Before we start

Before we begin working with Polymer we need to review ES2015 classes and look at the Custom Elements V1 specification.

All browsers are building or have shipped support for the Custom Element and Shadow DOM V1 specifications; Chrome shipped an earlier implementation of the specs (known as V0) and will support it for the forseeable future but you should do all your work with the new specs and syntax and use polyfills for browsers that do not support it yet.

We will also look at extending classes, using mixins to add functionality and how to incorporate other specifications like HTML templates and Shadow DOM V1 into the mix to make our lives easier and create trully reusable code.

Because we use classes and string template literals we will only work with ES6. There will be no ES5 examples

Revisiting ES2015 Classes

I was surprised to find classes as part of ES6... Ever since the language was created developers have had to learn to work with prototypal inheritance, how to make that chain work for us and what are the pitfalls to avoid when doing so. The first question I asked when I started looking at ES6 was what are classes in ES6?

They're syntactic sugar on top of prototypical inheritance. Everything we'll discuss in this section is built on top of the traditional prototypal inheritance framework and the parser will use that under the hood while we do all out shiny work with classes. This will be our example to look at how classes work. It defines a class `Person` with three default elements:

- `first` name
- `last` name
- `age`

It also defines 2 methods:

- `fullName` returning the first and last name
- `howOld` returing the string `${this.name}` is `${this.age}` years old

Note that the methods use template literals with string interpolations. The backticks are required for string literals and not an artifact of writing in Markdown

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


  fullName() {
   return `${this.first} ${this.last}`;
  }

  howOld() {
    return `${this.first} is ${this.age} years old`;
  }
}
```

To instantiate a new object of class Person we must use the new keyword like so:

```
const carlos =  new Person('carlos', 'araya', 49);
```

Now we can use the class methods like this:

```
carlos.fullName();
// returns "carlos araya"
carlos.howOld();
// returns "carlos is 49 years old"
```

Subclasses, extends and super

Now let’s say that we want to expand on the Person class by assigning additional attributes to the constructor and add additional methods. We could copy all the material from the Person class into our new class, call it Engineer, but Javascript saves us from having to do so. The `extends` keyword allows us to use a class as the basis for another one.

To continue the example, we'll extend `Person` to describe an `Engineer`. We'll say that the Engineer is a person with all the attributes and methods of the Person class with 2 additions:

- They belong to a `department`
- They have a favorite programming language represented by the `lang` parameter

The Engineer class is presented below:

```
class Engineer extends Person {
  constructor(first, last, age, department, lang) {
    // super calls the parent class constructor for the listed attributes
    super(first, last, age);
    this.department = department;
    this.language = lang;
  }

  affiliation() {
     return `${this.first} is in the ${this.department} department`;
  }

  favoriteLanguage() {
     return `${this.first} favorite language is ${this.language}`;
  }`${this.first} is in the ${this.department} department`
```

In the constructor we pass the same three values as for the Person Class but, rather than assign them parameters like we did in the Person class, we simply call the constructor of the parent class using the `super` keyword.

The instruction `super(first, last, age)` tells the parser to hand the attributes to the parent class for processing. The attributes that are particular to the Engineer class (`department` and `language`) are assigned normally.

We then add the methods specific to the Engineer class: What department they belong to and what's their favorite programming language.

The instantiation is the same as before

```
const william = new Engineer('William', 'Cameron', 49, 'engineering', 'C+
```

Now the cool part. Even though we didn't add the methods `fullName` and `howOld` to the Engineer class we get them for free becaure they are defined in the parent. With william (defined above) we can call methods from the parent class:

```
william.fullName();
// returns "William Cameron"

william.howOld();
// returns "William is 49 years old"
```

And we also get the methods that are exclusive to our Engineer class:

```
william.affiliation();
// returns "William is in the engineering department"

william.favoriteLanguage();
// returns "William favorite language is C++”
```

static class methods

This is something that still trips me but that will be necessary when working with Polymer elements later on.

According to MDN:

> Static methods are called without instantiating their class and are also not callable when the class is instantiated. Static methods are often used to create utility functions for an application.
>
> — MDN Javascript Reference: static keyword

In order to call a static method within another static method of the same class, you can use the this keyword.

```
class StaticMethodCall {
  static staticMethod() {
     return 'Static method has been called';
}
static anotherStaticMethod() {
    // Note the use of this to reference staticMethod()
    return this.staticMethod() + ' from another static method';
  }
}

StaticMethodCall.staticMethod();
' from another static method'// 'Static method has been called'ticMethod(
// 'Static method has been called from another static method'
'Static method has been called from another static method'
```

Static methods are not directly accessible using just the this keyword. You need to call them by using the class name in front: CLASSNAME.STATIC_METHOD_NAME (like other calls from outside of class) or by using the constructor property: `this.constructor.`STATIC_METHOD_NAME.

```
class StaticMethodCall{
  constructor(){
     console.log(StaticMethodCall.staticMethod());
     // 'static method has been called'

     console.log(this.constructor.staticMethod());
     // 'static method has been called'
  }

  static  staticMethod(){
     return 'static method has been called.';
  }
}
```

Static Method Example

In this example we create three classes:

- Triple is our base class. It defines a static `triple` method
- BiggerTriple extends Triple and defines its own version of the `triple` method that returns a different value than the parent class
- TP is defined as a literal object

```
class Triple {
  static triple(n) {
     if (n === undefined) {
       n = 1;
     }
     return n * 3;
  }
}

class BiggerTriple extends Triple {
  static triple(n) {
     return super.triple(n) * super.triple(n);
  }
}
```

The first example calls `Triple.triple` without a value wich causes it to default to one. The function will return 3 as the value

```
console.log(Triple.triple());        // 3
```

The second example calls `Triple.tripe` with a value which will be multiplied by 3 and return 18.

```
console.log(Triple.triple(6));       // 18
```

This example calls `BiggerTriple.triple` which is a derived class and uses its own definition of the `triple` method.

```
console.log(BiggerTriple.triple(3)); // 81
```

We next define a new instance of the Triple class

```
var tp = new Triple();
```

`BiggerTripple.triple` still returns a value using its own definition of the triple method that uses the static methods from the parent class.

```
console.log(BiggerTriple.triple(3));
// 81 (not affected by parent's instantiation)
```

However `tp.triple` will fail because the class has not been defined, only instantiated.

```
console.log(tp.triple());
// 'tp.triple is not a function'.
```

If we wanted this to work we would have to instantiate the `tp` class and create its own version of `triple`. Something like this:

```
class TP extends Triple {
  static triple(n) {
     return super.triple(n);
  }
}
```

Class expressions

Just like with functions we can use literal expressions to create class definitions. We can use unnamed and named declarations. The advantage of using named classes when creating them is that we can reuse them in other parts of our code.

```
// unnamed
let Polygon = class {
   constructor(height, width) {
      this.height = height;
      this.width = width;
   }
};

// named
let Polygon2 = class Polygon {
   constructor(height, width) {
      this.height = height;
      this.width = width;
   }
};
```

These will become important when we start building mixins in the next section. We'll use anonymous class expressions to create the mixin functions before we "mix" them with our classes.

Mixins

This is a concept I borrowed from Polymer, I'm researching if this can be used in regular web components and standardES6 classes. I will not document it any further.

Mixins are another name for abstract classes. We use mixins to get around the fact that ECMAScript can have only one superclass. This makes it impossible to create classes with multiple parents, each of which adds a small piece of functionality to improve our code. By creating mixins we can create these smaller "library" classes and we can add them to our tools.

Any function with a superclass as input and a subclass extending that superclass as output can be used to implement mix-ins in ECMAScript. Below are some literal class expressions that create mixins:

```
var engineerrMixin = Person => class extends Person {
  affiliation() { }
  language() { }
};

var programmerMixin = Person => class extends Person {
  editor() { }
  platofrm() { }
};
```

A class that uses these mix-ins can then be written like this:

```
class Person { }
class Employee extends engineerMixin(ProgrammerMixin(Person)) { }
```

The code is a little hard to understand, particularly if you're not familiar with how to extend a class with multiple mixin files. What is says is that class `Employee` uses both `engineerrMixin` and `programmerMixing` with `Person` as the base class. Whenever using `Employee` we have access to the methods of the base class `Person` the methods in both mixin classes (`engineerMixin` and `programmerMixin`) and any methods exclusive to `Employee`.

A fully realized `engineerMixin` could look like this:

```
var engineerMixin = Person => class extends Person {
  affiliation() {
      return `${this.first} is in the ${this.department} department`;
  }

  favoriteLanguage() {
     return `${this.first} favorite language is ${this.language}`;
  }
};
```

And then we can use the mixin by using when defining another subclass of Person. We get all the methods in the Person base class, all the methods of the engineer mixin and whatever methods we add to the Employee subclass.

```
class Employee extends (engineerMixin(Person)) {
  // Define methods of employee here
}
```

Notes, caveats and potential pitfalls

I found several gotchas to consider when working with classes. Here are some of them:

Classes do not hoist

When working with functions is ok to use a functions before it’s defined:

```
drawRectangle();

function drawRectangle() {
  // Definition goes here
};
```

In classes this will return a Reference Error as illustrated in the example below. The class must be defined before it’s used either in the same file, an external file imported in to the class or in the same inline script.

```
var p = new Polygon();
// This will produce: ReferenceError: Polygon is not defined

class Polygon {}
```

To make it work define the class first then the definition will work.

```
class Polygon {}

// This will work as intended
var p = new Polygon();
```

Always runs on strict mode

The bodies of class declarations and class expressions are executed in strict mode. This may introduce additional complications if you’re not familiar with strict mode in ES5.

Mozilla provides a good definition of strict mode and a transition guide from non-strict to strict mode

Custom Elements v1

There's a new version of custom elements going through the standards process, it is known as V1 to distinguish it from the original version implemented in Chrome and familiar if you've used Polymer 1.x. before. Custom Elements and Shadow DOM V1 are the specs making it to browsers so it pays to learn it now and get up to speed before the specs and the libraries supporting them upgrade.

All examples in this chapter will extend the basic `HTMLElement` interface. We do this for convenience and to provide a basic set of functionality for the `athena-card` components.

We can be more specific in what element definitions we extend. The full list of HTML element interfaces is part of the HTML spec maintained by WHATWG.

For example our custom elements that behave like buttons could extend `HTMLButtonElement` instead of the basic `HTMLElement`. In the end all elements will inherit from `HTMLElement` but the more specialized interfaces defined in the specification can do more for your content than inheriting from the generic base interface.

The basics

```
class AthenaDiv extends HTMLElement {
    constructor() {
      // call HTMLElement constructor so we can take advantage
      // of HTMLElement features
      super();
    }

    connectedCallback() {
      this.innerHTML = `
        <style>
          div.card {
            border: 5px solid red;
            display: inline-block;
            margin: 10px;
```

```
             paddin`
         <style>
           div.card {
             border: 5px solid red;
             display: inline-block;
             margin: 10px;
             padding: 10px;
             max-width: 300px;
             width: 100%;
           }
         </style>

         <div class="card">
           <h2>Default Title</h2>
         </div>
       `// Defines the athena-card element and
// associates it with the AthenaDiv class
window.customElements.define('athena-card', AthenaDiv);
```

You can use this element as is. Call it using the tag name we defined for it:

```
<athena-card></athena-card>
```

Defining the element Javascript API

Once we decide what our element is what it will do we can build it’s API. Think of this API as a contract between the developer and user of the element about what the element can do and how it is configured to do it.

The API can take advantage of all the ES2015 class features. In the example below we add a getter and setter for a disabled property, a constructor and a hypothetical toggleDisplay property to toggle the element’s visibility.

```
class AthenaDiv extends HTMLElement {
  // A getter/setter for a disabled property.
  get disabled() {
    return this.hasAttribute('disabled');
  }

  set disabled(val) {
    'disabled'// Reflect the value of the disabled property as an HTML at
      this.setAttribute('disabled', '' );
    } else {
      this.removeAttribute('disabled');
    }
  }

  // Can def'disabled'// Can define constructor arguments if you wish.
  constructor() {
    // If you define a constructor, always call super() first!
    // This is specific to Custom Elements and required by the spec.
    super();

    // Setup a click listener on <athena-card> itself.stener('click', e =>
      // Don't toggle the 'click'// Don't toggle the card if it's disable
        return;
      }
      this.toggleDisplay();
    });
  }

  toggleDisplay() {
    console.log('toggled');
  }
}

// Defines the athena-card 'toggled'// Defines the athena-card element and
window.customElements.define('athena-card', AthenaDiv);
```

Reaction Callbacks

Name	Called When
constructor	An instance of the element is created or upgraded.

Name	Called When
	Useful for initializing state, settings up event listeners, or creating shadow dom. See the spec for restrictions on what you can do in the constructor.
connectedCallback	Called every time the element is inserted into the DOM. Useful for running setup code, such as fetching resources or rendering. Generally, you should try to delay work until this time
disconnectedCallback	Called every time the element is removed from the DOM. Useful for running clean up code (removing event listeners, etc.).
attributeChangedCallback(attrName, oldVal, newVal)	An attribute was added, removed, updated, or replaced. Also called for initial values when an element is created by the parser, or upgraded in place. **Note**: only attributes listed in the **observedAttributes** property will receive this callback.
adoptedCallback()	The custom element has been moved into a new document (e.g. someone called document.adoptNode(el)).

The browser calls the attributeChangedCallback() for any attributes whitelisted in the observedAttributes array (see Observing changes to attributes). Essentially, this is a performance optimization. When users change a common attribute like style or class, you don't want to be spammed with tons of callbacks.

Reaction callbacks are synchronous. If someone calls el.setAttribute(...) on your element, the browser will immediately call attributeChangedCallback(). Similarly, you'll receive a disconnectedCallback() right after your element is removed from the DOM (e.g. the user calls el.remove()).

User content

As good as custom elements are are not enough to convey both structure and content of our elements and applications.

> We have three ways to add content to our custom elements:
>
> - Child content
> - Content created in the Shadow DOM
> - Using slots for fun and profit
> - Content created in templates

We'll explore each of these in turn.

Child Content

Custom elements can manage their own content by using the DOM APIs inside element code. Reactions come in handy for this. In this example we add HTML text to the element when it's connected to the DOM

```
customElements.define('athena-card-with-markup', class extends HTMLElemen
  connectedCallback() {
     this.innerHTML = "<strong>I'm an athena-card-with-markup!</strong>";
  }
});
```

This declaration will produce:

```
<athena-card-with-markup>
 <strong>I'm an athena-card-with-markup!</strong>
</athena-card-with-markup>
```

Shadow DOM

Shadow DOM provides a way for an element (custom or not) to own, render, and style a chunk of DOM that's separate from the rest of the page. This is how native elements such as video, audio, select hide all the elements they use to display the video natively.

Until custom elements and shadowDOM were introduced, hiding implementations this way was only possible for browser vendors

If we wanted to, we could hide an entire set of functionality inside a single tag:

```
<!-- athena-card's implementation details are hidden away in Shadow DOM. -
<athena-card></athena-card>
```

To use Shadow DOM in a custom element:

1. Call `super()` as the first line inside the constructor
2. Call attachShadow (using `this.attachShadow`) inside your constructor
3. Inside the shadow root: Define your scoped styles
4. Inside the shadow root: Define any content you want the element to have
5. Inside the shadow root: Define any slots to assign content to

```
customElements.define('athena-card', class extends HTMLElement {
  constructor() {
    super();                                                     // 1

    // Attach a shadow root to the element.
    const shadowRoot = this.attachShadow({mode: 'open'});  'open'// 2
    shadowRoot.innerHTML = `
       <style>:host { }</style>                                 // 3
       <strong>I'm in shadow dom!</strong>                      // 4
       <slot></slot>`;                                          `
       <style>:host { }</style>                                 // 3
       <strong>I'm in shadow dom!</strong>                      // 4
       <slot></slot>``
       <style>:host { }</style>                                 // 3
       <strong>I'm in shadow dom!</strong>                      // 4
       <slot></slot>`// 5
  }
});
```

Using the definition of athena-card we've worked with in this section, the p element in the example below will replace the slot element and will be part of the athena-card element stamped to the DOM along with whatever styles we've added for the `:host` pseudo element.

```
<athena-card>
  <p><strong>Us<p>s</strong> custom text</p>
</athe</strong>
<!-- renders as -->
<athena-card>
  <strong>I'm in shadow dom!</strong>
  <p><strong>User's</strong> custom text</p>
</athena-card>
```

We'll cover slots in more detail in the following section.

Slots for fun and profit

In our previous examples we could merge content from our light DOM into the shadow DOM but it was a heavy handed approach that didn't lend itself very well to composing elements and applications.

Composability is really big when working with web components. Custom Elements V1 gives us slots as a primitive to compose elements of the Light DOM (the DOM we use in our day to day HTML elements) into the Shadow DOM.

When using slots we can use named slots to match content in the light DOM or we can create generic slots (without names) that will work as pass through for elements in the light DOM that don't have a named slot associated with them.

In the example below for `athena-card` we do the following tasks:

1. We create the class `AthenaDiv` and extend `HTMLElement`
2. In the `constructor` we call `super()` as required to use HTMLMElement
3. In the `connectedCallback` we attach the ShadowDom to the element using `this.attachShadow` and set its mode to open
4. In the shadow root's `innerHTML` we add the styles for our element
5. In the shadow root's `innerHTML` we add the layout of our element and default data that will be used when we enter no data in the light DOM
6. Define the custom element and associate it with the class we just created

```
class AthenaDiv extends HTMLElement {                                // 1
    constructor() {
      // call HTMLElement constructor so we can take advantage
      // of HTMLElement features
      super();                                                       // 2
    }

    connectedCallback() {
      let shadow = this.attachShadow({mode: 'open'});                'open'//
      shadow.innerHTML = `
        <style>                                                      // 4
          div.card {
            border: 5px solid red;
            display: inline-block;
            margin: 10px;
            padding: 10px;
            max-width: 300px;
            width: 100%;
          }

          ::slotted(h2) {
 `
```

```
        <style>                                                      // 4
          div.card {
            border: 5px solid red;
            display: inline-block;
            margin: 10px;
            padding: 10px;
            max-width: 300px;
            width: 100%;
          }

          ::slotted(h2) {
            text-align: center;
          }
          slotted(p) {
            font-size: 1.125em;
          }
        </style>

        <div class="card">                                      // 5
          <h2><slot name="card-header">Default Title</slot></h2>
          <p><slot name="card-body">Default content</slot></p>
          <p class="footer">
            Written by <slot name="card-author">carlos</slot>
          </p>
          <slot></slot>
        </div>
      `// Defines the athena-card element and associates it with the Athe
window.customElements.define('athena-card', AthenaDiv);         // 6
```

The HTML examples below give three possible definitions of the `<athena-card>` component.

The first example is fully populated from the light DOM. We've given all the data and, using the `slot` attribute we've told the custom element how to compose the light and shadow DOM elements.

```
<athena-card>
<h2 slot="card-header">Card Header</h2>

<p slot="card-body">
  "Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusan
  doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inven
  veritatis et quasi architecto beatae vitae dicta sunt explicabo. Nemo e
  ipsam voluptatem quia voluptas sit aspernatur aut odit aut fugit, sed q
  consequuntur magni dolores eos qui ratione voluptatem sequi nesciunt."

<p slot="card-author">
  <a href="https://twitter.com/domfarolino">domfarolino</a>
</p>
</athena-card>
```

The second example uses a partial mix. The `card-header` and `card-body` attributes are taken from the light DOM. The `card-author` slot will be filled from the shadoDOM since the light side doesn't provide a value for it.

```
<athena-card>
<h2 slot="card-header">Another card</h2>

<p slot="card-body">
  "Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusan
  doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inven
  veritatis et quasi architecto beatae vitae dicta sunt explicabo. Nemo e
  ipsam voluptatem quia voluptas sit aspernatur aut odit aut fugit, sed q
  consequuntur magni dolores eos qui ratione voluptatem sequi nesciunt."

<p slot="card-author"></p>
</athena-card>
```

The final example is populated exclusively from the shadow DOM. The light side provides no data to compose so the shadow DOM takes over and uses the default values to populate the content.

```
<athena-card></athena-card>
```

Template elements

For those unfamiliar, the template element allows you to declare fragments of DOM which are parsed, remain inert when a user loads the page, and can be activated later at runtime.

In the “not so distant future ™” we will be able to use templates everywhere; they are already supported in all major browsers (holdouts are IE 11, Opera Mini and UC browser for Android according to caniuse.com) without workarounds and strange uses of scripts and other strategies

The idea of using templates with custom elements is that we populate the template with the CSS and HTML we want to use and then insert that into the shadow DOM

```
<template id="athena-card-from-template"> // 1
  <style><style> // 2
    div.card {
      border: 5px solid red;
      display: inline-block;
      margin: 10px;
      padding: 10px;
      max-width: 300px;
      width: 100%;
    }
    h2 {
      text-align: center;
    }
    p {
      font-size: 1.125em;
    }  >

  <div class="card"> // 3
    <h2><slot name="card-header">Default Title</slot></h2>
    <p><slot name="card-body">Default content</slot></p>
    <p class="footer">
      Written by <slot name="card-author">carlos</slot>
    </p>
    <slot></slot>
  </div>
</template>

<script> // 4
customElements.define('athena-<div class="card">', class extends HTMLEleme
constructor() {
  super(); // always call super() first
  let shadowRoot = this.attachShadow({mode: 'open'}); // 5
  const t = document.querySelector('#athena-card-from-template'); // 6
  const instance = t.content.cloneNode(tru // 4
customElements.define('athe'athena-card-from-template'ss extends HTMLEleme
constructor() {
  super(); // always call super() first
  let shadowRoot = this.attachShadow({mode: 'open'}); 'open'// 5st t = do
 '#athena-card-from-template'// 6
  const instance = t.content.cloneNode(true); // 7
  shadowRoot.appendChild(instance); // 8
}
});
```

The script does a lot and it's not intuitive to understand. Let's unpack it and see what it's actually doing:

1. We create a template and assign it an id of `athena-card-from-template`
2. Inside the template assign the styles for the element
3. Inside the template build the layout for the element using slots for composition along with any default data
4. In the script associated with the element define the element and what class it extends. In this example we use an anonymous class to define the element
5. Create the shadow root using `this.attachShadow`
6. Query the document for the template using `document.querySelector` and using the id of the template as a parameter
7. Clone the content of the template
8. Append the cloned content to the shadow root
9. Instantiate the element

Reflecting properties to attributes

Whenever we set a value for an attribute of an HTML tag it is reflected back to the DOM representation of our element. The most common I've seen is when assigning a `src` attribute to an image tag. In the example below we'll assign an `id` and `src` attributes to an image tag

> For example, when the values of hidden or id are changed in JS:

```
let dinosaur01 = new Image();
dinosaur01.id = 'image001';
dinosaur01.src = 'images/image001.png';
```

the values are applied to the live DOM as attributes:

```
<img id="image001" src="images/image001.png">
```

This is called “reflecting properties to attributes”. Almost every property in HTML does this. Why? Attributes are also useful for configuring an element declaratively and certain APIs like accessibility and CSS selectors rely on attributes to work.

Reflecting a property is useful anywhere you want to keep the element’s DOM representation in sync with its JavaScript state. One reason you might want to reflect a property is so user-defined styling applies when JS state changes.

Recall our `<athena-card>`. A consumer of this component may want to dim it out and prevent user interaction when it’s dimmed:

```
athena-card[dimmed] {
  opacity: 0.5;
  pointer-events: none;
}
```

When the disabled property is changed in JS, we want that attribute to be added to the DOM so the user’s selector matches. The element can provide that behavior by reflecting the value to an attribute of the same name.

Using this setter/getter pair we can change and retrieve the value of the disabled attribute. We’ll discuss why this is important when we talk about observing changes to attributes.

Observing changes to attributes

HTML attributes are a convenient way for users to declare initial state for our components. In the example below, `athena-card` a **dimmed** to indicate whether the card is active or not.

```
<athena-card dimmed></athena-card>
```

Elements can react to attribute changes by defining a attributeChangedCallback. The browser will call this method for every change to attributes listed in the observedAttributes array.

```
class AthenaDiv extends HTMLElement {
```

```
<template id="athena-card-from-template">
  <style>
    .card {
      border: 5px solid red;
      display: inline-block;
      margin: 10px;
      padding: 10px;
      max-width: 300px;
      width: 100%;
    }

    h2 {
      text-align: center;
    }

    p {
      font-size: 1.125em;
    }


  </style>

  <div class="card">
    <h2><slot name"card"-header">Default Title</slot></h2>
    <p><slot name="card-body">Default content</slot></p>
    <p class="footer">
      Written by <slot name="card-author">carlos</slot>
    </p>
    <slot></slot>
  </div>
</template>

<script>
  customElements.define('athena-card-from-template', class extends HTMLEle
    constructor() {
      super(); "card-body"// always call super() first shadowRoot = this.
      const t = document.querySelector('#athena-card-from-template');
      const instance = t.content.cloneNode(true);
      shadowRoot.appendChild(instance);
    }
    static get observedAttributes() {
      return ['dimmed'];
    }

    get dimmed() {
      return this.hasAttribute('dimmed');
    }

    set dimmed(val) {
      if (val) {
      // Reflect 'open'// Reflect the value of `dimmed` as an attribute.
        this.setAttribute('dimmed', '');
      } else {
        this.removeAttribute('dimmed');
      }
    }

  attributeChangedCallback(name, oldValue, newValue) {
   if (this.dimmed) {
     this.setAttribute('tabindex', '-1');
     this.setAttribute('aria-disabled', 'true');
   } else {
      this.setAttribute('tabindex', '0');
      this.setAttribute('aria-disabled', 'false');
   }
  }

  });
</script>

  <athena-card-from-template dimmed>
    <h2 slot='card-header'>Surprise</h2>

    <p slot='card-body'>Even more surprises!</p>
  </athena-card-from-template>
```

In the example, we’re setting additional attributes on `<athena-card>` when we add or remove the dimmed attribute. We also change accessibility attributes to make sure it works as intended for assistive technology. Depending on your application you may want to add additional attributes using the same formula shown for `tab-index` and `aria-disabled`.

If the attribute is not listed in our `ObservedAttributes` array the attributeChangedCallback event will not fire.

Styling custom elements

Even if your element defines its own styling using Shadow DOM, users can style your custom element from their page. These are called “user-defined styles”.

```
<!-- user-defined styling -->
<style><style>
  athena-card {
    display: flex;
  }
  panel-item {
    transition: opacity 400ms ease-in-out;
    opacity: 0.3;
    flex: 1;
    text-align: center;
    border-radius: 50%;
  }

  panel-item:hover {
    opacity: 1.0;
    background: rgb(255, 0, 255);
    color: white;
  }
</style>

<athena-card>
  <panel-item>Do</panel-item>
  <panel-item>Re</panel-item>
  <panel-item>Mi</panel-item>
</athena-card>
```

You might be asking yourself how CSS specificity works if the element has styles defined within Shadow DOM. In terms of specificity, user styles win. They’ll always override element-defined styling. See the section on Creating an element that uses Shadow DOM.

Loading custom elements: HTML Imports

One of the most contentious aspects of web components is how to load them.

HTML Imports has made it into two browsers (Chrome and Opera) and 2 browsers (Firefox and Safari) will not support it until ES6 Modules are implemented in browsers so they can evaluate whether modules will work for importing custom elements into applications.

At its simplest, HTML Imports add a new value for the `rel` attribute of the `link` tag.

If the file `athena-card.html` contains a custom element then we can use the following link to import it into our document.

```
<link rel="import" href="athena-card.html">
```

By declaring `rel="import"` and providing a reference to a URL containing a web component we can import the component, and all the data inside it, to use locally on our application.

Bundling libraries with imports

One of the biggest pains when using libraries like Bootstrap or Foundation is the number of files we have to download and how hard it is to customize. Using a single HTML import we can bundle the content from Bootstrap into a single download

The content of the imported file may look something like this:

```
<link rel="stylesheet" href="bootstrap.css">
<link rel="stylesheet" href="fonts.css">
<sc<link rel="stylesheet" href="fonts.css">rc="bootstrap.js"></script>
<script src="bootstrap-tooltip.js"></script>
<script src="bootstrap-dropdown.js"></script>
```

```
<!-- scaffolding ma</script><!-- scaffolding markup -->
<template>
  ...
</template>
```

and be called

```
<link rel="import" href="bootstrap.html">
```

Using part of an imported file

content of warning.html

```
<div class="warning">
  <style><style>
    h3 {
      color: red !important;
    }
  </style>
  <h3>Warning!</h3>
  <p>This page is under construction</p>
</div>

<div class="outdated">
  <h3>Heads up!</h3>
  <p>This content may be out of date</p>
</div>
```

importing `warning.html` and a way to use portions of the imported file

```
<head>
  <link <link rel="import" href="warnings.html">d>
<body>
  ...
  <script><body>
    var link = document.querySelector('link[rel="import"]');
    var content = link.import;

    // Grab DOM from warning.html's document.
    var el = content.querySelector('.warning');

    document.body.appendChild(el.cloneNode(true));
  '.warning'</script>
</body>
```

Components

import.html

```
<template>
  <h1>Hello <h1>d!</h1>
  </h1><!-- Img is not requested until the <template> goes live. -->mg sr
  <script>alert("Executed when the template is activated.");</script>
</template>
<script>alert("Executed when the template is activated.");
```

index.html

```
<head>
  <link <link rel="import" href="import.html">d>
<body>
  <div id="container"></div>
  <script><body>
    var link = document.querySelector('link[rel="import"]');

    // Clone the <template> in the import.
    var template = link.import.querySelector('template');
    var clone = document.importNode(template.content, true);

    document.querySelector('#c'template'.appendChild(clone);
  </script>
</body>
```

Sub-imports

It can be useful for one import to include another. For example, if you want to reuse or extend another component, use an import to load the other element(s).

Below is a real example from Polymer. It's a new tab component () that reuses a layout and selector component. The dependencies are managed using HTML Imports.

paper-tabs.html (simplified):

```
<link rel="import" href="iron-selector.html">
<link rel="import" href="classes/iron-flex-la<link rel="import" href="clas
    <iron-selector class="layout horizontal center">
      <content select="*"></content>
    </iron-selector>
  </template>
  <script></script>
</dom-module>

<iron-selector class="layout horizontal center">
```

App developers can import this new element using:

```
<link rel="import" href="paper-tabs.html">
<paper-tabs></paper-tabs>
```

When a new, more awesome comes along in the future, you can swap out and start using it straight away. You won't break your users thanks to imports and web components.

Success and error handling

```
<script>
  function handleLoad(e) {
    console.log('Loaded import: ' + e.target.href);
  }
  function handleError(e) {
    console.log('Error loading import: ' + e.target.href);
  }
</script>

<link rel="import" href="file.html"
      onload="handleLoad(event)" onerror="handleError(event)">
```

Deduplication

Polyfills and testing for support

This set of technologies requires ES2015/ES6 and fairly modern browsers. The level of support will depend on the browser you need to support with your application.

Custom elements

There is a polyfill available.

Shadow DOM

the shadydom and shadycss polyfills give you v1 feature. Shady DOM mimics the DOM scoping of Shadow DOM and shadycss polyfills CSS custom properties and the style scoping the native API provides.

HTML Imports

There is a polyfill for HTML Imports. This is labeled as **work in progress** in the repository so use at your own risk.

HTML Templates

Templates are supported across major browsers so we don't need a polyfill.

Using the polyfills

To use the polyfills we need to import them using Bower like this:

```
bower install --save https://github.com/webcomponents/html-imports.git
bower install --save webcomponents/custom-elements
bower install --save webcomponents/shadydom webcomponents/shadycss
```

For the HTML Imports polyfill we're installing directly from Github because I couldn’t find a package int he Bower search engine.

We then create constants to test if Custom Elements and Shadow DOM.

```
function supportImports() {
  return 'import' in document.createElement('link');
}

function supportsCustomElementsV1() {
  return 'customElements' in window;
}

function supportsShadowDOMV1() {
  return !!HTMLElement.prototype.attachShadow;
}
```

We could create one constant that tests all the elements, something like so

```
const supportStyledElements = supportImports()
                            && supportsCustomElementsV1()
                            && supportsShadowDOMV1();
```

But it makes it harder to work when a browser support one specification but not the other, so we’ll stick to the two separate constants.

We then define a script loader function that returns a promise that will resolve when the polyfill loads and reject if it doesn’t.

```
function loadScript(src) {
 return new Promise(function(resolve, reject) {
   const script = document.createElement('script');
   script.async = true;
   script.src = src;
   script.onload = resolve;
   script.onerror = reject;
   document.head.appendChild(script);
 });
}
```

Finally we use our loadScript function to lazy load the polyfills if needed. We first test if the browser supports the feature; if it does we log it to console and are done. If it doesn't then we load the script or scripts and log the result to console as well. We can then proceed with the instantiation and use of the components.

```
if (supportsImports) {
    console.log('browser supports imports natively');
} else {
    loadScript('bower_components/html-imports/html-imports.min.js')
    .then(e => {
    'bower_components/html-imports/html-imports.min.js'ssfully')
    })
    .catch(error => {
        console.log('Error loading polyfill')
    });
}

if (supportsCustomElementsV1) {
  console.log('Native support for custom elements. Good to go');
} else {
    loadScript('/bower_components/custom-elements/custom-elements.min.js'
    'Error loading polyfill'le.log('Custom ELements Polyfill loaded succes
    })
    .catch(error => {
        console.log('Error loading polyfill')
    });
}

'Custom ELements Polyfill loaded successfully'// Lazy load the polyfill i
if (supportsShadowDOMV1) {
    console.log('Native support for shadow DOM. Good to go');
} else {
    loadScript('/bower_components/shadydom/shadydom.min.js')
    .then(e => loadScript('/bower_components/shadycss/shadycss.min.js'))
    .then(() => {
      console.log('Custom Elements Polyfill loaded successfully');
    })
    .catch(error => {
        console.log('Error loading polyfill')
    });
}
```

Links and resources

- ES6 Classes, how they work and how we extend them
    - Links and resources
        1. ES6 Classes in depth - MDN
        2. ES6 Classes in depth - Ponyfoo
        3. “Real” Mixins with JavaScript Classes - Justin Fagnani
        4. Custom Elements v1: Reusable Web Components - Eric Bidelman
        5. Shadow DOM v1: Self-Contained Web Components - Eric Bidelman
        6. <card-swiper> custom element
        7. Fancy tabs web component - shadow dom v1, custom elements v1, full a11y- Eric Bidelman
        8. Custom Elements Specification
            1. Autonomous example
            2. Customized builtin example
            3. Drawbacks of autonomous custom elements
        9. Shadow DOM Specification **Note that this specification has part that are being upstreamed to other specs (DOM and HTML). As such its usefulness as a research and learning tool is limited**
    - Shadow DOM examples for v0 and v1
    - WebKit announcement on Slot-based ShadowDOM