CSE 390 B Spring 2021

OS, Debugging, Project 7 Tips, Stress & Wellness

Debugging, Tips for Project 7, Stress & Wellness

Significant material adapted from www.nand2tetris.org. © Noam Nisan and Shimon Schocken.

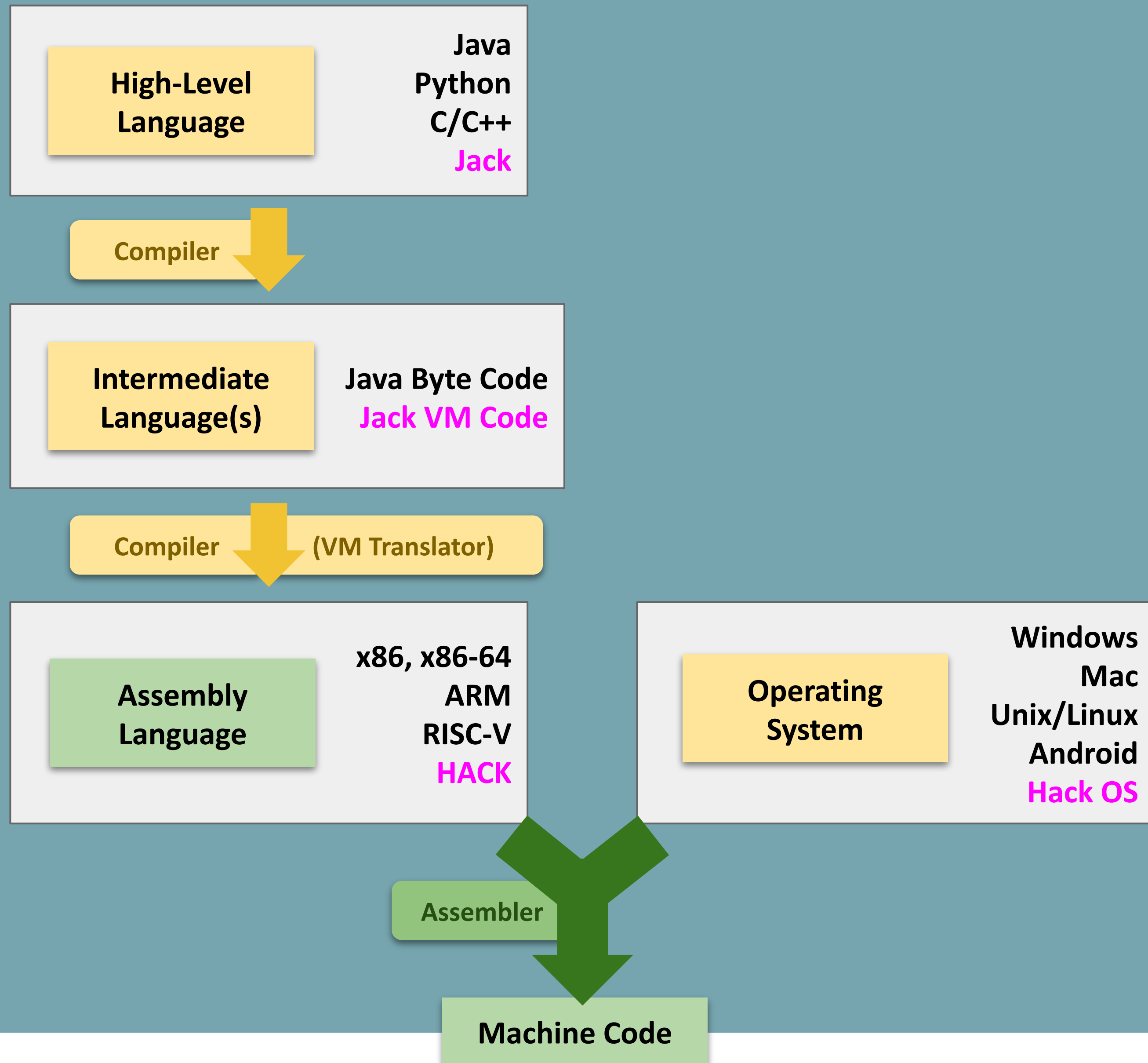
Agenda

- OS Overview
- Debugging
- Project 7 Tips
- Stress & Wellness Discussion

Agenda

- **OS Overview**
- Debugging
- Project 7 Tips
- Stress & Wellness Discussion

Software Overview



SOFTWARE

The Operating System

- Just another piece of software!
  - A massive, complex piece of software
  - In the end, uses the same machine language your code does
- OS is more trusted than the rest of the software that runs on your computer
- User programs/applications "invoke" or "ask" the OS to perform operations they are not trusted or allowed to
  - Means the OS has to be super secure!

Why an OS?

- Directly interacts with the hardware
  - Benefit: **Abstraction**
    - Provides high-level functionality for messy hardware devices
    - OS must be ported to new hardware; but user-level programs can then be portable
  - Benefit: **Protection**
    - OS is trusted to touch hardware; user-level programs are not
    - User-level programs cannot “break things”
    - Maintains security between programs and user accounts

OS: Abstraction

- Many abstractions provided by real-world Operating Systems!
- File System
  - File contents = just bits in the “giant array” that is the **hard drive** (“permanent” storage, as opposed to temporary storage in RAM that disappears when computer is turned off).
  - OS keeps a record of which ones fall into which “files”.
- Network Stack
  - Communicating with network devices ≈ communicating with screen/keyboard memory map
  - OS handles messy, time-sensitive protocols
- Processes
  - Only one process can run at once on a CPU
  - OS switches very quickly, illusion of running both “at once”

OS: Protection

- The CPU has different "privilege" levels when it is executing (controlled by a register on the CPU)
- OS code and memory can only be executed by an OS privilege level
  - Your applications run at a lower level and cannot access OS code and memory
- This helps prevent applications from crashing your entire system
  - For example, if your web browser crashes, usually it doesn't crash your entire computer!
  - Also helpful for security purposes

OS: Processes

- A "process" is an application running on your computer!
  - e.g. your web browser, terminal, word, etc.
- Each application instance is contained in one or more processes
- The OS manages these processes!
- Multiple processes are "running" at the same time, but it's really just the OS switching super fast between them (faster than the eye can see)
- A process only has access to its memory, and cannot access the memory of other processes
  - This is helpful because if one process crashes or is malicious, it makes it more difficult to crash or corrupt other processes too

Why *not* an OS?

- The Hack computer we've built is… small
  - Uses the same *principles* as your laptop CPU
  - But in terms of *scale*, closer to a microprocessor or small embedded chip

- For embedded systems, often an OS is overkill -- instead, designed to be programmed with/run a single program at a time
  - Pro: developer gets complete control over the device
  - Con: re-implement OS features, no protection

Mac vs. Linux vs. Windows - What's the big deal?

- Really three different ways to do pretty much the same thing
  - Everyone has their own preference!
- Each have their own benefits/tradeoffs
  - Work on varying types of hardware, provide different levels of customization, different features, work better with different softwares, open source vs. proprietary, etc.
- You could choose to do some research next time you are deciding on a laptop/computer/OS
  - Only if you think it's something you WANT to do - could also just ignore it entirely

Agenda

- OS Overview
- **Debugging**
- Project 7 Tips
- Stress & Wellness Discussion

Debugging!

Source / Acknowledgements

This is a subset+adaptation of a CSE331 lecture

If you've taken CSE331, this is perspective seeing more than once

- Plus we are going to "make you do it"
- And it's closely connected to metacognition

If you haven't taken CSE331, this is a helpful sneak peek

- Indeed, it would help if our 100-level classes taught debugging more explicitly

Acknowledgements: CSE331 instructors, notably Michael D. Ernst, Hal Perkins, …

Debugging Pre-discussion

- Breakouts! Discuss the following questions:
- How often do you run into bugs when writing programs?
- How intentional have you been with debugging in the past? In other words, when you run into a bug, do you have strategies that you consistently use to find it?
    - For those who have taken 331, maybe think back to before you had the debugging lecture
- What debugging strategies have you come across in the past?

A Bug's Life

defect – mistake committed by a human

error – incorrect computation

failure – visible error: program violates its specification

Debugging starts when a failure is observed

- During testing
- In the field

Goal is to go *from failure **back** to defect*

Testing vs. debugging

Testing ≠ debugging

- *test*: reveals existence of problem (failure)
- *debug*: pinpoint location + cause of problem (defect)

See CSE331 for:

- How to write code that has fewer bugs (so less debugging)
- How to write code that is easier to test (so easier to reveal bugs)
- How to make testing easier (so you do it more often)
- How to write code that is easier to debug (so less time spent debugging)

These are all incredibly valuable engineering skills

Last (inevitable) resort: debugging

Defects happen – people are imperfect

- Industry average (?): 10 defects per 1000 lines of code

Defects happen that are not immediately localizable

- Found during integration testing
- Or reported by user

Cost of an error increases by orders of magnitude during program lifecycle

step 1 – Clarify symptom (simplify input), create "minimal" test
step 2 – Find and understand cause
step 3 – Fix
step 4 – Rerun *all* tests, old and new

The debugging process

step 1 – *find small, repeatable test case that produces the failure*

- May take effort, but helps identify the defect *and* gives you a regression test
- Do *not* start step 2 until you have a simple repeatable test

step 2 – *narrow down location and proximate cause*

- *Loop*: (a) Study the data (b) hypothesize (c) experiment
- Experiments often involve changing the code
- Do *not* start step 3 until you understand the cause

step 3 – *fix the defect*

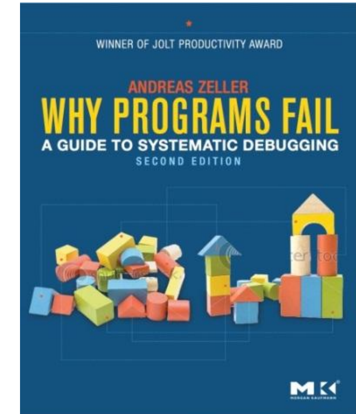
- Is it a simple typo, or a design flaw?
- *Does it occur elsewhere?*

step 4 – *add test case to regression suite*

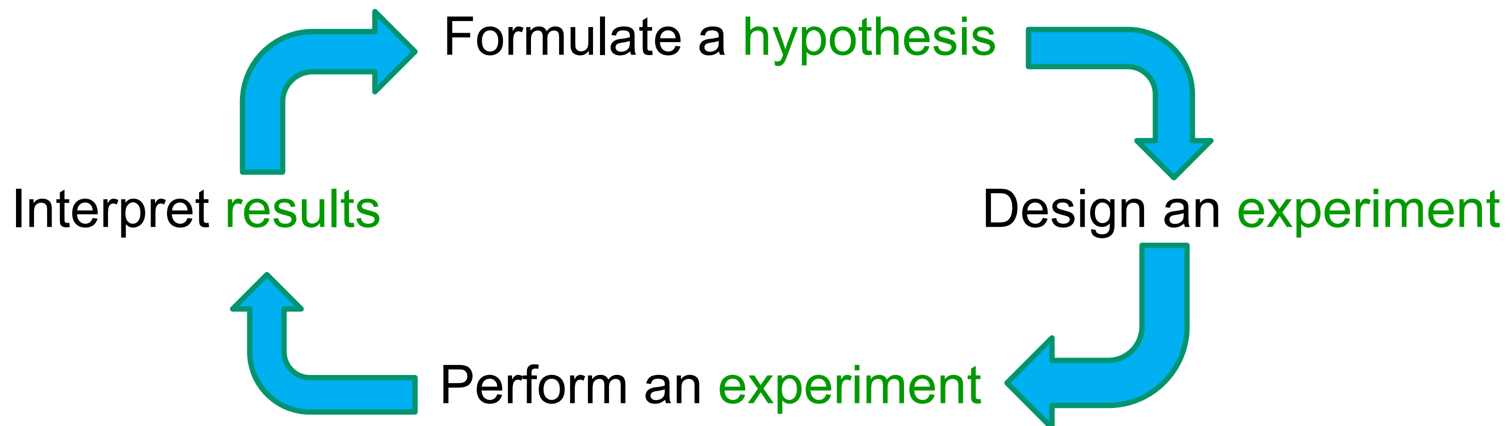
- Is this failure fixed? Are any other new failures introduced?

Debugging and the scientific method

- Debugging should be *systematic*
  - Carefully *decide* what to do
    - Don't flail!
  - Keep a *record* of everything that you do
  - Don't get sucked into fruitless avenues



- Use an iterative scientific process:



Example

```
// returns true iff sub is a substring of full
// (i.e. iff there exists A,B such that full=A+sub+B)
boolean contains(String full, String sub);
```

User bug report:

It can't find the string "**very happy**" within:

"Fáilte, you are very welcome! Hi Seán! I am very very happy to see you all."

Poor responses:

- Notice accented characters, panic about not knowing about Unicode, begin unorganized web searches and inserting poorly understood library calls, …
- Start tracing the execution of this example

Better response: simplify/clarify the symptom…

Reducing *absolute* input size

Find a simple test case by divide-and-conquer

Pare test down:

Cannot find "**very happy**" within

"**Fáilte, you are very welcome! Hi Seán! I am very very happy to see you all.**"

"**I am very very happy to see you all.**"

"**very very happy**"

Can find "**very happy**" within

"**very happy**"

Cannot find "**ab**" within "**aab**"

Reducing relative input size

Can you find two almost identical test cases where one gives the correct answer and the other does not?

Cannot find "**`very happy`**" within

`"I am very very happy to see you all."`

Can find "**`very happy`**" within

`"I am very happy to see you all.”`

General strategy: simplify

In general: find simplest input that will provoke failure

- Usually *not* the input that revealed existence of the defect

Start with data that revealed the defect

- Keep paring it down ("binary search" can help)
- Often leads directly to an understanding of the cause

When not dealing with simple method calls:

- The "test input" is the set of steps that reliably trigger the failure
- Same basic idea

Localizing a defect

Take advantage of modularity

- Start with everything, take away pieces until failure goes away
- Start with nothing, add pieces back in until failure appears

Take advantage of modular reasoning

- Trace through program, viewing intermediate results

Binary search speeds up the process

- Error happens somewhere between first and last statement
- Do binary search on that ordered set of statements

Binary search on buggy code

```
public class MotionDetector {
    private boolean first = true;
    private Matrix prev = new Matrix();

    public Point apply(Matrix current) {
        if (first) {
            prev = current;
        }
        Matrix motion = new Matrix();
        getDifference(prev,current,motion);
        applyThreshold(motion,motion,10);
        labelImage(motion,motion);
        Hist hist = getHistogram(motion);
        int top = hist.getMostFrequent();
        applyThreshold(motion,motion,top,top);
        Point result = getCentroid(motion);
        prev.copy(current);
        return result;
    }
}
```

no problem yet

Check intermediate result at half-way point

problem exists

Binary search on buggy code

```
public class MotionDetector {
    private boolean first = true;
    private Matrix prev = new Matrix();

    public Point apply(Matrix current) {
        if (first) {
            prev = current;
        }
        Matrix motion = new Matrix();
        getDifference(prev,current,motion);
        applyThreshold(motion,motion,10);
        labelImage(motion,motion);
        Hist hist = getHistogram(motion);
        int top = hist.getMostFrequent();
        applyThreshold(motion,motion,top,top);
        Point result = getCentroid(motion);
        prev.copy(current);
        return result;
    }
}
```

no problem yet

Check intermediate result at half-way point

problem exists

Detecting Bugs in the Real World

Real Systems
- Large and complex (duh ☺)
- Collection of modules, written by multiple people
- Complex input
- Many external interactions
- Nondeterministic

Replication can be an issue
- Infrequent failure
- Instrumentation eliminates the failure
- No printf or debugger

Errors cross abstraction barriers

Large time lag from corruption (error) to detection (failure)

Heisenbugs

In a sequential, deterministic program, failure is repeatable

But the real world is not that nice…
- Continuous input/environment changes
- Timing dependencies
- Concurrency and parallelism

Failure occurs randomly
- Depends on results of random-number generation
- Hash tables behave differently when program is rerun

Bugs hard to reproduce when:
- Use of debugger or assertions makes failure goes away
  - Due to timing or assertions having side-effects
- Only happens when under heavy load
- Only happens once in a while

Logging Events

Log (record) events during execution as program runs (at full speed)

Examine logs to help reconstruct the past

- Particularly on failing runs
- And/or compare failing and non-failing runs

But don’t spend too much time manually reading enormous, confusing logs

More Tricks for Hard Bugs

Rebuild system from scratch, or restart/reboot

- Find the bug in your build system or persistent data structures

Explain the problem to a friend (or to a rubber duck)

Make sure it is a bug

- Program may be working correctly and you don't realize it!

Face reality

- Debug reality (actual evidence), not what you think is true

And things we already know:

- Minimize input required to exercise bug (exhibit failure)
- Add more checks to the program
- Add more logging

Where is the defect?

The defect is not where you think it is

- Ask yourself where it cannot be; explain why
- Self-psychology: look forward to being wrong!

Look for simple easy-to-overlook mistakes first, e.g.,

- Reversed order of arguments
- Spelling of identifiers
- Same object vs. equal: `a == b` versus `a.equals(b)`
- Uninitialized data/variables
- Deep vs. shallow copy

Make sure that you have correct source code!

- Check out fresh copy from repository; recompile everything
- Does a syntax error break the build? (it should!)

When the going gets tough

Reconsider assumptions

- Debug the code, *not* the comments
  - Ensure that comments and specs describe the code

Start documenting your system

- Gives a fresh angle, and highlights area of confusion

Get help

- We all develop blind spots
- Explaining the problem often helps (even to rubber duck)

Walk away

- Trade latency for efficiency – **sleep!**
- One good reason to start early

Key Concepts

Testing and debugging are different

- Testing reveals *existence of failures*
- Debugging pinpoints *location of defects*

Debugging should be a systematic process

- Use the *scientific method*

Understand the source of defects

- To find similar ones and prevent them in the future

Learn from the debugging process

- It's inevitable and you have some control over how you approach the frustration

Debugging Post-discussion

- Breakouts! Discuss the following questions:
- How useful was this lecture? What “gaps” are there or what still seems hard to incorporate into your own practices?
- What strategies and concepts resonated most with you from today’s lecture?

Agenda

- OS Overview
- Debugging
- **Project 7 Tips**
- Stress & Wellness Discussion

Project 7 Buggy Compiler Overview

0	Read starter code	
1	**Implement** NumberLiteral.java	~4 Lines
2	**Debug** Plus.java	2 Bugs
3	**Implement** Minus.java	~13 Lines (similar to Plus)
4	**Implement** NotEquals.java	~21 Lines (similar to Equals)
5	**Implement** ArrayVarAccess.java	~3 Lines
6	**Debug** If.java	2 Bugs
7	**Implement** While.java	~14 Lines

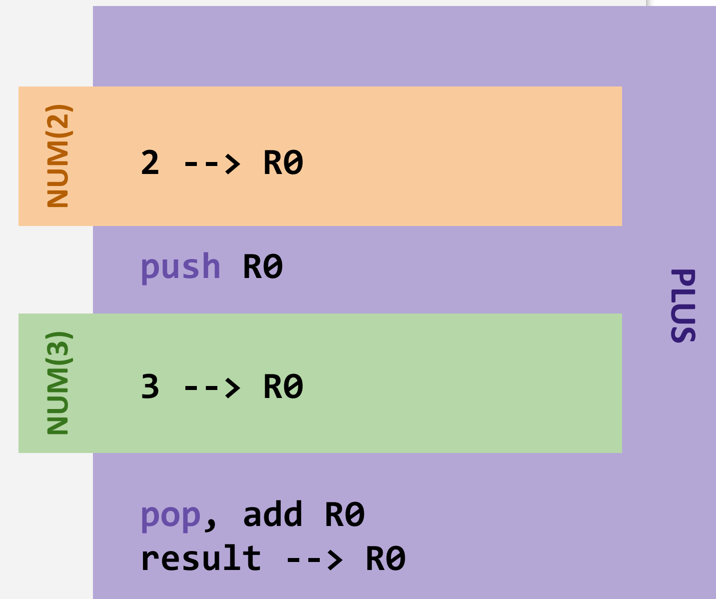
Example: Plus (Step 2)

```
public class Plus extends Expression {
    public Expression left;
    public Expression right;

    @Override
    public void printASM() {
        comment("Start Plus");

        left.printASM();

        instr("@R0");
        instr("D=M");

        right.printASM();

        push();

        instr("@R1");
        instr("A=M");

        instr("D=D+A", "perform the addition");

        ...
```

Example: Plus (Step 2)

1 Structural Bug: Map to abstract diagram for Plus:

```
public class Plus extends Expression {
    public Expression left;
    public Expression right;

    @Override
    public void printASM() {
        comment("Start Plus");

        left.printASM();

        instr("@R0");
        instr("D=M");

        right.printASM();

        push();

        instr("@R1");
        instr("A=M");

        instr("D=D+A", "perform the addition");

        ...
```



1 Detail Bug: Step through generated code, Check state at each step

Project 7: MicroJack Language Gotchas

- Can't write a negative integer literal
  - Instead, use subtraction from zero: `0 - 1`
- All variable declarations must come before all regular statements
  - (Why? Simplifies concept of a "defined" variable)
- No defined operator precedence
  - If order matters for an operation, use parentheses
- Arrays are very simple
  - `arr[index]` is really just calculating an address: take address of `arr` variable and add `index` to it as an offset
  - No array bounds checking -- just lets you run off the end
- "Booleans" are just 0 (false) and non-zero (true)

Project 7: Debugging Tips

- Try walking through the general `printASM` code to understand *why* each line is there
  - Add comments to the assembly as you go! *Much* easier to understand resulting file
- Find the smallest example you can
  - Provided tests get progressively more complex, but you may want to write your own tiny test case to isolate
  - ASM gets long fast -- we've added comments so you can isolate to the section you're working on
- "Play Computer": as you step through the code, write down the state you expect after each instruction, then advance and see if the CPUEmulator agrees

More Project 7 Tips

- When debugging assembly, a good first step is to try understanding the code and adding comments to the assembly as you go -- much easier to understand resulting file!
- A `printDebug` method has been implemented for you on all AST Nodes. Use it to visualize exactly what the Parser is giving you, but also as a basis for `printASM`! Both need to do processing on the current node and strategically recurse on its children.
- Pushing/popping from the stack is intimidating, but formulaic -- understand it once, copy/paste afterward!
  - push() and pop() are already implemented for you!
- We provide only a few Micro-Jack test files. You'll definitely want to write more of your own.
  - Can use Sandbox.* to write more tests, or create your own files!

Agenda

- OS Overview
- Debugging
- Project 7 Tips
- **Stress & Wellness Discussion**

Stress & Wellness Discussion

- Reading was to listen/read Episode 1 of the Feminist Survival Project podcast
- Topic was the stress response cycle and dealing with stress vs. dealing with stressors
- Lots of information to process despite being a relatively short episode!
  - Discussion with others can be a great way to debrief something that introduces a potentially new way of thinking about things

Stress & Wellness Discussion Prompt 1

- We can often trick ourselves into thinking that stress is something that can be willed away by our minds, a notion challenged by Emily and Amelia's theme “Wellness is not a state of mind nor is it a state of being. It is a state of action”.
  - How much time do you set aside for taking action towards wellness?
  - How much have you reflected on what actions do/don't contribute to your wellness?
  - What might be an example of treating wellness as a state of mind/being vs. treating wellness as a state of action?

Stress & Wellness Discussion Prompt 2

- One focus of the podcast was the importance in distinguishing between addressing our stress response and addressing the stressors in our life.
  - Reflecting on your own life, identify some of the common stressors that come up for you both externally (e.g. traffic) and internally (e.g. self-criticism)
  - How do you typically respond to stressors – physically, emotionally and cognitively?
  - Where do you feel you could focus on addressing your stress response and situations where you could focus on addressing your stressors? Why might those situations be more suitable to focusing on one or the other?

Wrapping Up

What's in store for Week 9?

- Design Processes
- Networking
- Final Project Overview

Reminders

- Midterm Corrections Due tonight 11:59PM PDT (No late days!)
- Professor Meeting Report Due next Thursday, May 27th
- Project 7 Released, due Tuesday, June 1st