# Hi-Lo Guess Game

*A Number Puzzle Game - ( DOM API + Events)*

**Table of Content:** *Implementing a Hi-Lo Guessing Game*

# Lab Introduction

## Prerequisites

**DOM / Events [ Lectures ]** . Software Requirements: Chrome browser, any code editor/IDE.

## Motivation

Learn to use Document Object Model API & JavaScript to make interactive HTML-based apps.

## Goal

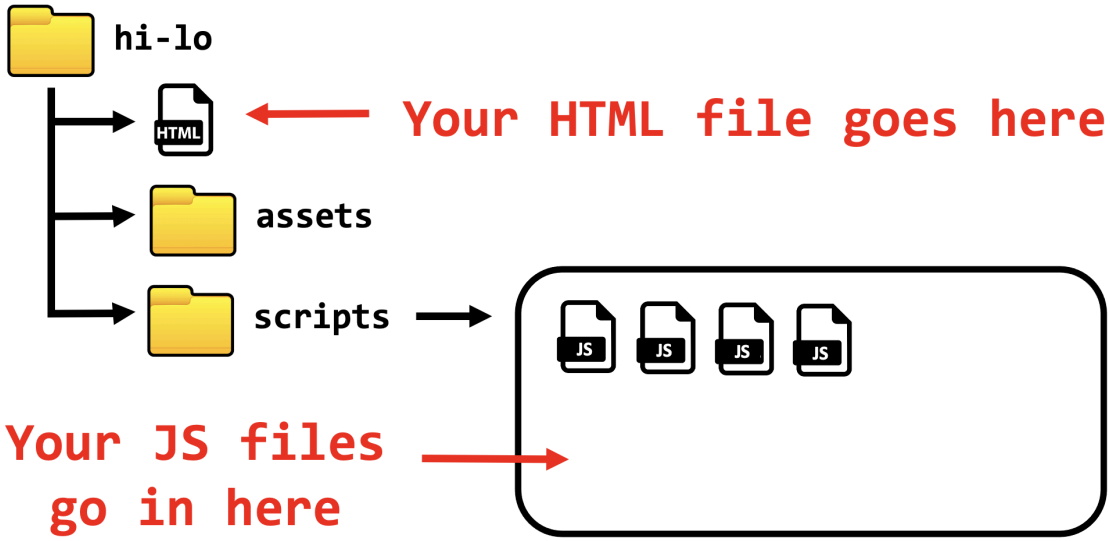
Build a compelling, fun version of the Hi-Lo Game that runs in Browser using JavaScript & DOM API.

## Learning Objectives

- Document Object Model API to access & update HTML
- Event-driven System design
- Agile practices: Minimal Viable Product
- Model-View-Controller Design Pattern
- UI/UX considerations/improvements
- Timed Events and Date object

## Project Architecture:

Start this project by downloading the starter files from github. See the project structure below.




**Download Starter files:**
[https://github.com/scalemailted/hilo-js-dom/archive/master.zip](https://github.com/scalemailted/hilo-js-dom/archive/master.zip)

## Document Object Model (DOM)

The DOM gives JavaScript access to the HTML document. The DOM is a built-in object with functions for manipulating the HTML. The DOM converts HTML elements into JavaScript objects.

- **document object:** The global object that is the entry point to the DOM API.
- **element objects:** The document object accesses element objects via ***getElementById(id)***. These element objects are the JS models of the HTML elements from the web page.
- **innerHTML:** A property of element objects that maintains its HTML in markup-notated strings. By changing the value of the innerHTML, the element may add, change, delete HTML from DOM.

---

## Browser (HTML) Events

Web browsers are event-driven systems. A browser generates & manages events based on certain actions or triggers. Those events may be used by JS apps to trigger actions or behaviors.

- **Event listener:** An event listener registers an element object to listen to the event queue for specific types of event. When that event occurs it invokes a callback function
- **Callback function**: a function within the JS app that is passed as a reference to the event system. When a event occurs the event system invokes the callback function
- **Event object**: Whenever a callback function is invoked, the browser passes an Event object as a parameter. The Event object has properties specific to that event.
- **Event type:** Every event object has a type. The event listener matches the Event objects type to a callback function, similar to a key-value pair. Event types are strings such as "click" or "load"

---

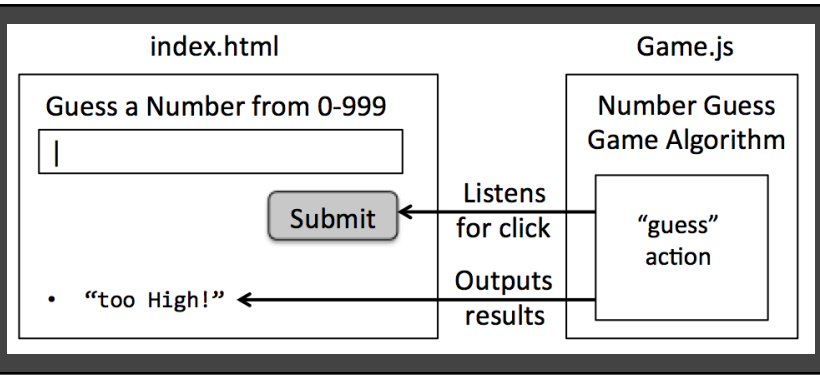
## Model-View-Controller (MVC)

MVC is a design pattern where an app's responsibilities are divided into Model, Controller and View.

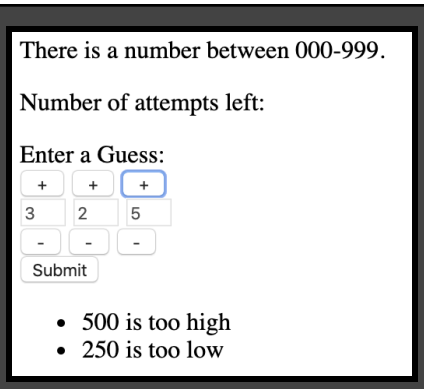
- **Model:** The model manages the app's core logic without concerns for handling input or output.
- **Controller:** The controller handles user input to send data & invoke actions within the model
- **View:** The view manages outputting data from the model & presenting it to the user.

# Goal 0: Single Page App - Iterative Builds

## Part 1: Minimal Viable Product (MVP) → Focus development cycle on app's necessities.




## Part 2: Refactor App with Model-View-Controller Pattern & Improve UI




This version refactors codebase into 3 parts:

- model: manages app logic
- view: manages app's output
- controller: manages app's inputs

This version also ensures that only valid inputs are allowed and that all devices are supported.

## Part 3: Refactor App into Time-based Challenge with Graphics

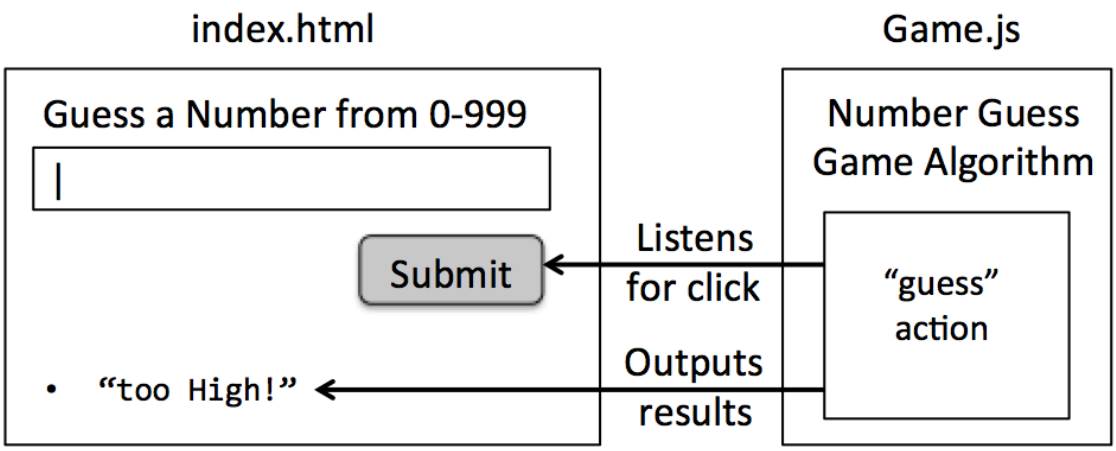

This version refactors codebase to improve UX.

From a Turn-based game into a Time-based for a novel & compelling version of Hi-Lo. Try to hack the system before the clock runs out!

Use graphics instead of HTML inputs

# Part 1:

Turn-based, Text-based with M.V.P. design



# Goal 1-0: Minimal Viable Product (MVP)

**Summary:**
This lab uses an agile approach for developing the Hi-Lo game, whereby we'll focus the initial build on delivering a Minimal Viable Product (MVP). A MVP is a version of an app with just enough features to be usable by early users who can then evaluate it for future development.

**MVP Specifications:**
The specification defines the necessary features for this version of the application

- Capture input from user through browser's viewport
- Update the browser's viewport with the player's results
- Implement the core game logic for executing the Hi-Lo game

# Goal 1-1: HTML for Hi-LO Game

## 'Approach' → Plan Phase

Make an HTML document that gives the game's instructions & its user inputs

## 'Apply' → Do Phase

Step 1: Create an index.html file & implement it with the base HTML content.

*index.html*

```html
<html>
  <head></head>
  <body>
    <p>There is a number between 000-999.</p>
    <p>You have 10 guesses.</p>
    <span>Enter a Guess:</span>
    <input type="text">
    <input type="button" value="Submit">
  </body>
</html>
```

## 'Approve' → Test Phase

Open the index.html document in the browser and ensure that it displays

There is a number between 000-999.

You have 10 guesses.

Enter a Guess: [ ] Submit

# Goal 1-2: DOM for JS controller (Event Listener)

## 'Approach' → Plan Phase

In this iteration, the HTML inputs are accessed from JavaScript which "listens" for a button click and triggers a "callback" function in response.

## 'Apply' → Do Phase

Step 1: Link the game.js file to index.html & give its inputs id attributes to access them from JS

*index.html*

```html
<html>
  <head></head>
  <body>
    <p>There is a number between 000-999.</p>
    <p>You have 10 guesses.</p>
    <span>Enter a Guess:</span>
    <input type="text" id="guess-text">
    <input type="button" value="Submit" id="guess-button">
    <script src="scripts/game.js"></script>
  </body>
</html>
```

Step 2: Use document to get html by id & add an event listener to button with callback function

*game.js*

```javascript
/* Get HTML Elements as JS objects */
const button = document.getElementById("guess-button");
const number =   document.getElementById("guess-text");

/* Add Event Listener to button with callback function*/
button.addEventListener("click", guessNumber);

/* Callback function for event: Button click  */
function guessNumber() {
    const guess = number.value;
    console.log(guess);
}
```

## 'Approve' → Test Phase

There is a number between 000-999.

You have 10 guesses.

Enter a Guess: 500 Submit

*in the console:*

```
500
```

# Goal 1-3: Hi-Lo Game logic in JS

## 'Approach' → Plan Phase

Implement the basic algorithm for the Hi-Lo game. The game's output displays to the console.

## 'Apply' → Do Phase

Step 1: Create variables for the random passcode (0-999) & the number of tries remaining.

*game.js*

```
/* Hi-Lo Game Data*/
const passcode = Math.floor( Math.random() * 1000 );
let tries = 10;
```

Step 2: Implement the logic for evaluating a guess to determine if user won, lost or gets a clues

*game.js*

```
/* Callback function for event: Button click  */
function guessNumber() {
   const guess = number.value;
   tries--;
   console.log(`Number of attempts left: ${tries}`);
   if ( guess == passcode){
     console.log(`You win! Got it in ${10-tries} attempts`);
   }
   else if (tries < 0){
       console.log(`You lose! The passcode was ${passcode}`);
   }
   else{
       giveClue(guess)
   }
}
```

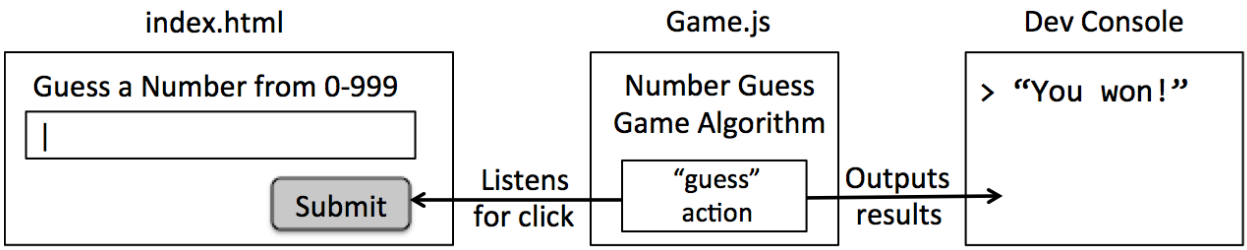
Step 3: Define a function that gives a clue, i.e. whether a guess was too low or too high.

*game.js*

```
/*Give Clue */
function giveClue(guess){
   if (guess > passcode){
       console.log(`${guess} is too High!`);
   }
   else{
       console.log(`${guess} is too Low!`);
   }
}
```

## 'Approve' → Test Phase

Play the game. Submit your guesses using the button & read result from the console.



# Goal 1-4: DOM for JS view (innerHTML)

## 'Approach' → Plan Phase

Use DOM API to update the HTML elements from the JavaScript

## 'Apply' → Do Phase

Step 1: Add an empty list for clues, with an id, also add an id to the paragraph for attempts

*index.html → <body>*

```
<body>
   <p>There is a number between 000-999.</p>
   <p id='attempts'>You have 10 guesses.</p>
   <span>Enter a Guess:</span>
   <input type="text" id="guess-text">
   <input type="button" id="guess-button" value="Submit">
   <ul id='clues'></ul>
   <script src="scripts/game.js"></script>
</body>
```

Step 2: Use the document object to access the HTML elements as JavaScript objects.

*game.js*

```
const attemptsView = document.getElementById("attempts");
const cluesView = document.getElementById("clues");
```

Step 3: Overwrite innerHTML of attempts text after a guess, & overwrite the body on gameover

*game.js*

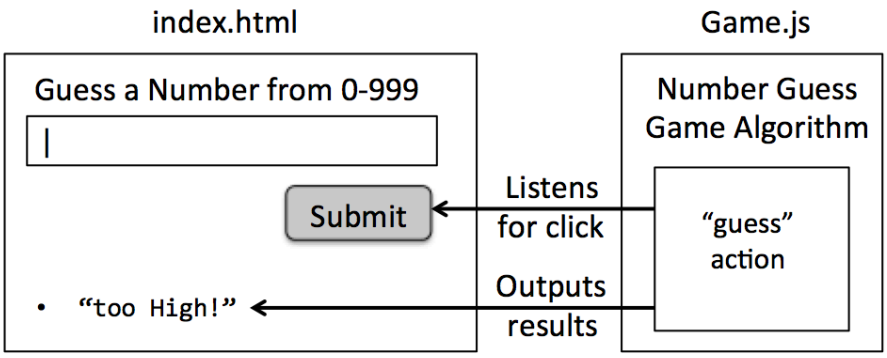
```
/* Callback function for event: Button click  */
function guessNumber() {
   const guess = number.value;
   tries--;
   attemptsView.innerHTML = `Number of attempts left: ${tries}`;
   if ( guess == passcode){
       document.body.innerHTML = `<h1>You win!</h1> <p>Got it in ${10-tries} attempts</p>`;
   }
   else if (tries < 0){
       document.body.innerHTML = `<h1>You lose!</h1> <p>The passcode was ${passcode}</p>`;
   }
   else{
       giveClue(guess)
   }
}
```

Step 4: Append to the innerHTML of the clues list with a new clue in a list item.

*game.js*

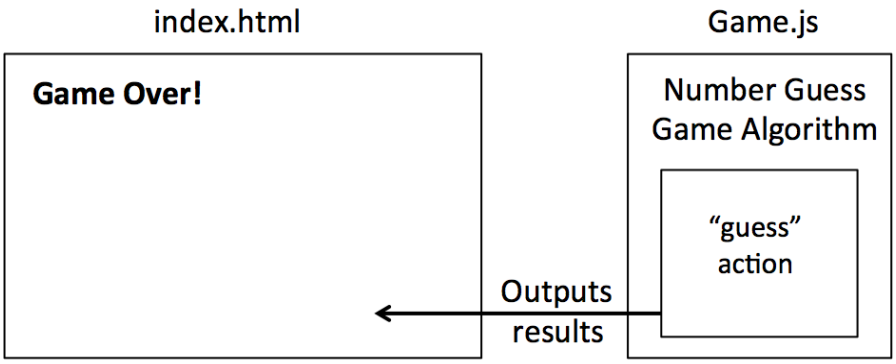
```
/*Give Clue */
function giveClue(guess){
   if (guess > passcode){
       cluesView.innerHTML += `<li>${guess} is too High!</li>`;
   }
   else{
       cluesView.innerHTML += `<li>${guess} is too Low!</li>`;
   }
}
```

## 'Approve' → Test Phase




*Game appends clues to page during play*

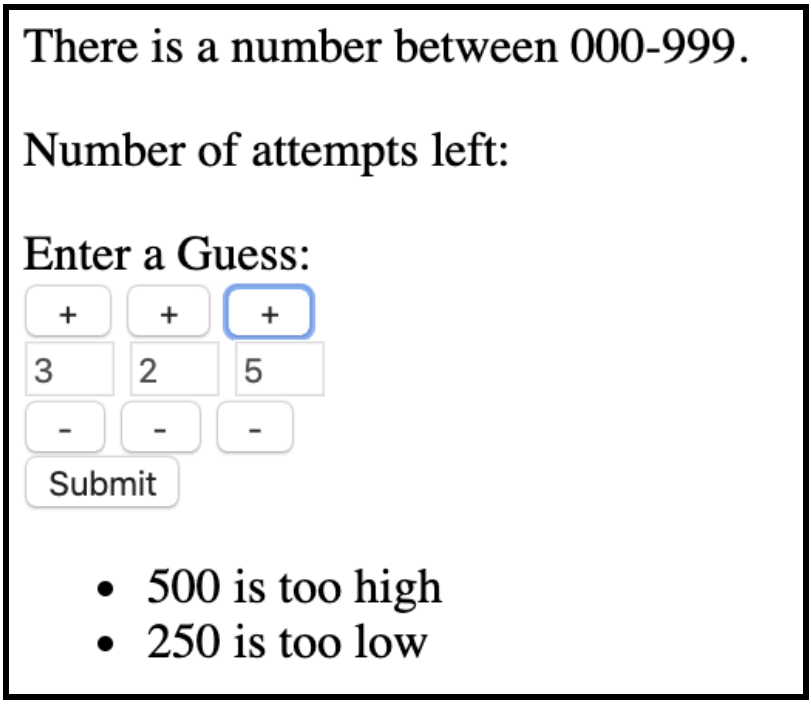
---




*Game overwrites page for Gameover*

# Part 2:

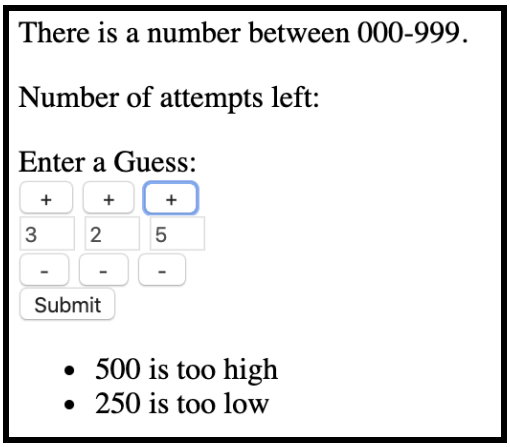
Turn-based, Text-based with MVC pattern



# Goal 2-0: MVC Design Pattern

**Summary:**
Let's create a new mockup for an improved version of the Puzzle Game. Each iterative goal will revolve around adding in those features.

**Countdown Challenge Game Mockup**




**Specifications:**

- *Improved Code Maintainability:* Adopt better software engineering principles and refactor code into three responsibilities: model, views, controllers.

- *Improved Controls:* Fault-tolerant user inputs that prevent invalid values. Ensure that inputs do not require a keyboard, but also support mobile users.

- *Improved Views:* Better UI/UX with more graphics and styling

*Responsibilities across M-V-C App*

| | |
|---|---|
| **View** | Manages the output from model to user |
| **Controller** | Manages the input from user into model |
| **Model** | The program's logic (game rules only, no input/output concerns) |

# Goal 2-1: Refactor with MVC Pattern

## 'Approach' → Plan Phase

Model-View-Controller (MVC) Architecture:

*Divide code for Hi-Lo game into three files*

| | model.js | view.js | controller.js |
|---|---|---|---|
| **variables** | `passcode`<br>`tries` | - | - |
| **functions** | `guessNumber()`<br>`giveClue()` | `printAttemptsRemaining()`<br>`printClue()`<br>`printGameOver()` | `initControls()`<br>`buttonEvent()` |

## 'Apply' → Do Phase

Step 0: In `scripts` folder, make 3 new files named: `model.js, controllers.js, views.js`

Step 1: [JS] Implement the Model → Hi-Lo game logic

*model.js*

```
const passcode =  Math.floor( Math.random()*1000 );
let tries = 10;

function guessNumber(guess){
   tries--;
   if ( guess == passcode ){
       printGameOver('WIN');
   }
   else if (tries <= 0){
       printGameOver('LOSE');
   }
   else{
       printAttemptsRemaining(tries);
       giveClue(guess);
  }
}

function giveClue(guess){
    if (guess > passcode){
        printClue('HI', guess);
    }
   else{
        printClue('LO', guess);
   }
}
```

## Step 2: [JS] Implement the Controller → Inputs for Hi-Lo

*controllers.js*

```
initControls();

function initControls(){
   const  button = document.getElementById("guess-button");
   button.addEventListener("click", buttonEvent);
}

function buttonEvent(){
   const number = document.getElementById("guess-text");
   guessNumber(number.value)
}
```

## Step 3: [JS] Implement the View → Outputs for Hi-Lo

*views.js*

```
function printAttemptsRemaining(tries){
   const attemptsText = document.getElementById("attempts");
   attemptsText.innerHTML = `Number of attempts left: ${tries}`;
}

function printClue(status, guess){
   const clueText = document.getElementById("clues");
   const clue = (status === 'HI') ? `<li>${guess} is too high</li>` : `<li>${guess} is too low</li>`;
   clueText.innerHTML += clue;
}

function printGameOver(status){
   if (status === 'WIN'){
       var message = `<h1>You Win!</h1> <p>Got it in ${10-tries} tries.</p>` ;
   }
   else{
       var message = `<h1>You Lose!</h1> <p>The number was: ${passcode}</p>`;
   }
   document.body.innerHTML = message;
}
```

## Step 4: [HTML] Update the index.html file's script tags to import the three new JS files.

*index.html → <body>*

```
<body>
   <p>There is a number between 000-999.</p>
   <p id="attempts">Number of attempts left: 10</p>
   <span>Enter a Guess:</span>
   <input type="text" id="guess-text">
   <input type="button" id="guess-button" value='Submit'>
   <ul id="clues"></ul>
   <script src="scripts/views.js"></script>
   <script src="scripts/model.js"></script>
   <script src="scripts/controllers.js"></script>
 </body>
```

# 'Approve' → Test Phase

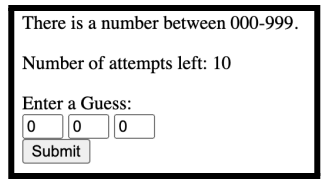
**Play the game!** Make sure it still works as it did before the refactoring.

# Goal 2-2: Refactor User Inputs

## 'Approach' → Plan Phase

New UI that improves precision & ensures valid input. Scheme is based on a combination lock.


*Combination Lock*




*New UI*

## 'Apply' → Do Phase

Step 1: **[ HTML ]** Refactor the text input tags from one to three number fields.

*index.html → <body>*

```
<body>
  <p>There is a number between 000-999.</p>
  <p id="attempts">Number of attempts left: 10</p>
  <span>Enter a Guess:</span>
  <div>
     <input type="number" min="0" max="9" value="0" id="digit-100s">
     <input type="number" min="0" max="9" value="0" id="digit-10s">
     <input type="number" min="0" max="9" value="0" id="digit-1s">
  </div>
  <input type="button" value='Submit' id="guess-button">
  <ul id="clues"></ul>

  <script src="scripts/views.js"></script>
  <script src="scripts/model.js"></script>
  <script src="scripts/controllers.js"></script>
</body>
```

Step 2: **[ JS ]** Update the buttonEvent function to get the guess

*controllers.js → buttonEvent()*

```
function buttonEvent(){
   const hundreds = document.getElementById("digit-100s");
   const tens =  document.getElementById("digit-10s");
   const ones =  document.getElementById("digit-1s");
   const number = "" + hundreds.value + tens.value + ones.value;
   guessNumber(number);
}
```

## 'Approve' → Test Phase

**Play the game!** Compatible with mouse/keyboard. Note: keyboards may input invalid numbers. Doesn't work with touch devices. Improve UI to support touch and prevent invalid values from keyboard

# Goal 2-3: Define a class Guess

## 'Approach' → Plan Phase

Model guess as a class maintained by the model that the view/controller references

## 'Apply' → Do Phase

**Step 1: [HTML]** Buttons to increment guess *(Controller)* & Read-only field to display it *(View)*

*index.html → <body>*

```
<body>
   <p>There is a number between 000-999.</p>
   <p id="attempts">Number of attempts left: 10</p>
   <span>Enter a Guess:</span>

   <div>
     <input type="button"  id="up-100s" value=' + '>
     <input type="button"  id="up-10s"  value=' + '>
     <input type="button"  id="up-1s"   value=' + '>
   </div>
   <div>
       <input disabled type="number" min="0" max="9" value="0" id="digit-100s">
       <input disabled type="number" min="0" max="9" value="0" id="digit-10s">
       <input disabled type="number" min="0" max="9" value="0" id="digit-1s">
   </div>
   <input type="button" value='Submit' id="guess-button">

   <ul id="clue"></ul>

   <script src="scripts/Guess.js"></script>
   <script src="scripts/views.js"></script>
   <script src="scripts/model.js"></script>
   <script src="scripts/controllers.js"></script>
</body>
```

**Step 2: [JS]** Create a Guess class that models the attributes & behaviors of a guess object

*Guess.js*

```
class Guess{
   constructor(){
     this.hundreds = 0;
     this.tens = 0;
     this.ones = 0;
   }
   toString(){
     return "" + this.hundreds + this.tens + this.ones;
   }
   increment(key){
     this[key] = (this[key] + 1) % 10;
   }
}
```

*Note: Objects are Dictionaries, so can access any attribute given its name as key in brackets.*

**Step 3: [JS]** Instantiate an instance of Guess in the model.js

*model.js → global variables*

```
const passcode =  Math.floor( Math.random()*1000 );
let tries = 10;
const guess = new Guess();
```

**Step 4: [JS]** Add a function to print the state of guess with DOM in view.js

*views.js*

```
function printDigits(){
   document.getElementById("digit-100s").value = guess.hundreds;
   document.getElementById("digit-10s").value = guess.tens;
   document.getElementById("digit-1s").value = guess.ones;
}
```

**Step 5: [JS]** Add a callback function (controller) to increment the guess and print it to the view

*controllers.js*

```
function incrementEvent(key){
   guess.increment(key);
   printDigits();
}
```

**Step 6: [JS]** Add event listeners to the '+' buttons that trigger the callback function to increment

*controllers.js*

```
function initControls(){
  const button = document.getElementById("guess-button");
  button.addEventListener("click", buttonEvent);
  const up100s = document.getElementById("up-100s");
  const up10s = document.getElementById("up-10s");
  const up1s = document.getElementById("up-1s");
  up100s.addEventListener("click", () =>  incrementEvent('hundreds'));
  up10s.addEventListener( "click", () =>  incrementEvent('tens')    );
  up1s.addEventListener(  "click", () =>  incrementEvent('ones')    );
}
```

**Step 7: [JS]** Get the state of guess from the instance and send to the game logic on a button event

*controllers.js*

```
function buttonEvent(){
   const number = guess.toString();
   guessNumber(number)
}
```

## 'Approve' → Test Phase

**Play the game!** This iteration now works on all devices and prevents invalid states. However, you can only increment the value up which doesn't offer a good UX.

# Goal 2-4: Decrement Controls

## 'Approach' → Plan Phase

Improve User Experience (UX) by adding a decrement control into the game.

## 'Apply' → Do Phase

**Step 1: [HTML]** Create HTML buttons for decrementing.

*index.html → <body>*

```
<body>
   <p>There is a number between 000-999.</p>
   <p id="attempts">Number of attempts left: 10</p>
   <span>Enter a Guess:</span>
   <div>
       <input type="button"  id="up-100s" value=' + '>
       <input type="button"  id="up-10s"  value=' + '>
       <input type="button"  id="up-1s"   value=' + '>
   </div>
   <div>
       <input disabled type="number" min="0" max="9" value="0" id="digit-100s">
       <input disabled type="number" min="0" max="9" value="0" id="digit-10s">
       <input disabled type="number" min="0" max="9" value="0" id="digit-1s">
   </div>
   <div>
       <input type="button"  id="down-100s" value=' - '>
       <input type="button"  id="down-10s"  value=' - '>
       <input type="button"  id="down-1s"   value=' - '>
   </div>
   <input type="button" value='Submit' id="guess-button">
   <ul id="clues"></ul>
   <script src="scripts/Guess.js"></script>
   <script src="scripts/views.js"></script>
   <script src="scripts/model.js"></script>
   <script src="scripts/controllers.js"></script>
</body>
```

**Step 2: [JS]** Add a decrement method into the body of the Guess class

*Guess.js*

```
decrement(key){
    this[key] = (this[key] > 0) ? this[key]-1 : 9;
}
```

*Note: Objects are Dictionaries, so can access any attribute given its name as key in brackets.*

**Step 3: [JS]** Add a callback function in controllers to decrement guess and print its value

*controllers.js*

```
function decrementEvent(key){
  guess.decrement(key);
  printDigits();
}
```

**Step 4: [JS]** Add event listeners to the down buttons that trigger the callback function to decrement

*controllers.js*

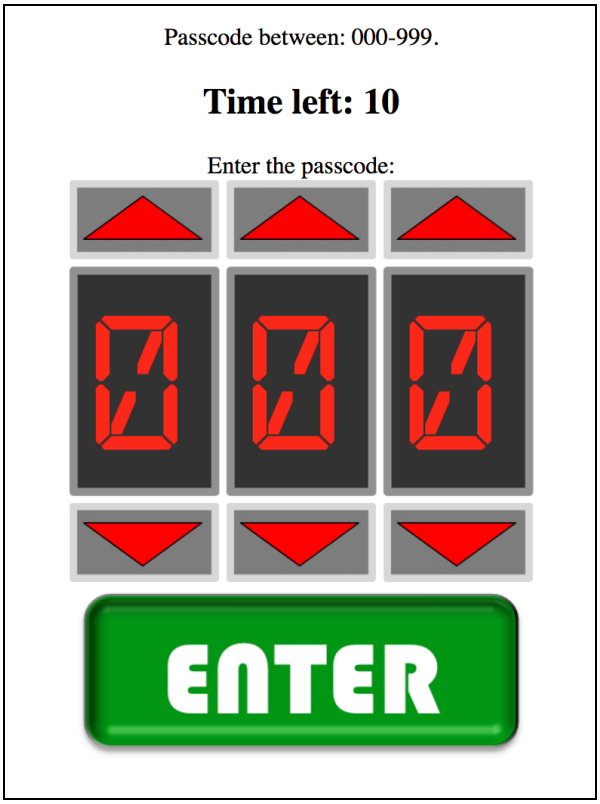
```
function initControls(){
  const button = document.getElementById("guess-button");
  button.addEventListener("click", buttonEvent);

  const up100s = document.getElementById("up-100s");
  const up10s = document.getElementById("up-10s");
  const up1s = document.getElementById("up-1s");
  up100s.addEventListener("click", () =>  incrementEvent('hundreds'));
  up10s.addEventListener( "click", () =>  incrementEvent('tens')    );
  up1s.addEventListener(  "click", () =>  incrementEvent('ones')    );

  const down100s = document.getElementById("down-100s");
  const down10s = document.getElementById("down-10s");
  const down1s = document.getElementById("down-1s");
  down100s.addEventListener("click", () => decrementEvent('hundreds')  );
  down10s.addEventListener( "click", () => decrementEvent('tens')      );
  down1s.addEventListener(  "click", () => decrementEvent('ones')      );
}
```

## 'Approve' → Test Phase

Play the game. This iteration now works on all devices and prevents invalid states. This version offers superior UX with increments/decrements & rollovers. This game may still be improved with graphics & time events.

# Part 3:

## Time-based, Graphics-based with MVC pattern

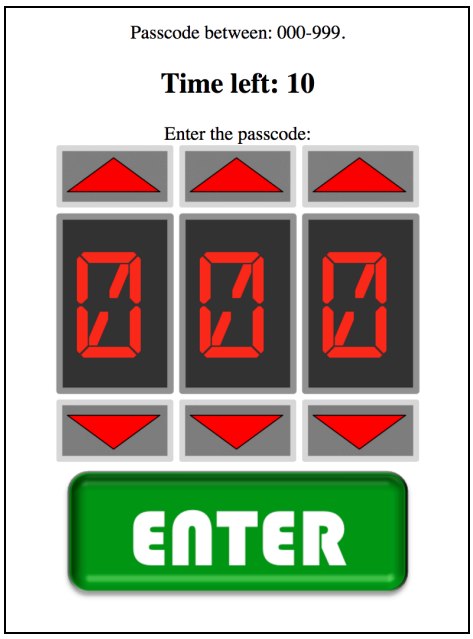


# Goal 3-0: Real-time Design

**Summary:**

Let's create a new mockup for an improved version of the Puzzle Game. Each iterative goal will revolve around adding in those features.

**Countdown Challenge Game Mockup**




**Specifications:**

- *Improved Gameplay:* Design a more compelling game experience by forcing the player into inputting choices quicker, where attempts are limited by a timer instead of a count.

- *Improved Views:* Replace all HTML elements with art & animated graphics.

# Goal 3-1: Graphical Controllers

## 'Approach' → Plan Phase

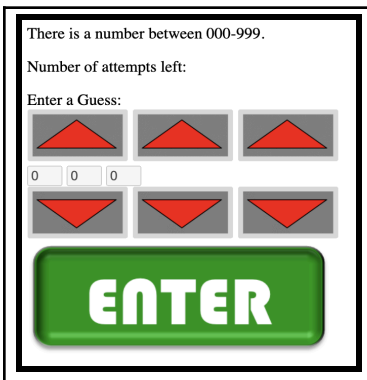
Replace the HTML button elements with rendered graphics as the controllers

## 'Apply' → Do Phase

**Step 1 [HTML]** Replace the buttons tags and input tags with image tags.

*index.html → <body>*

```
<body>
    <p>There is a number between 000-999.</p>
    <p id="attempts">Number of attempts left:</p>
    <span>Enter a Guess:</span>

    <div>
        <img src="assets/button-up.gif" id="up-100s" width="100"/>
        <img src="assets/button-up.gif" id="up-10s"  width="100"/>
        <img src="assets/button-up.gif" id="up-1s"   width="100"/>
   </div>
    <div>
        <input disabled type="number" min="0" max="9" value="0" id="digit-100s">
        <input disabled type="number" min="0" max="9" value="0" id="digit-10s">
        <input disabled type="number" min="0" max="9" value="0" id="digit-1s">
    </div>
    <div>
        <img src="assets/button-down.gif" id="down-100s" width="100"/>
        <img src="assets/button-down.gif" id="down-10s"  width="100"/>
        <img src="assets/button-down.gif" id="down-1s"   width="100"/>
   </div>
   <div>
        <img src="assets/button-enter.png" id="guess-button" width="300">
   </div>

    <ul id="clue"></ul>

    <script src="scripts/Guess.js"></script>
    <script src="scripts/views.js"></script>
    <script src="scripts/model.js"></script>
    <script src="scripts/controllers.js"></script>
</body>
```




## 'Approve' → Test Phase

**Play the game**! It now has graphical animated buttons!

- The view is ugly disabled HTML inputs at this point.

# Goal 3-2: Graphical View - Digits

## 'Approach' → Plan Phase

Render the view graphically: The Guess's digits.

## 'Apply' → Do Phase

**Step 1: [HTML]** Replace the input tags with image tags.

*index.html → <body>*

```
<body>
    <p>There is a number between 000-999.</p>
    <p id="attempts">Number of attempts left:</p>
    <span>Enter a Guess:</span>

    <div>
        <img src="assets/button-up.gif" id="up-100s" width="100"/>
        <img src="assets/button-up.gif" id="up-10s"  width="100"/>
        <img src="assets/button-up.gif" id="up-1s"   width="100"/>
   </div>
    <div>
       <img src="assets/0.png" id="digit-100s" width="100"/>
       <img src="assets/0.png" id="digit-10s"  width="100"/>
       <img src="assets/0.png" id="digit-1s"   width="100"/>
    </div>
    <div>
        <img src="assets/button-down.gif" id="down-100s" width="100"/>
        <img src="assets/button-down.gif" id="down-10s"  width="100"/>
        <img src="assets/button-down.gif" id="down-1s"   width="100"/>
   </div>
   <div>
        <img src="assets/button-enter.png" id="guess-button" width="300">
   </div>

    <ul id="clues"></ul>

    <script src="scripts/Guess.js"></script>
    <script src="scripts/views.js"></script>
    <script src="scripts/model.js"></script>
    <script src="scripts/controllers.js"></script>
</body>
```

**Step 2: [JS]** Refactor the printDisplay function to update the src attribute for the image.

*views.js*

```
function printDigits(){
   document.getElementById("digit-100s").src = `assets/${guess.hundreds}.png`;
   document.getElementById("digit-10s").src = `assets/${guess.tens}.png`;
   document.getElementById("digit-1s").src = `assets/${guess.ones}.png`;
}
```

# 'Approve' → Test Phase


**Play the game!**
It now has graphical controllers & views.

- However the clues are still an ugly HTML list.

# Goal 3-3: Graphical View - Clue

## 'Approach' → Plan Phase

Render the view graphically: Clues.

## 'Apply' → Do Phase

**Step 1: [HTML]** Remove the `<ul id='clues'></ul>` element

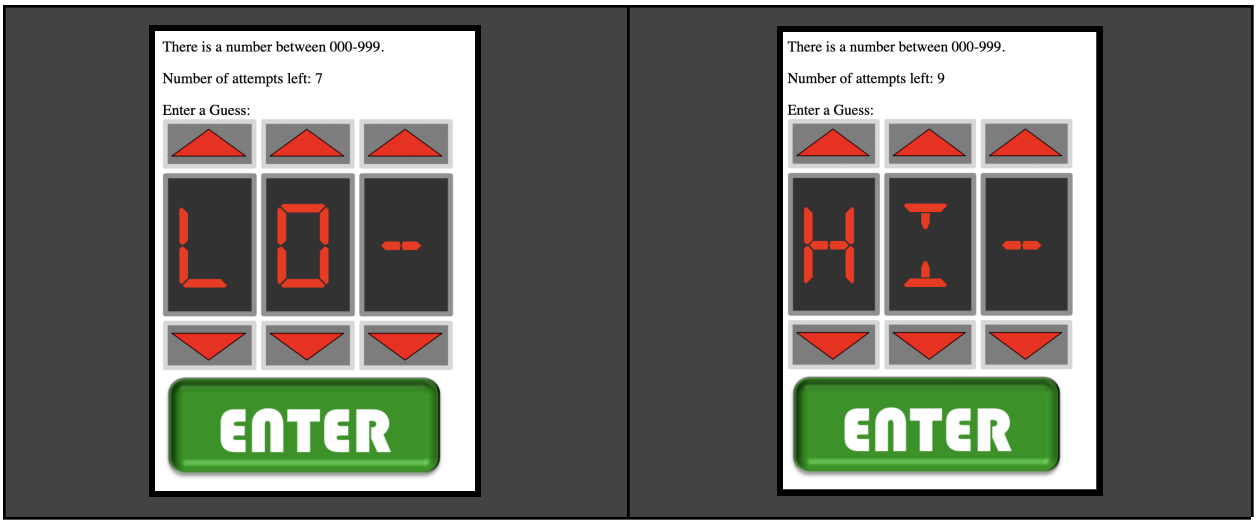
**Step 2: [JS]** Refactor the printClue() function to update the src attribute of the image. Destructuring is used in order to assign the three individual variables.

*views.js*

```
function printClue(status, guess){
   const [digit100,digit10,digit1] = status=='HI' ? ['H','I','-'] : ['L','O','-'];
   document.getElementById("digit-100s").src = `assets/${digit100}.png`;
   document.getElementById("digit-10s").src = `assets/${digit10}.png`;
   document.getElementById("digit-1s").src = `assets/${digit1}.png`;
}
```

## 'Approve' → Test Phase

**Play the game!** The game has graphic controllers & graphical views for the digits and clues. However, the clue obfuscates the view of the digits until the player increments/decrements the guess. This is poor UI/UX!



# Goal 3-4: Time Event: Overwrite View

## 'Approach' → Plan Phase

*(Goal):* Create a timer that overwrites the clue-view back into digits-view after 1 second.

To measure elapses in time, continually invoke a function & check the difference between 'now' time & 'then' time, if the duration is met, then perform the action & reset the 'now' time.

## 'Apply' → Do Phase

**Step 1: [JS]** Use Date class to get the current time in epoch time *(number of milliseconds since 1970)*

*model.js → global variables*

```
const passcode =  Math.floor( Math.random()*1000 );
let tries = 10;
const guess = new Guess();
let then = Date.now();
```

**Step 2: [JS]** Define a main looping function that prints the digits after a second has passed. The loop recursively occurs via the requestAnimationFrame function.

*model.js*

```
function main(){
   const now = Date.now();
   if (now - then > 1000){
       printDigits();
   }
   requestAnimationFrame(main);
}
main();
```

**Step 3: [JS]** Update the 'then' time everytime the clue is displayed to start the 1000ms countdown.

*views.js*

```
function printClue(status, guess){
   const [digit100,digit10,digit1] = status=='HI' ? ['H','I','-'] : ['L','O','-'];
   document.getElementById("digit-100s").src = `assets/${digit100}.png`;
   document.getElementById("digit-10s").src = `assets/${digit10}.png`;
   document.getElementById("digit-1s").src = `assets/${digit1}.png`;
   then = Date.now();
}
```

## 'Approve' → Test Phase

**Play the game.** The clue view should display for only a second before refreshing back to the digit view

# Goal 3-5: Time Event: Turn-based to Time-based

## 'Approach' → Plan Phase

*(Timer logic)* Use a timer instead of a counter for managing game over conditions. To manage game state over time, use a variable that tracks if the game is over or not.

## 'Apply' → Do Phase

**Step 1: [JS]** Initialize a timeleft variable to 30 & Initialize a gameover variable to false

*model.js → global variables*

```
const passcode =  Math.floor( Math.random()*1000 );
let tries = 10;
const guess = new Guess();
let then = Date.now();
let timeLeft = 30;
let gameover = false;
```

**Step 2: [JS]** Update gameover variable in win condition of guessNumber

*model.js*

```
function guessNumber(guess){
  tries--;
  if ( guess == passcode ){
      gameover = true;
      printGameOver('WIN');
  }
  else{
      giveClue(guess);
  }
}
```

**Step 3: [JS]** Update view to report in time left instead of attempts remaining

*views.js*

```
function printAttemptsRemaining(tries){
   const attemptsText = document.getElementById("attempts");
   attemptsText.innerHTML = `<h2>Time left: ${timeLeft}</h2>`;
}
```

**Step 4: [JS]** Update view to report gameover win with timeleft instead of tries left

*views.js*

```
function printGameOver(status){
   if (status === 'WIN'){
       var message = `<h1>You Win!</h1> <p>Got it in ${30-timeLeft} seconds.</p>` ;
   }
   else{
       var message = `<h1>You Lose!</h1> <p>The number was: ${passcode}</p>`;
   }
   document.body.innerHTML = message;
}
```

**Step 5: [JS]** Update main method to refresh the time left view after every second

*model.js*

```
function main(){
   const now = Date.now();
   if (gameover){
       return;
   }
   else if (timeLeft <= 0){
         printGameOver('LOSE');
   }
   else if (now - then > 1000){
       timeLeft--;
       printDigits();
       printAttemptsRemaining();
       then = Date.now();
   }
   requestAnimationFrame(main);
}
```

## 'Approve' → Test Phase

**Play the game!** The game should now be time-based instead of turn-based. Fin.

# Concluding Notes

## Graphics-view approach

Since the app was divided into MVC architecture, converting to a graphics view was trivial. Note the strategy we employed to programmatically manage which image asset to load.

## Future Improvements

- Improve the style of the game with CSS such as background colors and centering
- Store fastest time in leaderboard & save in local storage.
- Add a difficulty mode that supports either larger passcodes or different number bases
    - For the number of guesses to solve use: ceiling of log2(maxValue).

## Lab Submission

Compress your project folder into a zip file and submit on Moodle.