PLTW COMPUTER SCIENCE

Activity 1.1.4

Local and Global Variables: Guessing Game - Two Player

goals

- Learn when to use local variables and global variables
- Write programs as pseudocode and in a natural language
- Develop an app independently for creative expression

description of app

Create an app that allows one user to enter a number and a second user to guess the number.

Essential Questions

1. How does the variable scope influence the structure of an algorithm?
2. Why are user stories and user-centered design so important when creating an app?
3. What arithmetic and logical concepts do I keep using over and over?

Essential Concepts

- Local Variables and Global Variables
- User-centered Design
- Iterative Design and Debugging
- Natural Language and Pseudocode
- Algorithms, Variables, Arguments, Procedures, Operators, Data Types, Logic, and Strings

Resources

Two-player Guessing Game app icon

User Stories and Desired Features

Developers tend to identify a market or a need before developing a program, starting with the user story to describe the desired app. The initial user story is then broken down into actionable items or features that can be created. Each actionable item of the user story is an item in a list called a backlog. A backlog breaks down what the user wants into a sequential and prioritized list of what needs to be done to make the app do what the user wants.

During this course, you will take user stories and break them into small manageable pieces that you can code. By completing each smaller piece one by one, you can test the app and check with the user to make sure the app development meets the user's needs. While you break down the user story of needs, these are the types of questions to ask yourself about the design of the app:

- What features will you need?

- What should the user interface look like?
- How big should the screen be?
- Will the user ever need to scroll down?
- How does the accelerometer provide inputs on which way the sprite should move?
- How does the app know where the sprite is and where the sprite should move to?

Using the steps below, you will build the user interface for an app. An example image of the user interface is provided. Yours may look a little different, and that's okay. You may choose to lay out these components differently by dragging and dropping them as you see fit. Just make sure you include all the components, based on the activity's directions.

Design Overview: Guessing Game

User Story

A group of students wants a game that they can play on their Android™ device while riding the bus. Work to develop a game app that they like. The initial user story is defined, but like all user-centric developments, you will create the app incrementally and get feedback from the users. They may even provide suggestions to make your app better.
In the game, the user will have a limited number of attempts to guess a number programmed into the code. You will add a two-player mode where player two can enter a number and player one tries to guess it.

App Overview
The app will allow one user to enter a number and a second user to try to guess the number.

Initial Backlog Breakdown
The user needs:

- ☐ Text box to enter the number guessed.
- ☐ Button to input the number.
- ☐ Feedback label that prompts the user to guess higher or lower.
- ☐ Feedback label that tells the user how many guesses are left.
  - ☐ Feedback label that lets the user know whether they won or lost.
- ☐ A way to limit the number of guesses.
- ☐ Reset/play again button.
  - ☐ Text box to enter the number the player is trying to guess.
- ☐ Button to input the number the player is trying to guess.

Constraints
The app must be created using MIT App Inventor.

Refer to your downloadable resources for this material. Interactive content may not be available in the PDF edition of this course.

Important: The rotating images above are examples of using *HorizontalArrangements*.

1. Navigate to MIT App Inventor and log in.
2. Create a new project using the naming convention set forth by your teacher.
   Example: *LastNameFirstInitial114GuessingGame1*
3. From the *Layout* drawer, drag out the *HorizontalArrangement*.

   The arrangements in the *Layout* drawer allow you to center items across the screen, put multiple components next to each other, and control how much of the screen is used. The horizontal arrangement will align multiple user interface components horizontally next to each other.

4. Drag a *Button* from the *User Interface* drawer to drop into the horizontal arrangement.
    - Rename the button "GuessButton*".*
    - In *Properties*, change the Text property to: *Guess!*
5. Drag a TextBox from the *User Interface* drawer to drop into the horizontal arrangement next to the button.
    - Rename the TextBox, "GuessInput"*.*
    - In *Properties*, change the hint to "Enter your guess here!"
6. Take a moment to look at the properties of the horizontal arrangement. Adjust the properties to make the components easy to read for a user. For example, in Properties, select the width option **Fill parent***,* which will automatically make the layout fill the width of the user's tablet.
7. Add a horizontal arrangement that will hold feedback labels. Drag out the following *Labels* from the *User Interface* drawer*.*
    - *LowerHigherLabel* to provide feedback when the user needs to guess higher or lower.
    - *WinLoseLabel* to provide feedback to let the user know whether they won or lost.
    - *GuessesLeftLabel* to provide feedback about how many guesses the user has left before the game ends.
8. Drag a *Button* from the *User Interface* drawer and drop onto the screen.
    - Rename the button, "PlayButton"*.*
    - In *Properties*, change the Text property to "Reset/Play Again"*.*
9. Add a way for the second player to pick the number that the first player will guess.
    - Drag out a horizontal arrangement. Rename the arrangement to "MultiPlayerArrangement".
    - Drag a *textbox* into the horizontal arrangement and rename it "EnterNumText". In *Properties*, select **Text only**.
    - Drag a *button* into the horizontal arrangement and rename it "EnterNumButton". See the rotating images at the start of the procedure for an example.

PLTW DEVELOPER'S JOURNAL After reviewing the design considerations and adjusting your user interface, take some notes on things you have learned that will help with future design of user interfaces.

Defining Your Variables

Refer to your downloadable resources for this material.	Refer to your downloadable resources for this material.

Interactive content may not be available in the PDF edition of this course.	Interactive content may not be available in the PDF edition of this course.

Global Variables

A **global variable**can be accessed and referenced by any part of the program, which means it has a global scope. At any point in the code, the program can access or “get” the value. At any point in the code, the program can change or “set” the value. In this program, you will set up two global variables to store information that all parts of the program will need.

Using a large number of global variables adds complexity to your program, because all parts of the program can change global variables. If a global variable is providing unexpected results, it can take longer to track down what part of the program is changing that global variable in the undesired way. The global variables for this program are:

- *guesses* - A global variable to track the number of guesses (Each player starts with 20 guesses.)
- *number* - A global variable to set the number the user will try to guess. Initially, we will set the number at 55. Later, we will modify the app so that another player (Player Two) can enter a number that Player One must guess so that the number is not always 55.

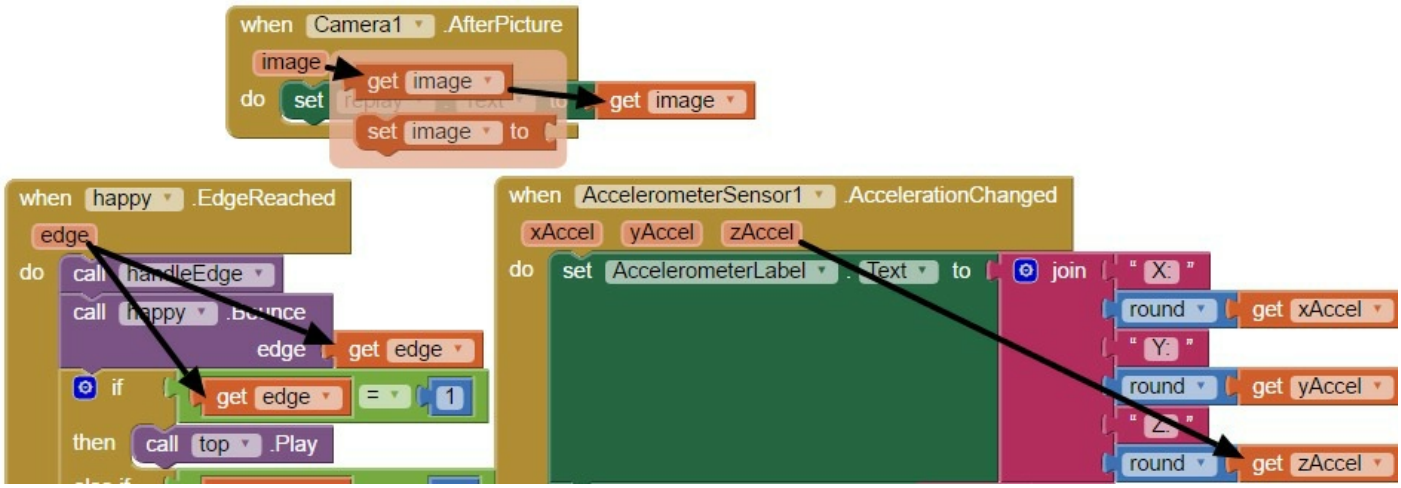
In the *Variables* drawer in the *Blocks* view, you will see *set* and *get* blocks. You can use these blocks anywhere in the code after you initialize them. Initializing a variable is setting a starting value for the computer to use and change as you establish the program.

10. Use *Math* blocks to initialize the following global variables:
    - *guesses*: 20
    - *number*: 55

Local Variables

There are also local variables that cannot be accessed by all parts of the program. You have used local variables every time you drag an orange block out of an event handler.

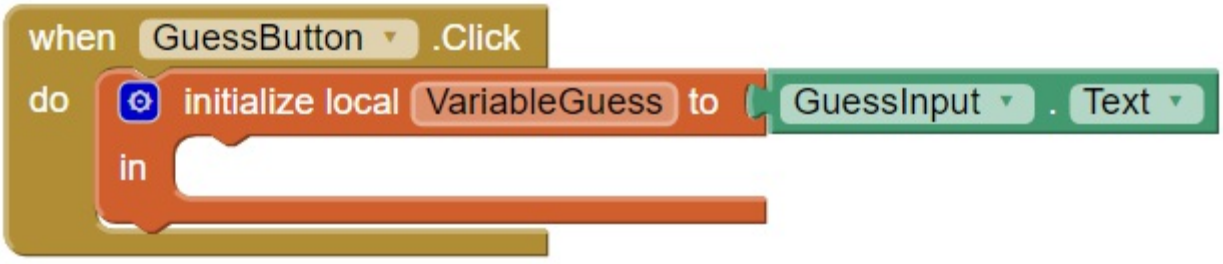
In previous activities, you have used abstracted variables. These variables were defined for you in MIT App Inventor. You did not need to know how they worked, just that they held information that you needed. You could get the value held in the variable (call the variable) or you could have the program put a value into (set) the variable from an input. Now, you will define and modify a variable with your own values. You should always set a variable to an initial value (initialize a variable).



A **local variable**has what is called a local **scope**. Local variables can only be accessed in a specific part of the program. Limiting access to variables can sometimes reduce the complexity of your program. It allows you to quickly determine how a local variable is being changed and what part of the program is changing it. This will allow you to spend less time debugging, because you can focus on the local scope rather than search through the entire program if you had used a global variable.

You are now going to define a local variable *VariableGuess*. This is the number the user guesses as the number the other player stored in the global variable. The number a user guesses does not need to be used by all parts of the program, and you want to limit the scope of the guessed variable from impacting the other parts of the program. Therefore, the user input guess will only be temporarily stored in a local variable, *VariableGuess,* when the user clicks the **Guess!** button in the user interface.

11. From the *Variables* drawer in the *Blocks view*, pull out the *local* variable.
12. Give the *local* variable a name such as *VariableGuess*.
13. Set the *initialize local VariableGuess* to the input from the *GuessInput* text box.
14. Add the *initialize control* block to the *GuessButton.Click* control block variable.

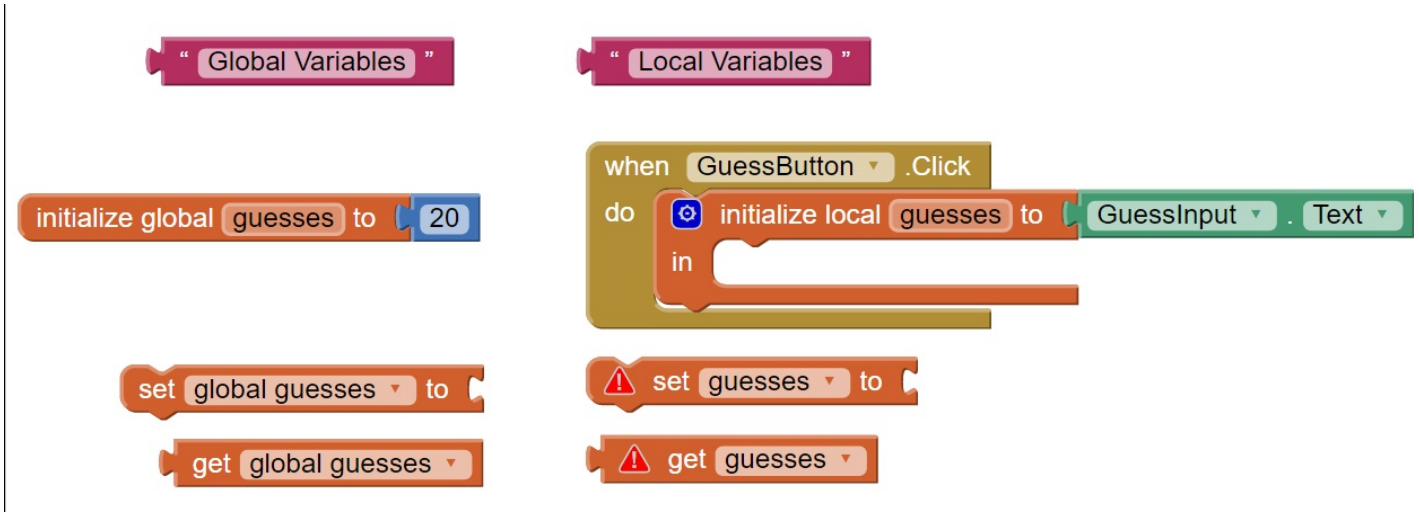


Why Use a Local Variable?

When a user clicks the **Guess!** button, the value the user guesses in the text box will be set and stored to the local variable *VariableGuess.* From a programming standpoint, the design will continue to expand, allowing people to play with a person near them, or in the next activity, play against a

computer-generated number. In both of these circumstances, the user wins if they guess the correct number, and loses if they cannot guess the number before running out of guesses. By making the win and lose conditions contained in a local variable, you are helping to secure the game from being won or lost under different circumstances or different types of game play as you expand the game play options.

Important: Be careful with variable naming. In lower programming abstraction levels, such as *Python*®, naming two variables the same thing causes problems. App Inventor will let you name a local and a global variable with the same name, but they have different scopes of influence.



In the image, notice that the global and local variables have the same name for guesses. They both have *set* blocks, which allow a part of the program to change the value that is held inside the variable, and *get* blocks, which allow the program to get the value stored inside that variable. However, as a higher level of abstract programming, MIT App Inventor automatically changes the global guesses set and get to include "global" in the name and provides an error for the local guesses that states it will not work outside the code. The blocks automatically adjust to keep from confusing these variables even though they have the same name. Not all programming languages do this, so it is important to start using different names for local and global variables.

As you progress into lower levels of abstracted programming, avoiding global variables helps to eliminate renaming variables and getting errors. When variables are local, they will not set or pull from other parts of the code like global variables will. Naming is easier when using local variables, and the chance of changing a different part of the code also decreases.

Using Variables in a Conditional

Now that a global variable holds the number the user is trying to guess (55), and the local variable *VariableGuess* is defined, you need to add a conditional statement. Conditional statements are a series of conditions that a computer moves through until it finds the one that is true. This is exactly what the computer needs to do to compare the *VariableGuess* (input by the user) with the number stored in the global variable (55) to see whether the user guessed the correct number.

Using Natural Language to Describe Code

A conditional uses the value stored in the variables to direct the computer to determine what to do next. In natural language, the way the computer would process the information is something like

this:

- Get the local variable and get the global number and see if they are equal to each other.
- *If* the local variable named *VariableGuess* is equal to the global variable *number*,
    - What relational operator would you pick from the *Logic* drawer to compare these two variables?
- *Then* the *WinLoseLabel* should say “You Win!”
- *Else*, the *WinLoseLabel* should say “Try Again”.

If we move down from the natural language version of how the computer compares information with the values in the variables, we get to **pseudocode**, which is more like how it would look in block form:

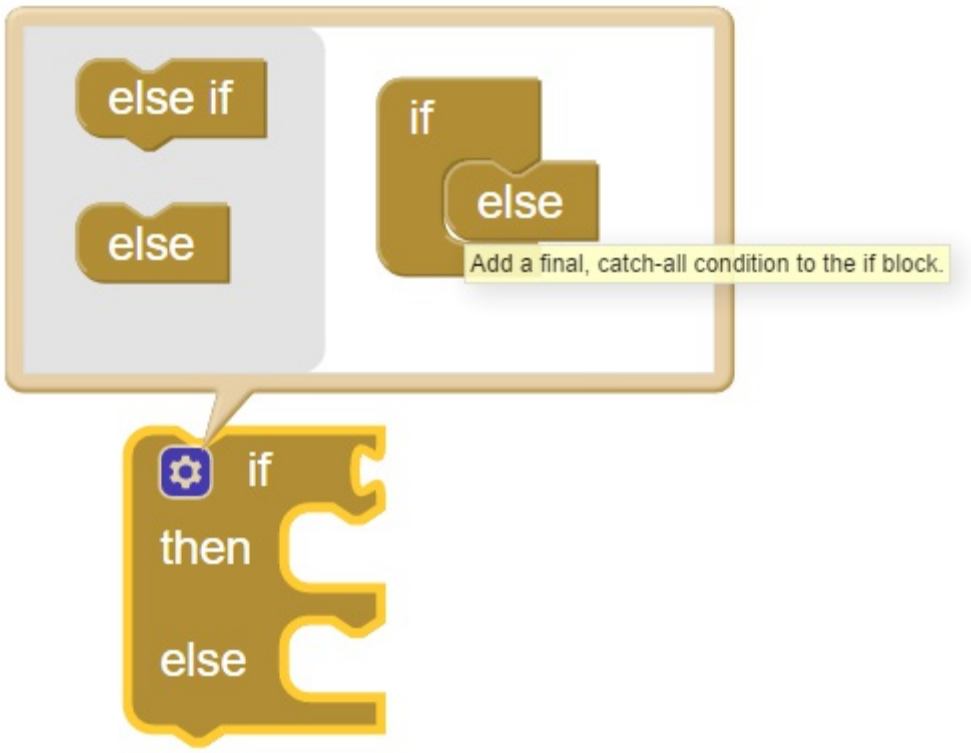
```
If get local VariableGuess = get global number:
    Then
        set WinLoseLabel to “You win!” message
    Else
        set WinLoseLabel to “Try Again” message
```

Later in the course, you will move to even lower levels of abstraction and do the same type of coding that App Inventor abstracts in the blocks. It is important that you start making the shift from natural language to the block language. Making this transfer now will help you get to lower levels of abstraction, because understanding how and when conditional statements may modify local and global variables is an essential part of computer science.

PLTW DEVELOPER’S JOURNAL Take some notes comparing different language abstractions of the same process of an *If*, *Then*, *Else* conditional.

15. From the *Logic* drawer, drag out and create an *If*, *Then*, *Else* block.

    Pull an *If-Then* block out of the *Control* drawer. Use the *mutator* to change the *control* block so that it includes an *else* statement.



16. Add the necessary relational operator, get the local variable. and compare it to the global variable as the *If* statement.

17. Set the *WinLoseLabel* to the appropriate output for the *then* statement and the *else* statement.

18. Test, debug, and adjust.
    - ☐ Enter a few wrong numbers in the Guess Input text box and click the **Guess**! button. Make sure you get the correct message.
    - ☐ Enter the correct number. If you get the correct message each time, your conditional is set up and you may proceed to the next iteration.

Important: To establish a history of successful program iterations, remember to save your project with a new name before you continue.

Program Iterations to Add Features

Iteration 1: Restarting the Game

The game is easier to test and more fun to play, if you can play more than once without resetting the connection.

Consider what replaying the game would involve? After you think about it in natural language, move from natural language to block-based coding for what replaying the game would look like.

19. Use the *PlayButton* event handler to create the replay option.

- From the *PlayButton* block drawer. drag out the *PlayButton* event handler into the *Blocks* view.
- Set the *WinLoseLabel* to a blank string.
- Set the *LowerHigherLabel* to a blank string.
- Set the *GuessesLeftLabel* to a blank string.

Iteration 2: Adding the Guess Higher or Lower Conditional Feedback

Conditionals examine a single factor and select the outcome defined by the programmer, but sometimes you will want to examine more than one factor with conditionals. Checking multiple factors in a single process is called a “chained conditional statement”. Chained conditionals are conditional statements linked one after the other. The computer will evaluate them to determine which one is true and act on the one that evaluates to true. You will add chained conditionals to the existing app to increase the efficiency of the app.

Add a new feature that tells the user which way they need to guess (higher or lower) if they don’t guess the right answer. Here are the possibilities that can happen after the user makes a guess:

- The user guesses the correct number stored in the variable.
- The user guesses higher than the number stored in the variable.
- The user guesses lower than the number stored in the variable.

20. Right now the *else* statement returns, “Try again” if the guess is incorrect. You will now change this response using chained conditionals to indicate what the user should do next. If the guess is too low, “Guess higher”. If the number is too high, “Guess lower”. Each part of the conditional compares the number guessed to a random number stored in the global variable. If the user’s guess is higher or lower than the number stored in the global variable, the program should respond differently.
21. Adjust your *if-then-else* conditional statement to remove the *else* statement and add two *else if* conditions.

> Refer to your downloadable resources for this material. Interactive content may not be available in the PDF edition of this course.

22. Use relational operators from the *Math* drawer. Get the local variable *guessed* and get the global variable. Compare the two variables with a math operator and provide the correct response based on the input:
    - *Else If* the guess is too low, set the *HighLowLabel* to provide the text feedback “Guess Higher”.
    - *Else If* the guess is too high, set the *HighLowLabel* to provide the text feedback “Guess Lower”.
    - Make sure that all feedback is clear in letting the user know whether they have won or are still playing. Keep the feedback professional and appropriate.
23. Set the *GuessInput.Text* to a blank string every time the **Guess!** button is tapped. This way,

the input box will be empty and ready for the next guess each time the **Guess!** button is tapped.

24. Drag a *set GuessInput* block into your code. Try adding it to the *GuessButton* event handler so that it clears the input text box with each guess (regardless of whether the guess is too high or too low).

25. Test, debug, and adjust. Collaborate with your elbow partner to test your code and see whether it provides the outputs you want.

    - ☐ Enter a wrong number that is lower than the actual number in the *GuessInput* textbox and click the **Guess**! button. Make sure you get the message telling you to guess higher.
    - ☐ Enter a wrong number that is higher than the actual number. Make sure you get the message telling you to guess lower.
    - ☐ After each guess, the *GuessInput* textbox should clear to a blank string so the user can enter the next number.
    - ☐ Enter the correct number. If you get the correct message each time, your conditional is set up, and you may proceed to the next iteration.
    - ☐ Only proceed when the app responds appropriately to lead a user to the correct answer.

Important: To establish a history of successful program iterations, remember to save your project with a new name before you continue.

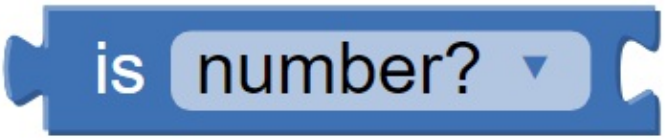
Iteration 3: Correcting an Error

Sometimes you want to test a big condition. For example, did the user enter a number before doing anything else in the program? Checking for these types of conditions can help prevent errors from occurring by making sure the program only executes if certain conditions are met.

- What happens when you touch the **Guess**! button without having entered a number?
- What data type does the program expect to be input?

Because App inventor cannot compare an integer to a string, it gives you an error. The operation cannot accept the arguments: , [*empty-string*], [55]

Each time the *GuessButtonClick* is activated, the program checks whether the input is a number or a string with the *is number?* logic block.



Refer to your downloadable resources for this material. Interactive content may not be available in the PDF edition of this course.

26. Add an *if-then-else* statement and logic to check whether the *GuessInput* value entered is really a number. If it is not, display to the user “Pick a number please!”

    Refer to your downloadable resources for this material. Interactive content may not be available in the PDF edition of this course.

27. Test, debug, and adjust.
    - ☐ Enter a wrong number that is lower than the actual number in the *GuessInput* textbox and click the **Guess**! button. Make sure you get the message telling you to guess higher.
    - ☐ Enter a wrong number that is higher than the actual number. Make sure you get the message telling you to guess lower.
    - ☐ Enter a letter, a word, a punctuation symbol. Make sure you get back the correct message.
    - ☐ After each guess, the *guess input* textbox should clear to a blank string so the user can enter the next number.
    - ☐ Enter the correct number. If you get the correct message each time, your conditional is set up and you may proceed to the next iteration.
    - ☐ Only proceed when the app responds appropriately to lead a user to the correct answer.

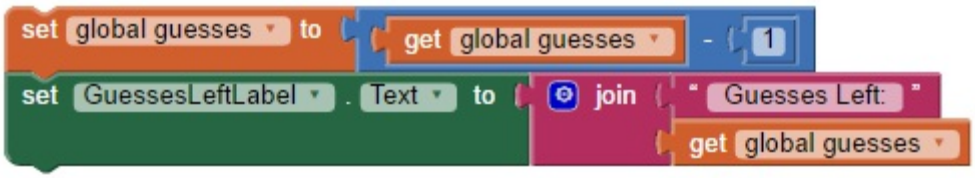
Important: Remember to save your project with a new name before you continue.

Iteration 4: Limiting Guesses

You have already set up a global variable that stores the number of guesses. Now, you are going to modify your chained conditional to subtract 1 from the number stored in the global variable each time the user guesses a wrong answer. This is an essential concept in computing called an **incremental counter**. Variables are a key component in this, because they have a value that the computer changes in a set process as outlined by the programmer.

While incrementing the variable, you also need to make sure the *GuessesLeftLabel* output to the user shows the value of the global variable. You will need to join a text string "Guesses Left" with the current value of the global guesses variable. (Remember, it decreases by one with each wrong guess.)

Here is an example of how you might do this:



Where in your program do you need to add these blocks to make it take away from the count with each guess?

28. Guess and test until the app behaves as you expect by subtracting one each time you enter a number. Consider that you did this same checking and clearing with the *GuessInput* text.

Complex Conditionals Using Variables

Complex conditionals are a series of checks that the computer moves through in sequence before setting the Boolean value to true or false to determine which process to do next.

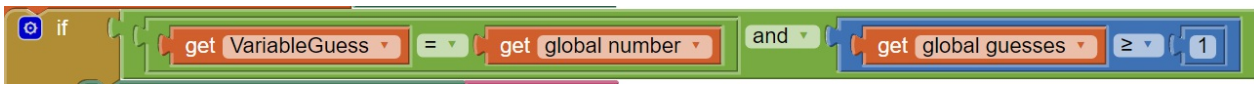
In a word processing program, nothing happens until you provide an input. The software is always waiting for user input. Such as:

- *If* the A key is pressed, *Then* output an a.
- The shift key adds an extra Boolean expression: *If* the A key is pressed, *AND* the shift key is pressed, *Then* output an A.

Using conditional logic and Boolean operators such as *AND*, *OR*, or *NOT* helps computers determine what to do next by increasing the number of conditions that must be met before taking a specific action. In this way, computers can determine how to compare, adjust, and when to access variables.

Combining both logic and arithmetic operators, you can give the conditional statement multiple variables to check to logically determine the next action to take.

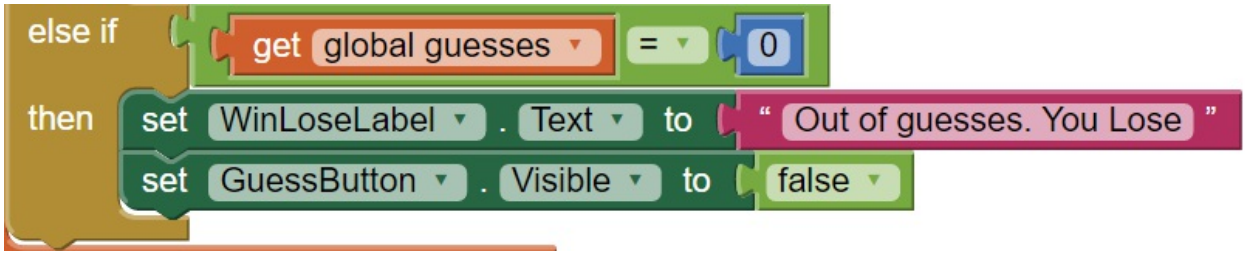
29. You want the computer to now check whether the user still has guesses and whether the user has input the correct answer. The conditional statement needs to evaluate whether there are still guesses left and whether the user guessed the correct number.
    - *If* the *global guesses* is greater than 0, *AND* the VariableGuess is equal to the global number 0:
      Then provide feedback in the *WinLoseLabel*: “You win!”
      Then provide feedback in the *LowerHigherLabel*: “Congratulations!”



Complex Conditional

    - *Else*, if there are still guesses left and the number guessed is lower than the number, *Then*, provide the message to the user to guess higher.
    - *Else*, if there are still guesses left and the number guessed is higher than the number, *Then*, provide the message to the user to guess lower.

- *Else*, the the user is out of guesses. *Then*, provide a message that they ran out of guesses so the game is over, and remove the Guess! button so the user cannot keep guessing.



30. Add to the *PlayButton* event handler to concatenate (join from the text drawer) a "Guesses Left: " string and *get global guesses*.

    - Even though you initialized the Global variables for guesses in the code, and the code starts fresh every time you reconnect, after one game, do the variables still have the same value?

      Important: A global variable changes based on any part of the program that sets it. As a user plays, they keep setting that variable down by 1. To play again, the variable needs to be reset back to 10.

    - To reset the *global guesses* variable to the original number of guesses to play again, what can you add to the *PlayButton* event handler?
    - How will the reset value of the global variable be shown to the user?

> Refer to your downloadable resources for this material. Interactive content may not be available in the PDF edition of this course.

31. Test, debug, and adjust.

    - ☐ Touching the **Play** button should restart the game by setting the values back the original amounts.
    - ☐ Enter a wrong number that is lower than the actual number in the *GuessInput* textbox and click the **Guess**! button. Make sure you get the message telling you to guess higher.
    - ☐ Enter a wrong number that is higher than the actual number. Make sure you get the message telling you to guess lower.
    - ☐ Enter a letter, a word, a punctuation symbol. Make sure you get back the correct message.
    - ☐ After each guess, the *guess counter* should count down by one.
    - ☐ After each guess, the *GuessInput* textbox should clear to a blank string so the user can enter the next number.
    - ☐ Running out of guesses ends the game by not allowing the user to guess anymore and displaying a message that they have lost.
    - ☐ Enter the correct number. If you get the correct message each time, your conditional

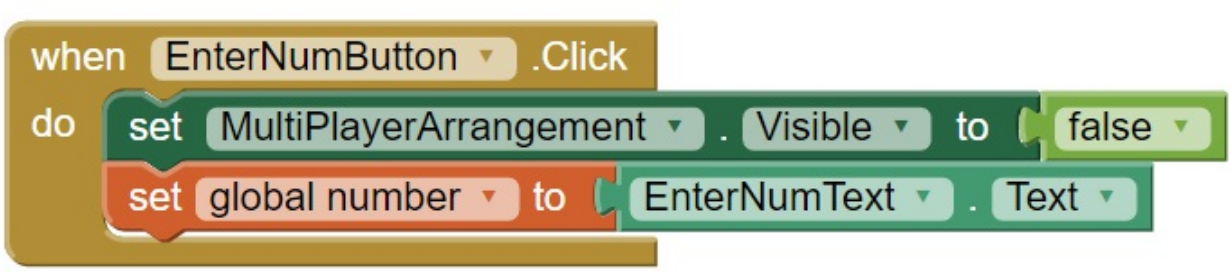
is set up and you may proceed to the next iteration.
- ☐ Only proceed when the app responds appropriately to lead a user to the correct answer.

Important: Remember to save your project with a new name before you continue.

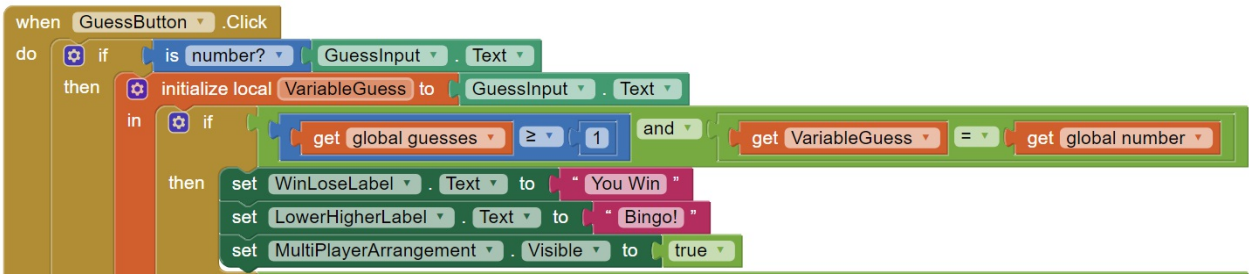
Iteration 5: Enable Local Play

Now that you have built and tested all the individual functions of the app, you are going to enable user input of a number to be stored in the global variable so the game can use that value.

32. Drag out the *EnterNumButton.click* event handler. Inside this event handler, set up the following:
    - So that Player Two cannot see what Player One entered, set the *MultiPlayerArrangement.visible* to “false”.
    - Set the global number to be the *EnterNumText.Text*



33. Because this will be another way to play, it is important to add all the replay features here, too. Add to the event handler:
    - Set the labels *WinLoseLabel* to be a blank string.
    - Set the *LowerHigherLabel* to be a blank string.
    - Set the *global guesses* to “10”.
    - Concatenate the *GuessesLeftLabel* to include a "Guesses Left: " string and the *get global guesses*.
34. In the *GuessButton.Click* event handler, add to the first *if* conditional:
    - To make sure the user can enter another number when the game ends, set *numberSet MultiPlayerArrangement Visible* to “true”.



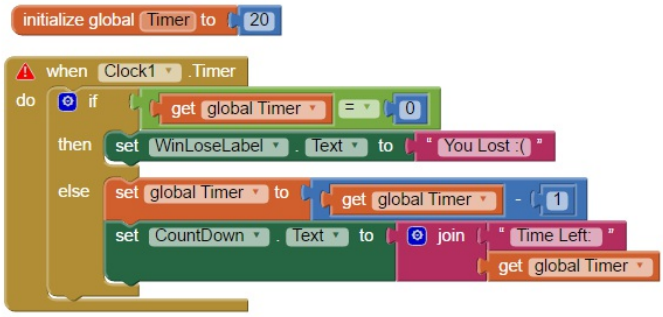
    - In what two other places should you add these blocks so that if Player Two loses, the user input becomes visible again for them to enter a number?
35. Once everything is working, trade with an elbow partner to test each other’s apps.

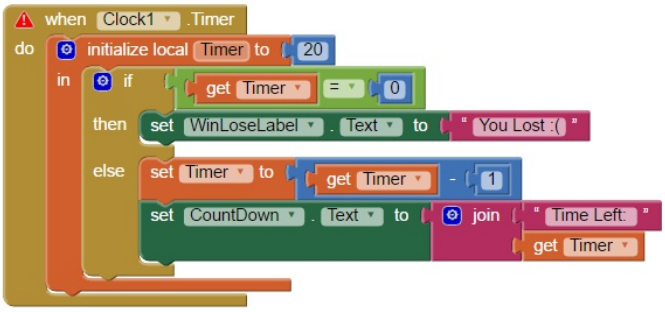
36. After you verify that your app works, let your teacher know.

PLTW DEVELOPER'S JOURNAL

In a later activity, you will expand this app to include a countdown timer that adds a level of challenge to the game. The countdown will need a start time that will be **decremented** until it reaches 0.

Consider the following image that shows two versions of code, one using a local variable and one using a global variable.





Will the countdown clock need to have a global scope or a local scope? Why?

> Refer to your downloadable resources for this material. Interactive content may not be available in the PDF edition of this course.

Conclusion

1. What would happen if a program tried to access a variable that was defined locally in another part of the program?
2. How does one of the algorithms in your program function? Use natural language to describe the algorithm as if you were explaining it to your friend who has not taken this course yet.

3. How did you interpret and respond to the **essential questions**? Capture your thoughts for future conversations.