YEAH Hours
Assignment 6: Adventure

—

11/27/2018

Jonathan Kula

Adventure: An Overview

- Adventure is a text-based adventure game!
- You'll be coding up a data-driven *framework* for running pre-written adventures.
- The player moves between rooms, picking up items in order to move through doors or other passageways.
- Your goal is to get to the end, collecting all the treasures along the way!

```
Welcome to Adventure!
You are standing at the end of a road before a small brick
building.  A small stream flows out of the building and
down a gully to the south.  A road runs up a small hill
to the west.
>
```

```
> NORTH
Slit in rock
> NORTH
Valley beside a stream
> NORTH
Outside building
>
```

JavaScript & Concepts

Objects: A New Take

- We use objects to represent real-world things!
  - Your **`enigma`** object represented the state of an enigma machine; complete with rotors and lamps!
  - We could represent a phonebook as an object – a mapping from name to phone number.
  - We could represent a Facebook Messenger profile as an object like this:

```
{
  "first_name": "Peter",
  "last_name": "Chang",
  "profile_pic": "https://fbcdn-profile-a.akamaihd.net/hprofile-ak-xpf1/v/t1.0-1/p200x200/1305
  "locale": "en_US",
  "timezone": -7,
  "gender": "male",
}
```

Objects: A New Take

- Unlike your **`enigma`** object, there are millions (billions?) of these profiles!
- Every profile has to stay perfectly consistent
- It would be nice if there was a way to give a name to this particular *type* of object.
- One might say the object is of a certain *classification*.

Classes

- ...and that's where we get *classes* from!
- A *class* is a description of a certain type of object (like GRect, or GOval).
- How do we represent this in JavaScript?

Factory Functions!

- We usually represent this by creating a *factory function* for the class.
- We use this function to create `Profiles`. If we change the function here, the structure of a Profile object changes everywhere!

```
function Profile(name, profileImage, language) {
    return {
        name: name,
        image: profileImage,
        language: language
    };
}
```

Factory Functions!

- We usually represent this by creating a *factory function* for the class.
- We use this function to create `Profiles`. If we change the function here, the structure of a Profile object changes everywhere!

```
let jonathan = Profile(“Jonathan Kula”, “http://image.url/”, “English”);

jonathan.name === “Jonathan Kula”         // true
jonathan.image === “http://image.url/”    // true
jonathan.language === “English”           // true
```

Factory Functions!

- You can add functionality to the objects you create, too!
  - Maybe you'd like to send a message to the user?

```
function Profile(name, profileImage, language) {
    let profile = {
        name: name,
        image: profileImage,
        language: language
    };
    profile.sendMessage = function(message) {
        // somehow send a message to this user
    };
    return profile;
}
```

Factory Functions!

- You can add functionality to the objects you create, too!
  - Maybe you'd like to send a message to the user?

```
let jonathan = Profile(“Jonathan Kula”, “http://image.url/”, “English”);

jonathan.sendMessage(“Hello, World!”);
```

Factory Functions!

- What about hiding information?
- Hiding refers specifically to *restricting read and/or write access to information*.
- For example: We might want to verify that the image url is valid!

```
function Profile(name, profileImage, language) {
    // Don’t create a profile with an invalid image.
    if(!isValidUrl(profileImage)) { return null; }

    let profile = {
        name: name,
        language: language  // no more image!
    };
    profile.getImage = function() {
        return profileImage;
    };
    profile.setImage = function(newImageUrl) {
        let valid = isValidUrl(newImageUrl);
        if(valid) {
            profileImage = newImageUrl;
        }
        return valid;
    };
    return profile;
}
```

Factory Functions!

- What about hiding information?

```
let jonathan = Profile(“Jonathan Kula”, “http://image.url/”, “English”);

jonathan.name           // “Jonathan Kula”
jonathan.image          // ERROR!
jonathan.getImage()     // “http://image.url/”

jonathan.setImage(“cat video”)  // false - not a valid url
jonathan.getImage()     // “http://image.url/”
```

Factory Functions!

- What about hiding information?

```
let jonathan = Profile(“Jonathan Kula”, “http://image.url/”, “English”);

jonathan.name           // “Jonathan Kula”
jonathan.image          // ERROR!
jonathan.getImage()     // “http://image.url/”

jonathan.setImage(“http://image.url/newImage”)  // true - a valid url
jonathan.getImage()     // “http://image.url/newImage”
```

Factory Functions!

- What about hiding information?
- Hiding refers specifically to *restricting read and/or write access to information*.
- Notice that *we are now in control*.

```
let jonathan = Profile(“Jonathan Kula”, “http://image.url/”, “English”);
```

Before:

```
jonathan.image = “cat video”;  // might crash website if it’s expecting a URL.
```

After:

```
jonathan.setImage(“cat video”);  // no problem - we caught it!
```

Objects vs Classes

- Remember, Classes describe a *type* of object. Classes are not objects.

```
let jonathan = Profile(“Jonathan Kula”, “http://image.url/”, “English”);
let ryan = Profile(“Ryan Eberhardt”, “http://image.url/”, “English”);

jonathan !== Profile    // true - they’re not equal!
ryan !== Profile        // true - they’re not equal!

jonathan !== ryan       // true - they’re not equal!
```

Objects vs Classes

- Remember, Classes describe a *type* of object. Classes are not objects.
- *Properties* of a class aren't part of the class itself

```
let jonathan = Profile("Jonathan Kula", "http://image.url/", "English");


jonathan.name    // "Jonathan Kula"
Profile.name     // ERROR!

jonathan.sendMessage("Hello World!")  // Sends the message "Hello World!"
Profile.sendMessage("Hello World!")   // ERROR!
```

You've seen this before!

- Just like with graphics objects!

```
let gl = GLabel("Hello World!");
gl.setFont("12px 'monospace'");
gl.getFont();                              // "12px 'monospace'"

GLabel.setFont("12px 'monospace'");        // ERROR!
GLabel.getFont();                          // ERROR!
```

XML

- XML is a way of encoding information. It looks like this:

```
<div>
    <object name=“Key” location=“River”>
        A shiny gold key, covered in sand
    </object>
</div>
```

XML

- XML is a way of encoding information. It looks like this:

```
<TAG>
    <TAG ATTRIBUTE=“VALUE” ATTRIBUTE2=“VALUE”>
        DATA
    </TAG>
</TAG>
```

XML

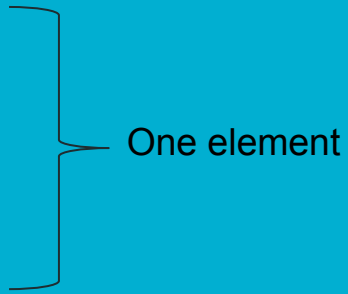
- XML is a way of encoding information. It looks like this:
- Each tag is matched by a closing tag!

```
<TAG>
    <TAG ATTRIBUTE=“VALUE” ATTRIBUTE2=“VALUE”>
        DATA
    </TAG>
</TAG>
```

XML

- XML is a way of encoding information. It looks like this:
- Each individual instance of a tag is called an element.

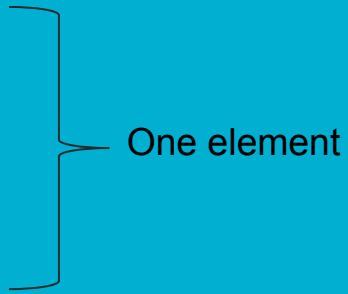
```
<TAG>
    <TAG ATTRIBUTE=“VALUE” ATTRIBUTE2=“VALUE”>
        DATA
    </TAG>
</TAG>
```



XML

- XML is a way of encoding information. It looks like this:
- Tags denote the type of element it is (e.g. “object” or “room”)

```
<TAG>
    <TAG ATTRIBUTE=“VALUE” ATTRIBUTE2=“VALUE”>
        DATA
    </TAG>
</TAG>
```



XML

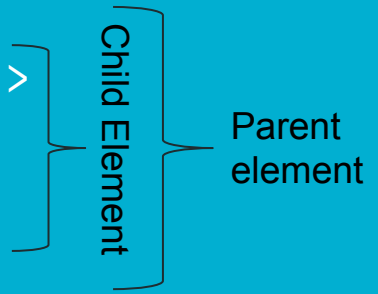
- XML is a way of encoding information. It looks like this:
- “Higher-up” elements are *parents* of what’s inside them.

```
<Parent>
    <TAG ATTRIBUTE=“VALUE” ATTRIBUTE2=“VALUE”>
        DATA
    </TAG>
</Parent>
```

XML

- XML is a way of encoding information. It looks like this:
- Elements may have elements inside them, called children.

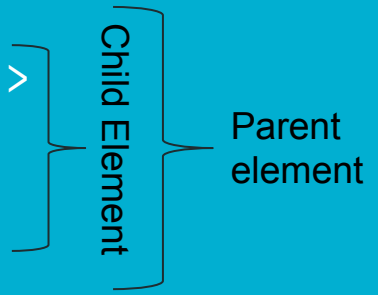
```
<Parent>
    <Child ATTRIBUTE=“VALUE” ATTRIBUTE2=“VALUE”>
        DATA
    </Child>
</Parent>
```



XML

- XML is a way of encoding information. It looks like this:
- Each element may have *zero* or more attributes (with all different names).
- Elements with the same tag usually have the same set of attributes

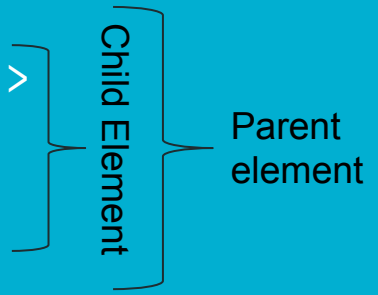
```
<Parent>
    <Child ATTRIBUTE="VALUE" ATTRIBUTE2="VALUE">
        DATA
    </Child>
</Parent>
```



XML

- XML is a way of encoding information. It looks like this:
- Values can totally share the same value, though!

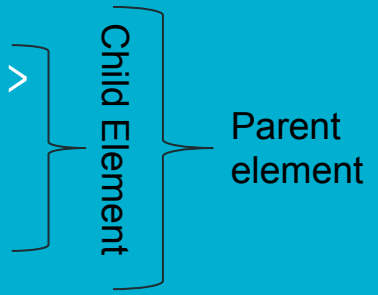
```
<Parent>
    <Child ATTRIBUTE="VALUE" ATTRIBUTE2="VALUE">
        DATA
    </Child>
</Parent>
```



XML

- XML is a way of encoding information. It looks like this:
- Finally, elements can have straight-up text inside of them.

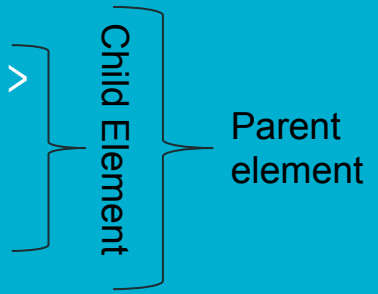
```
<Parent>
    <Child ATTRIBUTE="VALUE" ATTRIBUTE2="VALUE">
        DATA
    </Child>
</Parent>
```



XML

- XML is a way of encoding information. It looks like this:
- Finally, elements can have straight-up text inside of them.

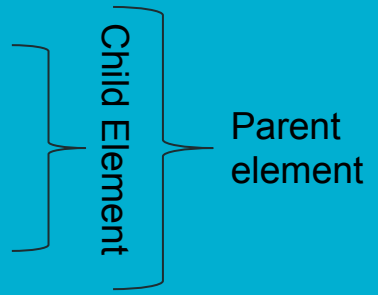
```
<Parent>
    <Child ATTRIBUTE="VALUE" ATTRIBUTE2="VALUE">
        This is all valid XML!
    </Child>
</Parent>
```



XML in Adventure

- XML is a way of encoding information – including an adventure!
- Here's something you'll see in your `index.html` file!

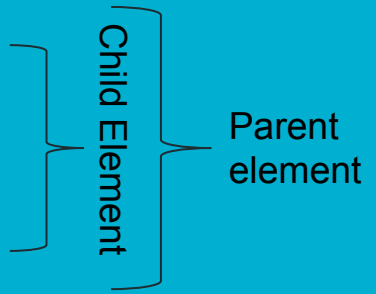
```
<div>
    <object name="Key" location="River">
        A shiny gold key, covered in sand.
    </object>
</div>
```



XML in Adventure

- XML is a way of encoding information – including an adventure!
- There's one special attribute name: id

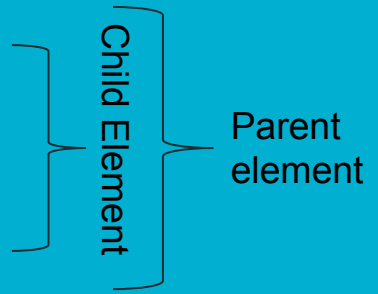
```
<div id="GameData">
    <object name="Key" location="River">
        A shiny gold key, covered in sand.
    </object>
</div>
```



XML in Adventure

- XML is a way of encoding information – including an adventure!
- The id attribute *uniquely* identifies a particular element.
- id is the only case where the attribute value has to be unique.

```
<div id="GameData">
    <object name="Key" location="River">
        A shiny gold key, covered in sand.
    </object>
</div>
```



The DOM

- The Document Object Model (or DOM) is just a fancy way to say “the way Javascript interacts with XML”
- Here are the four methods you’ll use with Adventure:

`document.getElementById(`*id*`)`	Returns the element with the specified id attribute.
element`.getElementsByTagName(`*name*`)`	Returns an array of the elements with the specified tag name.
element`.getAttribute(`*name*`)`	Returns the value of the named attribute.
element`.innerHTML`	Returns the HTML under the jurisdiction of an element.

The DOM: an example

```
<div id="GameData">
    <object name="Key" location="River">
        A shiny gold key, covered in sand.
    </object>
    <object name="Rope" location="Road">
        50ft of silk rope.
    </object>
</div>
```

```
let info = document.getElementById("GameData");
```

The DOM: an example

```
<div id="GameData">
    <object name="Key" location="River">
        A shiny gold key, covered in sand.
    </object>
    <object name="Rope" location="Road">
        50ft of silk rope.
    </object>
</div>
```

```
let info = document.getElementById("GameData");
let objects = info.getElementsByTagName("object");
// objects = [Key, Rope]
```

The DOM: an example

```
<div id="GameData">
    <object name="Key" location="River">
        A shiny gold key, covered in sand.
    </object>
    <object name="Rope" location="Road">
        50ft of silk rope.
    </object>
</div>
```

```
let info = document.getElementById("GameDatas");
let objects = info.getElementsByTagName("object");
// objects = [Key, Rope]

for(let i = 0; i < objects.length; i++) {
    let name = objects[i].getAttribute("name");
    let description = objects[i].innerHTML;
    console.log(name, description);
}
```

The DOM: an example

```
<div id="GameData">
    <object name="Key" location="River">
        A shiny gold key, covered in sand.
    </object>
    <object name="Rope" location="Road">
        50ft of silk rope.
    </object>
</div>
```

```
let info = document.getElementById("GameData");
let objects = info.getElementsByTagName("object");
// objects = [Key, Rope]

for(let i = 0; i < objects.length; i++) {  // i = 0
    let name = objects[i].getAttribute("name");
    let description = objects[i].innerHTML;
    console.log(name, description);   // "Key", "A shiny…"
}
```

The DOM: an example

```
<div id="GameData">
    <object name="Key" location="River">
        A shiny gold key, covered in sand.
    </object>
    <object name="Rope" location="Road">
        50ft of silk rope.
    </object>
</div>
```

```
let info = document.getElementById("GameData");
let objects = info.getElementsByTagName("object");
// objects = [Key, Rope]

for(let i = 0; i < objects.length; i++) {  // i = 1
    let name = objects[i].getAttribute("name");
    let description = objects[i].innerHTML;
    console.log(name, description);   // "Rope", "50ft of…"
}
```

Data-Driven Programming

- What is it?
- How is it different from what you've been doing?

Data-Driven Programming

- What is it? ***Letting the logic of your program be dictated by external data***
- How is it different from what you've been doing?

Data-Driven Programming

- What is it? *Letting the logic of your program be dictated by external data*
- **How is it different from what you've been doing?**

Data-Driven Programming

Before:

- Programs did one (complicated) thing.
  - They did it well, but were inflexible!
- Kinda like a prebuilt marble run, where everything is superglued together.

Data-Driven Programming

After:

- You're designing the building blocks.
- You get to dictate how everything fits together, and what each piece does.

- Your users get be as creative as they want using your program!

Data-Driven Programming & Adventure

- So, are you designing *an adventure*?

Data-Driven Programming & Adventure

- So, are you designing *an adventure*? **Nope!**
- You're designing a framework – a set of building blocks to make adventures with!

Data-Driven Programming & Adventure

- So, are you designing *an adventure*? Nope!
- You're designing a framework – a set of building blocks to make adventures with!
- *In a way, you're designing not just* an adventure, *but* ***all possible adventures****!*

Adventure

Adventure: An Overview

- Adventure is a text-based adventure game!
- You'll be coding up a data-driven *framework* for running pre-written adventures.
- The player moves between rooms, picking up items in order to move through doors or other passageways.
- Your goal is to get to the end, collecting all the treasures along the way

- In order to do this, you'll be reading in data about the adventure from XML, and using that data to construct an adventure!

Adventure: A Multi-File Project

- You'll be working with several files:
- **`Adventure.js`** – Defines where your program starts. You don't need to change this file at all!
- **`AdvGame.js`** – Defines a single game of adventure. Responsible for orchestrating the game, as well as reading in everything from XML. Depends on AdvRoom, AdvObject, and AdvPassage.
- **`AdvRoom.js`** – Defines a single room, and keeps track of everything related to the room.
- **`AdvObject.js`** – Defines a single Adventure object.
- **`AdvPassage.js`** – Defines a passage from one room to the next.

Adventure: XML Structure

```
<div id="GameData" style="display:none”>
    <object name="KEYS" location="InsideBuilding">
        a set of keys
    </object>
    <room name="InsideBuilding" short="Inside building">
        You are inside a building, a well house for a large spring.
        The exit door is to the south. There is another room to
        the north, but the door is barred by a shimmering curtain.
        <passage dir="SOUTH" room="OutsideBuilding" />
        <passage dir="OUT" room="OutsideBuilding" />
    </room>
    <synonym word="Q" definition="QUIT" />
</div>
```

Milestone #1: Cannibalize Teaching Machine

- Your goal is to cannibalize Teaching Machine's code (included in the starter code), and use it for Adventure.
- The code for Teaching Machine is very close to what you'll need in Adventure.
- TMCourse is very similar to AdvGame; and TMQuestion is close to AdvRoom.
- You'll be changing around variable names and method names, but that's about it!

Milestone #2: Implement Short Descriptions

- If someone re-visits a room, you don't want them to have to read the whole long description of the room again!
- Instead, you should give a `short` description!

```
<room name="InsideBuilding" short="Inside building">
```

- You can get this short description from the `short` attribute on a **room**.
- You'll also need a way to keep track of if the room has been visited or not!
  - Should this be a hidden attribute?

Milestone #2: Implement Short Descriptions

- If someone re-visits a room, you don't want them to have to read the whole long description of the room again!
- Instead, you should give a `short` description!

```
<room name="InsideBuilding" short="Inside building">
```

- You can get this short description from the `short` attribute on a **room**.
- You'll also need a way to keep track of if the room has been visited or not!
  - Should this be a hidden attribute? **Yes, because you don't want anyone to make the room unvisited.**

Milestone #3: Commands

- You want the user to be able to leave the game, view the description again, etc!
- You'll be implementing the three simplest – QUIT, HELP, and LOOK.
- These just require you to check if what's entered match any of these, before trying to go to a room.
- You'll want to match based on the **first word** of the commands
  - The "split" method will be your friend!
- Remember, these commands should be case insensitive!

Milestone #4: Objects

- You'll finally be reading in those object tags!
- You'll also need to distribute objects to their rooms!
- (which means you'll also need a way to keep track of which objects are in which room!)
    - What file should this be in?

Milestone #4: Objects

- You'll finally be reading in those object tags!
- You'll also need to distribute objects to their rooms!
- (which means you'll also need a way to keep track of which objects are in which room!)
  - What file should this be in? **AdvRoom.js**
- You'll make these four methods:
  - room.describeObjects()
  - room.addObject(obj)
  - room.removeObject(obj)
  - room.contains(obj)

Milestone #5: TAKE, DROP, & INVENTORY

- You'll be implementing commands that need you to *parse input*.
- You TAKE and DROP *objects*, which means you'll have to be able to check if the object they're trying to take/drop are either in the room or in the player's inventory.
- Speaking of which, you'll need to have an inventory for the player. How might you implement this?

Milestone #6: Synonyms

- Now, you'll be reading in synonyms.
- Once you read in the synonyms, before you start processing a command, you'll have to take **each word** of the input, and (if there's a matching synonym!) replace the **word** with the **definition**.

```
<synonym word="Q" definition="QUIT" />
```

Milestone #7: Locked Passages

- Passages sometimes have a **key** parameter.
- If a **key** parameter is defined, the player needs to have that object in their inventory in order to pass through.
- **This milestone also introduces the idea of multiple passages with the same direction.** You should take the *first matching passage* that the user is able to go through.

```
<room name="InsideBuilding" short="Inside building">
    <passage dir=”IN” key=”Key” room=”SecretRoom” />
    <passage dir=”IN” room=”MissingKey” />
```

Milestone #8: Forced Motion

- Finally, you'll deal with a *special* direction, called FORCED.
- If a passage's direction is **FORCED**, act as if they typed that in right away:

- First, print out the room description (either short or long, depending on if they've been here before)
- Try to go in the "FORCED" direction. *Just like regular directions, there may be multiple!*