
CSE 331

Software Design & Implementation

Hal Perkins

Based on slides by Soham Pardeshi, Andrew Gies

Winter 2023

HW9, Spark Java, Fetch

Administrivia

- HW8 due this week (Thur. 3/2 @ 11:00pm)
- Section this week: more about today's demo, react, and spark, and hw9 – do not miss!
- HW9 due a week later (Thur. 3/9 @11:00pm)
 - Spec released later today
 - Combination of ideas and code from HW7 + HW8
 - Get creative! Lots of cool opportunities to go above and beyond!
- Any questions?

Agenda

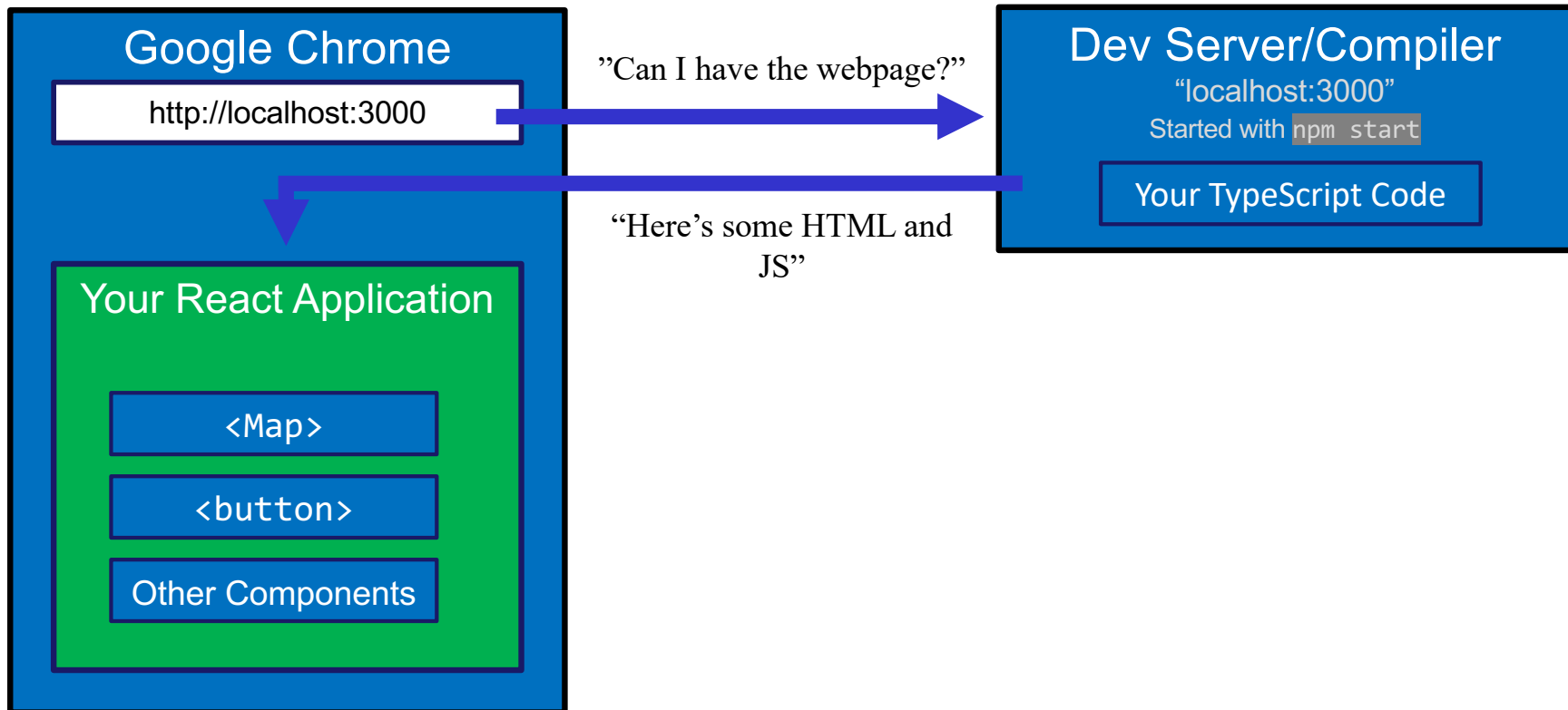
- HW9 Overview
- Anonymous Inner Classes
 - Common Java idiom – can make code easier to write
 - Come in handy when writing the Java server
- JSON
 - Brief overview
 - Helps share data between Java and JS
- Spark Java
 - How to turn your `hw-pathfinder` code into a Java web server
- Fetch
 - How your JS sends requests to the Java server

Homework 9 Overview

- Creating a new web GUI using React
 - Display a map and draw paths between two points on the map
 - Similar to your React app in HW8 – but you may add more!
 - New: Send requests to your (new) Java server to request building and path info
- Creating a Java server as part of your previous HW5-7 code
 - Receives requests from the React app to calculate paths and send data
 - Not much code to write here thanks to MVC
 - Reuse your CampusMap class from HW7

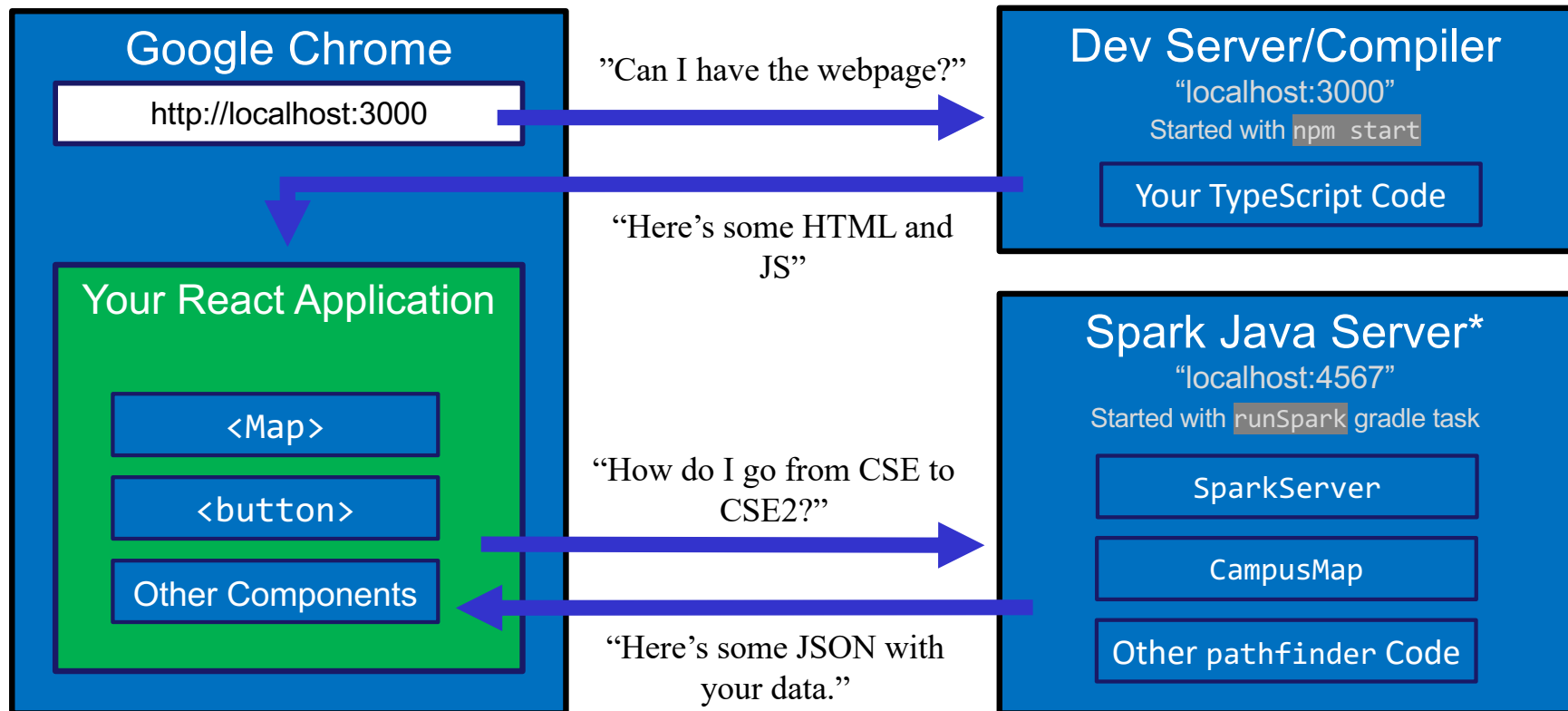
The Map Lines Stack

MapLines



The Campus Paths Stack

CampusPaths



**Note: This is not Apache Spark*

Any Questions?

- Done:
 - HW9 Basic Overview
- Up Next:
 - Anonymous Inner Classes
 - JSON
 - Spark Java
 - Fetch

Anonymous Inner Classes

- Helps put code closer to where it's used
- Makes sense when you aren't re-using classes
- The Example: sorting Strings by length instead of alphabetically
 - We need to make a Comparator – but how best to organize our code?
 - Start with what we're used to, then refine

Anonymous Inner Classes (Attempt 1)

```
public class StringSorter {  
  
    public static void main(String[] args) {  
        String[] strings = new String[]{"CSE331", "UW", "React", "Java"};  
        Arrays.sort(strings, new LengthComparator())  
        System.out.println(Arrays.toString(strings));  
    }  
}
```

StringSorter.java

```
public class LengthComparator implements Comparator<String> {  
  
    @Override  
    public int compare(String s1, String s2) {  
        return Integer.compare(s1.length(), s2.length());  
    }  
}
```

LengthComparator.java

Attempt 1 – Pros/Cons

- Pros:
 - Easy to reuse (assuming we want to)
- Cons:
 - Polluting the namespace with a whole extra top-level class
 - Understanding the main method requires viewing two separate Java files

Anonymous Inner Classes (Attempt 2)

```
public class InnerStringSorter {  
  
    public static void main(String[] args) {  
        String[] strings = new String[]{"CSE331", "UW", "React", "Java"};  
        Arrays.sort(strings, new InnerLengthComparator());  
        System.out.println(Arrays.toString(strings));  
    }  
  
    public static class InnerLengthComparator implements Comparator<String> {  
  
        @Override  
        public int compare(String s1, String s2) {  
            return Integer.compare(s1.length(), s2.length());  
        }  
    }  
}
```

InnerStringSorter.java

Attempt 2 – Pros/Cons

- Pros:
 - In a single Java file now – easier to read/understand
 - Still reusable outside this file, but more annoying syntax:
 - `new InnerStringSorter.InnerLengthComparator()`
 - `new Path<E>.Segment()`
- Cons:
 - If we're not reusing it, this is unnecessary indirection
 - Reader has to find and read a new class to understand what the code in main means, even if we only ever do this sorting in one place

Anonymous Inner Classes (Attempt 3)

```
public class AnonymousStringSorter {  
  
    public static void main(String[] args) {  
        String[] strings = new String[]{"CSE331", "UW", "React", "Java"};  
        Arrays.sort(strings, new Comparator<String>() {  
  
            @Override  
            public int compare(String s1, String s2) {  
                return Integer.compare(s1.length(), s2.length());  
            }  
        });  
        System.out.println(Arrays.toString(strings));  
    }  
}
```

AnonymousStringSorter.java

Anonymous Inner Classes (Attempt 3)

```
public class AnonymousStringSorter {  
  
    public static void main(String[] args) {  
        String[] strings = new String[]{"CSE331", "UW", "React", "..."  
        Arrays.sort(strings, new Comparator<String>() {  
  
            @Override  
            public int compare(String s1, String s2) {  
                return Integer.compare(s1.length(), s2.length());  
            }  
        });  
        System.out.println(Arrays.toString(strings));  
    }  
}
```

AnonymousStringSorter.java

Creating and using the class, all at once! No need to give it a name.

Attempt 3 – Pros/Cons

- Pros:
 - Still in a single Java file
 - Puts the meaning of the code right where it's being executed - easy to see exactly what the `Arrays.sort` is going to do
 - Very useful if you need to make many different Comparators (or objects that extend other classes)
 - Like in HW9 😊
- Cons:
 - Not reusable (there's no name!)
 - Anonymous inner classes only make sense in certain circumstances, like when you need to make an object for one specific situation
 - Can be harder to read if overused
- Note: Java 8 adds a whole bunch of additional ways to write these sorts of things
 - Not going to discuss them, but you're welcome to learn and use them if you'd like!

Any Questions?

- Done:
 - HW9 Basic Overview
 - Anonymous Inner Classes
- Up Next:
 - JSON
 - Spark Java
 - Fetch

JSON

- We have a whole application written in Java so far:
 - Reads CSV data, manages a Graph data structure with campus data, uses Dijkstra’s algorithm to find paths
- We’re writing a whole application in JavaScript:
 - React web app to create an interactive GUI for your users
- Even if we get them to communicate (discussed later), we need to make sure they “speak the same language”
 - JavaScript and Java store data *very* differently
- JSON = JavaScript Object Notation
 - Can convert JS Object → String, and String → JS Object
 - Bonus: Strings are easy to send inside server requests/responses

JSON ↔ Java

Java Object

```
public class SchoolInfo {  
  
    String name = "U of Washington";  
    String location = "Seattle";  
    int founded = 1861;  
    String mascot = "Dubs II";  
    boolean isRainy = true;  
    String website = "www.uw.edu";  
    String[] colors = new String[]  
        {"Purple", "Gold"};  
  
}
```

JSON String

```
{"name":"U of Washington",  
 "location":"Seattle","founded":1861,  
 "mascot":"Dubs II","isRainy":true,  
 "website":"www.uw.edu",  
 "colors":["Purple","Gold"]}
```



- Use Gson (a library from Google) to convert between them
 - Tricky (but possible) to go from JSON String to Java Object, but we don't need that for this assignment

```
Gson gson = new Gson();  
SchoolInfo sInfo = new SchoolInfo()  
String json = gson.toJson(sInfo);
```

JSON ↔ JS

Javascript Object

```
let schoolInfo = {  
  
  name: "U of Washington",  
  location: "Seattle",  
  founded: 1861,  
  mascot: "Dubs II",  
  isRainy: true,  
  website: "www.uw.edu",  
  colors: ["Purple", "Gold"]  
  
}
```

JSON String

```
{"name":"U of Washington",  
"location":"Seattle","founded":1861,  
"mascot":"Dubs II","isRainy":true,  
"website":"www.uw.edu",  
"colors":["Purple","Gold"]}
```



- Can convert between the two easily (we'll see how later)
- This means: if the server sent back a JSON String, it'd be easy to use the data inside of it – just turn it into a JS Object and read the fields out of the object

JSON – Key Ideas

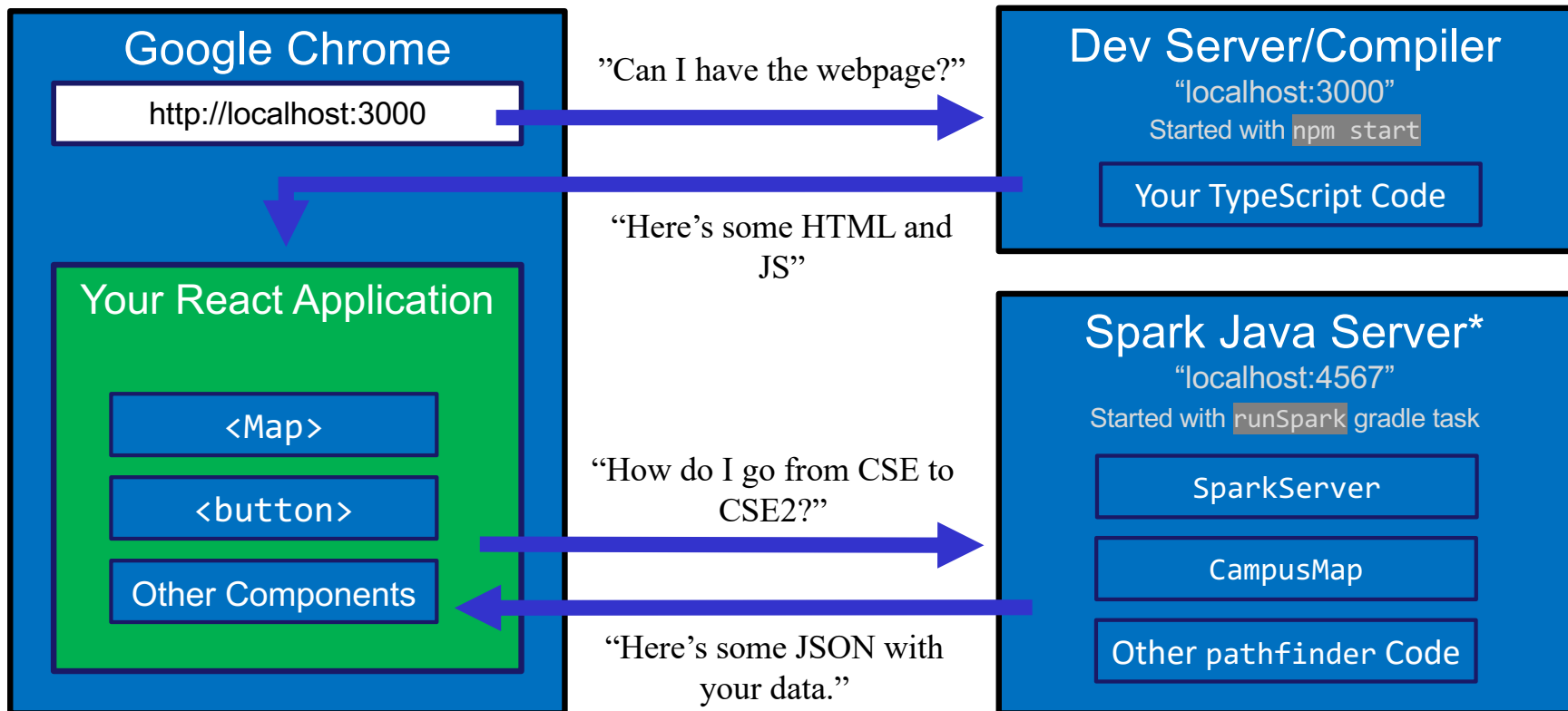
- Use Gson to turn Java objects containing the data into JSON before we send it back to the requesting app
 - The Java objects don't have to be simple, like in the example. Gson can handle complicated structures.
- Easy to turn a JSON string into a Javascript object so we can use the data (fetch can help us with that)

Any Questions?

- Done:
 - HW9 Basic Overview
 - Anonymous Inner Classes
 - JSON
- Up Next:
 - Spark Java
 - Fetch

The Campus Paths Stack

CampusPaths



**Note: This is not Apache Spark*

Spark Java

- Using the Spark Java framework – designed to make this short & easy
 - *Note: there's also something called Apache Spark. Completely different, careful what you Google*
- Create the server by creating “routes” in the main method of your server program
 - A route is an instruction that tells the server what to do when it gets a particular request
 - Create Route objects and override their abstract `handle()` method
 - Remember anonymous inner classes? ☺
 - Users can request information. The handle method gets information about these requests, can set information about the response, then return something that will be sent back to the user.

What is a Request

- Basically just a URL:
 - When you type a URL into your browser, it makes a GET request to that URL, the response to that request is sent by the website itself (i.e., the HTML, JS, etc.)
 - A "GET" request says "Hey server, can I get some info about _____?"
 - We're going to make a request from inside Javascript / React to ask our campus map (hw7) spark server for data about paths on campus
 - There are other kinds of requests, but we're just using GET. (It's the default for fetch)
- Each "place" that a request can be sent is called an "endpoint"
 - Your Java server will provide multiple endpoints – one for each kind of request that your React app might want to make.
 - Find a path, get building info, etc...

Forming a Request

Server Address: `http://localhost:4567`

- Basic request with no extra data: `http://localhost:4567/getSomeData`
 - A request to the `/getSomeData` endpoint in the server at `localhost:4567`
 - `localhost` just means “on this same computer”
 - `:4567` specifies a port number – every computer has multiple ports so multiple things can be running at a given time.
 - (`4567` is the port we’re using in this example – no further significance beyond that)
- Sending extra information in a request is done with a query string:
 - Add a `?`, then a list of `key=value` pairs. Each pair is separated by `&`.
 - Query string might look like: `?start=CSE&end=KNE`
- Complete request looks like:
`http://localhost:4567/findPath?start=CSE&end=KNE`
 - Sends a `/findPath` request to the server at `localhost:4567`, and includes two pieces of extra information, named `start` and `end`
- You don’t need to name your endpoints or query string parameters anything specific, the above is just an example
- Note: we are using basic, unencrypted http requests, not secure https that is standard on production websites these days. http is fine for what we need and https is complex to set up.

Forming a Request

Server Address: `http://localhost:4567`

```
http://washington.edu/about
```

```
http://localhost:4567/getSomeData
```

└──────────┬────────┬──────────┘
Hostname Port* Endpoint

Query Params*

```
http://localhost:4567/findPath?start=CSE&end=KNE
```

*Port and query params are technically optional

Our First Spark Route

```
public static void main(String[] args) {
    Spark.get("/hello-world", new Route() {
        @Override
        public Object handle(Request request, Response response) throws Exception {
            return "Hello, Spark!";
        }
    });
}
```

- Creating a new anonymous subclass of Route
 - Probably not going to have a whole bunch of different endpoints that all send back “Hello, Spark!” – so this makes sense
- Telling Spark to use that Route whenever it receives a GET request (`Spark.get`) to the “/hello-world” endpoint
 - Responds to the request: “http://localhost:4567/hello-world”

Demo Time!

- See that simple Spark route in action
- See a Spark route that can get info from a query parameter and use it
- See the node-fetch code that sends a request to the Spark endpoint that we just went over and displays it on the page
- There are more demos than we will go over in section – get the code from the website to see everything
 - LOTS of useful info in there

Any Questions?

- Done:
 - HW9 Basic Overview
 - Anonymous Inner Classes
 - JSON
 - Spark Java
- Up Next:
 - Fetch

Fetch

- Used by JS to send requests to servers to ask for info
- Uses Promises:
 - Promises capture the idea of “it’ll be finished later”
 - We can “pause” the currently executing function while we wait for the promise to complete
 - Asking a server for a response can be *slow*, so Promises allow the browser to keep working instead of stopping to wait
 - Getting the data out is a little more complicated
- We’re using `async/await` syntax to deal with promises

Sending a Request

```
let responsePromise = fetch("http://localhost:4567/findPath?start=CSE&end=KNE");
```

- The URL you pass to `fetch()` can include a query string if you need to send extra data
- `responsePromise` is a Promise object
 - Once the Promise “resolves,” it’ll hold whatever is sent back from the server
- How do we get the data out of the Promise?
 - We can `await` the promise’s resolution
 - `await` tells the browser that it can pause the currently-executing function and go do other things. Once the promise resolves, it’ll resume where we left off.
 - Prevents the browser from freezing while the request is happening

Getting Useful Data

“This function is
pause-able”

Will eventually
resolve to an
actual JS object
based on the
JSON string

Once we have
the data, store it
in a useful place

```
async sendRequest() {
  let responsePromise = fetch("...");
  let response = await responsePromise;
  let parsingPromise = response.json();
  let parsedObject = await parsingPromise;
  this.setState({
    importantData: parsedObject
  });
}
```


Error Checking

Every response has a 'status code' (e.g. 404 = Not Found)
This checks for 200 = OK

On a complete failure (i.e. server isn't running) an error is thrown

```
async sendRequest() {  
  try {  
    let response = await fetch("...");  
    if (!response.ok) {  
      alert("Error!");  
      return;  
    }  
    let parsed = await response.json();  
    this.setState({  
      importantData: parsed  
    });  
  } catch (e) {  
    alert("Error!");  
  }  
}
```

Things to Know

- Can only use the `await` keyword inside a function declared with the `async` keyword
 - `async` keyword means that a function can be “paused” while `await`-ing
- `async` functions automatically return a `Promise` that will eventually contain their return value
 - This means that if you need a return value from the function you declared as `async`, you’ll need to `await` the function call
 - But that means that the caller also needs to be `async`
 - Therefore: generally best to not have useful return values from `async` functions (for 331 that is; there are lots of use cases outside of this course, but can get complicated fast)
 - Instead of returning, consider calling `setState` to store the result and trigger an update (like in the example)

More Things to Know

- Error checking is important
 - If you forget, the error most likely will disappear without actually causing your program to explode
 - This is BAD! Silent errors can cause tricky bugs
 - This happens because errors don't bubble outside of promises, and the async function you're inside is effectively "inside" a promise
 - Means that if you don't catch an exception, it'll just disappear as soon as your function ends

More More Things to Know

- The return value of `await response.json()` will be any
 - As we know, this is dangerous! (No TypeScript checks)
- To solve, we create an interface describing what the server will respond with (e.g. a `Path`) and cast the value to that type:

```
interface Path { ... }  
const parsed: Path = await response.json() as Path;
```

- Note: This does *not* check that the value actually has this type
 - If the server sends back something different, could crash later
 - A true solution would check the object before casting
 - Can get pretty complicated – not required for hw9
 - If you're curious – libraries like `io-ts` can help with this

Any Questions?

- Done:
 - HW9 Basic Overview
 - Anonymous Inner Classes
 - JSON
 - Spark Java

Wrap-Up

- Don't forget:
 - HW8 due tomorrow (Thur. 3/2 @ 11:00pm)
 - HW9 due a week later (Thur. 3/9 @ 11:00pm)
 - Sections tomorrow – much more about today's topics/demos
- Use your resources!
 - Office Hours
 - Links from HW specs
 - React Tips & Tricks Handout (See “Resources” page on web; look for interesting sections; probably not worth reading all)
 - Other students (remember academic honesty policies: can't share/show/copy code, but discussion is great!)
 - Google (carefully, always fully understand code you use)