Lecture 03

Ready or Not

Practice: Boolean Expressions and Conditional if-then Statements

Go to poll.unc.edu

Sign-in via this website then go to pollev.com/compunc

VSCode: Open Project -> View Terminal -> npm run pull -> npm start

Quiz News

- We will hand back quizzes via Gradescope tomorrow!
- Last semester, the average score on midterm 0 was an 80
- Quiz 0 - average score...

  90!

- The material does get more challenging, but staying on top of videos, coming to lecture prepared, and knocking out assignments is the key to success.

Videos 4 through 7 Graded Warm-up Questions

- You may have out a single sheet of handwritten notes
- Complete the 10 questions in order
  - Caution: PollEv does not allow going back to change responses
- We will spend 10 minutes on this. When complete, please do not do anything to distract your neighbors.
- Open **pollev.com/compunc**
  - Register quickly if needed.

Challenge Question #0 - pollev.com/compunc

- Solve for yourself with paper/pencil then talk with your neighbors to see if you came to the same answer.
  - Don't use an interactive programming REPL!

- What values of a, b, and c would cause the following expression to evaluate to `true`?

```
((a && b) || c) && ((a || b) && !c)
```

Challenge Question #1 - pollev.com/compunc

- Solve for yourself with paper/pencil then talk with your neighbors to see if you came to the same answer.
  - Don't use an interactive programming REPL!

- What values of a, b, and c would cause the following expression to evaluate to `true`?

```
(a > 9) && (a < c) && (c < 12 && !b)
```

Lecture Readiness - "Pulling" Class Materials

- When you come into lecture each day, the routine we'll get into is:

1. Open PollEv.com/compunc

2. Open VSCode -> View -> Terminal

3. In the Terminal, first run: **npm run pull**
   - This downloads the latest lecture materials.

4. Then run: **npm start**
   - This starts the development compiler and server allowing us to see the output of our code.

Hands-on #0

- Open 03-scope-and-conditionals / 00-scope-app.ts
- Directly after the the TODO comment...
- Write a sequence of variable declaration and assignment statements to:
- Swap the values stored in variables **a** and **b**
- Check-in on PollEv.com/compunc when complete.

Block Statements, Scope, and Shadowing

Block Statements (1 / 2)

- A **block** is a special kind of statement that groups multiple, related statements.

- Blocks are **enclosing curly braces** that "contain" its statements.

```
{
   // This is a block statement
   print("Statement one");
   print("Statement two");
}
```

- Blocks do not end with a semicolon after the closing curly brace. The closing curly signals the end of the block.

Block Statements (2 / 2)

- Anywhere you can write a statement, you can also write a block statement.
  - Thus, you can nest inner blocks inside of outer blocks.

```
{
    print("Statement one");
    {
        print("Statement two");
    }
    print("Statement three");
}
```

- **Important** formatting rule: **each statement inside of a block is indented one additional level**!

Variable's **Block Scope** Rule

- A variable is only accessible after it is declared in the same block or in an outer, containing block.

```
{
   let x = 0;
   print(x); // OK! x declared in same block
   {
      print(x); // OK! x declared in outer block
   }
}

print(x); // ERROR! x declared in different block
```

Follow-Along: Hiding temp

- Let's perform the swap in 00-scope-app.ts inside of a block and convince ourselves the temp variable only exists for as long as it was needed:

```
{
   let temp = a;
   a = b;
   b = temp;
}
```

Variable Scoping Intuition

- Why have these specific rules?

- Block statements are like building blocks
  - Programs are constructed through many block statements

- Declaring a variable *in a block* prevents unrelated blocks from
  - Needing to worry about the existence of unrelated variables
  - Accidentally changing and mucking up another block's variables

- The rules help you avoid accidental logical errors

Warning: Variable Shadowing (1 / 2)

- You cannot declare two variables with the same name in the same block.

```
{
   let x = 0;
   let x = 1; // ERROR! x declared prev in same block
}
```

- Why not? It helps you avoid accidents in longer blocks of code.
  - i.e. you forgot you declared a variable of the same name and used it for another purpose

Warning: Variable Shadowing (2 / 2)

- You *can* declare a variable of the same name in a nested, inner block. This is called **"variable shadowing"**.
- The inner variable is a completely separate variable from the outer variable.
- When the processor returns to the outer block, **x** refers to the original **x** variable and its contents are unchanged.

```
let x = 0;
print(x); // Prints 0
{
  let x = 10;
  print(x); // Prints 10
}
print(x); // Prints 0
```

Variable shadowing is confusing and can usually be avoided by choosing meaningful variable names.

Challenge Question #2 - pollev.com/compunc
What is the output of these programs?

```
let x = 17;
if (x < 18) {
   print("A");
}

if (x > 13) {
   print("B");
} else {
   print("C");
}
```

…

```
let x = 17;
if (x < 18) {
      print("A");
} else {
   if (x > 13) {
      print("B");
   } else {
      print("C");
   }
}
```

Challenge Question #3 - What input at the prompt would cause "C" to print?

```
let x = await promptNumber("Enter a value for x");
if (x < 18) {
      print("A");
} else {
   if (x > 13) {
      print("B");
   } else {
      print("C");
   }
}
```

Generating Random Numbers

- The **`introcs`** Library has a special function for generating random numbers called... ***`random`***

- Before using `random`, we must import it into our program like **`print`**:

```
import { print, random } from "introcs";
```

- The random function generates a random number, so we can use it anywhere we can use a number:

```
let response: number = random(0, 2);
```

"Let choice be a number variable that is assigned the result of calling the random function with the arguments 0 and 2."

- The two numbers we "give" to the random function specify the bounds of the random number generated (a number between 0 and 2, inclusive).

Hands-on: Magic 8-Ball

- Open: **01-magic-8-ball-app.ts**

- Write a nested if-then-else statement (syntax below) at TODO #1 that will:

  if the **`response`** variable is equal to zero, **then** print "Very doubtful"
  otherwise,
  if `response` is equal to one, then print "Ask again later",
  otherwise, print "It is certain"

- **`if-then-else`** statement syntax:

```
if (<test>) {
     // then block
} else {
     if (<test>) {
         // then block
     } else {
         // else block
     }
}
```

- Check-in on pollev.com/compunc when your program prints one of these 3 messages

Pattern: Nesting **if-then** in an **else** Pattern

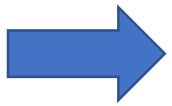
- It is commonly useful to nest additional if-then-else statements inside of subsequent else-blocks

- Why? It allows us to choose one next step from many possible options.
  - "If *this* then do X, otherwise if *that* do Y, *otherwise* do Z."

```
if (response === 0) {
    print("Very doubtful");
} else {
    if (response === 1) {
        print("Ask again later");
    } else {
        print("It is certain");
    }
}
```

This is so common and useful, we tend to use simpler syntax for it…

```
if (response === 0) {
    print("Very doubtful");
} else {
    if (response === 1) {
        print("Ask again later");
    } else {
        print("It is certain");
    }
}
```

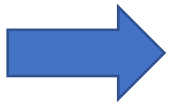


```
if (response === 0) {
    print("Very doubtful");
} else
    if (response === 1) {
        print("Ask again later");
    } else {
        print("It is certain");
    }
```

1. First we remove the curly braces surrounding the if-then that is nested inside of the else-block.

This is so common and useful, we tend to use simpler syntax for it…

```
if (response === 0) {
    print("Very doubtful");
} else
    if (response === 1) {
        print("Ask again later");
    } else {
        print("It is certain");
    }
```

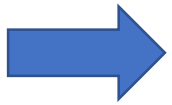


```
if (response === 0) {
    print("Very doubtful");
} else if (response === 1) {
    print("Ask again later");
} else {
    print("It is certain");
}
```

2. Then we clean up the spacing.

Using the **else-if** pattern is a change of *style* only. These two listings of code have the ***exact same logic***.

```
if (response === 0) {
    print("Very doubtful");
} else {
    if (response === 1) {
        print("Ask again later");
    } else {
        print("It is certain");
    }
}
```



```
if (response === 0) {
    print("Very doubtful");
} else if (response === 1) {
    print("Ask again later");
} else {
    print("It is certain");
}
```

Notice the code is visually simpler and cleaner by using else-if.

Follow-Along) Using the else-if Syntax Pattern

- Still in 01-magic-8-ball-app.ts
- Reformat the conditional logic to use the else-if syntax pattern.
- Step 1) Remove the curly brace directly following the *first* else and its matching closing curly brace.
- Step 2) Clean up the spacing by bringing the nested if to directly follow else and unindenting.
- Check-in when complete! pollev.com/compunc

Many, independent `if-then-else` statements

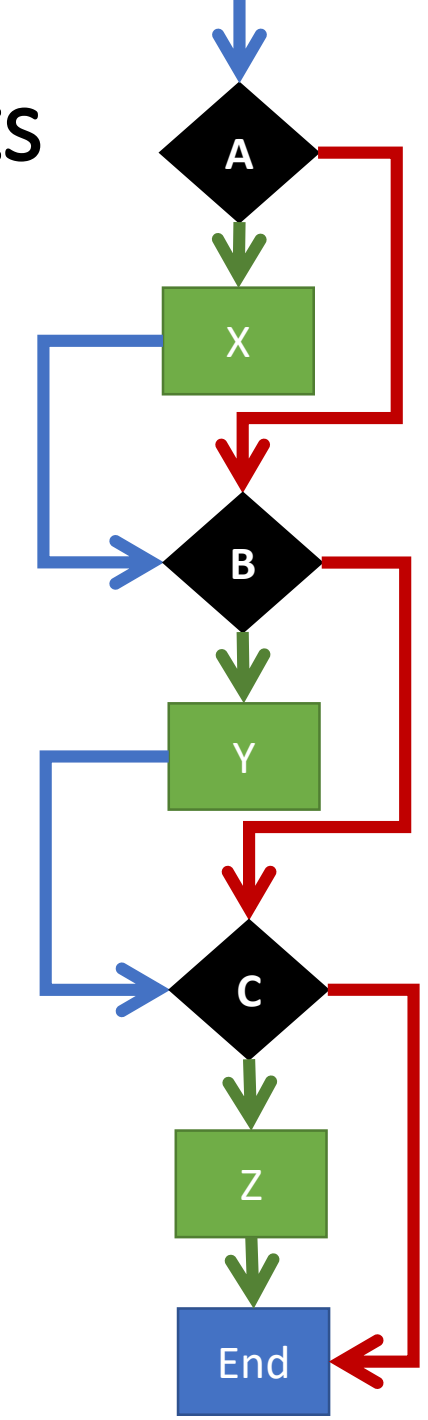
- When two or more if-then-else statements are *not* nested, they are independent statements of one another.
- Each `boolean` test expression will be evaluated.
- Notice in the diagram that there is a path through *every* block X, Y, Z.

```
if (A) {
  print("X");
}

if (B) {
  print("Y");
}

if (C) {
  print("Z");
}

print("End");
```



Tracing through `else-if` statements

- The previous slide does not apply to else-if statements *because…*
  - An else-if is a nested if-then
  - It is nested in the else-block

- Each `boolean` test expression will be evaluated **until one evaluates to true**. The rest are then skipped.

- Notice in the diagram that there is a path through *only one* outcome X, Y, Z.

- Useful when there are many possible next steps but you only want to choose one.

```
if (A) {

    print("X");

} else if(B) {

    print("Y");

} else if(C) {

    print("Z");

}

print("End");
```