Table of Contents

devY

Overview

Recommended Grades

5th-8th

Mission

The goal of devY is to provide educators with the tools and curriculum to create a unique project based learning environment to teach coding. We believe that an offline exploration of concepts enhances student understanding, and engaging projects cement learning.

Unit structure

In each devY unit, students create a project that includes all of the concepts taught in the unit. The project is built in pieces, with each lesson mapping to a particular concept needed to create the larger project. Some lessons may take several classes to implement so the units are designed to let students move at their own pace.

Lesson structure

devY lessons are composed of the following sections:

Explore

Students engage with the coding concept in an offline way.

Questions

Map explore exercise to Javascript concept.

Explain

Explicit instruction on how to implement the Javascript.

Engage

Use the new concept in a real word situation. During this time, students also break off to coding stations to review/reinforce learning of a particular concept.

Tools

All of the platforms used in these lessons are 100% free.

cycle shell

Using cycle shell students can interact with their computer in a terminal like environment. This allows students to build meaningful web applications with only and introductory knowledge of Javascript.

cycle shell tutorial

codewars (optional)

Codewars is an online platform where users solve real coding challenges. This curriculum utilizes codewars to reinforce concepts when students break out into coding stations.

codewars tutorial

weo (optional)

Weo is an educational platform that makes it easy to create and grade assignments. The devY curriculum employs this to create quizzes and short review exercises.

weo tutorial

devY white

devY White is an introduction course that introduces students to the cycle shell coding environment and basic Javascript concepts.

Concepts

- functions (read, return, parameters, create)
- variables (create, use, update)
    - strings
    - numbers
- string templates
- simple conditionals

Project

Students demonstrate their knowledge of cycle-shell and these concepts to create a guess the number game.

Core Features

- [ ] **Check user guess**
    - Check to see if the user guessed the correct number and return an appropriate response.
    - *Skills learned:*
        - conditionals
        - parameters
        - return
        - strings
- [ ] **Store correct answer**
    - Create a variable and store the correct answer value
    - *Skills learned:*
        - variables
        - numbers
- [ ] **Track the number of guesses**
    - Keep track of the number of guesses that the user tries before finding the correct solution

  - *Skills learned:*
    - variables
    - incrementing numbers

- [ ] **Hints**
  - The game should supply the user with hints. The hint should be either "higher" or "lower" depending on the user's guess.
  - *Skills learned:*
    - conditionals
    - strings
    - functions
      - create
      - return

- [ ] **Set maximum number of guesses**
  - Create a maximum number of guesses. Once the user has reached the limit the game is over.
  - *Skills learned:*
    - variables
      - create
      - update
    - conditionals

- [ ] **Score**
  - The game should calculate a score for the user based on the number of guesses it took them to find the correct number
  - *Skills learned:*
    - variables
    - arithmetic

- [ ] **Get user name**
  - Get the name of the user before the game starts and use it to customize the messages throughout the game.
  - *Skills required:*
    - variables
      - create
      - update

        - string templates

Additional Features

- [ ] **Make correct answer a random number**
    - Use the random function to create a random number when the user starts a new game.
    - *Skills required:*
        - functions
            - call
        - variables
            - create
            - number
            - set
- [ ] **Reset**
    - Add the ability for the user to reset the game once the game has ended:
    - *Skills required*
        - variables
            - update
        - functions
            - create
            - call
            - return
- [ ] **Grades**
    - Grade the user based on the number of number of guesses and the number of attempts the user took.
    - *Skills required:*
        - conditionals
        - arithmetic
        - strings
        - functions
            - create
            - return

- [ ] **Set difficulty**
  - Create a way of the user setting the difficulty by changing: maximum number of tries, number range, score
  - *Skills required:*
    - functions
      - create
      - return
    - variables
      - update
      - numbers

cycle shell

Overview

In this lesson, students will use cycle shell to create simple applications.

Learning Goals

By the end of the lesson, students will be able to:

- read a function
- identify parameters
- explain how to return a value from a function
- concatenate strings

pseudocode planning

Distribute the features checklist and introduce the guess the number game.

Activity 1
This is the first activity blah blah blah

Materials: This is the list of materials to use blah blah

length:
5-10 minutes

teacher actions	**student actions**
1 **On Paper** Ask students to describe the steps to playing a number in game if they were playing with another person. The list should be similar to: 1. more 2. stuff 3. whatever	1 stuff
2 Tell students that the game is going to be built in stages as they learn new concepts.	
3 To create this game the first step is to learn about functions and cycle-shell.	3 the most stuff

Activity 2

This is the second activity

Materials: This is the materials list of materials to use blah blah

length:

20 minutes

teacher actions	student actions
1 **On Paper** Ask students to describe the steps to playing a number in game if they were playing with another person. The list should be similar to: 1. more 2. stuff 3. whatever	1 stuff
2 Tell students that the game is going to be built in stages as they learn new concepts.	2 the most stuff
3 To create this game the first step is to learn about functions and cycle-shell.	

Functions

Students go to http://cycle.sh which should have the following starting code:

```
function main (input) {
  return input
}
```

1. The first line of cycle-shell sets up the interactive page on the right side of the screen.
2. A function is a reusable set of instructions. A function is a block of code and has curly braces `{}` to mark the beginning and end of the code block. Code inside of the code block should be indented. The function is written as:

```
function name (parameter1, parameter2) {
     return value
}
```

1. Parameters are variables that only exist inside of the function.
    - In cycle shell the parameters have the value of whatever is written in the input box.

   - Each word that is separated by a space will be the value of the next parameter.
2. The `return` statement ends the function and sends back a value. In cycle shell this value is displayed.

3. Walk through an example of creating an echo application. After coding push the `Run Code` button in cycle shell.
4. Walk through the code as the computer would.

   - Type the word `hello` into the input box on the right side of the screen but do not push `enter` yet.
   - Ask students, 'When I push enter, what is the value of `input`?'

     - `input` = `'hello'`
   - Ask students, 'What value gets returned from this function?'

     - `hello`

Strings

1. To return words return ` `` `

   ```
   function main (input) {
     return `Hello`
   }
   ```

   ```
   function main (input) {
     return `Hello, Daniel`
   }
   ```

   i. To use a variable in a string, use a dollar sign followed by curly braces such as `${variableName}`

   ```
   function main (name) {
   return `Hello, ${name}`
   }
   ```

Engage

1. As a group, create an echo application that returns the user's input.

```
function main (input) {
  return input
}
```

i. Create a pluralizer that adds an 's' to the user's input.

```
function main (input) {
return `${input}s`
}
```

ii. Create a greeting app that returns 'Hello, -------'

```
function main (input) {
return `Hello, ${input}`
}
```

iii. Create a compliment machine app that returns a compliment with the name the user inputs.

```
function main (input) {
return `${input}, you have a wonderful smile.`
}
```

iv. Create a mad lib that takes multiple user inputs and places them in a sentence.

```
function main (adjective, noun, verb) {
return `The ${adjective} ${noun} ${verb} to the pond.`
}
```

Justify code

As a group discuss step by step what happens in one of the example programs. After going through one as a group, students practice explaining one of the other examples to a partner.

Check user guess

Learning Goals

By the end of the lesson, students will be able to:

- write simple conditionals
- return values based on input
- use conditionals to check if the user won in their guess the number game

Overview

This lesson introduces students to the process of adding new features to their projects. Start by leading a pseudocode planning time. Once the concepts required have been outlined, introduce students to conditionals using the conditionals lesson. After practicing conditionals in their smaller projects, refocus students on guess the number project. Students get with a partner and discuss strategies for implementing conditionals in their project to check if the player has guessed the correct number.

Suggested stations

1. String templates
2. Function syntax

plan (5 minutes)

1. Lead a discussion with students about how the guessing part of the game would work in the human context. The final solution should look similar to:
   i. Person 2 guesses a number.
   ii. Person 1 (game) checks if the guess is correct. (conditionals)
      - If the guess is correct person 2 wins (return, strings)
      - Otherwise person the guess is incorrect

1. Label each of the steps with a javascript concept (in the parentheses above)
2. Discuss which concepts are new.

discover (30-45 minutes)

conditionals lesson

produce (30 minutes)

pseudocode

```
FUNCTION main (guess)
  IF guess is equal to answer
    RETURN win message
  ELSE
    RETURN lose message
```

With their partner, students plan and attempt to implement a solution for checking if the user guess is correct.

1. Fill in handout
2. Discuss plans
3. Attempt a solution
4. Justify reasoning

implement (15 minutes)

1. Discuss with students their ideas. Make sure to have students from each group explain what they attempted or any ideas they had on how they could use their knowledge to implement a solution.
2. After hearing their solutions, introduce students to the teacher solution.

```
function main (guess) {
  return guessNumber(guess)
}

function guessNumber (guess) {
  if (checkGuess(guess)) {
    return `You win!` // or any winning message
  } else {
    return `Nope!` // or any losing message
  }
}

function checkGuess (guess) {
  if (guess === 4) { // could be any number
    return true
  } else {
    return false
  }
}
```

1. Students contrast the teacher solution with their own ideas.
2. Student implement a working solution in their project.
3. Students add comments to their code explaining what each line means.

justify (5 minutes)

1. As a group, explain the code that they have written. Make sure to decompose each block and expression that was added to the code.
2. Tell students that in the future, they are expected to justify their code in this way after each feature that they add.
3. Students practice by explaining their code to a partner.

test

1. Students play the game of their partner.
2. When students are testing a game, they are looking for:
    - bugs
    - feature improvements
3. When students find a bug they fill out a bug report:
    - What were you doing in the game?
    - What did you type into the input box?
    - What did you expect to happen?
    - What actually happened?

Adding hints

Learning Goals

By the end of the lesson, students will be able to:

- define a function
- use string templates
- use inequalities in a conditional

Overview

Students learn how to add hints to their game. During this lesson students learn how to create a new function and use it to implement a new feature in their game.

Suggested stations

- conditional syntax
- function syntax

plan (5 minutes)

1. Lead a discussion with students about how the guessing part of the game would work in the human context. The final solution should look similar to:

- [x] Person 2 guesses a number.
- [x] Person 1 (game) checks if the guess is correct. (conditionals)
    - [x] If the guess is correct person 2 wins (return, strings)
    - [ ] Otherwise person 1 give person 2 a hint: higher or lower. (function, return, strings, conditional)
    - Label each of the steps with a javascript concept (in the parentheses above)
    - Discuss which concepts are new.

discover (30-45 minutes)

creating functions lesson

produce (20 minutes)

Students create four functions: 1. `guessNumber` - calls checkGuess to check if the player got it right. If the player wins call renderPlayerWins else call renderHint 2. `checkGuess` - takes a number and returns either true or false 3. `renderHint` - takes a number and returns a hint message: `too high` or `too low` 4. `renderPlayerWins` - returns a message to display when the player wins

pseudocode:

```
FUNCTION main (guess)
  RETURN guessNumber(guess)

FUNCTION guessNumber (guess)
  IF checkGuess(guess) is true
    RETURN renderPlayerWins()
  ELSE
    RETURN renderHint(guess)

FUNCTION checkGuess (guess)
  IF guess is equal to answer
    RETURN true
  ELSE
    RETURN false

FUNCTION renderHint (guess)
  IF guess [is greater than] answer
    RETURN too big hint
  ELSE
    RETURN too small hint

FUNCTION renderPlayerWins
  RETURN win message
```

With their partner, students plan and attempt to implement a solution for rendering hints if the user guesses incorrectly.

1. Fill in handout
2. Discuss plans
3. Attempt a solution
4. Justify reasoning

implement (15 minutes)

1. Discuss with students their ideas. Make sure to have students from each group explain what they attempted or any ideas they had on how they could use their knowledge to implement a solution.

2. After hearing their solutions, introduce students to the teacher solution.

   Teacher solution:

```
function main (guess) {
   return guessNumber(guess) // pass the guess to the guessNumber function
}

function guessNumber (guess) {
 if (checkGuess(guess)) {
   return renderPlayerWins()
 } else {
   return renderHint(guess) // call the getHint function
 }
}

function checkGuess (guess) {
   if (guess === 6) { // could be any number
     return true
   } else {
     return false
   }
}

function renderHint (guess) {
   if (guess > 6) { // if the guess is larger than the answer
     return `${guess} is too high` // the guess is too high
   } else {
     return `${guess} is too low` // the guess is too low
   }
}

function renderPlayerWins () {
 return `Correct! You win!`
}
```

   i. Students contrast the teacher solutions with their own ideas.
3. Student implement a working solution in their project.

justify (5 minutes)

1. As a group, explain the code that they have written. Make sure to decompose each block and expression that was added to the code.
2. Tell students that in the future, they are expected to justify their code in this way after each feature that they add.
3. Students practice by explaining their code to a partner.

test

1. Students play the game of their partner.
2. When students are testing a game, they are looking for:
    - bugs
    - feature improvements
3. When students find a bug they fill out a bug report:
    - What were you doing in the game?
    - What did you type into the input box?
    - What did you expect to happen?
    - What actually happened?

store answer

Learning Goals

By the end of the lesson, students will be able to:

- define a variable
- use a variable to store a value
- use a variable in a conditional
- increment a variable

Overview

Students learn how to use variables to store the correct answer to their guess the number game. This lesson starts by discussing the advantages of storing information in a variable. If this is the first variable lesson for your students, use the variable concept lesson to introduce variables. Then students implement storing the correct solution in their game.

Suggested stations

- variable syntax
- calling functions

plan (5 minutes)

1. Have students look at their code from last time. code:

```
function main (guess) {
   return guessNumber(guess) // pass the guess to the guessNumber function
}

function guessNumber (guess) {
   if (guess === 6) { // could be any number
     return `Correct!`
   } else {
     return getHint(guess) // call the getHint function
   }
}

function getHint (guess) {
   if (guess > 6) { // if the guess is larger than the answer
     return `${guess} is too high` // the guess is too high
   } else {
     return `${guess} is too low` // the guess is too low
   }
}
```

   i. Ask students, 'What would you do if they wanted to change the answer in your game? What if you wanted to use a random number?'

2. Today students are going to learn how to create variables to set the value of the correct answer in one place and use it in their games.

discover (30-45 minutes)

- variables lesson

produce (20 minutes)

Students have two goals: 1. Store the correct answer as a variable 2. Keep track of how many guesses the user has attempted

pseudocode:

```
SET answer to number
SET guesses to 0

FUNCTION main (guess)
  guessNumber(guess)

FUNCTION guessNumber (guess)
  INCREMENT guesses
  IF checkGuess(guess) is true
    RETURN renderPlayerWins()
  ELSE
    RETURN renderHint(guess)

FUNCTION checkGuess (guess)
  IF guess is equal to answer
    RETURN true
  ELSE
    RETURN false

FUNCTION renderHint (guess)
  IF guess is greater than answer
    RETURN too big hint
  ELSE
    RETURN too small hint

FUNCTION renderPlayerWins ()
  RETURN win message
```

With their partner, students plan and attempt to implement a solution for storing the correct answer in a variable.

1. Fill in handout
2. Discuss plans
3. Attempt a solution
4. Justify reasoning

implement (15 minutes)

1. Discuss with students their ideas. Make sure to have students from each group explain what they attempted or any ideas they had on how they could use their knowledge to implement a solution.
2. After hearing their solutions, introduce students to the ideal solution. teacher solution:

```
var answer = 6
var guesses = 0

function main (guess) {
   return guessNumber(guess) // pass the guess to the guessNumber function
}

function guessNumber (guess) {
   guesses++ // Add one to guesses
   if (checkGuess()) { // could be any number
     return `Correct! It took you ${guesses} guesses to solve it!`
   } else {
     return renderHint(guess) // call the getHint function
   }
}

function checkGuess (guess) {
  if (guess === answer) {
    return true
  } else {
    return false
  }
}

function renderHint (guess) {
   if (guess > answer) { // if the guess is larger than the answer
     return `${guess} is too high` // the guess is too high
   } else {
     return `${guess} is too low` // the guess is too low
   }
}
```

   i. Students contrast the teacher solutions with their own ideas.

3. Student implement a working solution in their project.

justify (5 minutes)

1. As a group, explain the code that they have written. Make sure to decompose each block and expression that was added to the code.
2. Tell students that in the future, they are expected to justify their code in this way after each feature that they add.
3. Students practice by explaining their code to a partner.

test

1. Students play the game of their partner.

2. When students are testing a game, they are looking for:
    - bugs
    - feature improvements
3. When students find a bug they fill out a bug report:
    - What were you doing in the game?
    - What did you type into the input box?
    - What did you expect to happen?
    - What actually happened?

setting maximum guesses

Learning Goals

By the end of the lesson, students will be able to:

- keep track of the state of the game
- increment a variable

Overview

Students use their knowledge of defining functions, creating variables, and conditionals to add a maximum number of guesses to their game. If the maximum number of guesses has been exceeded, then the game returns a game over message.

Suggested stations

- variable values
- conditional syntax

plan (5 minutes)

1. Lead a discussion with students about how the guessing part of the game would work in the human context. The final solution should look similar to:

- [x] Person 1 thinks of a number. (variables)
- [ ] Person 1 decides the maximum number of guesses. (variables)
- [ ] If the person has guesses left. (conditionals)
    - [x] Person 2 guesses a number.
    - [x] Person 1 (game) checks if the guess is correct. (conditionals)
        - [x] If the guess is correct person 2 wins (return, strings)
        - [x] Otherwise person 1 give person 2 a hint: higher or lower. (function, return, strings, conditional)
- [ ] Otherwise game is over. (conditionals)
    1. Label each of the steps with a javascript concept (in the parentheses above)
    2. Discuss which concepts are new.

produce (20 minutes)

Remind students that their goals are: 1. Create a variable to store the maximum guesses 2. Create a function called `checkDone` that checks if the current number of guesses is greater than or equal to the maximum guesses.

- If it is tell the player the game is over.
- Otherwise check the guess.

  pseudocode:

```
SET answer to number
SET guesses to 0
SET maxGuesses to number

FUNCTION main (guess)
RETURN guessNumber(guess)

FUNCTION guessNumber (guess)
INCREMENT guesses
IF checkDone() is true
  RETURN lose message
ELSE
  IF checkGuess(guess) is true
    RETURN win message
  ELSE
    RETURN renderHint(guess)

FUNCTION checkDone (guess)
IF guesses is greater than or equal to maxGuesses
  RETURN true
ELSE
  RETURN false

FUNCTION checkGuess (guess)
IF guess is equal to answer
  RETURN true
ELSE
  RETURN false

FUNCTION renderHint (guess)
IF guess is greater than answer
  RETURN too big hint
ELSE
  RETURN too small hint
```

With their partner, students plan and attempt to implement a solution for creating a maximum number of guesses. If the maximum number is reached, the game should output 'Game over'

1. Fill in handout
2. Discuss plans
3. Attempt a solution
4. Justify reasoning

implement (15 minutes)

1. Discuss with students their ideas. Make sure to have students from each group explain what they attempted or any ideas they had on how they could use their knowledge to implement a solution.
2. After hearing their solutions, introduce students to the teacher solution.

```
var MAX_TRIES = 10
var guesses = 0
var answer = 6

function main (guess) {
   return guessNumber(guess)
}

function guessNumber (guess) {
   if (checkDone()) {
     return 'Game over'
   } else {
     if (checkGuess()) {
       return `Correct!`
     } else {
       return renderHint(guess)
     }
   }
}

function checkDone (guess) {
   if (guesses >= MAX_TRIES) {
     return true
   } else {
     return false
   }
}

function checkGuess (guess) {
   if (guess === answer) {
     return true
   } else {
     return false
   }
}

function renderHint (guess) {
   if (guess > answer) {
     return `${guess} is too high!`
   } else {
     return `${guess} is too low!`
   }
}
```

   i. Students contrast the ideal solutions with their own ideas.

3. Student implement a working solution in their project.

justify (5 minutes)

1. As a group, explain the code that they have written. Make sure to decompose each block and expression that was added to the code.
2. Tell students that in the future, they are expected to justify their code in this way after each feature that they add.
3. Students practice by explaining their code to a partner.

test

1. Students play the game of their partner.
2. When students are testing a game, they are looking for:
    - bugs
    - feature improvements
3. When students find a bug they fill out a bug report:
    - What were you doing in the game?
    - What did you type into the input box?
    - What did you expect to happen?
    - What actually happened?

get score

Learning Goals

By the end of the lesson, students will be able to:

- call a function from a template string
- use arithmetic to create a score for their player
- render the score for their player

Overview

Students add a feature to the game where the user is presented with a score based on the number of guesses they have attempted.

Suggested stations

- javascript arithmetic
- calling functions
- defining functions

plan (5 minutes)

1. Lead a discussion with students about how the guessing part of the game would work in the human context. The final solution should look similar to:

- [x] Person 1 thinks of a number. (variables)
- [x] Person 1 decides the maximum number of guesses. (variables)
- [x] If the person has guesses left. (conditionals)
  - [x] Person 2 guesses a number.
  - [x] Person 1 (game) checks if the guess is correct. (conditionals)
    - [x] If the guess is correct person 2 wins (return, strings)
      - [ ] The user receives a score based on the number of tries
    - [x] Otherwise person 1 give person 2 a hint: higher or lower. (function, return, strings, conditional)
- [x] Otherwise game is over. (conditionals)
  1. Label each of the steps with a javascript concept (in the parentheses above)

2. Discuss which concepts are new.

produce (20 minutes)

Students need to: 1. Create a function called `getScore` that calculates the score using the formula: `(guessesLeft / MAX_TRIES) * 100` 2. Create a function called `renderScore` that uses a string template to create a sentence out of the score. 3. Call `renderScore` when the player either wins or loses.

pseudocode:

```
SET answer to number
SET guesses to 0
SET maxGuesses to number

FUNCTION main (guess)
  RETURN guessNumber(guess)

FUNCTION guessNumber (guess)
  INCREMENT guesses
  IF checkDone() is true
    RETURN lose message
  ELSE
    IF checkGuess(guess) is true
      RETURN win message
    ELSE
      RETURN renderHint(guess)

FUNCTION checkDone (guess)
  IF guesses is greater than or equal to maxGuesses
    RETURN true
  ELSE
    RETURN false

FUNCTION checkGuess (guess)
  IF guess is equal to answer
    RETURN true
  ELSE
    RETURN false

FUNCTION renderHint (guess)
  IF guess is greater than answer
    RETURN too big hint
  ELSE
    RETURN too small hint

FUNCTION renderScore ()
  SET score to getScore()
  RETURN your score is: + score

FUNCTION getScore ()
  SET guessesLeft = maxGuesses - guesses
  RETURN (guessesLeft / maxGuesses) * 100
```

With their partner, students plan and attempt to implement a solution for getting and rendering a score based using the formula `(guessesLeft / maxGuesses) * 100`.

1. Fill in handout
2. Discuss plans
3. Attempt a solution

4. Justify reasoning

implement (15 minutes)

1. Discuss with students their ideas. Make sure to have students from each group explain what they attempted or any ideas they had on how they could use their knowledge to implement a solution.
2. After hearing their solutions, introduce students to the teacher solution.

```
var MAX_TRIES = 10
var guesses = 0
 //...

function checkDone (guess) {
  if (guesses >= MAX_TRIES) {
    var scoreMessage = renderScore()
    return `Game Over! ${scoreMessage}`
  } else {
    checkGuess(guess)
  }
}

function checkGuess(guess) {
  if (guess === answer) {
    var score = renderScore()
    return `Correct! ${score}`
  } else {
    var hint = renderHint()
    return `Wrong! ${hint}`
  }
}

function renderScore () {
  var score = getScore()
  return `Your score is: ${score}`
}

function getScore () {
  var guessesLeft = MAX_TRIES - guesses
  return (guessesLeft / MAX_TRIES) * 100
}
```

   i. Students contrast the ideal solutions with their own ideas.
3. Student implement a working solution in their project.

justify (5 minutes)

1. As a group, explain the code that they have written. Make sure to decompose each block and expression that was added to the code.
2. Tell students that in the future, they are expected to justify their code in this way after each feature that they add.
3. Students practice by explaining their code to a partner.

test

1. Students play the game of their partner.
2. When students are testing a game, they are looking for:
    - bugs
    - feature improvements
3. When students find a bug they fill out a bug report:
    - What were you doing in the game?
    - What did you type into the input box?
    - What did you expect to happen?
    - What actually happened?

get username

Learning Goals

By the end of the lesson, students will be able to:

- Use conditionals to check if the user has inputted a name
- Update a variable

Overview

Students add a welcome message to their game and ask the user to input a name. Students save the name and use it in their code to personalize the messages that get sent back to the user.

Suggested stations

- conditionals
- setting variables

plan (5 minutes)

1. Lead a discussion with students about how the guessing part of the game would work in the human context. The final solution should look similar to:

- [ ] Person 1 gets the name of the player 2 (variables, conditionals)
- [x] Person 1 thinks of a number. (variables)
- [x] Person 1 decides the maximum number of guesses. (variables)
- [x] If the person has guesses left. (conditionals)
    - [x] Person 2 guesses a number.
    - [x] Person 1 (game) checks if the guess is correct. (conditionals)
        - [x] If the guess is correct person 2 wins (return, strings)
            - [x] The user receives a score based on the number of tries
        - [x] Otherwise person 1 give person 2 a hint: higher or lower. (function, return, strings, conditional)
- [x] Otherwise game is over. (conditionals)
    1. Label each of the steps with a javascript concept (in the parentheses above)

2. Discuss which concepts are new.

produce (20 minutes)

pseudocode with students:

```
SET name

FUNCTION main (input)
  IF name is empty
    RETURN CALL setName(input)
  ELSE
    RETURN CALL checkGuess(input)

FUNCTION setName (input)
  SET name to input
  RETURN Ok + name + guess a number between 1 and 100
```

1. Create a variable called `name`
2. When the user inputs:
    - If there is no name saved in the variable `name`
        - call the `setName` function
        - the `setName` function should:
            - update the value of `name` to equal the input
            - return a message

              ```
              `Ok ${name}, guess a number between 1 and ${MAX_NUMBER}`
              ```

- Otherwise
    - the input is a guess and should call `checkDone`

With their partner, students plan and attempt to implement a solution for getting the user name.

1. Fill in handout
2. Discuss plans
3. Attempt a solution
4. Justify reasoning

implement (15 minutes)

1. Discuss with students their ideas. Make sure to have students from each group

explain what they attempted or any ideas they had on how they could use their knowledge to implement a solution.

2. After hearing their solutions, introduce students to the teacher solution.

```
require('cycle-shell')(main, {
   welcome: `Welcome to my game! Please enter your name!` //Add a welcome message to
your game
})

var name // create a name variable but do not assign a value or set as empty strings
var MAX_NUMBER = 10

function main (input) {
   if (!name) {
     return setName(input)
   } else {
     return checkDone(input)
   }
}

function setName (input) {
   name = input
   return renderWelcome()
}

function renderWelcome () {
   return `Ok ${name}, guess a number between 1 and ${MAX_NUMBER}`
}

//... checkDone ...
```

   i. Students contrast the ideal solutions with their own ideas.

3. Student implement a working solution in their project.

justify (5 minutes)

1. As a group, explain the code that they have written. Make sure to decompose each block and expression that was added to the code.
2. Tell students that in the future, they are expected to justify their code in this way after each feature that they add.
3. Students practice by explaining their code to a partner.

test

1. Students play the game of their partner.

2. When students are testing a game, they are looking for:
    - bugs
    - feature improvements
3. When students find a bug they fill out a bug report:
    - What were you doing in the game?
    - What did you type into the input box?
    - What did you expect to happen?
    - What actually happened?

reset

Learning Goals

By the end of the lesson, students will be able to:

- create a function that updates all of the variable values

Overview

Students add a reset function that resets the variable values back to their original state. This function should be called whenever the user enters the word 'reset'.

Suggested stations

- conditional syntax
- updating variable values

plan (5 minutes)

1. Lead a discussion with students about how the guessing part of the game would work in the human context. The final solution should look similar to:

- [ ] If at any points the user types 'reset' the game should be reset to it's initial state (variables, conditionals, function)
- [x] Person 1 gets the name of the player 2 (variables, conditionals)
- [x] Person 1 thinks of a number. (variables)
- [x] Person 1 decides the maximum number of guesses. (variables)
- [x] If the person has guesses left. (conditionals)
    - [x] Person 2 guesses a number.
    - [x] Person 1 (game) checks if the guess is correct. (conditionals)
        - [x] If the guess is correct person 2 wins (return, strings)
            - [x] The user receives a score based on the number of tries
        - [x] Otherwise person 1 give person 2 a hint: higher or lower. (function, return, strings, conditional)
- [x] Otherwise game is over. (conditionals)
    1. Label each of the steps with a javascript concept (in the parentheses above)

2. Discuss which concepts are new.

produce (20 minutes)

pseudocode:

```
FUNCTION main (input)
  IF input is equal to reset
    RETURN reset()
  ELSE IF name is empty
    RETURN setName(input)
  ELSE
    RETURN checkDone(input)

FUNCTION reset
  SET guesses to 0
  SET name to ''
  RETURN welcome message
```

- create a function called `reset`
  - reset should set all of the variables back to their original values
- the main function should check to see if the user inputted 'reset'
  - if the input === 'reset', call the reset function

With their partner, students plan and attempt to implement a solution for reseting the game.

1. Fill in handout
2. Discuss plans
3. Attempt a solution
4. Justify reasoning

implement (15 minutes)

1. Discuss with students their ideas. Make sure to have students from each group explain what they attempted or any ideas they had on how they could use their knowledge to implement a solution.
2. After hearing their solutions, introduce students to the teacher solution.

```
// ...variables

function main (input) {
  if (input === 'reset') {
    return reset()
  } else if (!name) {
    return setName(input)
  } else {
    checkDone(input)
  }
}

function reset () { // the reset function resets all of the changing variables back
to their original state
  guesses = 0
  name = ''
}

// Other functions..
```

   i. Students contrast the ideal solutions with their own ideas.
3. Student implement a working solution in their project.

justify (5 minutes)

1. As a group, explain the code that they have written. Make sure to decompose each block and expression that was added to the code.
2. Tell students that in the future, they are expected to justify their code in this way after each feature that they add.
3. Students practice by explaining their code to a partner.

test

1. Students play the game of their partner.
2. When students are testing a game, they are looking for:
    - bugs
    - feature improvements
3. When students find a bug they fill out a bug report:
    - What were you doing in the game?
    - What did you type into the input box?
    - What did you expect to happen?

- What actually happened?

render grade

Learning Goals

By the end of the lesson, students will be able to: -

Overview

Suggested stations

plan (5 minutes)

produce (20 minutes)

With their partner, students plan and attempt to implement a solution for checking if the user guess is correct.

1. Fill in handout
2. Discuss plans
3. Attempt a solution
4. Justify reasoning

implement (15 minutes)

1. Discuss with students their ideas. Make sure to have students from each group explain what they attempted or any ideas they had on how they could use their knowledge to implement a solution.
2. After hearing their solutions, introduce students to the teacher solution.

```
/**
... other functions ...

Get grade based on score
Assume grades work as follows:
>= 90 A
>= 80 B
>= 70 C
>= 60 D
< 60 F
*/

function getGrade () {
   var score = getScore()
   if (score >= 90) {
     return 'A'
   } else if (score >= 80) {
     return 'B'
   } else if (score >= 70) {
     return 'C'
   } else if (score >= 60) {
     return 'D'
   } else {
     return 'F'
   }
}

function renderGrade () {
   var grade = getGrade()
   return `Your grade was a: ${grade}`
}
```

   i. Students contrast the ideal solutions with their own ideas.

3. Student implement a working solution in their project.

justify (5 minutes)

1. As a group, explain the code that they have written. Make sure to decompose each block and expression that was added to the code.
2. Tell students that in the future, they are expected to justify their code in this way after each feature that they add.
3. Students practice by explaining their code to a partner.

test

1. Students play the game of their partner.

2. When students are testing a game, they are looking for:
    - bugs
    - feature improvements
3. When students find a bug they fill out a bug report:
    - What were you doing in the game?
    - What did you type into the input box?
    - What did you expect to happen?
    - What actually happened?

defining functions

Overview

This lesson starts with students exploring decomposition of a problem and creating functions. Then, students learn how to create a function using Javascript. Finally, students cement understanding by completing codewars exercises.

Objectives

By the end of the lesson, students will be able to:

- recognize proper function syntax
- create a function to solve a problem

Explore

Build functions for a day

1. Each student picks a relatively simple task such as getting ready for school in the morning. Get out of bed, get dressed, brush teeth, etc...
2. Break the activity down into smaller actions and write those tasks.
3. Have one student read the tasks to the instructor, who models them. Takes a long time, right? Hard to read all of those instructions?
4. Each student names their list of actions. And then one by one, they call out their function, and the instructor enacts them one after the other.
5. Now, pick your own order. Instead of writing all of individual actions, use the lists that your peers wrote. Put them together into a chain of actions to get ready in the morning.

Questions

1. Why was creating the functions useful?
2. How did you use a specific function?

Explain

1. Defining a function teaches the program a new task.
2. A function is a list of instructions, like the lists students generated during the explore

phase, that is stored with a name for later use.
3. The function won't run until it is called.

Hello World Example

Create a function that says 'Hello World'. Students follow along by coding this example on their own computers

1. Define a function called helloWorld

```
function helloWorld () {
// code goes here
}
```

   i. Add the list of actions inside the curly braces. In this case we want to use return "Hello World"

```
function helloWorld () {
   console.log('Hello World')
}
```

   i. Why is the program not doing anything?
2. The function won't run until it is called
3. Call the function by writing the function's name followed by parentheses

```
function helloWorld () {
   console.log('Hello World')
}

helloWorld()
```

   i. Functions can be called multiple times.

```
function helloWorld () {
   console.log('Hello World')
}

helloWorld()
helloWorld()
helloWorld()
```

Engage

1. http://www.codewars.com/kata/55f73be6e12baaa5900000d4/train
2. http://www.codewars.com/kata/55f73f66d160f1f1db000059/train
3. http://www.codewars.com/kata/563a5f4366fbf8cc6e00008b/train

Simple conditionals

Explore

Play a game where students react to different situations based on instructions. For example:

1. Start the game with just an if statement

   ```
   if (shoeColor === `white`) {
     raiseRightHand()
   }
   ```

   i. Then add in an else
2. Add an else-if

   ```
   if (shoeColor === `white`) {
     raiseRightHand()
   } else if (shoeColor === `blue`) {
     raiseLeftHand()
   } else {
     clap()
   }
   ```

   i. Students interpret the rules. If they do the wrong actions they sit down.

Questions

1. What was special about the equals signs used in the if statements?
2. Why are conditionals (if and else) important?
3. What do you think you could use conditionals for in your code?

Explain

As a group, build an application that only responds if the user inputs the word `hello`

1. Start by asking students, "How many inputs do we need to create this application? What should it be called?"

```
This application need one input.
A simple name could be input but the name of the parameter doesn't matter.
```

```
function main (input) {

}
```

i. How can the program check **if** the user inputed hello?

   Add an if statement. The if should check if input is equal to the string `hello`.

```
function main (input) {
   if (input === `hello`) {

   }
}
```

i. Choose how the application should responds

   ```
   function main (input) {
   if (input === `hello`) {
     return `Thanks for saying hello`
   }
   }
   ```

Engage

Create conditional codewars exercises

1. Comparing values handout - https://www.weo.io/activity/577409e042d5437c00d65614/public/preview/
2. Personalized greeting - https://www.codewars.com/kata/5772da22b89313a4d50012f7/train/javascript

   Solution:

```
function greet (name, owner) {
  if (name === owner) {
    return `Hello boss`
  } else {
    return `Hello guest`
  }
}
```

i. Calculator - https://www.codewars.com/kata/basic-mathematical-operations

Solution:

```
function basicOp (op, num1, num2) {
  if (op === `+`) {
    return num1 + num2
  } else if (op === `-`) {
    return num1 - num2
  } else if (op === `*`) {
    return num1 * num2
  } else if (op === `/`) {
    return num1 / num2
  } else {
    return `I can't do that operation`
  }
}
```

Conditional Continued

Learning Goals

By the end of the lesson, students will be able to:

- demonstrate learning by creating a calculator

Overview

Students review conditionals and continue to practice creating apps using cycle.sh

Explore

Students explore conditionals by playing a game of simon says with conditionals.

1. Write a series of if/else statements
2. Students follow along with the statements written on the board.
3. If a students does the wrong action they sit down.

Explain

Create conditional codewars 2 exercises

1. Review what students know about conditionals

```
if (condition) {
   // what to do if true
} else {
   // what to do if false
}
```

Engage

1. Assign letter grade

```
function main (grade1, grade2, grade3) {
   var sum = grade1 + grade2 + grade3
   var num = sum / 3
   if (num >= 90) {
     return 'A'
   } else if (num >= 80) {
     return 'B'
   } else if (num >= 70) {
     return 'C'
   } else if (num >= 60) {
     return 'D'
   } else {
     return 'F'
   }
}
```

i. Password unlock

```
var password = 'secret'

function main (input) {
     if (input === 'password') {
       return `Unlocked`
     } else {
       return `Locked`
     }
}
```

Variables (numbers + strings)

Explore

1. Set up 5 boxes and label them:
    - favoriteNumber
    - randomNumber
    - age
    - favoriteAnimal
    - leastFavoriteAnimal
2. Students write their personal answers for each of those subjects on small pieces of paper.
3. Have one student volunteer to read their favorite number paper aloud and then drop it in the corresponding box.
4. Ask students, “What is the value that is stored in favoriteNumber?”
    - The value of favoriteNumber is whatever is on that student’s paper
5. Continue until there is a paper in each of the boxes.
6. Ask students what they think will happen if we want to change the value of favorite number.
    - To change the value, take the old value out and drop the new value in.
    - Change the values stored in the boxes.
7. Ask students, “What is age + randomNumber?”
    - The answer is the value that is stored inside the age box added to the value that is stored in the randomNumber box.

Suggested Questions:

- Why was it easier to know what the numbers stored in the variables meant?
- What do you think this is trying to tell us about giving instructions to computers?

Explain

Students follow along with the following example on repl.it.

Introduce variables

1. A variable is a way to store data and give it a name so that it can be used later.
2. Draw a box on the whiteboard. Explain that a variable is like a piece of paper that you can store a value on.
3. Ask students, “What are some things that you think could be stored in a variable?”
4. Students first write down the information they used in the previous activity. The goal is to keep each item to one line and come up with their own instruction conventions.
5. Explain that humans can understand those instructions, but computers use their own language. Show how it’s done.
6. To create a variable start with the keyword ‘var’ and then add a name

   ```
   var myAge
   ```

Introduce numbers

1. Now that there is a variable named myAge it can be given a value
2. One type of value that can be stored is a number
3. Students store their age in the variable myAge using the ‘=’

   ```
   var myAge = 27
   ```

   i. To change the value of a variable once it has already been defined drop the keyword `var`

      ```
      var myAge = 27
      myAge = 28
      ```

   ii. You can also use math operations to change the value of a variable. Show students the example below and ask them what the value of myAge will be after the code is run. (29)

      ```
      var myAge = 27
      myAge = myAge + 2
      ```

Introduce strings

1. The information that students have stored are numbers.

2. Now students create a variable named favoriteAnimal and store the value from the explore phase.

   ```
   var favoriteAnimal = dog
   ```

   i. To save a word students need to add back tick (to the left of the one on the keyboard) marks. This tells the computer that the value is a word or a string.

      ```
      var favoriteAnimal = `dog`
      ```

   ii. Students add a second variable for their leastFavoriteAnimal

3. Variables can be added to strings by putting them inside of a pair of curly braces with a dollar sign in front. `${variable}`

   ```
   var phrase = `My favorite animal is ${favoriteAnimal}`
   ```

Engage

Create variables codewars

1. Split into smaller groups with tutors to complete variable worksheet. (http://bit.ly/1SebxsH)
2. Messi goals
3. Variable assignment debug

devY blue

prerequisites: devY white

devY blue focuses on continued exploration of introductory Javascript concepts to create a more advanced web application in cycle-shell.

Concepts:

- variables
- objects
- complex conditionals
- functions (writing)
- debugging

Inspiration

Students create a text based game in cycle-shell. The inspiration for this project is the game Zork. Check out an online version here to see how the game works.

Project Document

Look at the project document for more information on the order and structure of the game.

Object literals

Learning Goals

By the end of the lesson, students will be able to create a Javascript object using literal notation, get values at a specified key, and demonstrate knowledge by creating the first room in their game.

Overview

Students start by exploring books as objects. Students then learn how to create those same objects in Javascript using repl.it. Students use objects to complete the activities and then finish by creating the first room in their game.

Explore

Example level

In the second room hide a key and a sword cutout

Sample script: Room 1 - You wake up in a large empty room with no windows and no furniture. There is a musty smell in the air and you can't here anyone else around. At the far end of the room you see a door.

Room 2 - As you walk into the next room, the only light is coming from the door behind you. With that light you can barely see a light switch. (Wait for students to give you command to turn on the lights.) With the lights on you see (describe the things in the room that you hid the key and sword under).

Room 3 - Upon entering the room you hear strange noises in the distance. As you look around you see the room has a window and piles of old books. (When the students try to leave through the window...) A monster jumps through the window

1. Show students the example escape the room level.
2. The goal is to make it out of the room by giving instructions to the computer (instructor).
3. The only instructions that the computer will understand will be those given in verb + noun form. For example:
    - Take key

    - Open door
4. There can also be a few special verbs that work by themselves such as:
    - Look
    - Help
5. After each command, the computer responds with a new description of the room.

Play Zork

Give students 10 minutes try out Zork on the computer at tinyurl.com/9dotszork

Student created level

Creating an escape the room game.

1. Tell students they will be creating an escape the room game.
2. Split students into the following roles (roles will change for each level):
    - 1 player
        - Interacts with the computer to play the game.
    - 1 computer
        - Receives the level design from the designers and sets up the room.
        - Gives the player messages about their current state while they are playing.
    - Everyone else is a level designer
        - Create the level and hand it to the computer.
        - The level should have a description of the room and possible interactions.
        - Once the level is submitted the level designers can no longer give input about how the game should run.

1. Discuss with the group what verbs would be essential to play.
    - `Look` - Get a description of what the current situation of the room is.
    - `Help` - Get a list of commands available to use from the computer.
    - Any other verbs needed to interact with the objects around the room such as `Take` or `Use`
2. Everyone (including the computer and the player) creates a level.
3. Once the level designers hand the plans to the computer that player can start.
4. When the player has either finished the level or gotten stuck, reflect on how the level went. Some example questions include:

  - What worked well about the level?
  - What was confusing?
  - Was the level challenging?
  - What could be changed about the level design to make it easier to understand?
5. Change roles and try the exercise again.

Questions

Ask students questions to get them thinking about how to create this game on the computer.

1. What is the information the computer needs to run the game? (How does the computer keep track of the current room and score? How does the computer know what to do when the user inputs a command? How does the computer know how to set up the room?)
   - The computer needs to keep track of the state of the game
   - The game needs to use if/else to react to the user input
   - The room needs to be described in a way the computer can understand
2. What programming concepts will you need to be able to create the game?
   - variables (store temporary information / create rooms)
   - objects (store game state and room information)
   - complex conditionals (react to user input)
   - functions (writing)
   - debugging (work through problems)

Explain

How to create a Javascript object

Show students how to create an object on the computer. Do the example below with your group on repl.it. The code is all a continuation of one example.

1. Start with the keyword var and give it a name

   ```
   var room
   ```

   i. To create an object, use the curly braces

```
var room = {}
```

ii. For each room create a key and a value. The value can be a string, a number, or an object. Start with the name.

```
var room = {
name: 'Cellar'
}
```

iii. To add another key and value add a comma to the end of the previous one. Now add the completed property.

```
var room = {
name: 'Cellar',
completed: false
}
```

iv. Now students add the description.

```
var room = {
name: 'Cellar',
completed: false,
description: 'You are in a dark room and all you can see is a light switch on th
e wall next to you.'
}
```

v. To access information students use the name of the variable followed by a `.` followed by the key. For example to see if the room is completed use `room.completed`

2. Ask students, "How would you get the name of the room"
   - `room.name`
3. Ask students, "How would you get the description of the room"
   - `room.description`

Engage

Check for understanding

- Give students ten minutes at the end of the lesson to complete this challenge on codewars. http://bit.ly/1nqRoD5

The main function

Learning Goals

Students learn how to use the main function to take the user input and generate a new view.

Overview

This lesson starts with students revisiting the explore from the first day. Students then review what they learned about creating objects and getting/setting properties on those objects. After the review, introduce students to cycle-shell. Students will use this new tool to create the first level of their game.

Explore

Revisit the human version of the game

1. Show students the example escape the room level.
2. Ask students what kind of data type is used to create the room in Javascript.
    - Map out the room as an object. Students write down this object in the top section of their handout.
3. Ask students, 'How does the computer (instructor) know when to respond to the user?'
    - The computer waits for the user to input instructions
4. After each command, ask students the following question and chart students responses.
    - What message should the player receive?
5. Continue until the level is completed.

Questions

1. What Javascript data type would be good to keep track of the game information?
    - An object can keep track of all of the information that is stored about the game.
2. At what point in the game is the message changed?

- The game gets updated whenever the user inputs a command.

3. How did we decide which part of the state should be updated?
    - We check what command the user inputs, and then change the state of the game accordingly.

Explain

Review

1. Ask students, "How do you create an object?"
    - An object is create by using the curly braces.
2. Tell students to create the room object for their first level. This room should include the following properties:
    - a description
    - items: an object that contains the items in the room
        - description: a description of the item
        - takeable: true or false on whether or not the user can add item to inventory.

```
    var rooms = {
      basement: {
        description: 'The basement is dark. You can just manage to see a couch and a brief
case.',
        items: {
          couch: {
            description: 'big and comfy',
            takeable: false
          },
          briefcase: {
            description: 'old and rickety',
            takeable: true
          }
        }
      },
      kitchen: {
        description: 'Messy and stinky.',
        items: {
          knife: {
            description: 'sharp and shiny',
            takeable: true
          },
          oven: {
            description: 'old and dirty',
            takeable: false
          }
        }
      }
    }
```

1. Ask students, "How would you get value of the description of exampleRoom from this object?"

   ```
   rooms.exampleRoom.description
   ```

Using cycle-shell

The main function

1. Ask students, "Should the game act the same way for every input?"
   - No. The game should react to the user input.
2. Going back to the worksheet from the explore phase, student complete the second and third parts on the handout.
3. Introduce the main function on requirebin + cycle-shell
4. The main function takes the arguments input by the user and returns the next view.

Returns [number, string, array, object]

5. Each word in the input is split up and sent to the main function as arguments. For example, if the user inputs `hello cycle` the main function receives two arguments: `hello` and `cycle` .

```
function main (word1, word2) {
   console.log(word1) // hello
   console.log(word2) // cycle
}
```

   i. Whatever gets returned from the function will be displayed.

   ii. Example: To build a simple app that echoes the user input

```
function main (message) {
   return message
}
```

1. To make the function react to the input students need to implement conditionals.

```
if (verb === 'help') {
   return help()
}
```

   i. To add a second command students use `else-if`

```
if (verb === 'help') {
return help()
} else if (verb === 'look') {
return look()
}
```

   ii. Sample main function

```
function main (verb, noun) {
if (verb === 'help') {
  return help()
} else if (verb === 'look') {
  return look()
}
}
```

Setting current room

1. Students must set the currentRoom to their starting room. This way, users can move through the game and keep up to date with which room they're in. The functions created can access necessary information using less code.

```
var currentRoom = rooms.livingRoom
```

Additional functions

1. The main function calls the functions look and help. Now, we need to create those functions.
2. The look function must return the description of the room user is currently in.

```
function look () {
   return currentRoom.description
}
```

i. Help function will return a message providing hints to user about the level.

```
function help () {
return `
To play the game, input a verb and press enter.
Look: Describes setting of room you are currently in.
Help: Provides hints to complete the level.
`
}
```

ii. Students may realize that there is no way to access the descriptions of their items. To do so, we will need to create another function call called inspect.

```
function inspect (noun) {
if (noun === 'couch') {
  return currentRoom.items.couch.description
} else if (noun === 'briefcase') {
  return currentRoom.items.briefcase.description
}
}
```

iii. To move user from one room to the next, we will need to create a move function setting the new room as the current room.

```
function move () {
currentRoom = rooms[currentRoom.next]
return currentRoom.description
}
```

iv. The main function must be updated to reflect these changes.

```
function main (verb, noun) {
if (verb === 'help') {
  return help()
} else if (verb === 'look') {
  return look()
} else if (verb === 'inspect') {
  return inspect(noun)
} else if (verb === 'move') {
  return move()
}
}
```

Overall code

```
var rooms = {
  kitchen: {
    description: 'You are in an abandoned kitchen. Brad Pitt is sitting on a couch. There
are two dark rooms located to your left. You see a knife on the counter.',
    items: {
      knife: {
        description: 'The knife is rusted and old. It is sharp and deadly.',
        takeable: true
      },
      couch: {
        description: 'Brad Pitt is sitting on the couch. The couch is old and covered in h
oles.',
        takeable: false
      }
    }
  },
  livingRoom: {
    description: 'describe your living room',
    items: {
      item1: {
        description: 'describe item',
        takeable: false
      },
      item2: {
        description: 'describe second item',
        takeable: true
      }
    }
  }
}

var currentRoom = rooms.kitchen

function main (verb, noun) {
   if (verb === 'help') {
     return help()
   } else if (verb === 'look') {
     return look()
   } else if (verb === 'inspect') {
     return inspect(noun)
   } else if (verb === 'move') {
     return move()
   }
}

function look () {
   return currentRoom.description
}

function help () {
   return `
   To play the game, input a verb and press enter.
   Look: Describes setting of room you are currently in.
   Help: Provides hints to complete the level.
   `
}

function inspect (noun) {
   if (noun === 'knife') {
     return currentRoom.items.knife.description
   } else if (noun === 'couch') {
     return currentRoom.items.couch.description
   }
}

function move () {
   currentRoom = rooms[currentRoom.next]
   return currentRoom.description
}
```

Engage

Project milestone

Implement the first room using cycle-shell

Adding new features

To add new features to the game, students should:

- complete 'Creating a Function' worksheet
- create function in their code
- add necessary information to main function
- add necessary information to help function

The update function

Overview

Learning Goals

By the end of the lesson students will be able to:

- Explain the process to debug syntax errors

Explore

Use pre-made example of code with bugs in it to practice debugging skills. Make sure to ask students about the process by which they debug the code.

1. Show students an example of a Javascript object with syntax errors.
2. Students have one minute to write down what the mistake is.
3. Ask students:
    - "What did you do to find the error?"
    - "What is the error?"
    - "How can you fix it?"
4. Distribute examples of properly written Javascript objects and conditionals.

Explain

The main function

1. Ask students, "Should the game act the same way for every input?"
    - No. The game should react to the user input.
2. Going back to the worksheet from the explore phase, student complete the second and third parts on the handout.
3. Introduce the main function on requirebin + cycle-shell
4. The main function takes the arguments inputted by the user returns the next view.

    Returns [number, string, array, object]

5. Each word in the input is split up and sent to the main function as arguments. For example, if the user inputs `hello cycle` the main function receives two arguments `hello` and `cycle`.

   ```
   function main (word1, word2) {
      console.log(word1) // hello
      console.log(word2) // cycle
   }
   ```

   i. Whatever gets returned from the function will be displayed.

   ii. Example: To build a simple app that echoes the user input

   ```
   function main (message) {
         return message
   }
   ```

1. To make the function react to the input students need to implement conditionals.

   ```
   if (verb === 'help') {
      return room.help
   }
   ```

   i. To add a second command students use `else-if`

   ```
   if (verb === 'help') {
   return room.help
   } else if (verb === 'look') {
   return room.description
   }
   ```

Engage

Students continue work on the first room of their game.

Project milestone

The user can inspect the items around the room.

Accessing properties

Learning Goals

By the end of the lesson, students will be able to:

- Access object keys by variable

Overview

Debugging Review

Debugging review with examples.

Discuss order: 1. Syntax error 2. Logic mistakes

Explore

Create a human version of an object and if/else chain.

1. Go through an example with if/else chain similar to the one students create in the previous lesson. Announce your steps out loud. 1. Check the input 2. Go to the if/else chain and check the noun 3. If it is not the noun proceed to the next until you find the proper noun 4. Once the proper one has been found walk over to the example object to find the description. 5. Bring the description to the user.

2. Ask students, "Can you think of a better way to find the description?"

    - If students are having trouble, point out that the object already has a list of the available items.
3. Go through the example again, this time only checking the object. 1. Check the input 2. Go to the object 3. If the input is there bring back the description

Questions

1. Why is the checking the object important?
2. As your game gets bigger how many items do you think will be in the game?
3. Is it easier to check the list or create an if/else chain to check each word?

Explain

1. Review what students currently know about accessing properties of objects
2. In the generic example show students the syntax for accessing a Javascript object using a variable in repl.it

```
items: {
   light: 'There is a dim light',
   chair: 'The chair is old and green.',
   closet: 'Inside the closet you see a shiny key.'
}
```

   i. Students practice accessing the properties on the right hand side of the repl
   ii. First students access using a string

```
items['light']
```

      i. Then students create a variable and access that way

```
var input = 'light'
items[input]
```

      ii. Then create a function to do the same

```
function getItem (input) {
return items[input]
}
getItem('chair')
```

      iii. Students practice adding an item to an object using this notation

```
function addToInventory (input) {
inventory[input] = items[inventory]
}
```

Engage

1. Students work on refactoring their code
2. Students work on adding an inventory to their game
3. Students work on implementing an original feature

Adding Levels

Learning Goals

Overview

Begin with students talking in small groups to discuss what we learned last class, then teach the two new debugging tactics (syntax then concepts & isolating lines of code), and then work on building new rooms and expect that tutors will only help with debugging procedures.

Explore

1. Review debugging procedures.
2. Practice debugging logic mistakes
3. Move through the human version of the game.
    - Make a point of what information the game current has to keep track of and keep a list of the variables on the whiteboard:
        - inventory
        - room
    - Every time the user inputs a command make sure to follow the main function and update the variable values
4. Move to a second room.
    - Ask students to speculate about how to keep track of which room the user is in.
        - Add a current room variable to keep track of which room the user is in.
        - Change all of the instances of room to currentRoom in the main function.

Explain

Walk through the steps of moving to a new room.

- Solicit ideas from students
- Add a next property to each of the rooms. Ex: `next: 'second'`
- Add a `else if (verb === 'move')` to add a command
- Set `currentRoom = room[currentRoom.next]`

```
var rooms = { // Change rooms to an object with multiple rooms
  first: {
    description: 'First room',
    next: 'second', // Add a next parameter that stores a string with the name of the next
 room
    items: {
      light: 'The light',
      chair: 'The chair'
    }
  },
  second: {
    description: 'Second room',
    next: 'first', // go back to the first room
    items: {
      sofa: 'The sofa',
      table: 'The table'
    }
  }
}

var currentRoom = rooms.first

function main (verb, noun) {
  // Same code as before but add
  if (verb === 'move') {
    currentRoom = rooms[currentRoom.next] // move to whichever room is listed as next for
the current room
    return currentRoom.description // print out the description of the new current room
  }
}
```

Questions

1. How are we keeping track of which room we are in?
2. How can you keep track of the next room in the object?
3. How can you use `currentRoom.next` to move to the next room?
4. How can you make sure that the student can't move to the next room until something else is completed?

Engage

Students work on creating their multi-room games and add the ability to move between levels

Teaching order

1. Build sequentially
2. Build with functions
3. Build with loops
4. Mine with arrays

Intro

1. Explain the goal of the class
    - Students are working to create a city, a theme park, and an automatic mining facility
    - In the city, each student has a plot of land that they can build their house on.
    - There is one catch: for 45 minutes per day, students will not be able to build directly. They need to use programmable turtle to do their building!
2. Introduce students to turtles.
    - Turtles are programmable Minecraft blocks that allow you to automate different Minecraft actions such as:
        - building
        - mining
        - crafting
        - and more!
    - Turtles are programmed in a language called **Lua**.
        - Lua is a scripting language like Javascript.
        - Lua is used in lots of games such as World of Warcraft and Angry Birds.
        - This means the concepts will be the same as your coding classes but the syntax, or how it is written, will be different.
    - To call functions in Lua simple write the name of the function and the add parentheses (just like Javascript)
3. Students login to Minecraft.
    - Show students their plot of land.
    - Explain that during the first 45 minutes of the lesson the only way that they will be able to build is by using the turtles.
    - For the second 35 minutes of the lesson students will be able to build as well as use their turtles to help out.
    - The expectation is that students will work on their own plot of land. Destruction of anything that does not belong to them will not be tolerated. Turtles can not leave their plot of land to protect against accidents.
    - Distribute the turtles to each student.
4. Show students how to interact with a turtle.
    - You need a remote to control the turtle. Once you have a remote right-click on

the turtle you want to program.
    - You can use the remote to control the robot directly.
    - Show students the coding interface.
    - After adding code, students need to push the run button to make the robot run the code.
    - The undo button will reset the turtle back to where it started and undo everything from the last run.
5. Students test out turtles using the examples.
    - Students have 5 minutes to try out all of the example methods in the API. For example:
        - `turtle.forward()`
        - `turtle.back()`
        - `turtle.turnLeft()`
        - `turtle.digDown()`

1. Show students their movement challenges for the first day. When students finish a challenge, they raise their hand and have a tutor check their work to make sure that it functions correctly.

Ideas

introduction

- introduce concepts
- predict novel code

debugging

- make turtle for predicting and debugging
- when predicting act out turtle movements using physical representation or body
- running turtle in first person viewpoint

presentation

- Present features out of order
- Add thumbnails
- Concepts needed for feature

when building

- loop of
  - code
  - predict
  - observe
  - explain (when tutor is available)
  - loop to start
- share out findings with build partner
- partner explains code back (or to tutor)

end of day sharing

- 2 or 3 students
- look at code

class timing

- code (intro new concept)
- turtle build
- student build
    - r & d
    - use turtle remote control to research how to build their specific challenge
    - take notes on building process
- turtle build w/ new process

Roles

- student
- builder
- teacher

building with turtles

Learning Goals

By the end of the lesson, students will be able to loop through:

- Produce and revise code
- Predict the outcome of code
- Test code

Classroom norms

1. Discuss the different roles students adopt during each lesson:
    - Student mindset: Students learn a new concept
    - Builder mindset: Students use their knowledge to work on creating something new
    - Teacher mindset: Students teach their peers and their tutors by explaining how their creation works.
2. Whenever students are creating a new program they are expected to predict how that code works by using their paper cutout **before** executing the code in game.
3. When the students are frozen in Minecraft, they are expected to:
    - tilt computer screens down to a 45 degree angle
    - eyes to the front of the room prepared to listen to the teacher

Follow up questions

Use these questions to help students elaborate when explaining their code:

- "Can you say more about that?"
- "What do you mean by that?"
- "How can you take that idea a step further?"

---

student mindset - 20 minutes

introduce concept - 10 minutes

Introduce students to turtles

- Turtles are programmable Minecraft blocks that allow you to automate different Minecraft actions such as:
    - building
    - mining
    - crafting
    - and more!
- Turtles are programmed in a language called **Lua**.
    - Lua is a scripting language like Javascript.
    - Lua is used in lots of games such as World of Warcraft and Angry Birds.
    - This means the concepts will be the same as your coding classes but the syntax, or how it is written, will be different.
- To call functions in Lua write the name of the function and add parentheses (just like Javascript)
- Students still control their character, but instead of building directly, students program the turtle to build for them

predict novel code - 5 minutes

1. Students create their turtle cutout.
2. Show students the first code example. Students look at the code and use their cutout to predict what the code will do.
3. If everyone picks the same solution, ask for students to come up with alternative approaches (even if they think it is wrong).
4. After students finish their predictions, run the code on the projector.

```
turtle.forward()
turtle.turnLeft()
turtle.forward()
turtle.forward()
```

    i. Discuss with students what happened and why.

Teacher notes

- Play up how believable all student ideas are.
- Collect all student ideas before showing the correct answer.

- After running the code, have students assess their prediction.
    - did it match?
    - what was different?

---

builder mindset - 40 minutes

turtle build - 20 minutes

1. Students join the Minecraft game a receive a turtle and turtle remote.
2. Show students how to interact with a turtle.
    - You need a remote to control the turtle. Once you have a remote right-click on the turtle you want to program.
    - You can use the remote to control the robot directly.
    - Show students the coding interface.
    - After adding code, students need to push the run button to make the robot run the code.
    - The undo button will reset the turtle back to where it started and undo everything from the last run.
3. Students test out turtles using the move methods.
    - The move methods are:
        - `turtle.forward()`
        - `turtle.back()`
        - `turtle.up()`
        - `turtle.down()`
        - `turtle.turnRight()`
        - `turtle.turnLeft()`
    - Students create a first program using the move methods. It should have at least four lines. For example:
        - `turtle.forward()`
        - `turtle.back()`
        - `turtle.turnLeft()`
        - `turtle.up()`
    - When they are done, students get with a partner and predict the code that their partner wrote using their cutouts.
4. Introduce digging

  - The dig methods are:
    - `turtle.dig()` - digs directly in front of the turtle
    - `turtle.digUp()` - digs one space above the turtle
    - `turtle.digDown()` - digs one space below the turtle
  - Students write a program that digs a capital letter into the ground using the movement and dig methods.
  - When they are done, students get with a partner and predict the code that their partner wrote using their cutouts.
5. Introduce placing
  - The place methods are:
    - `turtle.place()` - place a block directly in front of the turtle
    - `turtle.placeUp()` - place a block one space above the turtle
    - `turtle.placeDown()` - place a block one space below the turtle
  - Students edit their dig program to place a block in each hole that robot digs for the letter

plan with teammate - 5 minutes

1. Students get a partner or a group (2-3).
2. Each student in the group picks one building challenge. At the end of the day, each person in the group explains their code to the group.
   - Successful teaching means that each of the members of the group can explain the code back to the group (or hopefully a tutor)

turtle build with new process - 15 minutes

Using their notes from the research time, student continue to work on their building challenge. When students are building, the process should be a loop of:

- code - create or revise the program.
- predict - predict the outcome of the code using the physical representations.
- observe - run the code in game and take notes on what happens using the worksheet.
- loop to start

Teacher notes

As students are working, rotate between students and discuss their code.

- If it did not work:
  - Why did it not work?
  - What line do you think the problem is on?
  - What kind of mistake is it?
    - syntax
    - logic
  - What can you change to fix it?
- If it did work:
  - Why does the code work now?
  - What was the most difficult part of this coding challenge?
  - What challenge are you going to work on next?

---

teacher mindset - 10 minutes

Students show their partner the code that they have created during the lesson. The goal is to teach the code well enough that the partner can then turn around and explain the code to someone else (tutor).

Each student should: 1. Show code to partner 2. Allow the partner to predict what the turtle will do 3. Clarify their partner's understanding of the code

---

Teacher notes

Ask students to explain your partner's idea in your own words.

---

loops

Overview

student mindset - 20 minutes

introduce concept - 10 minutes

Introduce loops

predict novel code - 5 minutes

Example of loop code

plan with teammate - 5 minutes

With teammate, plan which challenges to undertake

engineer mindset - 10 minutes

Use turtles to build item from building challenges

student build - 10 minutes

With building enabled, students create one of the building challenges. While building, students take notes on the procedures they used for their creation, and attempt to map that procedure list to code that the turtle can understand.

turtle build with new process - 30 minutes

Using their notes from the research time, student create their

teacher mindset - 10 minutes

share with partner - 5 minutes

Students show their partner the code that they have created during the lesson. The goal is to teach the code well enough that the partner can then turn around and explain the code to someone else.

share to group - 5 minutes

2-3 students per day share their code with the entire group.

Lua Examples

Move functions

~~~
```

#### To create a function that digs 3 times.
```js
function forward3 () -- create a function named forward3
  turtle.forward() -- first forward
  turtle.forward() -- second forward
  turtle.forward() -- third forward
end -- end function

forward3() -- call function
~~~

For loops are used to repeat an action a specific number of times. In the example below, the turtle moves forward 10 times.

```
for i=1, 10 do -- Repeat 10 times
  turtle.forward() -- Move forward
  turtle.placeDown() -- Place the currently selected block below the turtle
end -- end loop
```

Nested for loops (a loop inside of a loop) are useful for building shapes. In the example below, the turtle moves in a square.

```
for i=1, 4 do -- Repeat 4 times
  for i=1, 5 do -- Repeat 5 times
    turtle.forward() -- Move forward
  end -- end inner for loop
  turtle.turnRight() -- After moving forward 5 times turn right
end -- end outer for loop
```

If statements are used to check a condition. The turtles has a few methods that can be used to detect what is around the turtle.

- `turtle.detect()` -- Check to see if there is a block directly in front of the turtle. Returns true or false.
- `turtle.detectUp()` -- Check to see if there is a block directly above the turtle. Returns true or false.
- `turtle.detectDown()` -- Check to see if there is a block directly below the turtle. Returns true or false.

```
if turtle.detect() == true then -- If the turtle detects a block in front
  turtle.up() -- move up
else -- otherwise
  turtle.forward() -- move forward
end
```

Arrays are lists. They can be used to keep track of a list of actions (such as what movements your turtle has done.

```
local moves = {} -- Create an empty list to keep track of the moves

for i=1, 10 do -- loop 10 times.. each time the loop is executed the value of x increases
by 1
  turtle.forward() -- move turtle forward
  moves[i] = 'forward' -- add the string 'forward' to the move array at the index of i (wh
ich increases each time the loop is run)
end
```

Turtle challenges

Building challenges

Each building should be create as a separate file so that it can be used as a function.

rectangular shapes

- Tower (5)
- Tower (arbitrary)
- Waterfall (lavafall)
- Square (2x)
- Square (arbitrary size)
- Rectangle (2 x 3)
- Rectangle (function with width and height)
- Filled in square
- Filled in rectangle
- Cube (3x3x3)
- Cube (arbitrary)
- Rectangular prism (3x4x4)
- Rectangular prism (arbitrary)
- 4 Legged table with legs in the corner (2 x 3 x 3)
- 4 Legged table with legs in the corner (arbitrary sizes)
- 4 legged chair with back (table 2 x 3 x 3) (backrest height 3)
- Couch (arbitrary chair)
- Bridge

triangular shapes

- Staircase
- Spiral staircase
- Pyramid
- Triangle
- Triangular prism
- Waterslide

conditionals

- inventory
- switch materials
- switch to next available building block

house

- Hut
- Farm
- House
- Castle
- Pool
- Mote
- Window
- Roof

misc

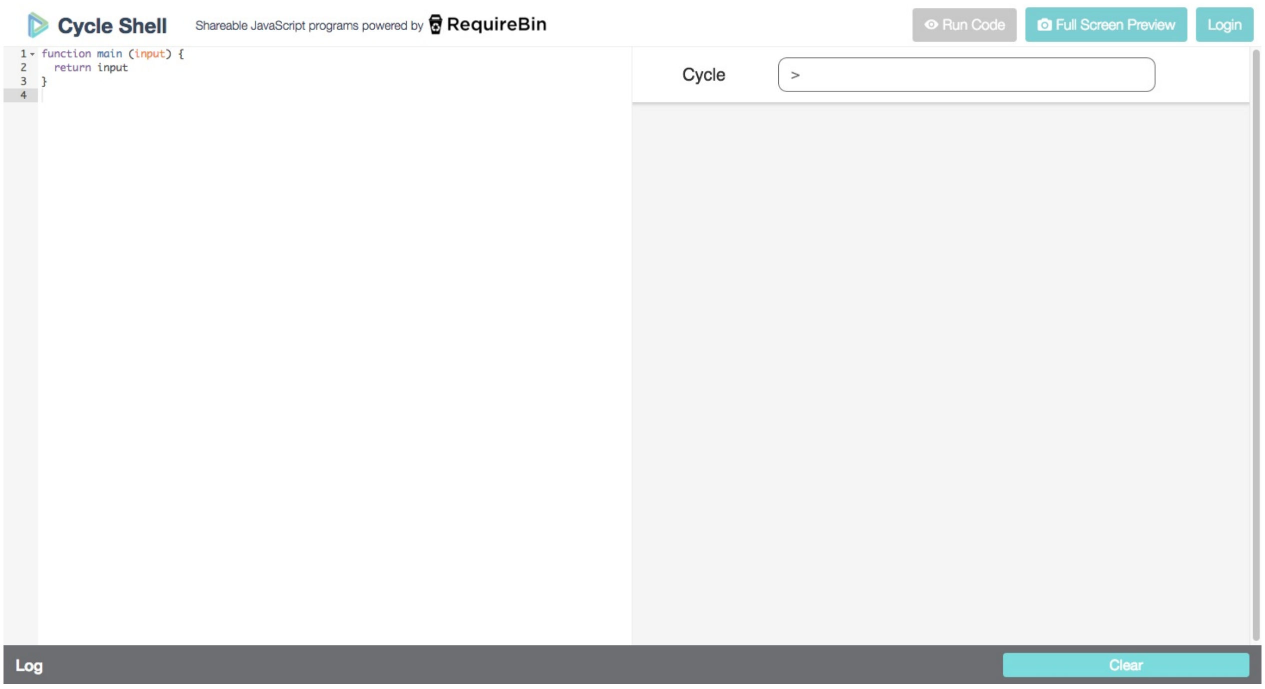
- Animal statue
- Block art
- Create a letter
- Create a number

student actions	teacher actions
asdf	asdf

Cycle shell

Using cycle shell students can interact with their computer in a terminal like environment. This allows students to build meaningful web applications with only and introductory knowledge of Javascript.

Authentication: Github



```
function main (input) {
  return input
}
```

Layout

Preview

The right side of the screen is an interactive preview of the program you created. You can also use the `Full Screen Preview` button to open a new tab that show only your program (no code editor).

Editor

The code editor is located on the left hand side of the website. This is where you create your program.

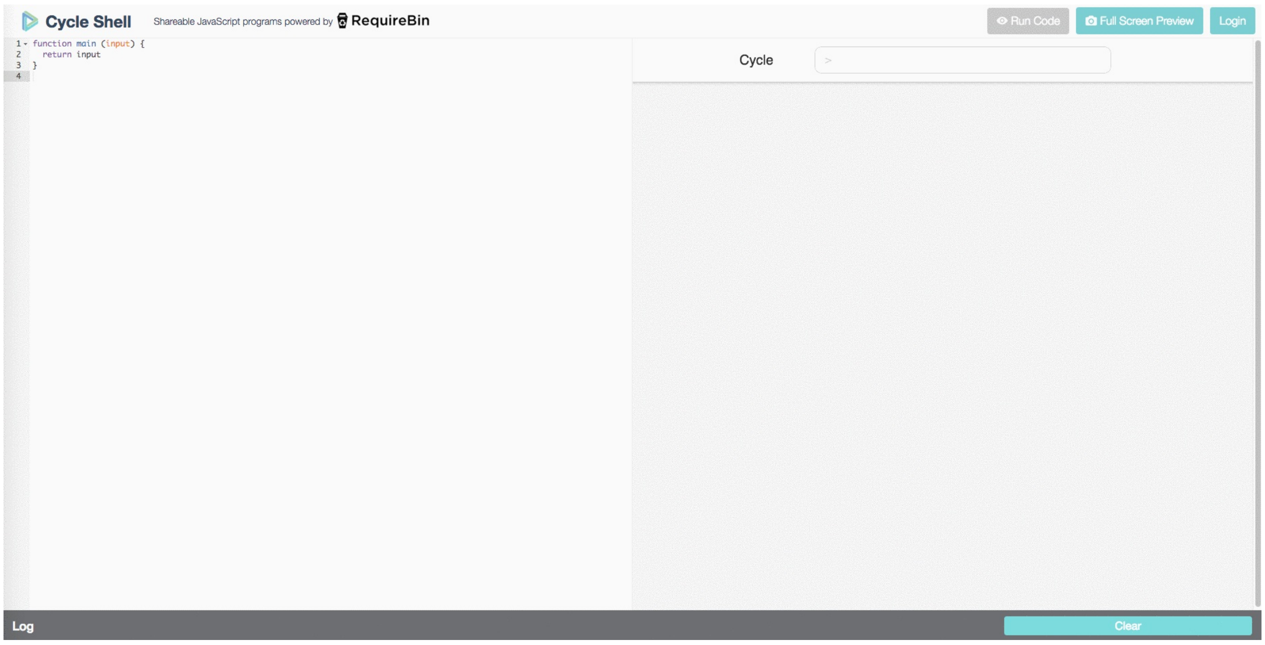
Log

At the bottom of the screen is a log that displays error and console messages. It alerts about a new message by adding a number next to the log indicating the number of unread messages.

How it works

The `main` function

The function `main` is required to make cycle-shell work. The function is called whenever the input box on the right side of the screen submits. The input is parsed (split at every space) and passed to the `main` function as parameters. The return value of the `main` function is outputted as a card element.



```
function main (input) {
  return input
}
```

Example:

```
input: hello
```

```
function main (input1) { //input1 = 'hello'
  return input1 // returns 'hello'
}
```

```
input: hello daniel
```

```
function main (input1, input2) { // input1 = 'hello' input2 = 'daniel'
  return input2 // returns 'daniel'
}
```

Running the code

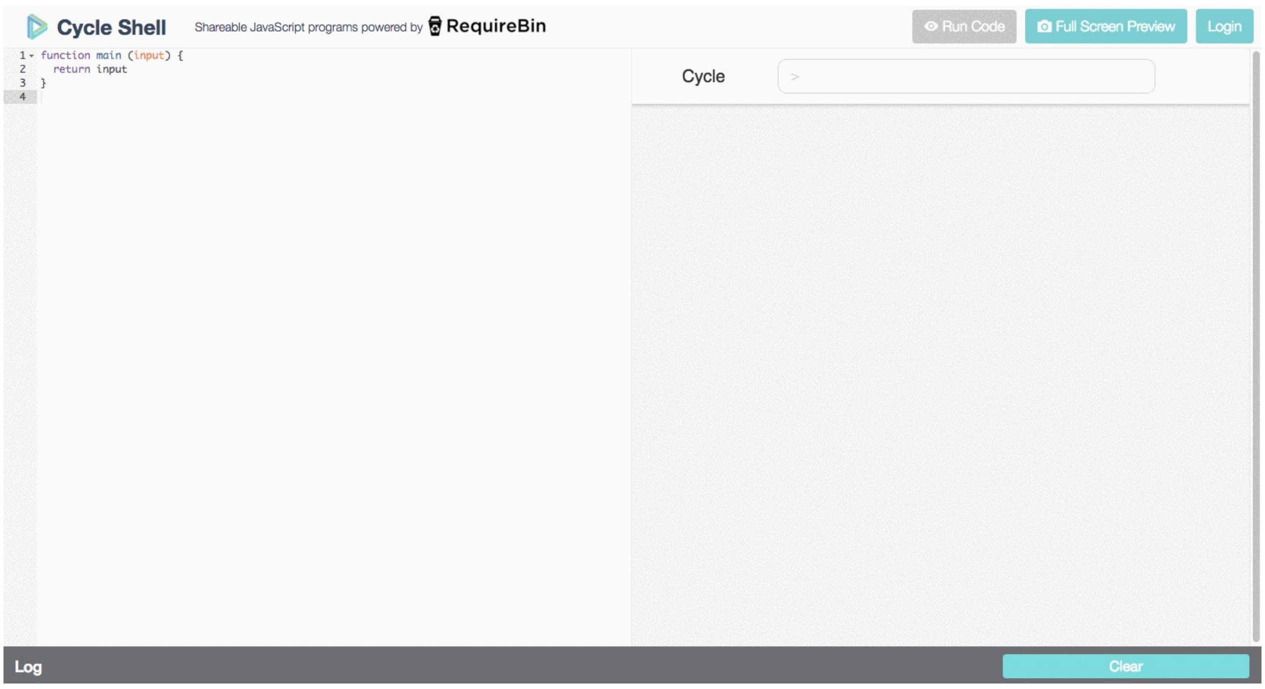
Anytime the code is altered, you must use the `Run Code` button to compile the new program. You can use the standard keyboard shortcut on your operating system to save.

Mac: `cmd + s`

Windows: `ctrl + s`

Errors

The code editor features live syntax error checking. Additional errors and console messages are printed in the log at the bottom of the screen.

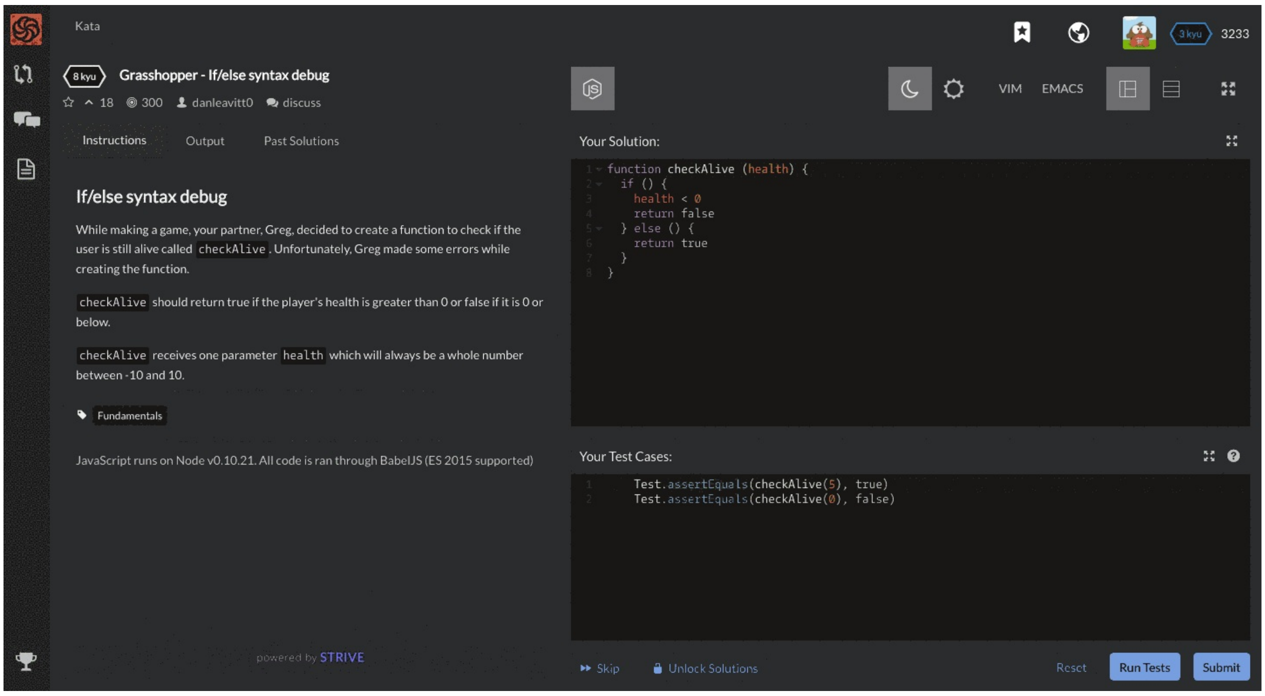


```
function main (input) {
  return input
}
```

Codewars

Codewars is an online platform where users solve real coding challenges. This curriculum utilizes codewars to reinforce concepts when students break out into coding stations.

Authentication: Github



Grasshopper - If/else syntax debug

18 300 danleavitt0 discuss

If/else syntax debug

While making a game, your partner, Greg, decided to create a function to check if the user is still alive called `checkAlive`. Unfortunately, Greg made some errors while creating the function.

`checkAlive` should return true if the player's health is greater than 0 or false if it is 0 or below.

`checkAlive` receives one parameter `health` which will always be a whole number between -10 and 10.

Fundamentals

JavaScript runs on Node v0.10.21. All code is ran through BabelJS (ES 2015 supported)

Your Solution:

```
function checkAlive (health) {
  if () {
    health < 0
    return false
  } else () {
    return true
  }
}
```

Your Test Cases:

```
    Test.assertEquals(checkAlive(5), true)
    Test.assertEquals(checkAlive(0), false)
```

Layout

Challenge

On the left half of the page there are instructions that describe the challenge the user needs to solve. This is also where feedback from tests is given.

Editor

The top right portion of the screen is the code editor. This is where the user writes code to complete the challenge.

Test cases

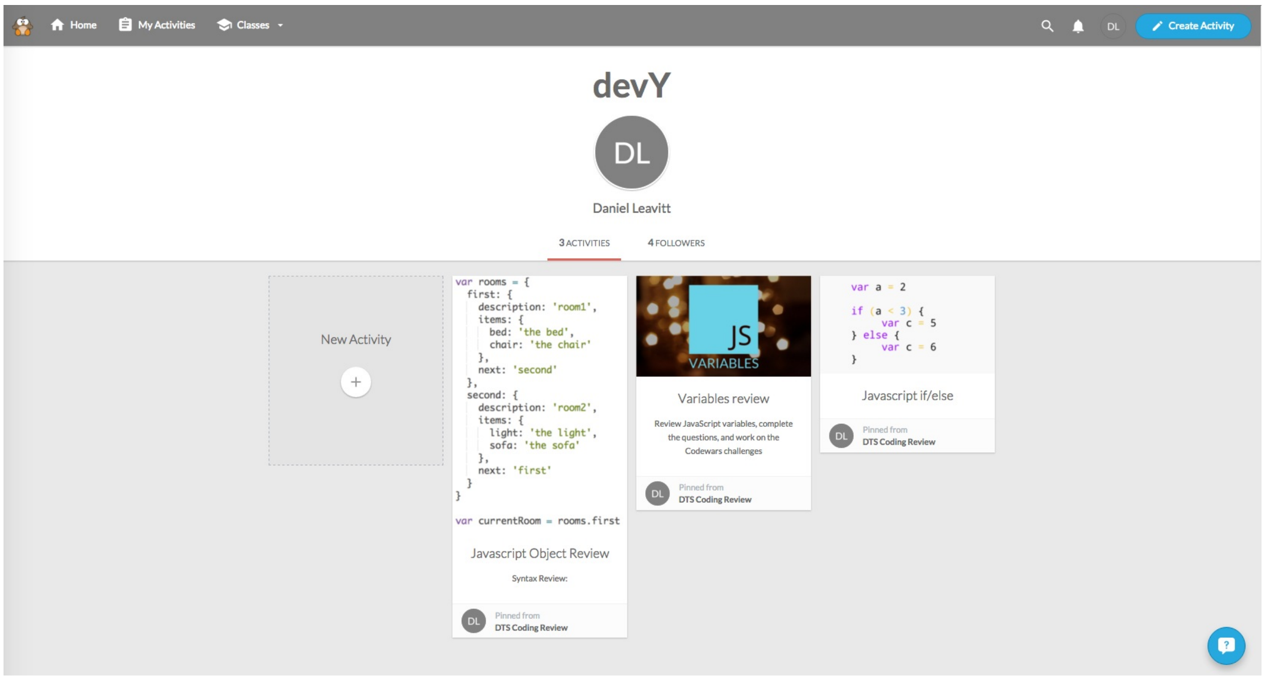
The bottom right area shows the example test cases. These are provided by the author of the exercise and ensure that the users code sufficiently solves the problem.

Weo

Weo is an educational platform that makes it easy to create and grade assignments. The devY curriculum employs this to create quizzes and short review exercises.

Authentication:

- For Teachers: Google, Facebook, or E-mail
- For Students: No email required



```
var rooms = {
  first: {
    description: 'room1',
    items: {
      bed: 'the bed',
      chair: 'the chair'
    },
    next: 'second'
  },
  second: {
    description: 'room2',
    items: {
      light: 'the light',
      sofa: 'the sofa'
    },
    next: 'first'
  }
}

var currentRoom = rooms.first
```

Javascript Object Review

Syntax Review:

Pinned from DTS Coding Review

Variables review

Review JavaScript variables, complete the questions, and work on the Codewars challenges

Pinned from DTS Coding Review

```
var a = 2

if (a < 3) {
    var c = 5
} else {
    var c = 6
}
```

Javascript if/else

Pinned from DTS Coding Review

Getting Started

Check out Weo's how it works page for information on how to get started with your account.

1. Create an account
2. Create a class
3. Add students to class
4. Follow the devY board
5. Assign the relevant assignment to your class

Escape the room

Goal

The goal for the project is build a text-based adventure games. There's no way for the game to include every possible feature so it is important to focus on building a stable game with the core features and then add the additional features one at a time. It is important to test the functionality of your game after each feature is added to ensure that it did not affect the rest of your game.

Core features

These features will be implemented with help from your tutors while learning the new coding concepts.

- [ ] **Three rooms with descriptions and items**
  - Your game should have three rooms. Each room should have a description and items.
  - *Skills learned:*
    - Creating objects
    - Nested objects (objects inside of objects)
    - Strings
- [ ] **Help**
  - The help command should return a list of the available commands of the game.
  - *Skills learned:*
    - Creating functions
    - Conditionals (if\/else)
    - Returning values from a function
    - Strings
- [ ] **Look**
  - When the user types look, the game should display the description of the current room.
  - *Skills learned:*
    - Returning a variable
    - Conditionals (if\/else)
    - Accessing object properties (dot notation)
- [ ] **Inspect items**
  - Inspecting an item should return the description of the item if it exists or return a message explaining that the item does not exist.
  - *Skills learned:*
    - Returning a variable
    - Conditionals (if\/else)
    - Accessing object properties (bracket notation)
- [ ] **Error**
  - If the verb is not recognized it should return a message explaining that it does not know how to perform that action.

  - *Skills learned:*
    - Returning a string
    - Template string
    - Conditionals (if\/else)

- [ ] **Move**
  - The user should be able to use the ‘move’ command to move to the next room.
  - *Skills learned:*
    - Set a new value in an existing variable
    - Accessing object properties (bracket notation)
    - Conditionals (if\/else)
    - Return a variable

Additional features

These features will be implemented independently once the core features of your game are completed. Each feature should be completely implemented and tested before moving on to another of the additional features.

- [ ] **Previous room**
    - The user can move back to a previous room by using the ‘back’ command.
    - *Skills required:*
        - Set a new value in an existing variable
        - Accessing object properties (bracket notation)
        - Conditionals (if\/else)
        - Return a variable
- [ ] **Take items**
    - The user should be able to take an item and place it in their inventory. If the item does not exist the game should provide a relevant error message.
    - *Skills required:*
        - Set a new value in an existing variable
        - Setting object properties (bracket notation)
        - Accessing object properties (bracket notation)
        - Conditionals (if\/else)
        - Return a string
- [ ] **Hidden items**
    - Add items to the game that are hidden at the beginning and that are only revealed when one of the other items in the room is inspected.
    - *Skills required:*
        - Setting object properties (bracket notation)
        - Accessing object properties (bracket notation)
        - Conditionals (if\/else)
- [ ] **View inventory**
    - The user should be able to use the ‘inventory’ command to look at what is currently being held in their inventory.
    - *Skills required:*
        - Return an object
- [ ] **Use Items**
    - The user should be able to use the item if it exists in their inventory.

- *Skills required:*
    - Accessing object properties (bracket notation)
    - Conditionals (if\/else)
    - Return a string

- [ ] **Teleport**
    - In certain rooms, the user should be able to use the teleport command to move to another room that is not adjacent to it.
    - *Skills required:*
        - Set a new value in an existing variable
        - Accessing object properties (bracket notation)
        - Conditionals (if\/else)
        - Return a string

- [ ] **Add images (HTML)**
    - Error icon
        - The game shows an error icon image when an illegal command is issued by the user
    - Item images
        - A relevant image is displayed when items are inspected
    - Room images
        - A relevant image is displayed when the user performs a ‘look’ command
    - *Skills required:*
        - Template strings
        - HTML - image tag

- [ ] **Add styles (HTML)**
    - Add custom styling to various messages using HTML and a style object
    - *Skills required:*
        - Template strings - embed variable
        - HTML - image tag
        - CSS styles
        - Create object

- [ ] **Check health**
    - The user should be able to check the status of their health.
    - *Skills required:*

        - Return a string
        - String concatenation

- [ ] **Combat**
    - Create a combat system for battling monsters in the game. This should include keeping track of the player’s health, the monster’s health, and a way to initiate and finish combat.
    - *Skills required:*
        - Variable numbers
        - Arithmetic
        - Passing parameters
        - Variables - set a new value
        - Return string