Unit 5 Lesson 1

Introduction to Event-Driven Programming

Resources

Unit 5 Lesson 2

Multi-Screen Apps

Resources

Unit 5 Lesson 3

Building an App: Multi-Screen App

Resources

Name(s)_____ Period _____ Date _____

Activity Guide - Multi-screen App

Multi-screen App

You will be **creating your own multi-screen app** to practice designing user interfaces and writing event-driven programs**.** You have a lot of freedom to choose what your application will be but some ideas might include:

- A personal app about you and your hobbies / interests
- A “Top 3” app for a category of your choosing
- An informational app for an organization or club
- A flash card app for studying for quizzes
- A short “choose-your-own adventure” game
- An app with a different game on each screen

Requirements

Your application must have the following components:

- Your app must have some kind of **purpose**
  - Even if the purpose is simple like “Celebrating all my favorite foods to eat”, there must be an underlying purpose that thematically ties the whole thing together.
  - The title of your app should make it pretty clear.
- Your app will have **at least 4 screens.**
- Your app should include **text, images, and buttons** (and optionally sound).
- **No “getting stuck” on a screen.**
  - It should always be possible to navigate from a screen in your app to some other screen.
  - The user should also be able to “get back to the start” somehow. There are many ways to do this (e.g., screens go in a cycle, or every screen can navigate back to the home screen, etc.) but you should make sure you plan accordingly.
- Your program code should follow good style, particularly by giving UI elements **descriptive and meaningful IDs.**
- Your user interface should be **intuitive to use.**

Process

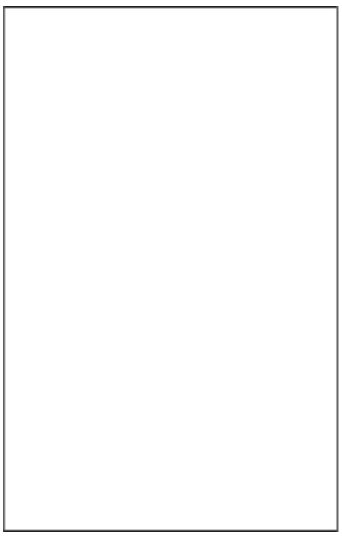
- **Choose** the theme and purpose of your app.
- **Complete** the Planning Guide to decide how you will display your information.
- **Informally Share** the sketch of your idea with a classmate to get some basic feedback and to see if they have any ideas you hadn’t thought of. Possible discussion points:
  - Does the way users navigate through pages intuitive?
  - Is the design / layout clear and present the information well?
  - Anything you would add? Anything you would take out?
- **Program** your app following the plan you develop in the Planning Guide.
- **Peer Review** at least one of your classmates’ apps using the Peer Review Rubric.

Planning Guide

Outline Your App

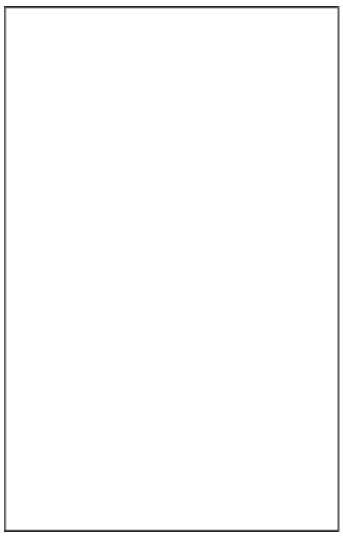
You will be **sketching out** the layout of your application using the rectangles below. Each rectangle represents a screen of your app. For each screen you should:

- **Decide** what information will be included on that screen.
- Give the screen a **descriptive ID.**
- Add any **notes** about the content that will be featured in that screen.
- Within each rectangle, **draw the elements** that will appear in that screen.
- **Draw arrows** to / from your screen showing how a user will be able to navigate through the app.



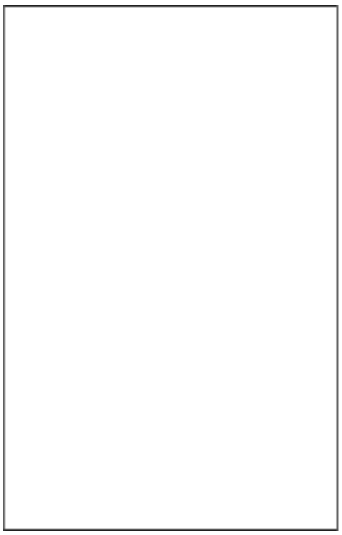
Screen ID:

Notes:



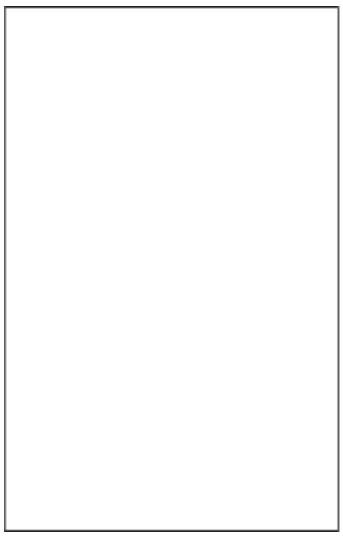
Screen ID:

Notes:



Screen ID:

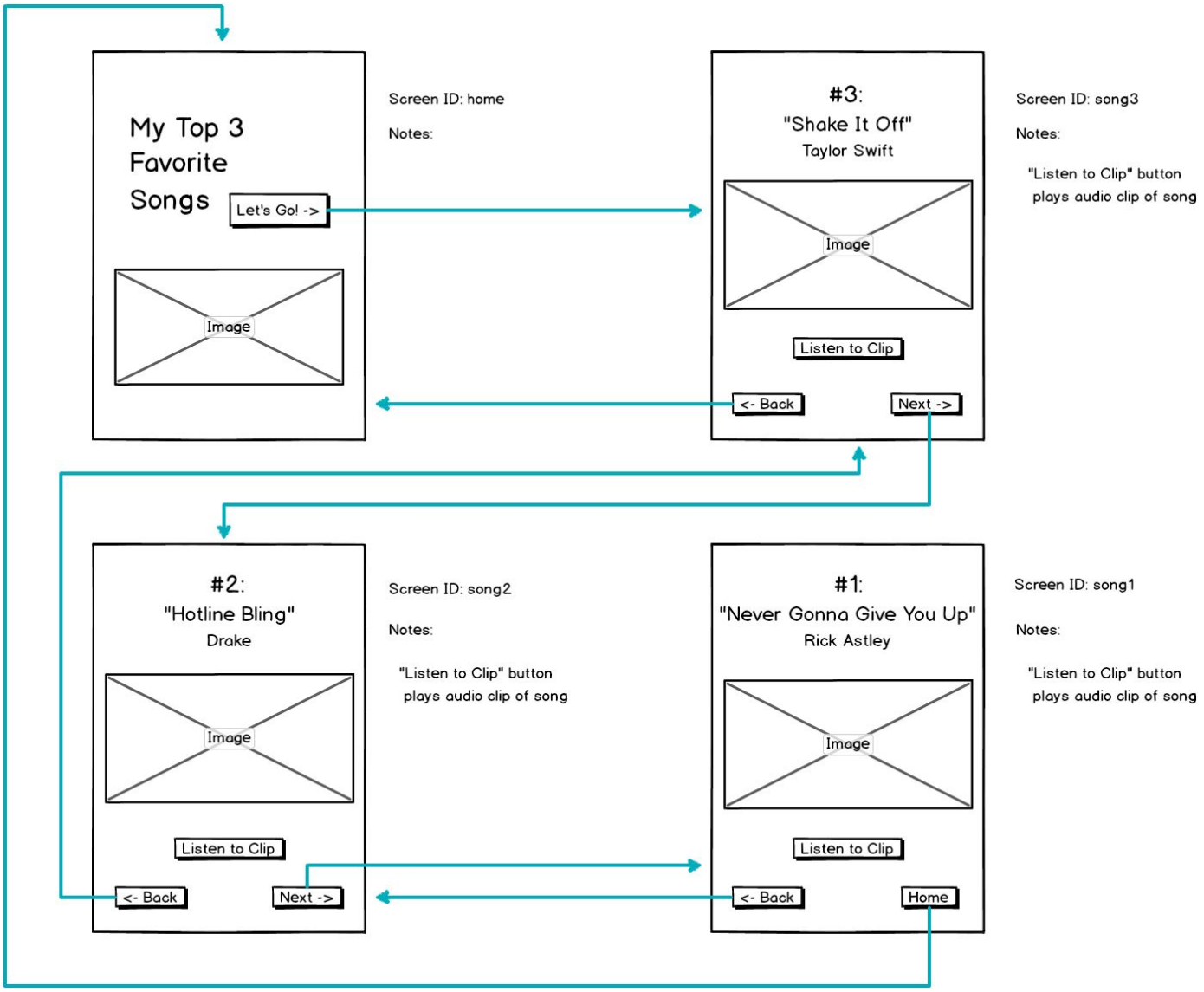
Notes:



Screen ID:

Notes:

Outline Example



Peer Review Rubric

Project being reviewed:

Reviewer:

Criteria	Yes	Almost	No	Comments
Intended purpose of the app is clear				
Project includes at least 4 screens.				
Application includes images, text, and buttons. (bonus points for sound).				
You cannot get “stuck” on any screen. It is always possible to get to the rest of the app.				
Text on screen is clear and descriptive.				
Element IDs are descriptive and meaningful. (Look at the code.)				
The app is visually appealing and the user interface is intuitive to use.				

Unit 5 Lesson 4

Controlling Memory with Variables

Resources

Unit 5 Lesson 5

Building an App: Clicker Game

Resources

Name(s)_____ Period _____ Date _____

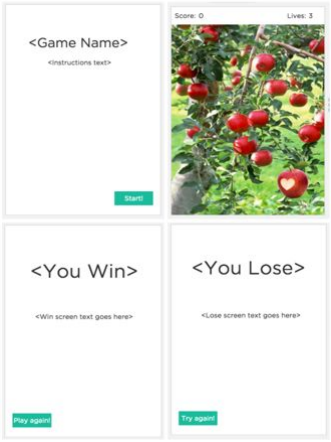
Activity Guide - The Clicker Game

The Clicker Game

You will be **creating your own "clicker" game** similar to the Apple Grabber game you worked on in this lesson. The general object of the game is to click on an element that jumps around every time you click it. You will pick your own theme and decide what the rules are and how to keep score.

Template

A template for the app is provided in Code Studio. The template has **4 screens** and some **basic navigation functionality and event handlers** set up for you.
The game play screen uses the images from the Apple Grabber game, but you should replace these with images related to your chosen theme.



Your **main tasks** are to:

- pick a theme for your game and add appropriate images and styling
- add variables to track some data during game play
- add code to event handlers to update the variables and display appropriately

Requirements

Your application must have the following components:

- Your **game must end** - there must be a way to "win" and a way to "lose."
- You **must use at least one variable** (but you may use as many as you like) to keep track of some data during game play (such as a score, or a number of attempts remaining, number of times a certain element was clicked, etc.).
- Your app will have at least **4 screens:**
    1. **A welcome screen** that explains what your game is and how to play, and lets the user start
    2. **A screen for game play** that displays some data on the screen that updates during play (such as the running score, number of attempts remaining, etc.)
    3. **A "win" screen** to show a congratulatory message if the player "wins" the game
    4. **A "loss" screen** to show when the player "loses" the game
- From the the win/loss screens, it **must** be possible to start the game over without simply re-running the app from scratch; this means resetting all variables and displays back to initial values.
- Your program code should follow good style, particularly by making sure to **create global variables in the first few lines of code** and giving UI elements and variables **descriptive and meaningful IDs.**
- Your user interface should be **intuitive to use.**

Process

- **Choose** the theme of your game: what is jumping around the screen that the user is trying to click? Many themes and metaphors are possible.
- **Program** your app: it's suggested you start by adding some functionality before style. Add one variable into the program, and add code to update and display it properly.
- **Have a peer test it out** to see if there are any more improvements you should make.
- **Make any necessary final adjustments.**
- **Peer Review:** you will review at least one of your classmates' apps using the Peer Review Rubric, and someone will review yours.

Advanced option: use `getTime()` to factor time into your game

In the toolbox you'll see a function called **`getTime()`** which returns a number that's called a *timestamp,* representing the moment in time (to the millisecond) when the function was called. If you store a timestamp in a variable when the player starts the game and then grab a timestamp when they win, you can calculate how long it took and you could factor this into your score. Read the documentation for `getTime()` to see how it works.

Peer Review Rubric

Project being reviewed:

Reviewer:

Criteria	Yes	Almost	No	Comments
Using the App / Playing the game				
Welcome screen explains the game, how to win or lose, and allows the player to start.				
At least one value is shown on screen that changes during game play (for example: a running score).				
It is possible to win, and the app switches to a “win” screen.				
It is possible to lose, and the app switches to a “loss” screen.				
It is possible to start the game over *without* having to restart the program (i.e., the win/loss screens allow you to navigate back to the welcome screen).				
When you start over, variables or other data and displays are properly reset to initial values.				
The app is visually appealing and the user interface is intuitive to use.				
The Code				
The code contains at least one global variable; it appears at or near the first lines of code.				
You can find the line or lines of code in an event handler function that updates a global variable.				
UI elements have meaningful/ descriptive IDs.				

Unit 5 Lesson 6

User Input and Strings

Resources

Name(s)_____ Period _____ Date _____

Activity Guide - Mad Libs®

Create a Mad Libs App

You will be creating an instructional Mad Libs® app that allows a user to submit input of different types (e.g., noun, verb, plural noun, etc.) and then combines them with text you've written to create a single (usually funny) message. The main criteria of your project are as follows:

- Your Mad Libs app should be structured as instructions for completing some task.
- You must include at least three steps in your instructions.
- You must accept at least three pieces of input from your user.
- You must set at least one piece of input to always be uppercase or lowercase.

Example

Here you can see how you might plan out the "Learn to Drive" example found in Code Studio.

Inputs:	**Output Text:**
• **[plural noun]** make lowercase • **[noun 1]** made lowercase • **[noun 2]** made lowercase • **[song]** make lowercase • **[verb]** made uppercase	Learning to drive is a tricky process. There are a few rules you must follow. 1. Keep two **[plural noun]** on the steering wheel at all times. 2. Step on the **[noun 1]** to speed up and the **[noun 2]** to slow down. 3. Your parents will just LOVE it if you play **[song]** on the radio. 4. Make sure to honk your horn when you see **[verb]** on a street sign.

Brainstorm

Use this space to brainstorm components of your own Mad Libs app.

Unit 5 Lesson 7

If-statements Unplugged

Resources

Name(s)_____ Period _____ Date _____

Activity Guide - Will it Crash?

Let's play a game: "Will it Crash?"

Each row in the table below presents a small program that uses if-statements and robot commands. Trace the code and plot the movements of the robot for the 3 scenarios shown to the right of the code. If the robot is directed to move onto a black square, it "crashes" and the program ends. If the robot doesn't crash, then draw a triangle showing its ending location and direction.

There are a few patterns to the ways if-statements are typically used:

- Basic If-statements
- Sequential If-statements
- Basic If-else statements
- Nested If and if-else statements.
- Combinations of all of the above

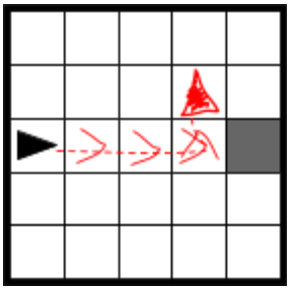
Each section below presents an example of one of these common patterns, followed by a few problems for you to try. For each type **study, and make sure you understand, the example** and why each of the 3 scenarios ends up in the state shown.

EXAMPLE: Basic If-statement

Code is executed sequentially from top to bottom. The code inside the if-block executes ONLY if the condition is true, otherwise the block is skipped and execution picks up on the first line after the if-block.

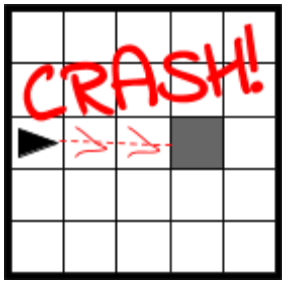
```
MOVE_FORWARD ()
IF (CAN_MOVE (forward))
{
   MOVE_FORWARD ()
   MOVE_FORWARD ()
}
ROTATE_LEFT ()
MOVE_FORWARD ()
```

Scenario 1:

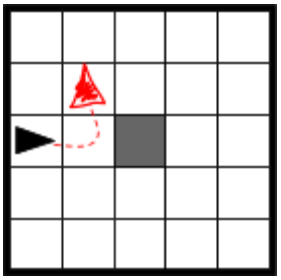


Use the diagram to trace each robot move.

Scenario 2:

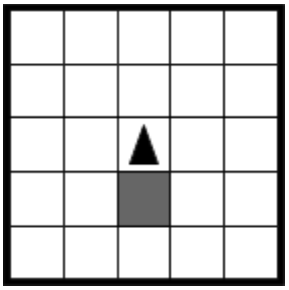


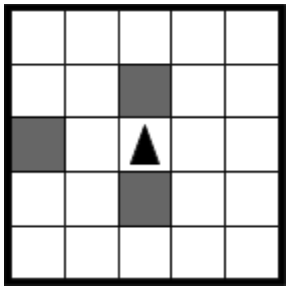
Scenario 3:

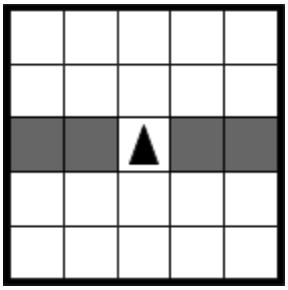


YOU TRY IT - Basic If-statement

```
ROTATE_LEFT ()
IF (CAN_MOVE (left))
{
   ROTATE_LEFT ()
}
MOVE_FORWARD ()
MOVE_FORWARD ()
```



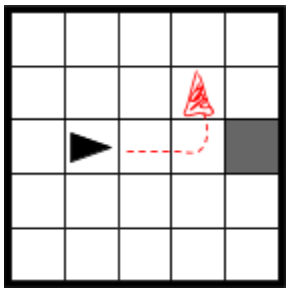


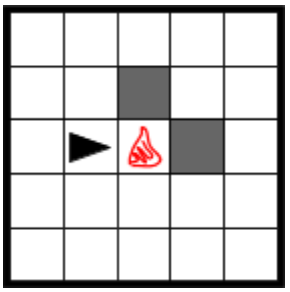


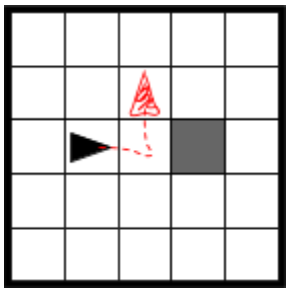
EXAMPLE: Sequential If-statements

Lines of code, including if statements, are evaluated separately, one at a time, in order from top to bottom. An if-block executes ONLY if the expression is true. Note that an earlier if-statement might change the state of the of world for an if-statement that comes later. This makes it hard to predict what will happen unless you trace the robot moves and take each line one at a time.

```
IF (CAN_MOVE (forward))
{
   MOVE_FORWARD ()
}
IF (CAN_MOVE (forward))
{
   MOVE_FORWARD ()
}
ROTATE_LEFT ()
IF (CAN_MOVE (forward))
{
   MOVE_FORWARD ()
}
```

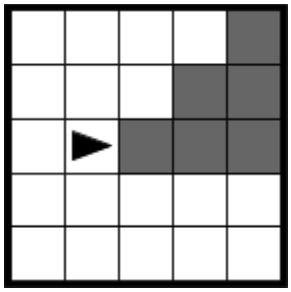


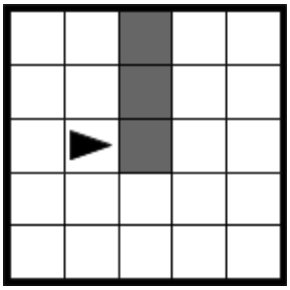


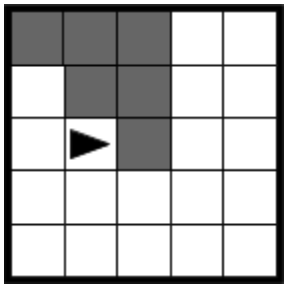


YOU TRY IT - Sequential If-statements

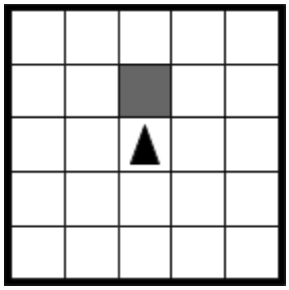
```
ROTATE_LEFT ()
IF (CAN_MOVE (forward))
{
   MOVE_FORWARD ()
}
ROTATE_RIGHT ()
IF (CAN_MOVE (forward))
{
   MOVE_FORWARD ()
}
ROTATE_LEFT ()
IF (CAN_MOVE (forward))
{
   MOVE_FORWARD ()
}
```

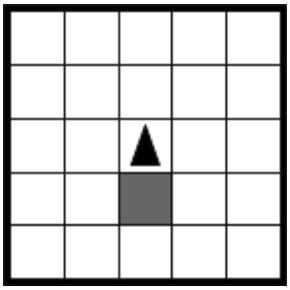


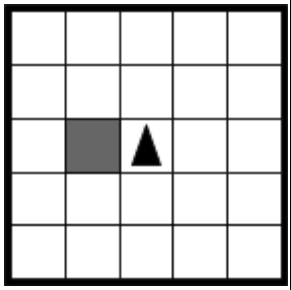




```
IF (CAN_MOVE ( left ))
{
   ROTATE_LEFT ()
   MOVE_FORWARD ()
}
IF (CAN_MOVE ( left ))
{
   ROTATE_LEFT ()
   MOVE_FORWARD ()
}
IF (CAN_MOVE ( left ))
{
   ROTATE_LEFT ()
   MOVE_FORWARD ()
}
```



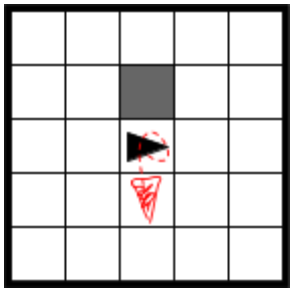


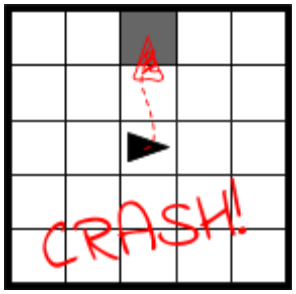


EXAMPLE: If-else Statement

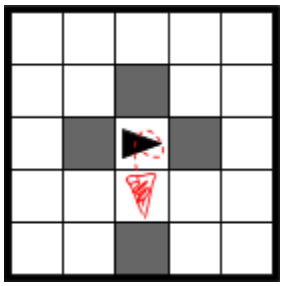
The code in the if-block executes ONLY if the expression is true, otherwise the code in the else block will run. But one or the other must execute. An else statement can be attached to a single if-statement.

```
ROTATE_LEFT ()
IF (CAN_MOVE (forward))
{
   MOVE_FORWARD ()
}
ELSE{
   ROTATE_LEFT ()
   ROTATE_LEFT ()
}
MOVE_FORWARD ()
```



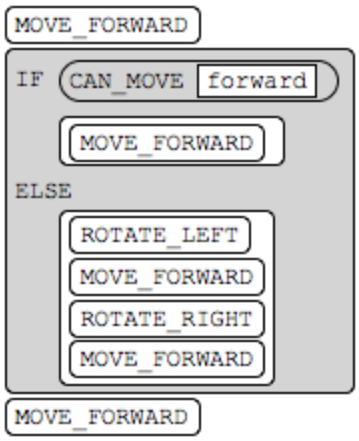


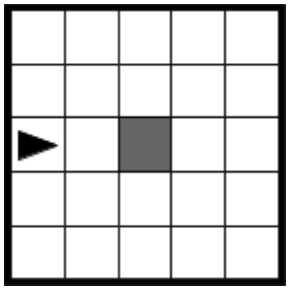
NOTE: Easy to miss MOVE_FORWARD on the last line

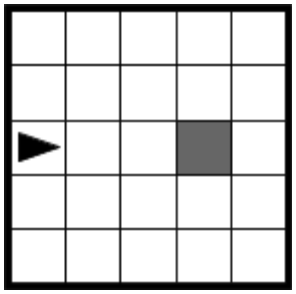


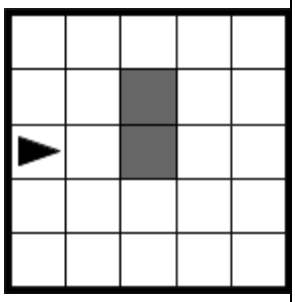
YOU TRY IT - Simple If-Else

Here is a block-based version of a very similar program.

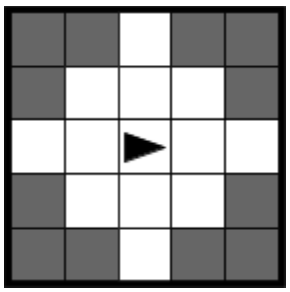


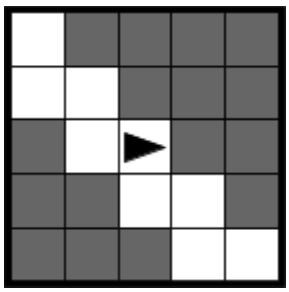


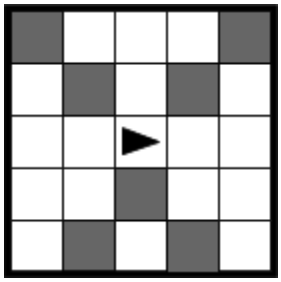




```
IF (CAN_MOVE (left))
{
   ROTATE_LEFT ()
   MOVE_FORWARD ()
}
ELSE
{
   ROTATE_RIGHT()
   MOVE_FORWARD ()
}
IF (CAN_MOVE (right))
{
   ROTATE_RIGHT ()
}
ELSE
{
   ROTATE_LEFT ()
}
MOVE_FORWARD ()
```



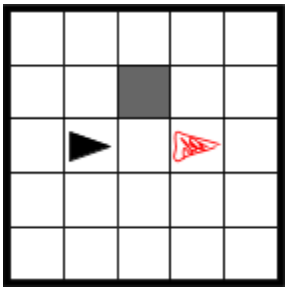


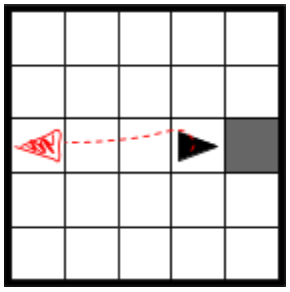


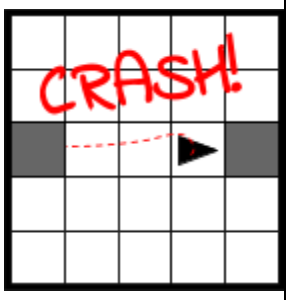
EXAMPLE: Nested Statements

You can put if- and if-else statements inside other if-statements. All previous rules apply, but tracing the code can be tricky.

```
IF (CAN_MOVE (forward))
{
   MOVE_FORWARD ()
}
ELSE
{
    IF(CAN_MOVE( backward ))
    {
       ROTATE_LEFT ()
       ROTATE_LEFT ()
       MOVE_FORWARD ()
    }
    MOVE_FORWARD ()
}
MOVE_FORWARD ()
```

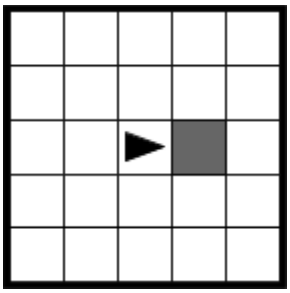


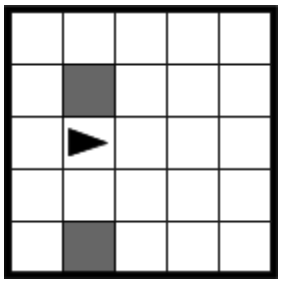


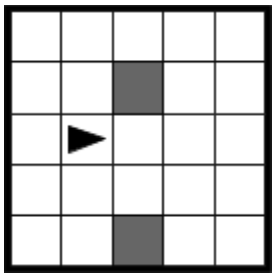


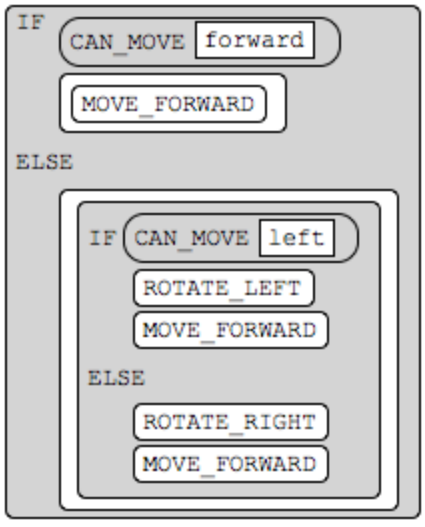
YOU TRY IT - Nested If-statements

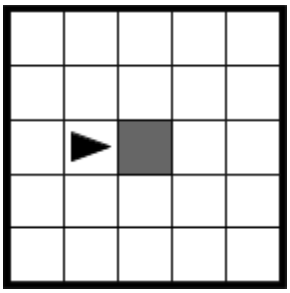
```
IF (CAN_MOVE (forward))
{
   IF (CAN_MOVE (left))
   {
      ROTATE_LEFT ()
   }
   ELSE{
      ROTATE_RIGHT ()
   }
   MOVE_FORWARD ()
}
ELSE
{
   ROTATE_LEFT ()
   ROTATE_LEFT ()
}
MOVE_FORWARD ()
```

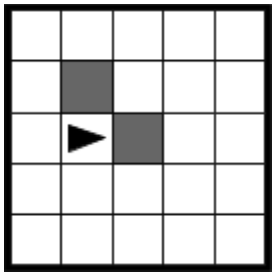


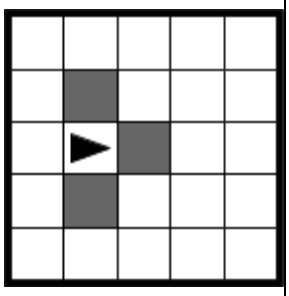




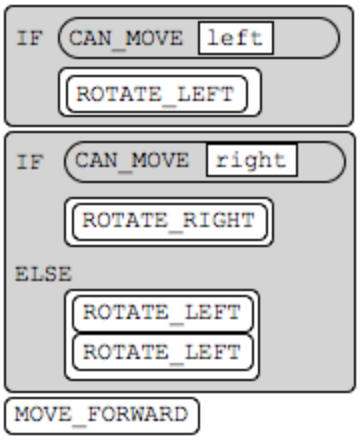


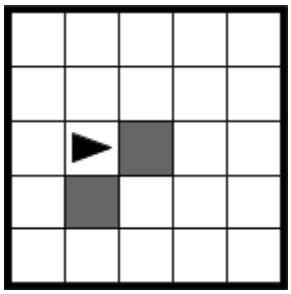


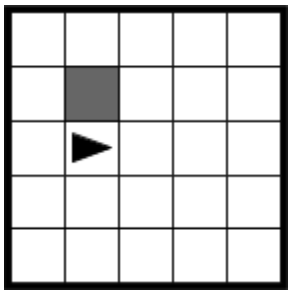


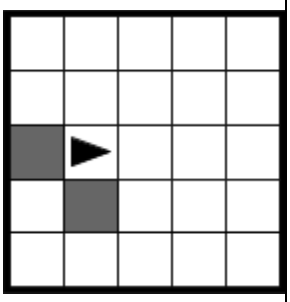


Challenge: putting it all together — if, if-else, sequential if, nested statements

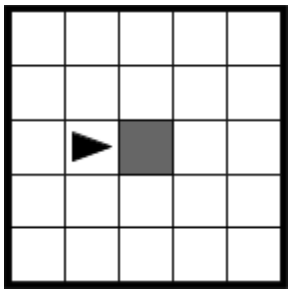


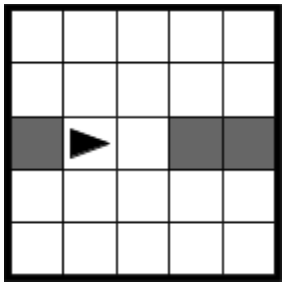


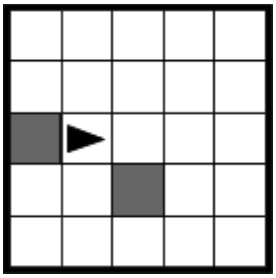




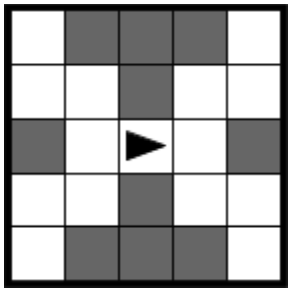
```
IF (CAN_MOVE (forward))
{
   MOVE_FORWARD ()
   IF (CAN_MOVE (left))
   {
      ROTATE_LEFT ()
      IF (CAN_MOVE (right))
      {
         ROTATE_RIGHT()
      }
   }
}
MOVE_FORWARD ()
```

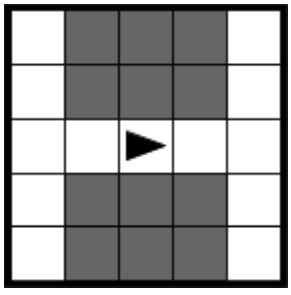


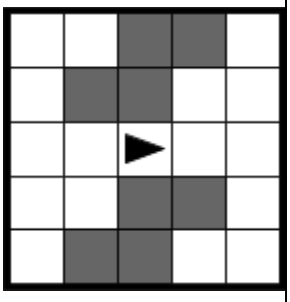




```
IF (CAN_MOVE (forward))
{
   MOVE_FORWARD ()
   IF( CAN_MOVE (left))
   {
      IF( CAN_MOVE (right))
      {
         ROTATE_RIGHT ()
      }
      ELSE
      {
         ROTATE_LEFT ()
      }
   }
   ELSE
   {
      ROTATE_RIGHT ()
   }
   MOVE_FORWARD ()
}
ELSE
{
   ROTATE_LEFT ()
   ROTATE_LEFT ()
}
MOVE_FORWARD ()
```





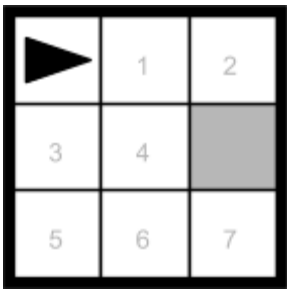


Now you try it!

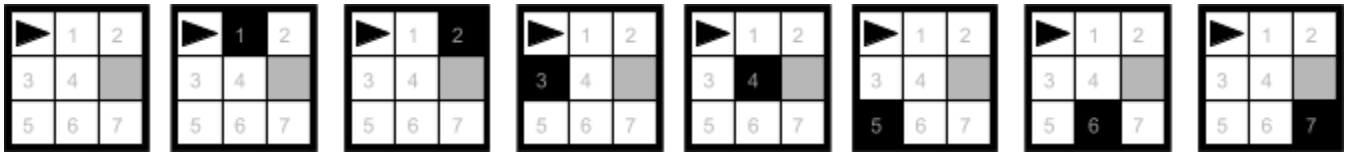
Now that you've had a bunch of practice reading and tracing code with if-statements, try writing your own pseudocode robot program that uses if-statements.

Problem statement:

Write a program to make the robot end up on the target gray square facing any direction...**but** your code must be able to handle the possibility of an obstacle that could appear in any one of the other squares (i.e. squares that aren't the start or target squares - numbered 1-7 in the diagram)

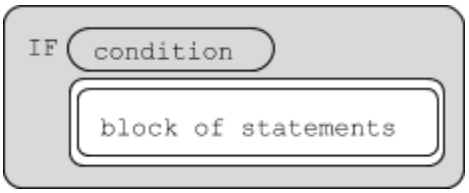


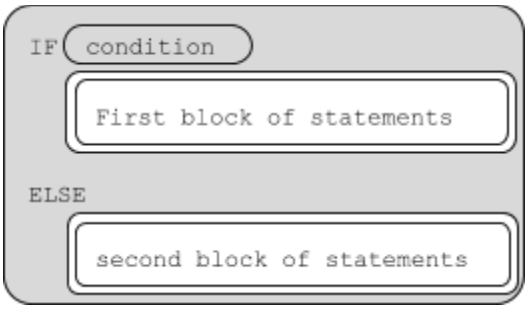
You must write the code without knowing ahead of time where the obstacle will be. In other words, you must *write one program* that can handle any possibility that might occur. For this exercise, there 8 possible locations where obstacle might be, we'll call them scenarios 0-7:

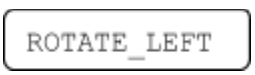


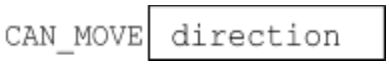
Write your program by hand below and test it by tracing it against each of the 8 possible scenarios. Your code should get the robot to the target no matter where the obstacle appears.
Goal: When the program ends, the robot is on the gray square -- it can be facing any direction
Tip: *When hand-writing code you don't need to follow the pseudocode syntax strictly, as long as your intent is clear. For example, using abbreviations and/or omitting the curly-braces and just indenting is fine.*

Selection	

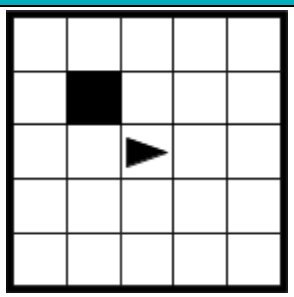
Text
`IF (condition)`
`{`
`  <block of statements>`
`}`
Block
 
	The code in block of statements is executed if the Boolean expression condition evaluates to true; no action is taken if condition evaluates to false.
	
	Commentary:
	An if-statement might execute some code or it might not. It checks to see IF something is true (the "condition"). If it is true, then run some code contained inside the if-statement. Otherwise, if the condition is false, *just ignore the whole block* and pick up executing the code that comes after it.

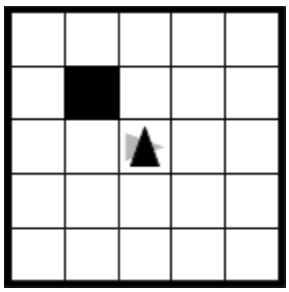
Text
`IF (condition)`
`{`
`  <first block of statements>`
`}`
`ELSE`
`{`
`  <second block of statements>`
`}`
Block
 
	The code in first block of statements is executed if the Boolean expression condition evaluates to true; otherwise the code in second block of statements is executed.
	
	Commentary:
	An if-else statement is an "either or". You are saying: "I want to run *either* this block of code *or* this other one". An if-else *guarantees* that one of the two blocks of statements *will* execute. If the condition is true, then run the first block of statements and ignore the other block. If the condition is FALSE, then ignore the first block and run the second one.
	
	Important Note: While an if statement can stand on its own (see above), an ELSE statement cannot - it *must* be "attached" to an IF statement, and no lines of code can be inserted between an if and else.

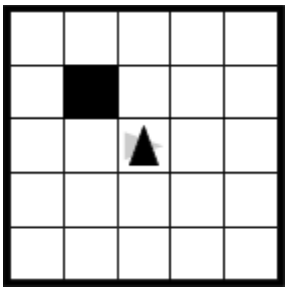
Robot	
Text: `MOVE_FORWARD ()` **Block**: 	The robot moves one square forward in the direction it is facing.
Text: `ROTATE_LEFT ()` **Block:** 	The robot rotates in place 90 degrees counterclockwise (i.e., makes an in-place left turn).
Text:	The robot rotates in place 90 degrees clockwise (i.e., makes an in-place

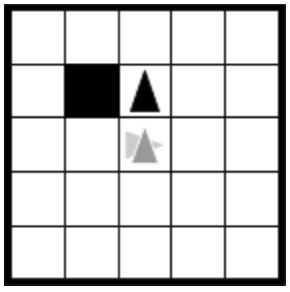
`ROTATE_RIGHT ()` **Block**: 	right turn).
Text: `CAN_MOVE (direction)` **Block:** 	Evaluates to **true** if there is an open square one square in the direction relative to where the robot is facing; otherwise evaluates to **false**. The value of direction can be left, right, forward, or backward. **Commentary**: CAN_MOVE is a *condition* that you can use in an if-statement - it will either be true or false. Example: IF(CAN_MOVE (forward)) is a way to check if the space in front of the robot is open.

Code	Robot Scenario	Commentary

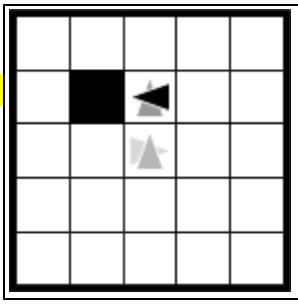
1 ROTATE LEFT ()
2 IF (CAN MOVE (forward))
3 {
4 MOVE FORWARD ()
5 }
6 ROTATE LEFT ()
7 IF (CAN MOVE (forward))
8 {
9 MOVE FORWARD ()
10 }
		**Before:**
		The starting scenario before any lines have been executed.
		
		Before reading the rest of the page, you might want to try to predict where the robot will end up.

1 > ROTATE LEFT ()
2 IF (CAN MOVE (forward))
3 {
4 MOVE FORWARD ()
5 }
6 ROTATE LEFT ()
7 IF (CAN MOVE (forward))
8 {
9 MOVE FORWARD ()
10 }
		**Line 1 executes:**
		Robot turns 90 degrees to the left

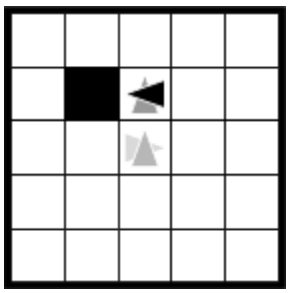
1 ROTATE LEFT ()
2 > IF (CAN MOVE (forward))
3 {
4 MOVE FORWARD ()
5 }
6 ROTATE LEFT ()
7 IF (CAN MOVE (forward))
8 {
9 MOVE FORWARD ()
10 }
		**Line 2 executes:**
		Check to see if the robot can move forward. TRUE - it can. So the code is said to “enter the if-statement block” ...

1 ROTATE LEFT () 2 IF (CAN MOVE (forward)) **3 {** **4 > MOVE FORWARD ()** **5 }** 6 ROTATE LEFT () 7 IF (CAN MOVE (forward)) 8 { 9 MOVE FORWARD () 10 }		**Line 4 executes:** Since the condition was TRUE, execute the line(s) of code inside the if-statement - in this case: move_forward. **NOTE: Lines 3 and 5:** The “curly braces” { } encapsulate the lines of code to execute if the condition is true.
1 ROTATE LEFT () 2 IF (CAN MOVE (forward)) 3 {		**Line 6 executes:** Rotate the robot 90 degrees to the left

```
4       MOVE FORWARD ()
5    }
6 >  ROTATE LEFT ()
7    IF (CAN MOVE (forward))
8    {
9       MOVE FORWARD ()
10   }
```



```
1    ROTATE LEFT ()
2    IF (CAN MOVE (forward))
3    {
4       MOVE FORWARD ()
5    }
6    ROTATE LEFT ()
7 >  IF (CAN MOVE (forward))
8    {
9       MOVE FORWARD ()
10   }
```

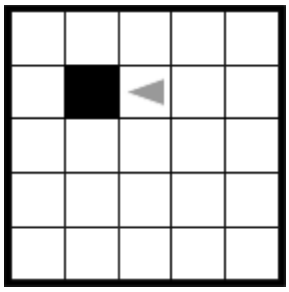


Line 7: Check to see if the robot can move forward now...

FALSE - it cannot move forward. The condition fails, so IGNORE the block of code in the curly braces. The robot actually does nothing here.

NOTE: the condition of an if-statement is *always* executed - the blocks inside it though, only run if the condition is TRUE.

```
1    ROTATE LEFT ()
2    IF (CAN MOVE (forward))
3    {
4       MOVE FORWARD ()
5    }
6    ROTATE LEFT ()
7    IF (CAN MOVE (forward))
8    {
9       MOVE FORWARD ()
10   }
11 >
```



End state:

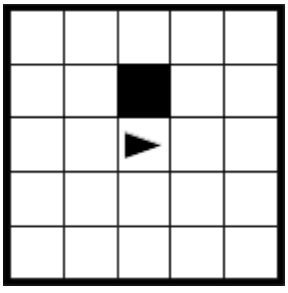
Since there are no more lines of code to execute, the program ends in this state.

All lines have been processed. If there were any code starting on line 11, we would attempt to execute that next.

You try it - Same Code, Different Scenario

Because the code that runs depends on the conditions at the time of execution, the same program might end up with different results given a different starting scenario. Try tracing through the same program again, but with a different initial robot setup. What happens? Where does the robot end up?

```
1    ROTATE LEFT ()
2    IF (CAN MOVE (forward))
3    {
4       MOVE FORWARD ()
5    }
6    ROTATE LEFT ()
7    IF (CAN MOVE (forward))
8    {
9       MOVE FORWARD ()
10   }
```



Unit 5 Lesson 8

Boolean Expressions and "if" Statements

Resources

Name(s)_____ Period _____ Date _____

Activity Guide - Flowcharts

The Types of Questions You Can Ask a Computer

When we make decisions as humans, we usually consider complex sets of conditions, like the circumstances surrounding the decision, past experience, or even just how we feel. When we want to write programs that make decisions, we need to represent our decisions in a way the computer can understand.

Decision Making with a Computer

- The computer evaluates some statement (also called an expression) that can only be **true** or **false**. This means that the result can be represented by a single bit.
- This result determines which of two parts of the program will run next.

The types of statements that a computer can evaluate as either true or false are also limited by the fact that information stored in the computer is binary. As a result, most true/false statements you will use in your programs are **comparing two values** in the computer's memory. Here are the **most common types of comparisons** you'll see:

__ is equal to __

__ is not equal to __

__ is greater than __

__ is less than __

__ is greater than or equal to __

__ is less than or equal to __

Creating Computer Questions

It can take a little practice to convert a **question** you might ask as a human into a **binary statement** that can be evaluated by a computer. Here are some examples:

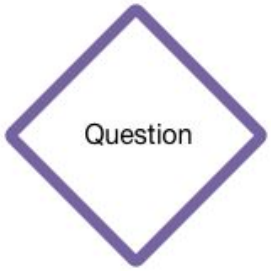
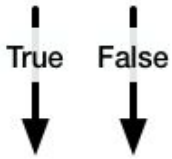
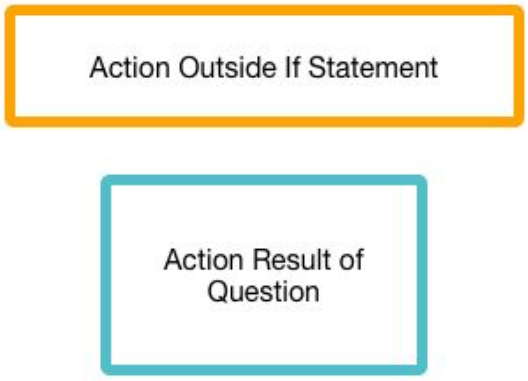
The Human Question	The Computer Question		The Human Question	The Computer Question
Are you old enough to drive?	**Is age greater than or equal to 16?**		**Is it lunch time?**	**Is the time equal to 12?**
Did I fail the test?	**Is grade less than 70?**		**Is it the weekend?**	**Is the day equal to Saturday? If not, is the day equal to Sunday?**
Was it a tied game?	**Was the home team score equal to the away team score?**		**Is this person a teenager?**	**Is age greater than 12? If yes, is age less than 20?**

Your Turn! Change the following human questions into computer questions:

Human Question	Computer Question
Could this water freeze right now?	
Did I get a baker's dozen (13) of donuts?	
Am I old enough to vote?	
Were you born before the millennium?	
Are you an only child?	

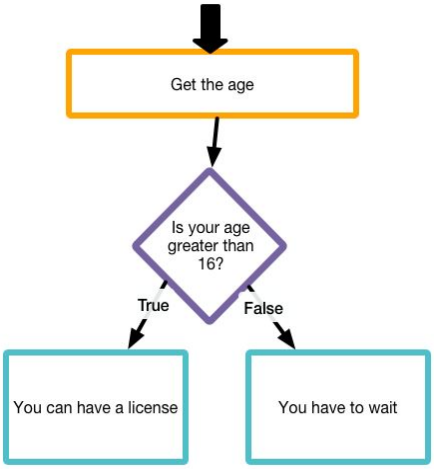
Flowchart Components

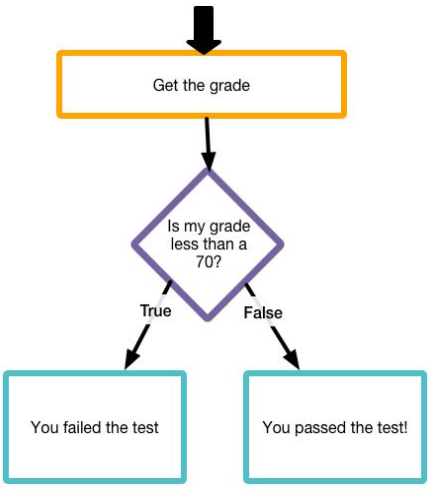
One way that programmers plan complex programs with decisions in them is to create diagrams called **Flowcharts**, which demonstrate the logic they want for their program. Flowcharts have a couple different components. Each component is a different shape to signify its purpose. Check out the table below as a build to flowcharts.

Component	Purpose
	A **diamond** represents a decision to be made within your program, based on a binary question or expression (one that has only two possible responses). We will see more about binary questions later in this activity.
	**True** and **False arrows** designate the paths taken, based on the result of a decision (diamond). Note once again that every decision may have only 2 possible paths that result from it, one for true and one for false.
	A **rectangle** represents an action that is performed. Some actions may be performed as the direct result of a decision, while others are performed every time a program is run. The examples we'll see usually use the following style: ● Action that is always performed: wide orange rectangle ● Action performed based on decision: narrow blue rectangle **When you draw your own flowcharts, you may use a single style of rectangle for all actions.**
	A **simple arrow** indicates that we are moving from one action to the next without considering any decision. These will generally be used to link a set of actions to be completed one after the other.

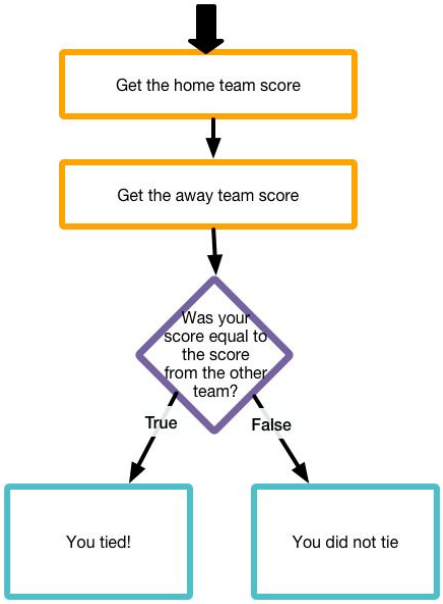
The Results of Questions You Can Ask a Computer

So far we have only created questions, but in order to use them to make decisions, we need to specify actions that get performed depending on the result. Check out the examples of flowcharts below:

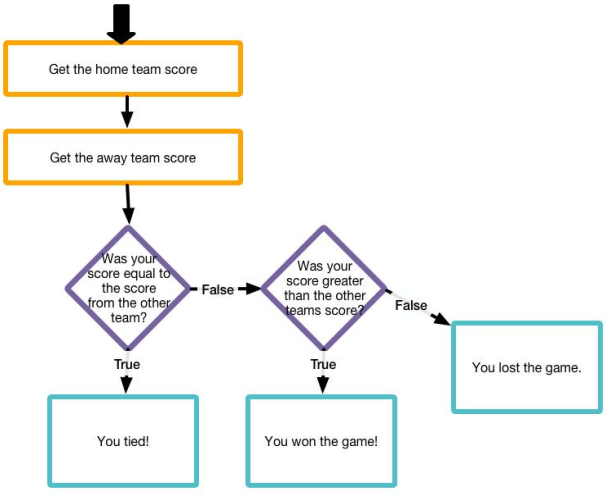


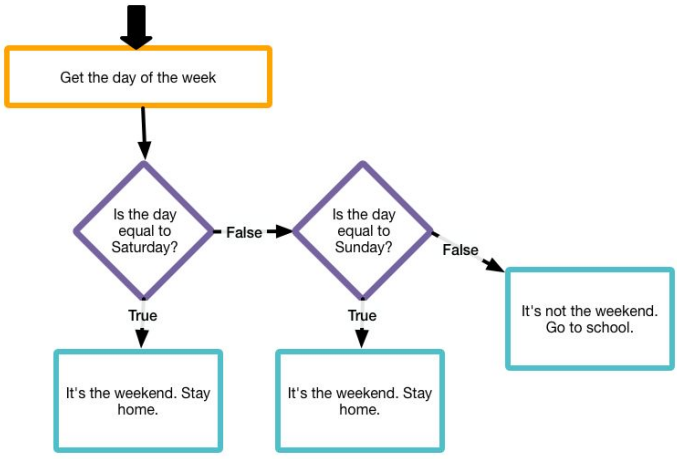


Sometimes you need to get multiple pieces of information to make your decision.



Sometimes you want to ask a follow up question to your original question based on a certain answer.





Your turn!
Refer back to your Human-Computer Questions you worked on before. Can you make flow charts for them? The questions are copied below for your reference.

Human Question	Flow Chart
Could this water freeze right now?	
Did I get a baker's dozen of donuts?	

Human Question	Flow Chart
Am I old enough to vote?	
Were you born before this millennium?	
Are you an only child?	

Unit 5 Lesson 9

"if-else-if" and Conditional Logic

Resources

Name(s)_____ Period _____ Date _____

Worksheet - Compound Conditionals

Chained and Nested Conditionals

In order to express more complex decisions, we've used **chained conditionals** (`else` / `else-if`) and **nested conditionals** (`if` statements inside of `if` statements). These are powerful tools which **can be combined to express any complex boolean condition.**

Practice: Using what you know about writing `if`, `if-else`, and `if-else-if` statements, write ***pseudocode*** to accomplish the tasks given below.[1] Assume that you are writing an application which asks the user to input the day of the week (a string) and his age (a number) and that these are stored in variables called `day` and `age`.

- **EXAMPLE:** If the day is Friday, write, "Thank goodness it's Friday." Otherwise, write, "How long till Friday?"

```
if day == Friday
        write "Thank goodness it's Friday."
else
        write "How long till Friday?"
```

- If the day is any day but "Monday" the program writes, "At least it's not Monday."

- If the user is a teenager, the program writes, "You are a teenager." Otherwise, it writes, "You are not a teenager." (Remember that you can assume there is a variable called **age** that you can refer to.)

- If the day is a weekend day, the program writes, "It is the weekend." Otherwise, it writes, "It is not the weekend."

[1] Writing **pseudocode** means you should use the structural conventions of a programming language, but it is intended for human reading rather than machine reading. Even if pseudocode sometimes looks very close to a real programming language, there is no expectation that it be syntactically perfect - it's about expressing and communicating your ideas for a program.

Improving Conditionals with Boolean Operators

As we noted before, **nested and chained conditionals can be used to create any possible boolean condition**. Unfortunately, however, the resulting code will often be long, cumbersome, and redundant. To help simplify the expression of complex conditionals we will be introducing three new logical operators: **NOT**, **AND**, and **OR**.

NOT

When creating more complex boolean expressions you will encounter instances when you want to know if a statement **is false** rather than true. You have seen the **!=** ("is not equal") operator before, but previously we might also have accomplished this by using the `else` statement. Sometimes, expressing logical conditions is a challenge and you might feel like you need to write an `if` statement *just* to use the `else` clause. Here is simple example.

Example:

```
if(day == "Monday"){
}
else {
  write("At least it's not Monday.");
}
```

But writing an empty `if` statement just to use the `else` clause is bad practice (some would call it ugly code). There are many instances in programming where an operation doesn't have a convenient logical inverse (like **==** and **!=**) and we simply want to say **if** some condition is **NOT** true. The single **!** is the **NOT** operator. Here's an example:

Operator	Description	Truth Table	JavaScript Syntax
NOT statement1	Evaluates to the opposite truth value of the statement provided as input	*expr* | *!(expr)* T | F F | T	`!`

Note the **truth table** shown above. A truth table is a simple tool used to show how every possible value of the input will be treated by a logical operation. Here we use "*expr*" to stand in for any expression that might evaluate to **true** or **false**. This table shows that when **NOT** is applied to a boolean expression, the result is the opposite truth value.

Also note that the **JavaScript syntax** for NOT is a single exclamation point. As good practice, you should place the expression to which you want to apply the NOT within parentheses. Here's our example from before, simplified with the **NOT** operation.

Updated Example:

```
if(!(day == "Monday")){
  write("At least it's not Monday.");
}
```

Practice: Circle whether each expression evaluates to `true` or `false`. The variable `day` is initialized as shown.

```
var day = "Monday";
```

1. `!(day == "Monday")` (Evaluates to `true`) (Evaluates to `false`)
2. `!(2 > 3)` (Evaluates to `true`) (Evaluates to `false`)
3. `!(day == "Tuesday")` (Evaluates to `true`) (Evaluates to `false`)

AND

When we've wanted to check whether **multiple conditions are true,** we have been making use of nested conditionals. Here's how we might approach determining whether a user is a teenager, based on the variable `age`.

Example:

```
if(age >= 13){
  if(age < 20){
    write("You are a teenager.");
   } else {
     write("You are not a teenager.");
   }
} else {
  write("You are not a teenager.");
}
```

Not only is this code long, but there is redundancy introduced by having to write "You are not a teenager." at two different points. We can improve the expression of this complex condition with the **AND** operator.

Operator	Description	Truth Table	JavaScript Syntax
statement1 **AND** statement2	Evaluates to true only when both boolean statements it connects are true	*expr1* | *expr2* | *expr1 && expr2* T | T | T T | F | F F | T | F F | F | F	`&&`

Notice that the **truth table** for **AND** includes columns for two statements, "a" and "b," and that every possible true / false combination of their values is shown in the columns.

When using the **AND** operator, we can significantly improve the code we wrote in the example above (see below). We've actually improved the code in two ways. First, notice that we've removed the redundant lines of code. Second, since all of the logic of the conditional can now be found in one line, it is easier to read and understand, making it a better expression of what you are trying say.

Updated Example:

```
if((age >= 13) && (age < 20)){
  write("You are a teenager.");
} else {
   write("You are not a teenager.");
}
```

It is recommended that you **place each boolean expression inside of its own parentheses**, as shown here. This ensures that the expressions are evaluated in the order you intend.

Practice: Circle whether each expression evaluates to `true` or `false`. The variable `age` is initialized as shown.

```
var age = 16;
```

1. `(age > 12) && (age < 18)` (Evaluates to `true`) (Evaluates to `false`)
2. `(age != 12) && (age > 18)` (Evaluates to `true`) (Evaluates to `false`)
3. `(age != 12) && (2 > 3)` (Evaluates to `true`) (Evaluates to `false`)

OR

When we've wanted to check whether **at least one** of many conditions is true we have been making use of chained conditionals. Here's how we might approach determining whether it is the weekend based on the variable "day".

Example:

```
if(day == "Saturday"){
  write("It is the weekend.");
} else if(day == "Sunday"){
  write("It is the weekend.”);
} else {
  write("It is not the weekend.");
}
```

Once again, this code is long and introduces redundancies. We can improve the expression of this complex condition with the **OR** operator.

Operator	Description	Truth Table	JavaScript Syntax
statement1 **OR** statement2	Evaluates to true as long as at least one of the boolean statements it connects is true	*expr1* | *expr2* | *expr1 || expr2* T | T | T T | F | T F | T | T F | F | F	`||`

The syntax for **OR** in JavaScript is **two vertical pipe characters** that you probably have not used very much outside of programming. When using the **OR** operator, we can significantly improve the code we wrote in the example above. Once again we've removed redundancy and improved the overall readability of our code.

Updated Example:

```
if((day == "Saturday") || (day == "Sunday")){
  write("It is the weekend.");
} else {
  write("It is not the weekend.");
}
```

Notice that once again we have **placed each boolean expression inside of its own parentheses**. This is not strictly necessary, but it makes it clear which expressions are being grouped together.

Practice: Circle whether each expression evaluates to `true` or `false`. Assume the variable day is initialized as shown.

```
var day = "Monday";
```

1. `(day == "Mon") || (day == "Monday")` (Evaluates to `true`) (Evaluates to `false`)
2. `(day == "Tues") || (day == "Tuesday")` (Evaluates to `true`) (Evaluates to `false`)
3. `(day == "Tues") || (5 < 10)` (Evaluates to `true`) (Evaluates to `false`)

Activity

Evaluating Compound Conditionals: Determine if the follow statements evaluate to `true` or `false`. Assume in every case that the two variables `age` and `day` have been initialized with the values shown.

```
var age = 16;
var day = "Monday";
```

4. `(age > 10) && (age < 20)` (Evaluates to `true`) (Evaluates to `false`)

5. `!(age > 10)` (Evaluates to `true`) (Evaluates to `false`)

6. `(day == "Tuesday") || (age < 12)` (Evaluates to `true`) (Evaluates to `false`)

7. `!((age == 16) || (day == "Monday"))` (Evaluates to `true`) (Evaluates to `false`)

8. `!((age == 16) && !(day == "Monday"))` (Evaluates to `true`) (Evaluates to `false`)

Challenge Problems

9. `((age == 16) && (day == "Monday")) && (day == "Tuesday")`

   (Evaluates to `true`) (Evaluates to `false`)

10. `((age == 16) && (day == "Monday")) || (day == "Tuesday")`

    (Evaluates to `true`) (Evaluates to `false`)

11. `((age > 10) && ((age + 5) > 20))`

    (Evaluates to `true`) (Evaluates to `false`)

Unit 5 Lesson 10

Building an App: Color Sleuth

Resources

Color Sleuth and the AP Create PT

How the Color Sleuth Project Meets the AP Create Performance Task Requirements

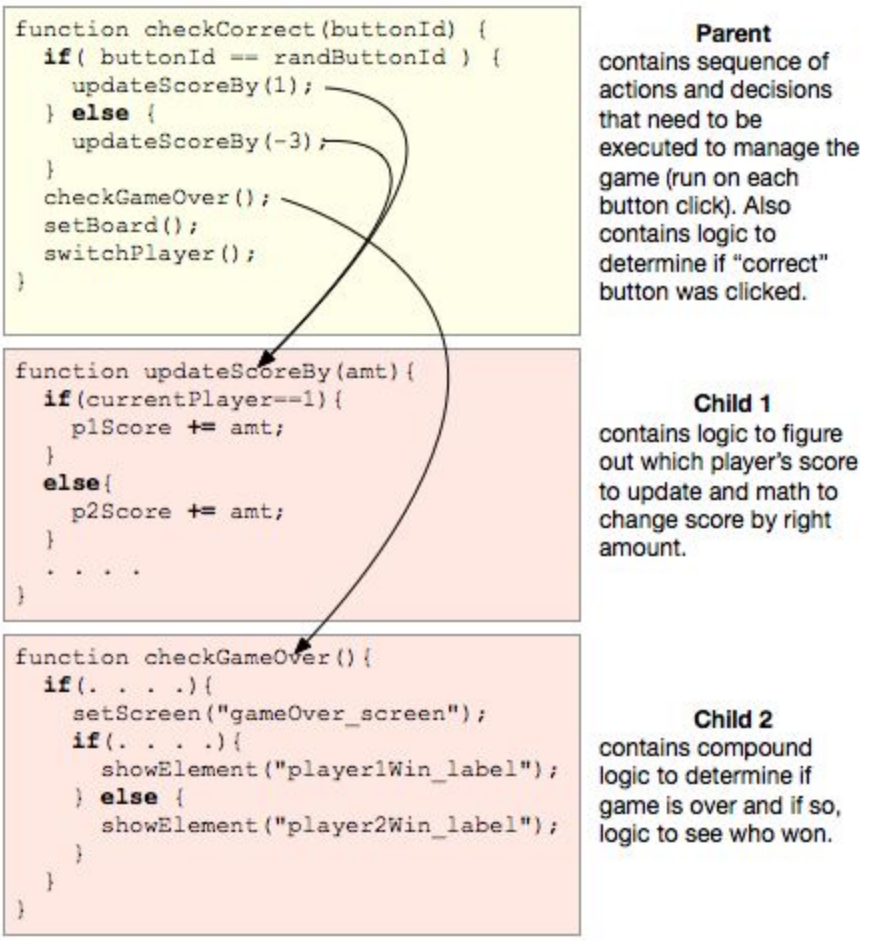
The Color Sleuth App, written as suggested in this lesson, is an example of a program that can meet the minimum bar for the AP Create Performance Task. Here's how.

Iterative Design Process (rows 2-3 of AP Create Task Scoring Guidelines) - for the AP you must discuss your overall "incremental and iterative development process" as well as two points along the way where you saw an opportunity, or some difficulty, that you worked out and it ended up in the final program.

Alexis and Michael's discussion throughout the tutorial is an excellent example of working collaboratively to iteratively write a program - they wrote in small parts, testing each part along the way, modifying it, or adding new functionality. The realization to use a parameterized function is a good opportunity to talk about. And any time they re-organized the code or changed their course of action is a response to some difficulty they were trying to overcome.

Algorithms (rows 4-6) - For the AP you need to show code of an algorithm that includes two or more algorithms where at least one of the included algorithms contains mathematical or logical concepts.

For a program structured like Color Sleuth, the main algorithm (or "parent") and included algorithms ("children") will likely be spread out across separate functions. An example of a choice that could be made along with arguments for the written responses is shown in the diagram. Note: the student would select all three of these functions as the "algorithm".



Abstraction (rows 7-8) - for the AP the code must contain a student-written abstraction that helps manage the complexity of the program. The functions in this program are strong evidence of using abstraction to manage complexity in the code.

A function with a parameter is often a good one to choose because the fact that the function has a parameter means that the problem has been abstracted so it can handle different types of input. `checkCorrect(buttonId)` and `updateScore(amt)` would be good choices that you should be able to justify easily in the written responses about how they help manage complexity.

Name(s)_____ Period _____ Date _____

Color Sleuth Project Rubric

Overview

The Color Sleuth lesson walks you through a scenario of two fictional students planning and writing code for an app of their design. You are asked to mimic the code they write by transcribing from the pseudocode sketches they make along the way (the tutorial). At the end you're on your own to write code that decides how the came ends and who wins.

Criteria	Yes	Almost	No	Comments
Features Covered in Tutorial Students are guided through building these components of the game by the tutorial. These features should be clear from quickly playing the game and reviewing the code.				
Game Board Changes: Buttons change colors when any one of them is clicked, one button has a slightly different color				
Score Updates: Clicking buttons updates and displays score for the correct player				
Switching Turns: Players switch turns after each click - whose turn it is clearly indicated				
Code Style: Code is neatly organized and broken into functions in the style suggested by the tutorial.				
Features Student(s) Write Independently Students are asked to independently write code to end the game. Assessing this feature may require reading code, playing the game to its conclusion, or changing the starting score of the game to force the game to end.				
Game over: UI indicates game is over				
Game over: Code written so that game *can* end - contains logic to check whether game is over.				
Who won: UI indicates winning player				
Who won: Code contains logic to determine which player won (if game is over).				

Unit 5 Lesson 11

While Loops

Resources

Name(s)_____ Period _____ Date _____

Activity Guide - Flowcharts with `while` Loops

Introduction

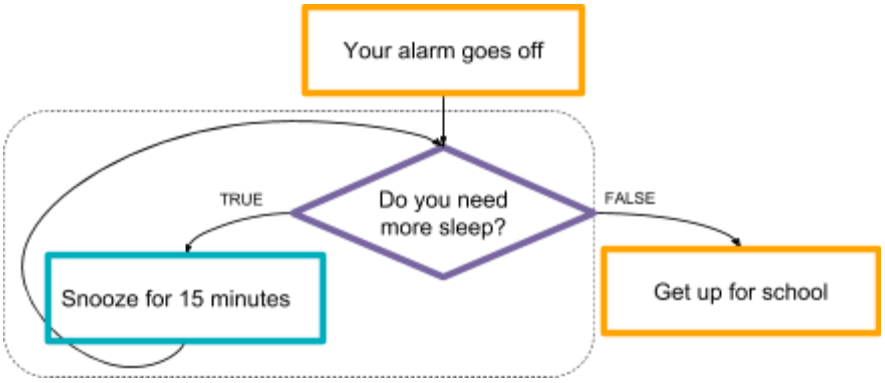
We are going to be learning a new programming structure today called **`while`** loops. **`while` loops allow us to control program flow by repeating a set of commands until a condition is met.** When we control program flow it is often helpful to think about the ideas in a visual way first. You will use flowcharts to begin thinking about **`while`** loops.

Flowchart Components

As a reminder we have included the flowchart components here as a reference for you as you work on this sheet.

Real-Life `while` Loops

Each flowchart below contains a "loop" that runs "while" a condition is true. Investigate each flowchart, mark how many times you believe the loop will run on the line provided, and then justify your reasoning.
Note: There may be many possible answers, so pick the one you believe makes the most sense.



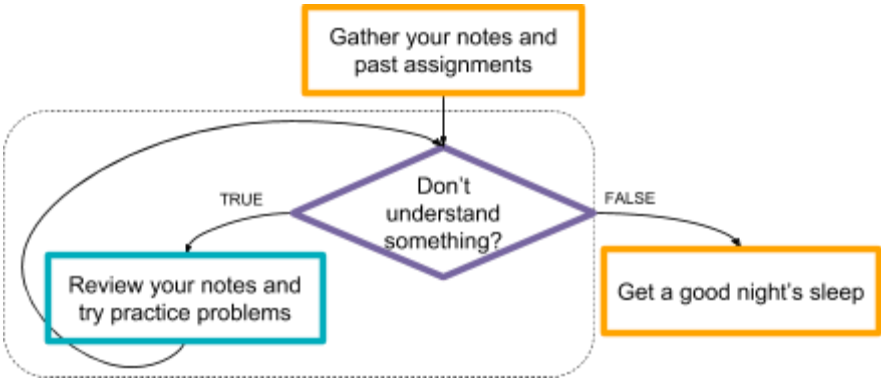
How many times will the loop run?

|----------|

Never A few times Forever

Justification:_____

_____



How many times will the loop run?

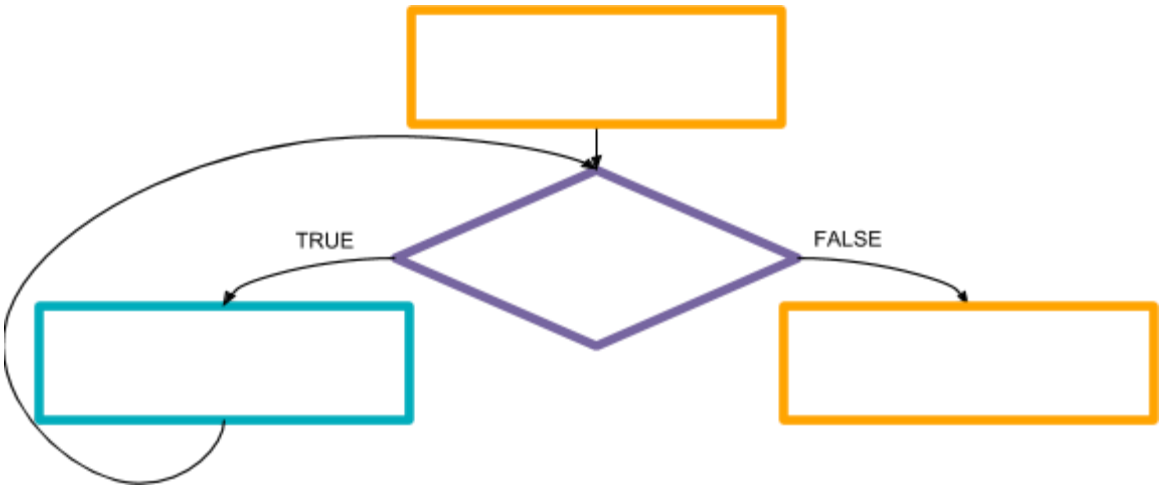
|----------|

Never A few times Forever

Justification:_____

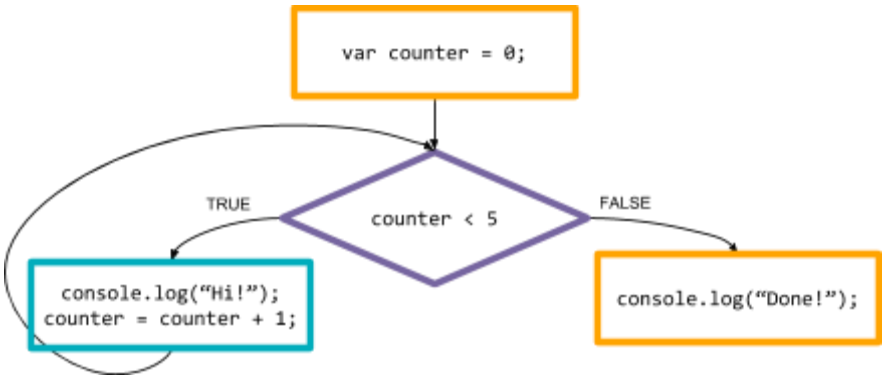
_____

Complete this flowchart with a real-life `while` loop of your own choosing. Exchange with a partner and see how many times they think your loop would run.

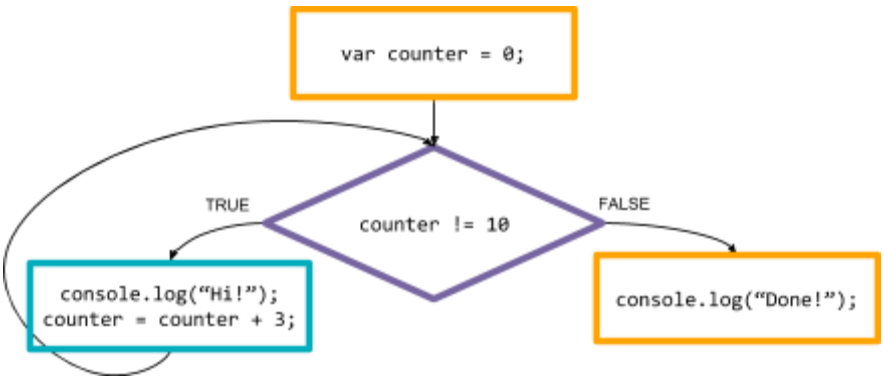


Programming with `while` Loops

The next two examples begin making the transition from real-life `while` loops to ones you might see while programming. **Use the space provided to write the output that would be generated from the program.**



Output:


var counter = 0;
counter != 10
TRUE
FALSE
console.log("Hi!");
counter = counter + 3;
console.log("Done!");

Output:

Unit 5 Lesson 12

Loops and Simulations

Resources

Name(s)_____ Period _____ Date _____

Worksheet - Flipping Coins

Flip a coin until you get 5 total heads. How many flips did it take? _____

Record your flips in the space below by writing “H” or “T” for each flip.

Flip a coin until you get 3 heads in a row. How many flips did it take? _____

Record your flips in the space below by writing “H” or “T” for each flip.

Predict

The **most flips** it took someone (including you) to get **5 total heads** _____

The **fewest flips** it took someone (including you) to get **5 total heads** _____

The **most flips** it took someone (including you) to get **3 heads in a row** _____

The **fewest flips** it took someone (including you) to get **3 heads in a row** _____

Make a Hypothesis

We’ll be developing a **simulation** that flips coins for us. As a result we can test how many flips it takes to get many more total heads and longer strings of heads. Before we make our simulation, predict the following.

What is the **most flips** you expect to see in order to get **10,000 total heads**? _____

What is the **fewest flips** you expect to see in order to get **10,000 total heads**? _____

Explain your reasoning:

What is the **most flips** you expect to see in order to get **12 heads in a row**? _____

What is the **fewest flips** you expect to see in order to get **12 heads in a row**? _____

Explain your reasoning:

Extend Your Hypothesis

You may have observed that flipping tens of thousands of coins takes some time on a computer. If we want to simulate even larger problems, it may be difficult for even a computer to complete the full simulation in a time frame that makes sense. Luckily, you should have developed some intuitions about how these problems develop, based on your earlier simulation. **Try to make a reasonable prediction** for the questions below, based on the results from the simulation that you ran today.

What is the **most flips** you expect to see in order to get **10,000,000 total heads**? _____

What is the **fewest flips** you expect to see in order to get **10,000,000 total heads**? _____

Explain your reasoning, making reference to the results of your simulation.

What is the **most flips** you expect to see in order to get **20 heads in a row**? _____

What is the **fewest flips** you expect to see in order to get **20 heads in a row**? _____

Explain your reasoning, making reference to the results of your simulation.

Unit 5 Lesson 13

Introduction to Arrays

Resources

Unit 5 Lesson 14

Building an App: Image Scroller

Resources

Unit 5 Lesson 15

Processing Arrays

Resources

Name(s)_____ Period _____ Date _____

Activity - Minimum Card Algorithm

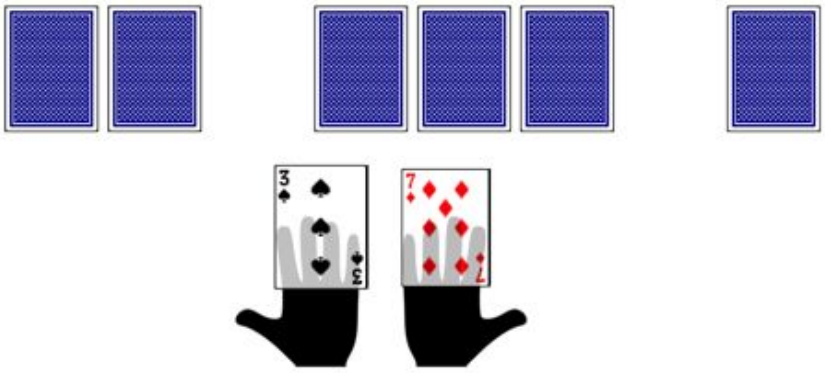
Algorithms with Lists

Once you know how to write programs that can store data in lists, the next step is to think about how you might write code, or develop algorithms, to process those lists to find or do interesting things with the data.

We often get started thinking about algorithms by trying to rigorously act them out ourselves. Because we know *something* about how computer programs work, we can keep the limitations of a computer in mind when we are "acting as the machine."

In this activity, you'll design an algorithm to find the smallest item in a list. Obviously, if we were really writing instructions for a person, we could simply tell them: "find the smallest item in a list." But that won't work for a computer. We need to describe the *process* that a person must go through when they are finding the smallest item. What are they *really* doing?

Setup and Rules:



- We'll use playing cards face down on the table to represent a list of items. Start with 8 cards face down in a row. Any card on the table ***must*** be face down.
- When acting as the machine, you can pick up a card with either hand, but you can *only hold one card at a time in each hand.*
- You can also compare the values on playing cards to determine which one is greater than the other (based on its face value).
- You can put a card back down on the table (face down), but once a card is face down on the table, you cannot remember (or memorize) its value or position in the list.

Task:

Write an algorithm to find the card with the lowest value in the row of cards.

- Goal: The algorithm must have a clear end to it. The last instruction should be to say: "I found it!" and hold up the card with the lowest value.
- The algorithm should be written so that it would theoretically work for any number of cards (1 or 1 million).
- Write your algorithm out on paper as a clear list of instructions in "pseudocode." Your instructions can refer to cards, and a person's hands, etc., but you must give a systematic way for finding the smallest card.

Algorithm To Find Minimum Card

Name(s)_____ Period _____ Date _____

Activity - Card Searching Algorithm

Analyzing Algorithms

Linear Search
A common way to process a list is to find out if it contains a specific item. We implemented one algorithm to do this called **linear search**. Linear search is pretty simple. You start at the beginning of the list, look at every item, one at a time, and see if it matches what you are looking for. Stop once you've found it, or when there are no more items to consider.

Cost and Efficiency
In computer science, we measure the "cost" or "efficiency" of an algorithm by how much work - roughly, how many primitive operations - the computer has to execute to arrive at the answer. All computers perform roughly the same set of primitive commands, but some are much faster at running them than others. Measuring the total number of primitive commands, rather than the time it takes to run a program, is a better way to compare one algorithm to another, since it does not depend on the speed of the computer running it.

Worst-Case Analysis
The same algorithm might take a different amount of time to run based on the input it is given. Consider linear search. This algorithm will likely require more work to run on larger lists, since we may need to look at every item. At the same time, we could always get lucky and find the item in the first place we look. To make it easier to compare efficiencies of algorithms, we usually **consider the worst-case scenario**. In other words, we consider the input that would cause the algorithm to do the most work. Then we can guarantee that our algorithm can't do worse than that.

Worst-Case Analysis of Linear Search
The worst case for linear search (and most searching algorithms) is that the item you're looking for is not in the list. This would make the algorithm look at every item once to verify that it wasn't there. We typically talk about the efficiency of an algorithm **in comparison to the size of the input, or data, that it must consider.** Therefore, we would say that if a list has *N* items, linear search requires that you look at all *N* items in the worst case. (For most searching algorithms, the only primitive command we consider is accessing an item in the list.) Based on this analysis, we can compare linear search to other searching algorithms.

Searching on Sorted Lists

Different Algorithms for Different Inputs
Linear search can find an item in any list, no matter how it's ordered. If we know the items in the list are in **sorted order**, however, then there are more efficient algorithms we could use to search for an item.

Challenge: Develop a searching algorithm for a sorted list
Develop an algorithm that searches for an item **in a sorted list.** Your algorithm should be as efficient as possible, as measured by a **worst-case analysis**. Note the list [1, 2, 3, 4, 5] is sorted, but so is [1, 1, 1, 1, 2] and [1, 2, 2, 2, 2].

How To Do It
On the next page you have space to **write out your algorithm in pseudocode**. Your algorithm should be written to work on any list, but you might want some manipulatives, such as **a deck of cards,** that you can use to test out your ideas as you go. If you are having trouble expressing your algorithm in pseudocode, it is fine to just describe it so that a friend could run it on a small row of cards.

My Algorithm for Searching a Sorted List

Write the steps of your algorithm below.

Analyze Your Algorithm

Worst-Case Analysis

Share your algorithm with a classmate. Once you understand one another's algorithms, try to come up with the worst-case input for each algorithm. What is the sorted list that would make it perform the most work?

Quantify Your Results

For the purposes of comparison, we'll consider the most important primitive command when comparing searching algorithms: **accessing an item in a list**. In the worst case, we know that linear search must look at every item in a list. Try to determine **how many items your algorithm must look at in the worst case** for the different-sized inputs shown below.

Number of Items Looked at in the Worst Case						
Algorithm	**8 items**	**16 items**	**32 items**	**64 items**	**100 items**	**1,000 items**
Linear Search	8	16	32	64	100	1,000
Your Algorithm						

Unit 5 Lesson 16

Functions with Return Values

Resources

Name(s)_____ Period _____ Date _____

Activity - Return Values with Go Fish

Functions with Return Values

Today you and your group are going to play the classic card game of **Go Fish**. You will work as a group to break down the game into functions in order explore writing our own functions with return values.

Rules of Go Fish

Goal: Obtain as many sets of 4-of-a-kind as possible.

Game Play:

- Each player is dealt 5 cards.
- Remaining cards are spread out in the middle.
- Moving clockwise, players take turns asking one other player in the group for a card of a particular rank. You can only ask for a rank that you currently have in your hand.
    - For example: if you have a 7 in your hand you could ask, “Sarah, do you have any 7s?”
- If you ask someone for a card, they must give up all the cards of that rank to you. Then you get to ask for another card from any of the players.
- If you ask someone for a card and they don’t have any, they will say “Go Fish!”, and you can pull one card from the table.
- Once you have obtained a set of 4-of-a-kind, remove them from your hand and place them in front of you.
- If you ever run out of cards in your hand, pick up 5 new cards from the middle of the table.
- The game is over when there are no more cards on the table.
- The winner is the person who has the most complete sets of 4-of-a-kind.

Task

1. Spend 5 minutes playing Go Fish with your group.
2. Answer the questions to explore the functions in Go Fish.

Reflection

1. What part(s) of your algorithm could be written using a list or array?

2. What part(s) of your algorithm could be written using a function?

There are two different types of people in the game of Go Fish:

1. Asker: The person asking the question
2. Responder: The person responding

We are going to focus on the algorithm for the Responder, as it is a function which gets called by the Asker.

3. What parameters would the Responder function have? i.e., What information does the Asker need to give the Responder?

4. What information/objects does the Responder have to give the Asker at the end of the function?

5. What programming commands have you seen which have behavior like the interaction between the Asker and Responder?

6. Write a pseudocode algorithm for the Responder, using the parameters you specified above.
    a. When you want to give information from the Responder back to the Asker, use the word **RETURN**.
        i. For example: RETURN card.
    b. **Note:** Obviously, if we were really writing instructions for a person, we could simply tell them: “Find all the cards that match the Asker’s request.” But that won’t work for a computer. We need to describe the *process* that a person must go through when they are finding all those items. What are they *really* doing?

Unit 5 Lesson 17

Building an App: Canvas Painter

Resources