Making Decisions 1

Intro

Overview

- Conditional Logic
- Loops
- Pair Programming
- GitHub

Conditional Logic

Logical Operators

If-statements, a common form of conditional logic, execute code based on if a certain condition is met. This is determined by the use of logical operators.

The most common logical operators are:

- `>` greater than
- `>=` greater than or equal to
- `<` less than
- `<=` less than or equal to
- `===` equal to
- `!==` not equal to

When we need to check if multiple conditions are true, we can use the `&&` (and) operator.

When we need to check if just one of multiple conditions are true, we can use the `||` (or) operator.

If-statements

The condition for an if-statement is contained within parentheses `()` while the code to be executed is contained in curly brackets `{}`. An example of an if-statement could be:

```
if (2 === 2) {
  console.log('Math still works')
}
```

- If 2 is equal to 2, when we execute this file, the string `'Math still works'` will be printed to the console. However, if we write the following if-statement:

```
if (2 === 2 && 2 === 4) {
  console.log('Math is broken')
}
```

- Because 2 will never be equal to both 2 & 4, we will never see this console log.

If-else statements

You can also provide `else` and `else-if` statements to an if-statement to handle exceptions. To build on our previous examples we could say:

```
if (2 === 2 && 2 === 4) {
  console.log('Math is broken')
} else {
  console.log('Math still works')
}
```

- In this case we would see the second console log.

if, else, else if

Else-if statements are useful when comparing many conditions and are structured identically to regular if-statements. For example:

```
var num = 50

if (num >= 0 && num < 25) {
  console.log('Greater than zero, less than 25')
} else if (num >= 25 && num < 100) {
  console.log('Greater than 25, less than 100')
} else {
  console.log('Greater than 100')
}
```

- Depending on the value assigned to the num variable, we will see a different console log.

Loops

For-loops

For-loops are statements that allow us to repeat a given block of code until a given condition fails. A for-loop is structured like this:

```
for (let i = 0; i < 5; i++) {
  //Code to execute goes here
}
```

Let's break that down a little bit. When we initialize a for-loop, we need to provide it 3 arguments separated by semicolons (;)

1. An iterator, usually represented by i. This is used to track how many times the for-loop has run.
2. A condition. As long as this condition evaluates to true, the cycle will continue. As soon as it fails, the for-loop will stop.
3. What we want to have happen at the completion of each loop. Most commonly we will just increment our iterator by saying i++.

Let's revisit the above example:

```
for (let i = 0; i < 5; i++) {
  console.log(i)
}
```

- What this loop will end up doing is console logging 0 1 2 3 4 because as long as i is less than 5, we will console log i, incrementing it by 1 after each loop. - - - This is obviously a very simple example, but for loops can be incredibly useful.

While-Loops

Another popular loop is a `while-loop`. While loops are a little more simplistic in the way they are built, but can also be very effective. Like for-loops, however, they can be dangerous to use if not written properly. Here is an example of while-loop:

```
let x = 10

while (x < 50) {
  console.log("x is small.")
}
```

In this while loop, our code will continue to log “x is small” to the console, until x is no longer less than 50. Of course, this is not great, because at no point in time is x being modified. Let’s fix this on the next slide.

```
let x = 10

while (x < 50) {
  console.log("x is small.")
  x += 1
}
```

In this example, each time the loops runs, x will increment 1, helping us guarantee that we will not get stuck in an infinite loop. In this example, we will log “x is small” over and over again until x is equal to 50. At that point, the loop will stop executing.

Pair Programming

What is Pair Programming?

Software developers often work using a strategy referred to as “Paired Programming.” So what is paired programming? Paired programming is the practice of coding with a partner developer, one person typing at a time. The most dedicated paired programmers use only one computer at a time, with one developer “driving”(typing) and the other developer observing.

The two developers openly and frequently communicate to determine what code needs to be written and/or edited. But doesn’t this practice just double the required time needed to write code? In the worst case scenario, yes, paired programming will only serve to make the experience longer. However, studies have indicated that it is more likely labor costs will decrease when paired programming is employed. After all, two minds are greater than one. With this approach, developers are actually able to code more effectively. Additionally, it is more likely that bugs and errors will be caught in the initial coding phases. This is much preferable to finding out that there is a problem with the code after the application has been made available to the public.

The General Idea

Without pair programming:

- You have to come up with an algorithm to solve problem
- *And* the syntax to express that algorithm at the same time

With pair programming:

- One person comes up with the algorithm
- The other person produces the syntax

Pair Programming in the Industry

Professional developers work with *huge* systems

- The code they write can have far-reaching side effects
  - It's impossible for one person to juggle all that in their head
- Also encourages knowledge sharing
- Studies show that it reduces bugs and increases code quality

Pair Programming at Devmountain

Learning a new language is hard!

- Pair programming helps you focus on one thing at a time
  - You're either thinking about the steps you'll need to follow to solve a problem
  - Or translating those steps into syntax
- It helps you retain knowledge and validate understanding by giving you more ways to engage with code
- Also, having a buddy helps you get unstuck — two heads are better than one!
- It'll force you to come up with a plan *before* you type

Also, pair programming requires you to communicate technical ideas effectively

- Communicating technical ideas — explaining and justifying your code — is something you'll be doing a *lot*
  - During technical interviews
  - On the job, working on a team of developers
- It's a difficult skill to master, but practice makes perfect!

The Driver & Navigator

This is a common framework for pairing

One person is the ***driver*** and the other person is the ***navigator***

- Navigators checks the map
- Drivers steer the ship

Driver

- Translates navigator's idea into code
- Focuses on writing correct syntax
- Asks clarifying questions

Navigator

- Provides plan of action
- Tells the driver what to write (at a high level)
- Keeps an eye out for any blindspots, pitfalls

Switch often! (~5 minutes, 2-3 lines of code)

General Guidelines

- Read the problem out loud
- Ask each other questions
- Decide on a plan together
- Silence is **deadly**

Helpful Tips

- Take breaks!
  - At the same time
  - More than you might usually
- Set ground rules for when to get help
  - If you are stuck for 20 minutes, reach out in the queue channel
  - After trying this one thing?
- Let your pair know what you want to do before you do it
  - Propose solutions
  - “Can we…?”
- Reflect on pairing process together
- Speak up for yourself
- Step Up / Step Back
- Be Gentle

What Can Go Wrong?

Pairing **will** feel weird because most of you have never done it before

It’s important to acknowledge that it can go wrong at first and the ways that might happen

- Your pair doesn’t explain what they’re doing before they do it
- You feel lost but it seems like your pair isn’t lost at all
- You’re worried that you’re being bossy and taking over a lot
- Only one person wants to ask for help

If You're Slower than Your Pair

- This is a great opportunity to learn from a peer!
- Make sure you're switching roles often
- Don't be afraid to ask for what you need

If You're Faster than Your Pair

- This is a great opportunity to solidify your knowledge!
- Explaining things to others helps deepen your own understanding
- Make sure you're switching roles often

If You Tend to be Quiet

- You're here to learn and understand
- Getting the most out of exercises is important
- Your ideas are valuable, and will help your pair
- Make sure you're switching roles often
- Remember — this is a safe place to practice speaking up
    - That way, you're prepared to do it in the workplace

If You Tend to be Assertive

- Make space for your pair to participate
- Check in to make sure they understand what's happening
- Make sure you're switching roles often
- Look for non-verbal cues
- Remember — this is a safe place to practice stepping up and stepping back

- That way, you're prepared to do it in the workplace

All types of folks should…

- Learn the exercise materials
- Learn **a lot** about collaborating on code
- Deepen communication skills

These are all key job skills!

Try - Before Pairing

Set the Scene for Collaboration!

- "What's your pairing style?"
- "How often would you like to switch roles?"
- "Can you let me know if you want to slow down/speed up?"
- "Do you feel comfortable speaking up? If not, is there a sign I should look for?"

Try - During Pairing

Keep Communicating!

- "Can we test that in the console? That might be easier to understand"
- "Can we walk through the pseudo-code again?"
- "Can we slow down? I would like to read more about this first"
- "I didn't get that. Could you explain it?"
- "Can we switch roles after each function/5 minutes?"
- "How's this going? Do you want to take a break?"

Try - After Pairing

Process and Improve!

- “Thanks for pairing!”
- “How was that for you?”
- “Could I have done anything better?”
- “I liked the way you …”
- “I would have liked more …”

Enjoy Pairing!

You’ll practice pairing in lab today!

GitHub

What is GitHub?

We have talked about git, but what is GitHub? GitHub?

GitHub is a website. It allows you to take your git repositories and upload/download them. It is essentially a cloud service for your git repositories.

Git commands for GitHub use

- `git clone REPO_URL`: copy a repo from GitHub and put it on my machine
- `git remote -v`: show names of repos on GitHub that I can push to
- `git push REPO_NAME`: put my local commits on GitHub

Collaborating on GitHub

- Most people use GitHub to collaborate across one coding project
- For now, we'll just use it on individual code projects
- This will help you learn the basics of git/GitHub before you begin collaborating

The End