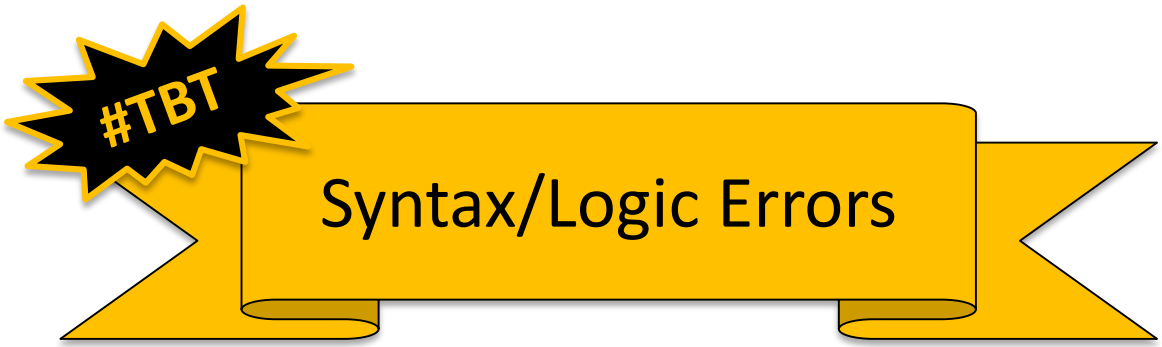
CMSC201
Computer Science I for Majors

Lecture 13 – Program Design

Last Class We Covered

- Two-dimensional lists
- Lists and functions
- Mutability



Any Questions from Last Time?

Today's Objectives

- To learn about modularity and its benefits
- To see an example of breaking a large program into smaller pieces
  - Top Down Design
- To introduce two methods of implementation
  - Top Down and Bottom Up

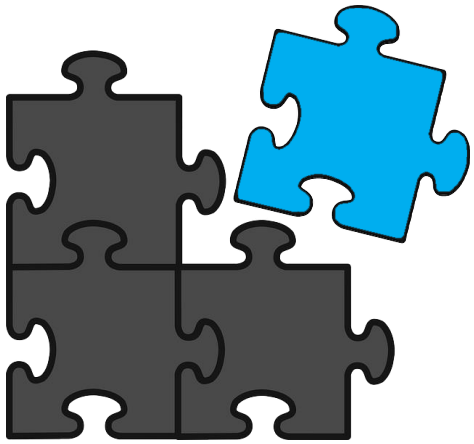
Modularity

Modularity

- A program being ***modular*** means that it is:
- Made up of individual pieces (modules)
  - That can be changed or replaced
  - Without affecting the rest of the system

- So if we replace or change one function, the rest should still work, even after the change

Modularity



- With modularity, you can reuse and repurpose your code

- What are some pieces of code you've had to write multiple times?
  - Getting input between some min and max
  - Using a sentinel loop to create a list
  - What else?

Functions and Program Structure

- So far, functions have been used as a mechanism for reducing code duplication
- Another reason to use functions is to make your programs more modular

- As the algorithms you design get increasingly complex, it gets more and more difficult to make sense out of the programs

Functions and Program Structure

- One option to handle this complexity is to break it down into smaller pieces

- Each piece makes sense on its own
- You can then combine them together to form the complete program

Helper Functions

- These are functions that assist other functions, or that provide basic functionality

- They are often called from functions other than **`main()`**

Planning `getValidInt()`

- What about a helper function that is called any time we need a number within some range?
  - Grades: 0 to 100
  - Menu options: 1 to N (whatever the last option is)

- What should it take in? What should it output?
  - Input: the minimum and maximum
  - Output: the selected valid number

Creating `getValidInt()`

- Here is one possible way to implement it:

```
def getValidInt(minn, maxx):
    message = "Enter a number between " + str(minn) + \
        " and " + str(maxx) + " (inclusive): "

    newInt = int(input(message))
    while newInt < minn or newInt > maxx:
        print("That number is not allowed.  Try again!")
        newInt = int(input(message))

    return newInt
```

Using `getValidInt()`

- Now that the function is written, we can use it
  - To get a valid grade

```
grade = getValidInt(0, MAX_GRADE)
```

  - To get a menu choice

```
printMenu()
choice = getValidInt(MENU_MIN, MENU_MAX)
```

  - To get a valid index of a list

```
index = getValidInt(0, len(myList)-1 )
```

Complex Problems

- If we only take a problem in one piece, it may seem too complicated to even begin to solve
  - Create a program that lets two users play a game of checkers
  - Search for and present user-requested information from a database of music
  - Creating a video game from scratch

Top Down Design

Top Down Design

- Computer programmers often use a ***divide and conquer*** approach to problem solving:
  - Break the problem into parts
  - Solve each part individually
  - Assemble into the larger solution
- One example of this technique is known as ***top down design***

Top Down Design

- Breaking the problem down into pieces makes it more manageable to solve

- ***Top-down design*** is a process in which:
  - A big problem is broken down into small sub-problems
    - Which can themselves be broken down into even smaller sub-problems
      - And so on and so forth…

Top Down Design: Illustration

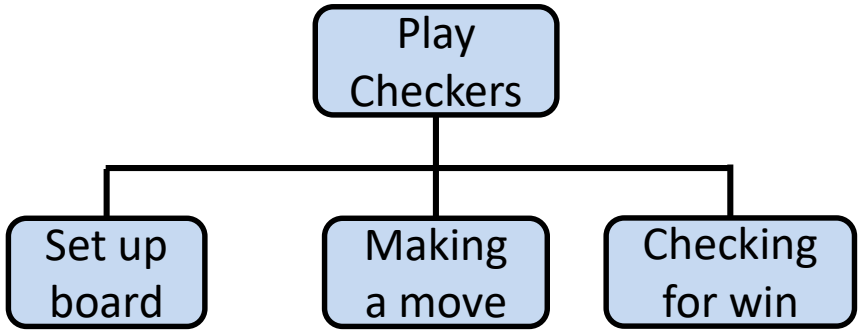
- First, start with a clear statement of the problem or concept

Play Checkers

- A single big idea

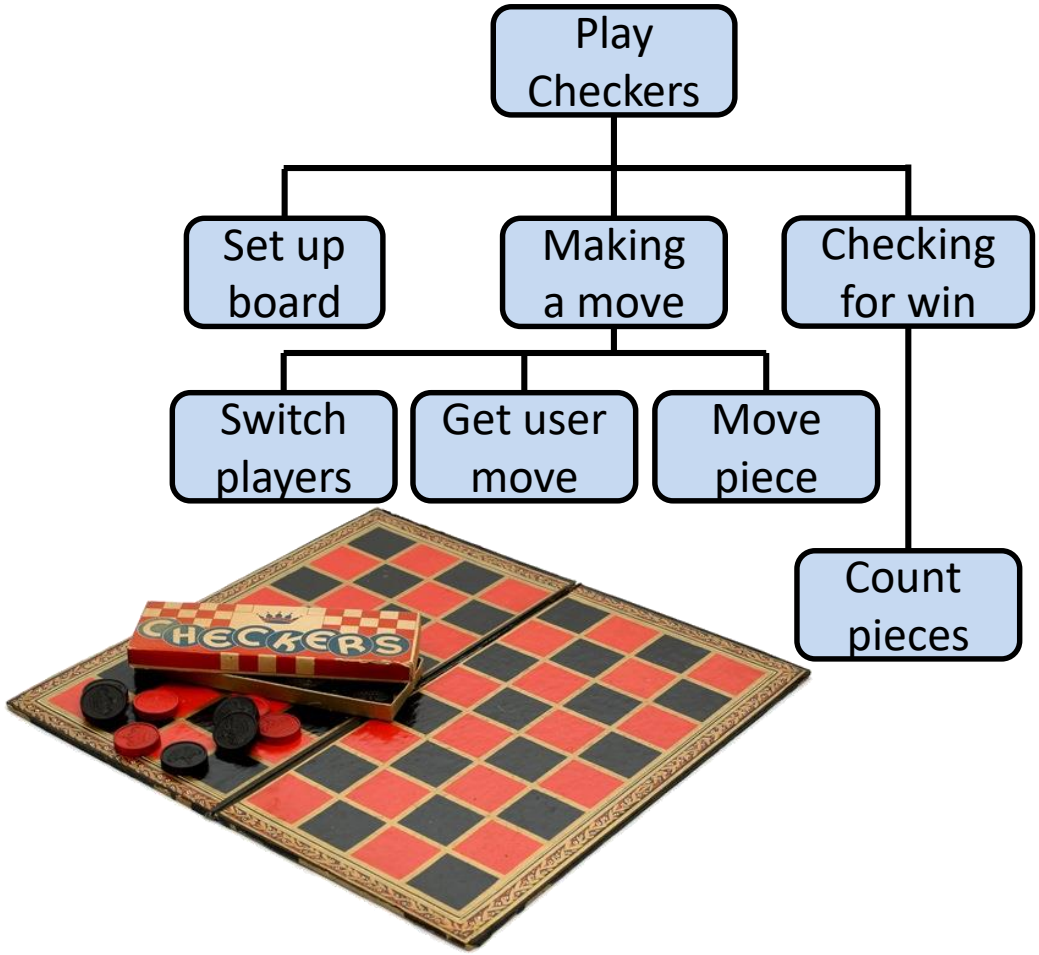
Top Down Design: Illustration

- Next, break it down into several parts



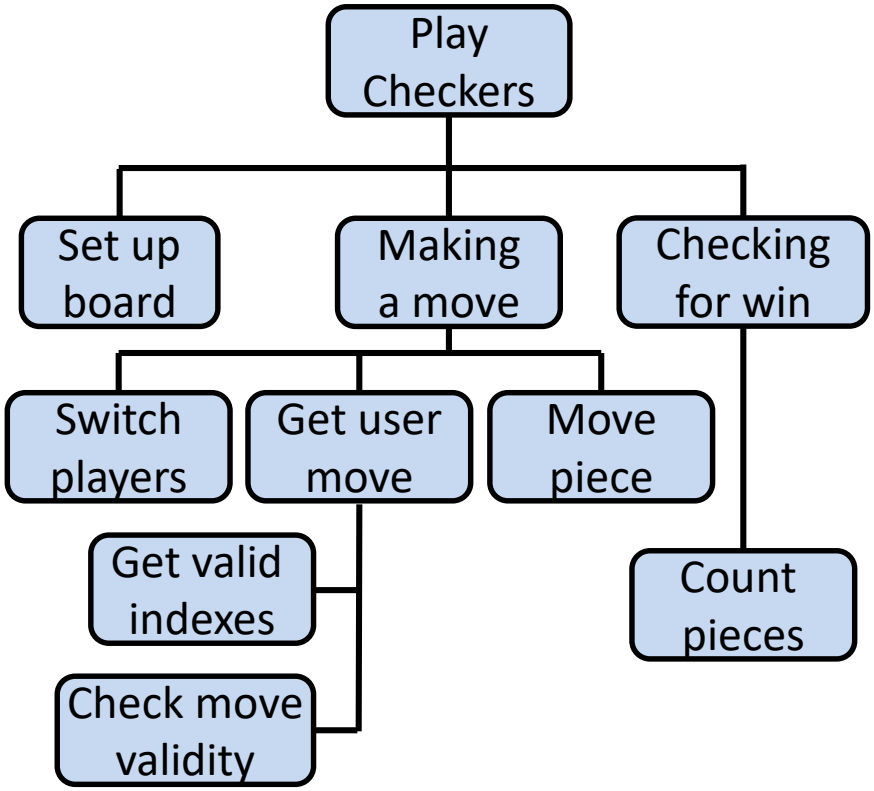
Top Down Design: Illustration

- Next, break it down into several parts
- If any of those parts can be further broken down, then the process continues…



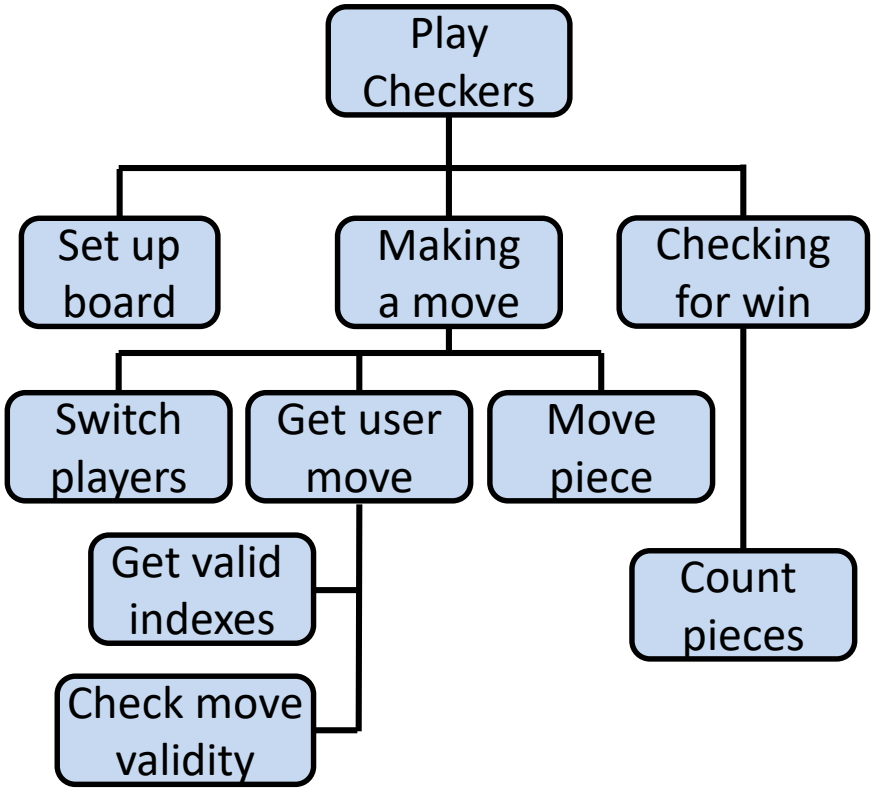
Top Down Design: Illustration

- And so on…



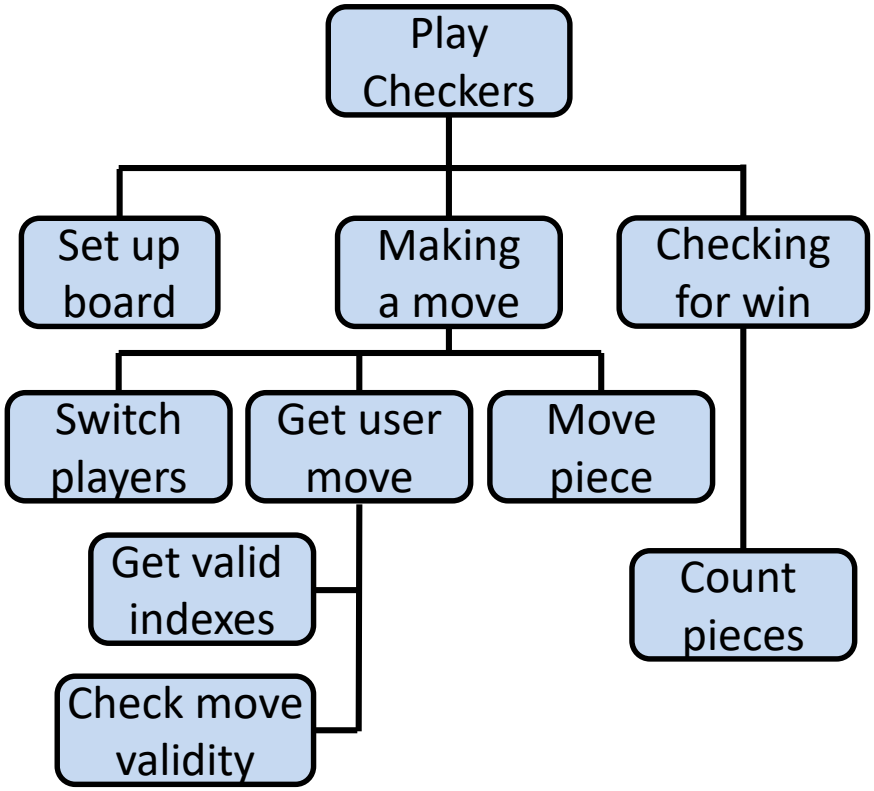
Top Down Design: Illustration

- Your final design might look like this chart, which shows the overall structure of the smaller pieces that together make up the “big idea” of the program



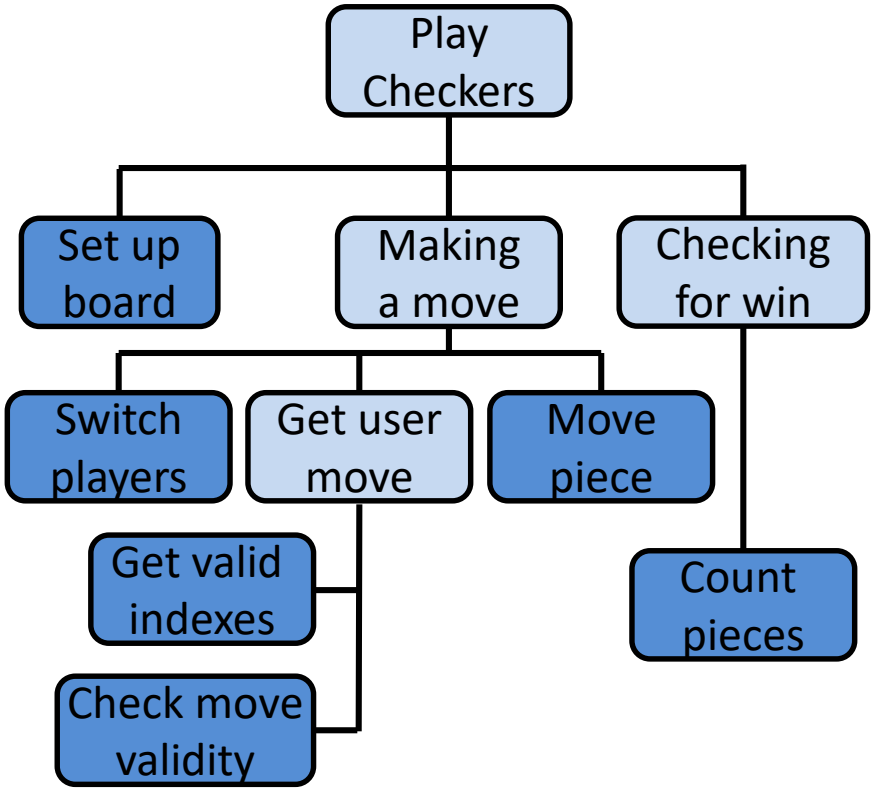
Top Down Design: Illustration

- This is like an upside-down "tree," where each of the nodes represents a single process (or a function)



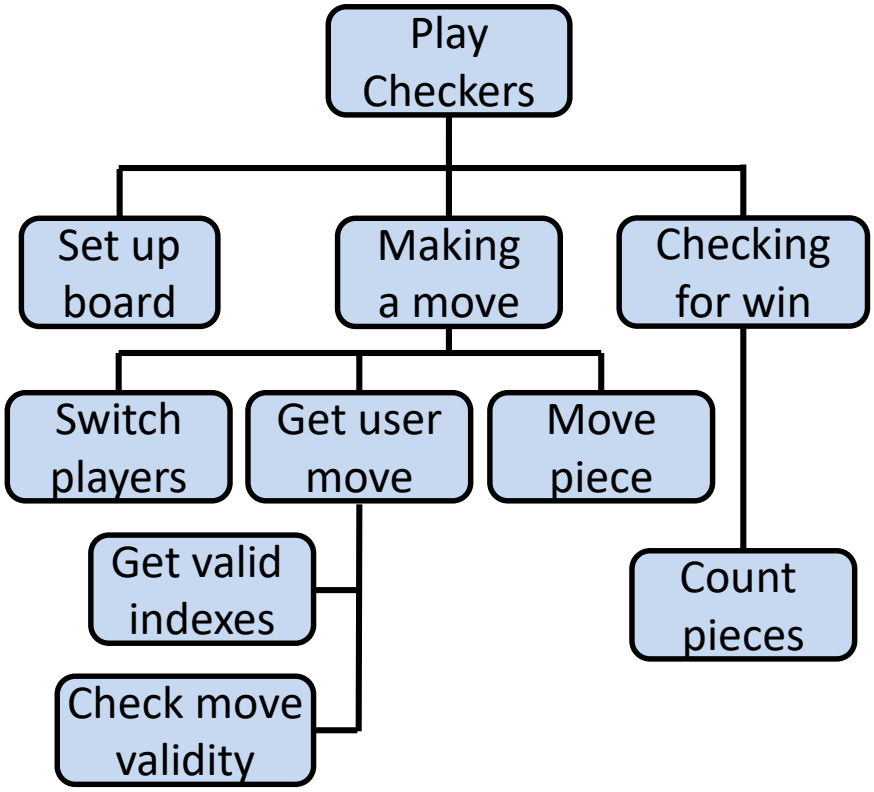
Top Down Design: Illustration

- The bottom nodes are “leaves” that represent pieces that need to be developed
- They are then recombined to create the solution to the original problem



Top Down Design

- We've created a simplified design that's easy to follow
- Still missing a couple pieces, but it's a start!
  - There's also no plan included for **`main()`** in this design



Analogy: Essay Outline

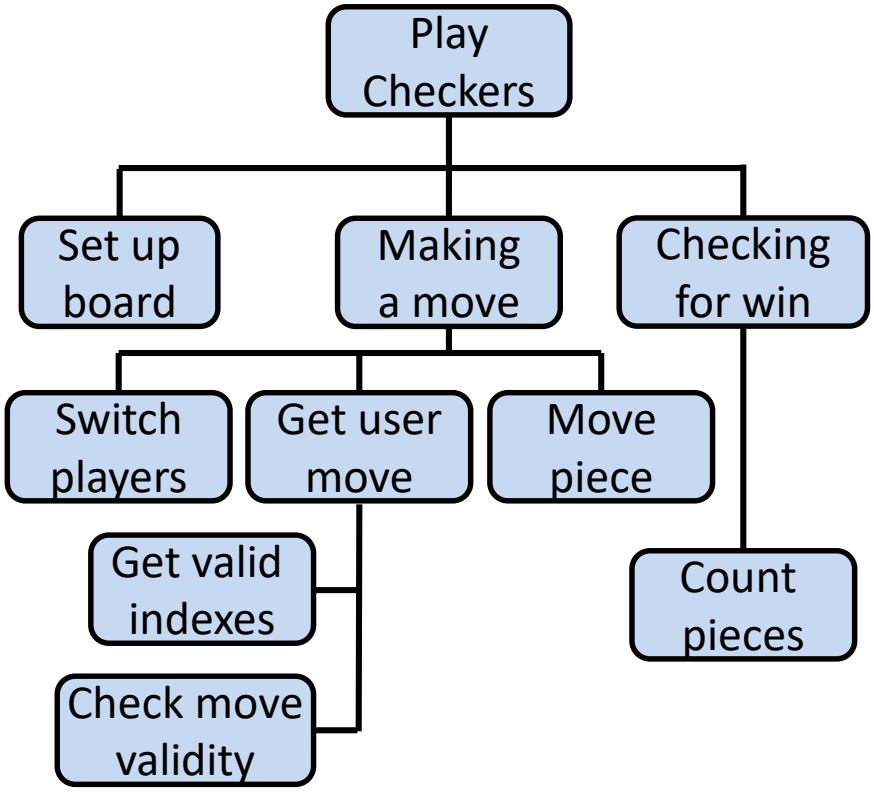
- Think of it as an outline for a essay you're writing for a class assignment

- You don't just start writing things down!
  - You come up with a plan of the important points you'll cover, and in what order
  - This helps you to formulate your thoughts as well

Implementing a Design in Code

Bottom Up Implementation

- Develop each of the modules separately
  - Test that each one works as expected
- Then combine into their larger parts
  - Continue until the program is complete



Bottom Up Implementation

- To test your functions, you will probably use **`main()`** as a (temporary) test bed
  - You can even call it **`testMain()`** if you want

- Call each function with different test inputs
  - How does the board setup work if it's 1x1?
  - Does the **`if/else`** work when switching players?
  - Ensure that functions "play nicely" together

Top Down Implementation

- Sort of the "opposite" of bottom up
- Create "dummy" functions that fulfill the requirements, but don't perform their job
  - For example, a function that is supposed to get the user move; it takes in the board, but simply returns that they want to move to 0, 0
- Write up a "functional" **`main()`** that calls these dummy functions
  - Helps to pinpoint other functions you may need

Which To Choose?

- Top down? Or bottom up?
- It's up to you!
  - As you do more programming, you will develop your own preference and style
- For now, just use something – don't code up everything at once without testing anything!

Daily emacs Shortcut

- **CTRL+V**
  - Moves the screen down one “page”
- **M+V**
  - Moves the screen up one “page”

Announcements

- HW 5 is out on Blackboard now
  - Due by Friday (March 16th) at 8:59:59 PM
- Lab 7 is online and available on the website
- Midterm is in class, next time we meet
  - Out-of-class reviews held Monday and Tuesday
  - Metacognition "quiz" available on Blackboard
    - You need to submit it for it to count!
    - Closes Tuesday night at 9 PM

Exam Rules

- The midterm is closed everything:
  - No books
  - No notes
  - No cheat sheets
  - No laptops
  - No calculators
  - No phones

Exam Rules

- Place your bag under your desk/chair
  - NOT on the seat next to you
- You may have on your desk:
  - Pencils, erasers
    - You **must** use a pencil, not a pen
  - Water bottle
  - **UMBC ID**
    - You **must** bring your UMBC ID with you to the exam! We won't accept your test without it.

Exam Rules

- Your TA or instructor may ask you to move at any time during the test
  - This doesn't mean we think you're cheating

- That being said, **DO NOT CHEAT!!!**
- Cheating will be dealt with severely and immediately
  - If a TA or instructor sees you looking at another student's paper they may take your test from you

Exam Seating

- Space allowing, you will sit every other seat, so that you are not next to another student
- Your instructor may have specific instructions for their lecture hall seating arrangements

Exam Advice

- Write down your name, sign the Academic Integrity agreement, and circle your section
  - Make sure your name is ***legible***

- Flip through the exam and get a feel for the length of it and the types of questions
  - The programming problems are the last questions on the exam – don't leave them until the last minute!

Exam Advice

- Most questions have partial credit
  - You should at least attempt every problem
  - If you don't know how to do one part of the problem, skip it and do the rest
  - You can use comments instead of code (like "`# get user input`") if you know what you want a piece of code to do but not how to do it

Exam Advice

- After you are done coding the programming problems, try "running" your program with some input and making sure it works the way you think it does

- If a problem is unclear or you think there is an error on the exam, raise your hand

Image Sources

- Puzzle pieces (adapted from):
  - https://pixabay.com/p-308908/
- Helping hands:
  - https://pixabay.com/p-40805/
- Checkers:
  - https://en.wikipedia.org/wiki/File:The_Childrens_Museum_of_Indianapolis_-_Checkers.jpg