

# Events Review

- Elements can have "events"
  - User triggered
  - Network triggered
- JS Code can add "listeners" for events
  - Call callback ("handler")
- JS Code can modify HTML
  - Including changing classes
- CSS styles based on current page
  - Changes applied automatically

# Events Example

```
<button class="toggle-active">Toggle</button>  
<div class="example">Example</div>
```

```
.example {  
  display: none;  
}  
  
.example.shown {  
  display: block;  
}
```

```
const buttonEl = document.querySelector('.toggle-active');  
const exampleEl = document.querySelector('.example');  
  
buttonEl.addEventListener('click', function() {  
  exampleEl.classList.toggle('shown');  
});
```

# We change appearance by changing classes

- We are NOT using the `style` attribute
- Many examples/tutorials online will
  - It "works"
  - But makes code hard to change over time
  - Mixing CSS inside JS

# Cleaner syntax

- "function() { }" is a lot to type
  - Programmers are lazy
  - `function` keyword add little value here
    - This function never used elsewhere
    - Distracts from "important" parts
      - event name
      - parameters
      - function body
- Solution: The Fat Arrow Function

# We are not body shaming our code!

- Coding languages have many odd symbol names:
  - Shuttle operator: `<=>`
  - Elvis operator: `&:`
  - Arrow operator: `->`
  - Fat arrow operator: `=>`

# What is a Fat Arrow function in JS

- Different way to declare a function
- ALWAYS without a name
- Avoids redeclaring `this`
  - Outside this course
  - But a big deal
- Not required TO USE for this course
  - Many examples will use
  - In and out of this course
  - Must be able to read at least
- See `/readings/js/` for more on fat arrow

# Basic Fat Arrow function

- No `function`
- No name
- Parameters in `()`
- Then fat arrow `=>`
- Then function body `{ ... }`
- Not a block

```
const greet = function ( message, target ) {  
  return `${message}, ${target}`;  
};
```

VS

```
const greet = (message, target) => {  
  return `${message}, ${target}`;  
};
```

# That doesn't seem so bad

- But we aren't done
- If exactly 1 parameter
  - `()` becomes optional
- If body is exactly 1 statement
  - AND you return that value
  - `{}` for body become optional

```
const greet = (message, target) => {  
  return `${message}, ${target}`;  
};  
  
const sayHello = target => {    // () optional w/1 param  
  return `${message}, ${target}`;  
};
```



# Fat Arrow Function with optional parens

- If exactly 1 parameter
  - `()` becomes optional

```
const greet = (target) => {  
  return `Hello, ${target}`;  
};  
  
const sayHello = target => {    // () optional w/1 param  
  return `Hello, ${target}`;  
};  
  
const greetWorld = () => { // not 1 param, () required  
  return `Hello World!`;  
};
```

# Fat Arrow function with optional body block

- If body is exactly 1 statement
  - AND you return that value
  - `{ }` for body become optional

```
const greet = (message, target) => {  
  return `${message}, ${target}`;  
};  
  
const sayHello = (message, target) => `${message}, ${target}`;
```

# All the options

```
const greet = (target) => {  
  return `Hello, ${target}`;  
};  
  
const sayHello = target => {  
  return `Hello, ${target}`;  
};  
  
const sayHi = (target) => `Hi ${target}`;  
  
const sayHey = target => `Hey ${target}`;
```

# Fat Arrow Functions are common

- **Very common when a function is defined inline**
  - Where lack of function name isn't problem
  - Not reused anywhere
  - Fat Arrow has benefits when used as callback
    - the `this` thing
- Some people use it a lot because it is "smaller"
- Some people use it for obvious variable creation
- Some people avoid it for lack of function name
- This course puts no restrictions/requirements
  - I will use it a lot for inline callbacks

# Limits of what we know so far

Assignment made one dropdown menu work on click

- But what if two+ menus?
- Would need unique class on each button
- Would need unique class on each dropdown
- Tedious is un-fun

# **Hover worked with many dropdowns**

- Only needed one instruction
- Each dropdown displayed
  - When parent was hovered
  - Change was based on relationship
    - Based on structure

We can do something similar with JS

# Learning about the event

When the handler (the callback) is called

- Passed an "event object"
- Our examples so far have ignored it
  - JS fine with passing values that aren't used
  - Only one function of a given name in scope
    - Nothing based on arguments

```
function test( a, b ) {  
  console.log(a, b);  
}  
  
test("one", "two"); // one two  
test("one"); // one undefined  
test("one", "two", "three"); // one two
```

# Event object

```
buttonEl.addEventListener('click', (event) => {  
  console.log(event);  
});
```

A common convention is to call it `e`

- They clearly didn't take my course :(

```
buttonEl.addEventListener('click', (e) => {  
  console.log(e);  
});
```



# The Event Target

- Event object has a LOT on it
- One important part is the `.target` property
  - DOM Node of element that got the event
  - Ex: the button element that was clicked
- But we already know this!
  - We had element node to add event listener

**Node w/listener NOT always event origin element**

# Event Propagation

Also known as "event bubbling"

Event happens to element

- Listeners on that node happen
- THEN event happens to parent element
  - Repeat, then to THAT element's parent
  - etc, until no more parent elements

**Event Propagation**

# Event Propagation has benefits

- If you have multiple targets
  - Ex: Many buttons for dropdown menus
- Can add one listener
  - On a common ancestor element
- When event triggers `event.target` is actual button
  - ...or some other descendant element
  - You are getting ALL the clicks
  - Want to filter out targets you don't care about
  - Check for a class identifying the category

# Propagation and Filtering Example

```
<div class="cards">
  <div class="card">
    <h2 class="card__title">Title</h2>
    
    <p class="card__text">Dolor sunt soluta suscipit praesentium perferendis. Expedi
    <button class="card__link" type="button">Activate</button>
  </div>
  <!-- more cards -->
</div>
```

```
const cardsEl = document.querySelector('.cards'); // ancestor
cardsEl.addEventListener('click', (e) => {
  if( e.target.classList.contains('card__link') ) {
    console.log('an activate button was clicked');
  } else {
    console.log('something else inside .cards was clicked');
  }
});
```

# When you have the right element event

- Now know when the element we want was clicked
  - Filtered out event on other elements
- Can handle many such target elements!
  - Any descendant of the ancestor
  - That passes filter
- But we don't want to alter that element!
  - Ex: To modify `.card` div when button clicked

# Can further select elements based on relationship

- Each element has `.querySelector()`
  - Will search descendants
- Each element has `.closest()`
  - Will search ancestors
- `~` and `+` selectors can search siblings

```
const cardsEl = document.querySelector('.cards'); // ancestor
cardsEl.addEventListener('click', (e) => {
  if( e.target.classList.contains('card__link') ) {
    // e.target is the button element
    const cardEl = e.target.closest('.card');
    cardEl.classList.toggle('card--active');
  }
});
```

# Why did our card use a button?

- This is semantically correct
  - Button for controls
  - Link for navigation
- What if we are styling HTML someone else made?
  - Okay, use card with link

```
<div class="cards">
  <div class="card">
    <h2 class="card__title">Title</h2>
    
    <p class="card__text">Dolor sunt soluta suscipit praesentium perferendis. Expedi
    <a class="card__link" href="/fake">Activate</a>
  </div>
  <!-- more cards -->
</div>
```

# The link will navigate!

- Leaving the page resets our page state
  - JS has to start over on reload
- If link doesn't navigate
  - Should not BE a link (semantically)
  - Can style a button as a link appearance
- But if we MUST use a link:
  - Could use `href="#"`
    - But this has complications!
    - Alters url
    - Considered an in-page scroll-to-element



# Prevent Default Action

- Navigation is the "default" for a link
- Form Submit is the "default" for a submit button
- Defaults happen AFTER other event handlers
  - Including on event ancestors
- event object has a `.preventDefault()` method

```
whateverEl.addEventListener('click', (e) => {  
  // Any other code  
  e.preventDefault();  
});
```

# What to use as href?

- Avoid problem: **only use links for navigation**
  - Use buttons for controls
  - Better accessibility!
- When existing href is there
  - Such as Progressive Enhancement
    - Making a page work without/with JS
  - Just leave existing href
  - It is a valid url to visit
- If you MUST use link AND there is no existing href
  - Use `#` to match all the sites that break a11y
  - But `e.preventDefault()` to stop navigation

# Many matching selectors

- `.querySelector()` returns first matching node
- `.querySelectorAll()` returns ALL matching nodes

```
const nodes = document.querySelectorAll('.card');
```

- Returns a NodeList
  - Any **array-like** collection of Nodes
  - indexed like an array
  - Lacks many array methods
  - If you need an actual array

```
const someArray = Array.from(arrayLike);
```

# querySelector/All search descendants

```
document.querySelector('.title')
```

- Searches entire document

```
someNode.querySelector('.title')
```

- Searches descendants of `someNode`

`.querySelector()` is reverse `.closest()`

- Same applies to `.querySelectorAll()`

# Looping

For all items in collection

- Do something

Looping is a very common need in programming

JS has many options for looping

# C-Style For Loop

Rarely a good choice!

- But very common despite better options

```
const cats = ['Jorts', 'Jean', 'Nyancat'];  
  
for ( let i = 0; i < cats.length; i++ ) {  
  console.log( cats[i] );  
}
```

- A lot going on
- Creates an **index variable** (*i*)
  - But we don't care about *i*
  - We just want the element of the array

# For...of loop

A `for...of` loop

- Loops over elements of **iterable objects**
  - arrays, strings, NodeLists, etc
  - Included arrays created on the spot
    - using `Object.keys()`, etc
- No index variable

```
const cats = ['Jorts', 'Jean', 'Nyancat'];  
  
for ( const cat of cats ) {  
  console.log(cat);  
}
```

# What kind of collection?

- Arrays
- Array-likes
- Objects

Can convert!

- Array-likes to arrays using `Array.from()`
- Object to arrays of keys using `Object.keys()`
- Object to arrays of values using `Object.values()`
- Objects to pairs using `Object.entries()`



# for...of with objects

```
const cat = {  
  name: 'Jorts',  
  age: 3,  
  color: "orange tabby",  
};  
  
for( const key of Object.keys(cat) ) {  
  console.log( `${key}: ${ cat[key] }` );  
}
```

# for...in works with objects

- Easy to confuse `for...in` with `for...of`
- Will iterate over inherited properties
  - But most objects don't inherit
- Much old advice: problems that don't happen
- `for...in` is rare
  - `for...of` `Object.keys()` more common

```
const cat = {  
  name: 'Jorts',  
  age: 3,  
  color: "orange tabby",  
};  
  
for( const key in cat ) {  
  console.log( `${key}: ${ cat[key] }` );  
}
```

# Array forEach

Arrays have a `.forEach()` method

- Array-likes may NOT
- Pass a callback
- Callback is called for each element
  - Callback can ALSO get an index variable

```
const cats = ['Jorts', 'Jean', 'Nyancat'];

cats.forEach( (cat) => {
  console.log(cat);
});

cats.forEach( (cat, index) => {
  console.log(`element ${index} is ${cat}`);
});
```

# **.forEach() vs for...of**

Which to use?

- Often Personal style
- `for...of` probably less overhead
  - But not enough to be important
- Do you need an index value?
- What are you trying to communicate with code?
  - `for...of` emphasizes loop
  - `forEach` emphasizes the callback contents

# forEach() vs map()

Both `.forEach()` and `.map()` loop with a callback

- `.forEach()` is NOT about creating a new array
- `.map()` is for creating a new array
  - Will use this later to generate HTML from data

# More JS Syntax

- Function defaults
- Destructuring
  - Named Function Params
- Spread operator
- Rest operator

# Function Defaults

- We already talked about "defaulting" a value
  - `name ||= "Jorts";`
  - same as `name = name || "Jorts"`
- Functions have options to default arguments

```
function greet( message = "Hello", target = "World" ) {  
  console.log(` ${message} ${target}`);  
}  
  
greet(); // Hello World  
greet("Hi"); // Hi World  
greet("Hi", "Class"); // Hi Class  
greet(undefined, "Class"); // Hello Class  
greet(null, "Class"); // null Class (!)
```

Only defaults on `undefined`, not nullish or falsy

# Destructuring

- Common: variables from object properties
  - Makes code easier to skim/read
  - May pass these variables to other functions

```
const name = cat.name;  
const age = cat.age;
```

- **Destructuring** (de-structure) does this
  - Can do to objects or arrays
  - Declares and assigns new variables



# Destructuring Objects

- Destructure objects using 

```
const cat = {  
  name: "Jorts",  
  age: 3,  
  color: "Orange Tabby",  
};  
  
const { name, age } = cat; // Declares and assigns  
  
console.log( cat ); // same as above  
console.log( name ); // "Jorts"  
console.log( age ); // 3  
// there is no "color" variable
```

# Destructuring Arrays

- Destructure using `[]`
  - Less common than objects
  - Will be used in React

```
const cats = [ 'Jorts', 'Jean', 'Nyancat' ];  
  
const [ first, second ] = cats; // Declares and assigns  
  
console.log( cats ); // same as above  
console.log( first ); // "Jorts"  
console.log( second ); // "Jean"  
// There is no variable for "Nyancat"
```

# Spread Operator

The **spread operator**

- is `...` before array or object (collections)
- replaces with the individual elements/pairs
- Used to copy/extend/merge objects
- Used to copy arrays
- Used to pass individual elements as parameters

# Shallow Copy of Objects using Spread

- spread operator allows a **shallow copy** of object
  - Items that are collections remain references

```
const cat = {  
  name: "Jorts",  
  age: 3,  
  toy: {  
    type: "pipe cleaner",  
    condition: "poor",  
  }  
};  
  
const copy = { ...cat };  
copy.age = 4;  
copy.toy.condition = "terrible";  
  
console.log(cat); // age is 3, toy.condition is terrible  
console.log(copy); // age is 4, toy.condition is terrible
```

# Extend/Merge Objects using Spread operator

- Objects created with repeated keys
  - Use "last" value for that key

```
const cat = {  
  name: "Jorts",  
  age: 3,  
  name: "Jean",  
};  
  
console.log(cat.name); // Jean
```

- This allows you to extend/merge objects


```
const feline = {  
  sleeping: true, // a "default", can be overwritten by cat  
  ...cat,  
  hungry: true, // will overwrite value in cat  
};
```

# Array Spread operator

- Used to copy arrays
- Used to nest array contents
- Could use array methods
  - This does not mutate array
  - That becomes important for React

```
const cats = [  
  "Jorts",  
  "Jean",  
  "Nyancat",  
];  
  
console.log( [ "Maru", cats ] ); // Nested array :(  
console.log( [ "Maru", ...cats ] ); // New, "flat" array
```

# Rest Operator

- Uses 
  - but is not spread operator
- Collects "remaining" elements into array
  - "the rest"
- Used for function arguments
- Used when spreading arrays
- Not used in this course
  - Good to recognize as different than spread

```
const cats = [ "Jorts", "Jean", "Maru" ];  
const [ first, ...otherCats ] = cats;  
  
console.log( otherCats ); // [ "Jean", "Maru" ]
```