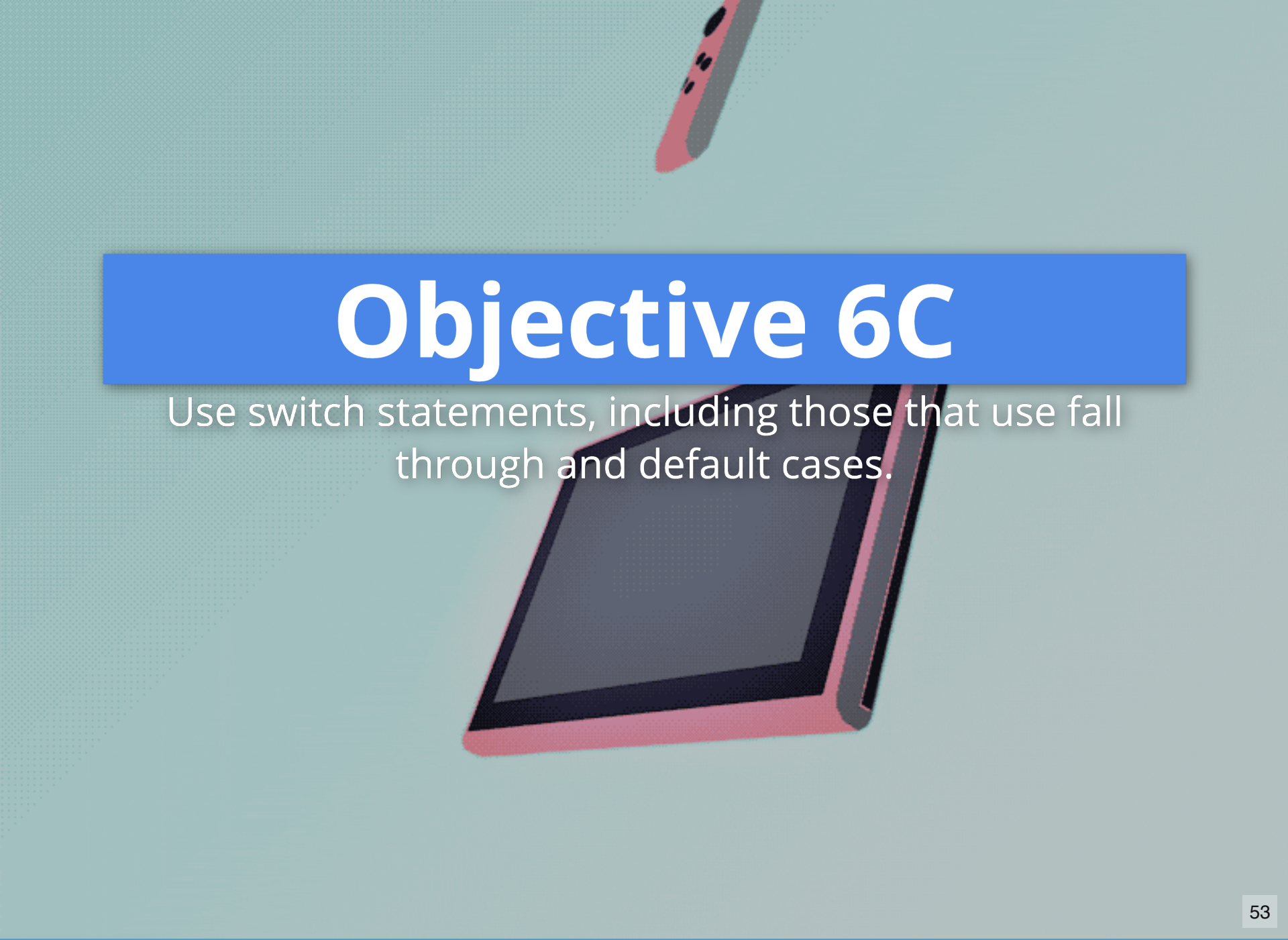
- This module, we'll introduce two JavaScript statements that provide additional control over loop structures:
  - **break**
    - The **break** statement ends a loop (in other words, it jumps out of a loop).
  - **continue**
    - The **continue** statement ends the current iteration of a loop, but allows the next iteration to proceed. In other words, it jumps to the start of the loop.
- When working with nested loops, the **break** and **continue** statements apply only to the loop that they're in.

The break Statement

```
1  // Example 1: The break statement in a while loop
2
3  let number = 0;
4  while (true) {
5      number = parseInt( prompt(
6                    "Enter a number from 1 to 10.") );
7      if ( isNaN(number) || number < 1 || number > 10 ) {
8          alert("Invalid entry. Try again.");
9      } else {
10         break;
11     }
12 }
13 alert(number);
14
```

The continue Statement

```
1  // Example 2: The continue statement in a while loop
2
3  let sum = 0;
4  let number = 0;
5  while ( number <= 40 ) {
6      number++;
7      if ( number % 5 !== 0 ) {
8          continue;      // if number isn't divisible by 5
9      }
10     sum += number;
11 }
12 alert(sum);
13 // displays sum of 5, 10, 15, 20, 25, 30, 35, 40
```



Objective 6C

Use switch statements, including those that use fall through and default cases.

The switch Statement

- The **switch** statement evaluates a **switch expression** in the parenthesis and then executes the code in the **case label** whose value matches the expression.
- Execution stops when it reaches a **break statement** or a **return statement**.
- If a case doesn't contain a break or return statement, execution will **fall through** to the next label.

```
switch (expression) {
  case "condition1":
    //code to run
    break;
  case "condition2":
    //code to run
    break;
  default:
    //code to run if nothing
    //else is true
}
```

A switch Statement With a Default Case

```
switch ( letterGrade ) {
    case "A":
        message = "well above average";
        break;
    case "B":
        message = "above average";
        break;
    case "C":
        message = "average";
        break;
    case "D":
        message = "below average";
        break;
    case "F":
        message = "failing";
        break;
    default:
        message = "invalid grade";
        break;
}
```

A Switch Statement With Fall Through

```
switch ( letterGrade ) {
    case "A":
    case "B":
        message = "Scholarship approved.";
        break;
    case "C":
        message = "Application requires review.";
        break;
    case "D":
    case "F":
        message = "Scholarship not approved.";
        break;
}
```

A Function That Uses A Switch Statement To Return A Value

```
1  const getTimeOfDay = ampm => {
2      switch( ampm.toUpperCase() ) {
3          case "AM":
4              return "morning";
5          case "PM":
6              return "evening";
7          default:
8              return "invalid";
9      }
10 };
```

Objective 6D

Use the conditional (ternary) operator for simple logic requirements.

The Conditional/Ternary Operator

- JavaScript has a *conditional* or *ternary operator*, which can be used to write concise code.
  - The *conditional/ternary operator* has three operands, which is why it's called the *ternary* operator.
  - `The syntax for the operator is`

    `(conditional_expression) ? value_if_true : value_if_false`

  - The *conditional/ternary* operator first evaluates the conditional expression. Then, if the expression is true, the value that results from the middle operand is returned. But, fi the expression is false, the value that results from the third operand is returned.
  - Although the use of the *conditional operator* can lead to cryptic code, JavaScript programmers commonly use this operator. For clarity, though, *conditional operators* can be written with *if* statements alone.

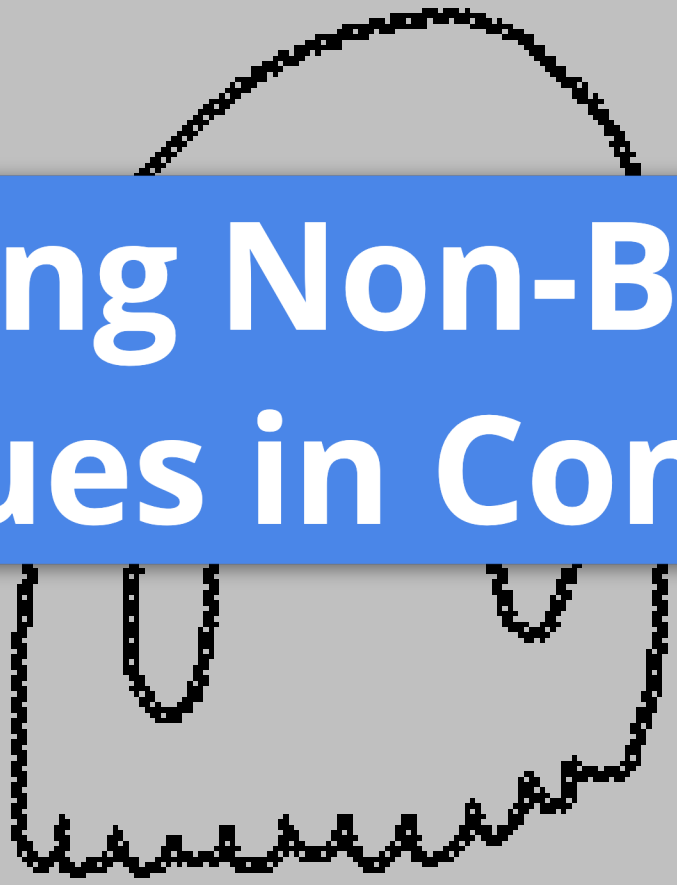
Examples Of Using The Conditional Operator

```
1  // Example 1: Setting a string based on a comparison
2  const message = ( age >= 18 ) ? "Can vote" : "Cannot vote";
3
4  // Example 2: Calculating overtime pay
5  const overtime = ( hours > 40 ) ? ( hours - 40 ) * rate * 1.5 : 0;
6
7  // Example 3: Selecting a singular or plural ending based on a value
8  const ending = ( errorCount == 1 ) ? "" : "s".
9  const message = "Found " + error_count + " error" + ending + ".";
10
11 // Example 4: Setting a value to 1 if it's at a maximum value
12 let value = ( value == maxValue ) ? 1 : value + 1;
13
14 // Example 5: Returning one of two values based on a comparison
15 return ( number > highest ) ? highest : number;
```

How Conditional Operators Can Be Rewritten With If Statements

```
1  // Example 1: Setting a string based on a comparison
2  const message = ( age >= 18 ) ? "Can vote" : "Cannot vote";
3
4  // Example 1 rewritten with an if statement
5  let message;
6  if ( age >= 18 ) {
7      message = "Can vote";
8  } else {
9      message = "Cannot vote";
10 }
11
12 // Example 4: Setting a value to 1 if it's at a maximum value
13 let value = ( value == maxValue ) ? 1 : value + 1;
14
15 // Example 4 rewritten with an if statement
16 if ( value == maxValue ) {
17     value = 1;
18 }
19 else {
20     value = value + 1;
21 }
```

Using Non-Boolean Values in Conditions



Using Non-Boolean Values in Conditions

- JavaScript allows you to use non-Boolean as well as Boolean values in conditions.
- Non-Boolean values that are used this way are called **truthy** or **falsey** to indicate that they aren't actual Boolean values but can still be used in conditions.
- You can use non-Boolean values to test for a number of conditions like uninitialized variables, strings that are empty, and properties that are nonexistent.

Non-Boolean Values Evaluated As False

- 0
- "" (empty string)
- `null`
- `undefined`
- an empty array

Non-Boolean Values Evaluated As True

- Any number other than 0
- Any string that isn't empty
- Any array that isn't empty
- Any object
- Any symbol

Using Non-Boolean Values in Conditions

```
1  // Example 1: An if statement that checks if a variable is initialized
2
3  let val;
4  if (!val) {
5      alert("Variable 'val' is not initialized");
6  }
7
8  // Example 2: An if-else statement that checks for an empty string
9  const name = prompt("Please enter a name");
10 if (name) {
11     alert(name);
12 } else {
13     alert("You did not enter a name");
14 }
15
16 // Example 3: An if statement that checks for the existence of a property
17
18 if (window.navigator.geolocation) {
19       // code that uses the geolocation property
20       // of the window object's Navigator object
21 }
```

Additional Techniques for Using the Logical Operators

- When the **AND** and **OR** logical operators are used to create a compound condition, the result of the last condition that was evaluated was returned.
- The logical operators use **short-circuit evaluation** to determine the result that's returned by both Boolean and non-Boolean values. The result can then be stored in a constant or variable.
  - If the result is a Boolean value, it can also be used as a condition in a control statement.
- The **OR** operator is commonly used with non-Boolean values to return a default value if a value isn't provided.
- The **AND** operator is commonly used with non-Boolean values to make sure that a property exists.
- ES2020 provides a **nullish coalescing operator (??)** that can be used in the place of the **OR** operator and an **optional chaining operator (?)** that can be used in place of the **AND** operator. Both of these features are supported by major browsers (see speaker's notes).

Speaker notes

Support for nullish coalescing operator: https://caniuse.com/mdn-javascript_operators_nullish_coalescing
Support for optional chaining operator: https://caniuse.com/mdn-javascript_operators_optional_chaining

Using Non-Boolean Values in Conditions

```
1  // Statements that store a true or false value
2  // Example 1: Using the OR operator
3  const selected = state === "CA" || state === "NC";
4
5  // Example 2: Using the AND operator
6  const canVote = age >= 18 && citizen;
7
8  // Statements that store a value and provide a default value
9  // Example 3: Using the OR operator
10 const name = prompt("Please enter a name") || "N/A";
11
12 // // Example 4: Using the nullish coalescing operator
13 const name = prompt("Please enter a name") ?? "N/A";
```

Using Non-Boolean Values in Conditions

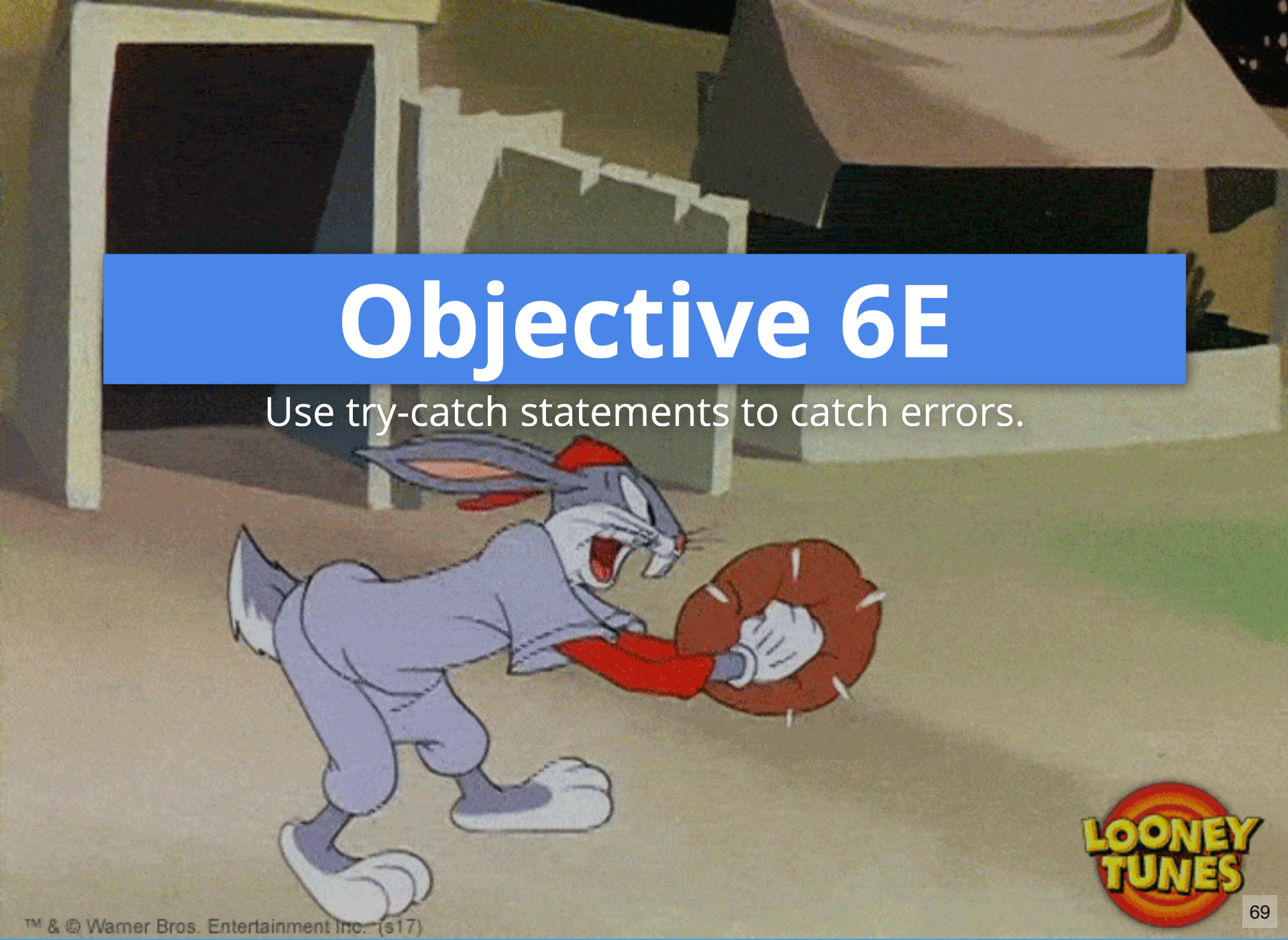
```
1 // Statements that check for the existence of a property
2
3 // Example 5: Using the AND operator
4 if (window && window.navigator &&
5     window.navigator.geolocation) { ... }
6
7 // Example 6: Using the optional chaining operator
8 if (window?.navigator?.geolocation) { ... }
```

Using Non-Boolean Values in Conditions

```
1  // Statements that store the value of a property
2  // or a default value if the property doesn't exist
3
4  // Example 7: Using the AND and OR operators
5  const initial = book && book.author &&
6        book.author.middleInitial || "No Middle Initial";
7
8  // Example 8: Using the nullish coalescing
9  // and optional chaining operators
10 const initial = book?.author?.middleInitial ??
11       "No Middle Initial";
12
```

Objective 6E

Use try-catch statements to catch errors.



Exceptions

- Runtime errors can also be referred to as **exceptions**.
- Because **exceptions** shouldn't occur in real-world applications, most programming languages provide a way to handle **exceptions** that are **thrown** by an application so that the applications don't crash. This is known as **exception handling**.
- You can use a **try-catch statement** to process any errors that are thrown by an application.
  - In a **try-catch statement**,
    - You code a **try block** that contains the statements that may throw exceptions.
    - Then, you code a **catch block** that contains the statements that are excepted when an exception is thrown in the try block.
    - The optional **finally block** is executed whether or not the statements in the catch block are executed.

Try-Catch Statement Syntax

- The syntax for a **try-catch** statement is summarized in the figure to the right.
  - The **errorName** parameter in the catch block gives you access to the Error object. This object has two properties:
    - **name**
      - The type of error
    - **message**
      - The message that describes the error
  - If you don't need the Error object, you can omit the **errorName** parameter.

```
try {
  //statements
} catch([errorName]) {
  //statements
} [ finally {
   //statements
} ]
```

A Try-catch Statement For A calculatefv() Function

```
const calculateFV = (investment, rate, years) => {
    try {
        //let's try this block of code....
        let futureValue = investment;
        for (let i = 1; i <= years; i++ ) {
            futureValue += futureValue * rate / 100;
        }
        return futureValue.toFixed(2);
    }
    catch(error) {
        //if there's an error in the block of code above...
        //catch it and then execute the following
        alert (error.name + ": " + error.message)
    }
};
```

Objective 6F

Create and throw Error objects.

Create and Throw Error Objects

- In some cases, you will want to **throw** your own exceptions.
  - For example, if you're creating a utility library that will be used by other programmers, you can throw exceptions to alert those using the library that something went wrong.
- Three reasons for using throw statements:
  1. To test the operation of a try-catch statement
  2. To throw an error from a function that lets the calling code know that one or more of the arguments passed were invalid.
  3. To perform some processing after catching an error and then throw the error again.
- To create an error object, you use its constructor with a string as a parameter.
  - The string corresponds to the error message the Error object will store in the Error object's **message** property.
- To trigger a runtime error, you use the **throw statement.**

Creating a New Error Object

- The syntax for creating a new error is summarized in the figure on the right.
  - This is a generic error with whatever error message you pass as a string.
  - JavaScript supports a number of pre-built errors in addition to this base **Error** object.
    - In addition, you can define custom error types; however, this requires knowledge of **inheritance**, which is a topic we haven't covered yet.

```
1 new Error(message);
```

Some JavaScript Error Types . . .

Type	Thrown when
`RangeError`	A numeric value has exceeded the allowable range
`ReferenceError`	A variable is read that hasn't been defined
`SyntaxError`	A runtime syntax error is encountered
`TypeError`	The type of a value is different from what was expected

```
// A statement that throws a RangeError object

throw new RangeError("Annual rate is invalid.");
```

A calculatefv() Method That Throws A New Error Object

```
const calculateFV = ( investment, rate, years ) => {
    if ( isNaN(investment) || investment <= 0 ) {
        throw new Error(
            "calculateFV requires investment greater than 0.");
    }
    if ( isNaN(rate) || rate <= 0 ) {
        throw new Error(
            "calculateFV requires annual rate greater than 0.");
    }
    let futureValue = investment;
    for (let i = 1; i <= years; i++ ) {
        futureValue += futureValue * rate / 100;
    }
    return futureValue.toFixed(2);
};
```

A try-catch Statement That Catches The Error Object That Has Been Thrown

```
try {
    $("future_value").text =
        calculateFV(investment, rate, years);
}
catch(error) {
    alert (error.name + ": " + error.message);
}
finally {
    $("investment").focus();
}
```

Fin

Make sure to review the lecture videos and lab demos before starting your homework. The videos will provide more information on topics discussed in this presentation.