

# Unit 5 - Building Apps

This unit continues to develop students' ability to program in the JavaScript language, using Code.org's App Lab environment to create a series of small applications (apps) that live on the web, each highlighting a core concept of programming. In this unit students transition to creating event-driven apps. The unit assumes that students have learned the concepts and skills from Unit 3, namely: writing and using functions, using simple repeat loops, being able to read documentation, collaborating, and using the Code Studio environment with App Lab.

## Chapter 1: Event-Driven Programming

### Big Questions

- How do you program apps to respond to user "events"?
- How do you write programs to make decisions?
- How do programs keep track of information?
- How **creative** is programming?
- How do people develop, test, and debug programs?

### Enduring Understandings

- 1.1 Creative development can be an essential process for creating computational artifacts.
- 1.2 Computing enables people to use creative development processes to create computational artifacts for creative expression or to solve a problem.
- 1.3 Computing can extend traditional forms of human expression and experience.
- 2.2 Multiple levels of abstraction are used to write programs or create other computational artifacts
- 4.1 Algorithms are precise sequences of instructions for processes that can be executed by a computer and are implemented using programming languages.
- 5.1 Programs can be developed for creative expression, to satisfy personal curiosity, to create new knowledge, or to solve problems (to help people, organizations, or society).
- 5.2 People write programs to execute algorithms.
- 5.3 Programming is facilitated by appropriate abstractions.
- 5.4 Programs are developed, maintained, and used by people for different purposes.
- 5.5 Programming uses mathematical and logical concepts.
- 7.1 Computing enhances communication, interaction, and cognition.

Week 1



## Lesson 1: Introduction to Event-Driven Programming

Students are introduced to Design Mode in App Lab, which allows students to easily design the User Interface (UI) of their apps and add simple event handlers to create a simple game.

## Lesson 2: Multi-Screen Apps

Students improve the chaser game by learning how to add multiple “screens” to an app and by adding code to switch between them. Students learn to use `console.log` to display simple messages for debugging purposes.

## Lesson 3: Building an App: Multi-Screen App

Students design and create a 4-screen app on a topic of their choosing. Students may collaborate with a classmate as a “thought partner,” similar to the recommendation for the Create Performance Task.

## Week 2

## Lesson 4: Controlling Memory with Variables

Students learn to create and assign values to variables and are navigated through common misconceptions.

## Lesson 5: Building an App: Clicker Game

Students learn about global versus local variables, and use variables to track the score in a simple game.

## Lesson 6: User Input and Strings

Students develop a simple Mad Libs® app, learning to collect and process text strings as input from the user.

## Week 3

## Lesson 7: If-statements unplugged

Students trace simple robot programs on paper to develop a sense of how to read and reason about code with if statements in it. The code is the same pseudocode used on the AP exam.

## Lesson 8: Boolean Expressions and “if” Statements

Students learn how to write and use if statements in JavaScript by debugging common problems, solving simple problems, or adding conditional logic into an existing app or game.

## Week 4

## Lesson 9: “if-else-if” and Conditional Logic

Students are introduced to the boolean (logic) operators NOT, AND, and OR as well as the if-else-if construct as tools for creating compound boolean conditions in if statements.



# Lesson 10: Building an App: Color Sleuth

## Programming | Conditionals | App Lab

Students follow an imaginary conversation between two characters, Alexis and Michael, as they solve problems and make design decisions in the multiple steps required to construct the "Color Sleuth" App. Students must implement elements of the code along the way.

## Chapter Commentary

### Unit 5 Chapter 1 - What's the story?

This chapter establishes the basic story of "What's an app?" The first week is dedicated to introducing App Lab's **design mode**, and becoming familiar with the **event-driven mindset** for programming. The largest difference between this unit and previous programming unit (unit 3) is the the **event-driven** paradigm for programming. In Unit 3 (turtle programming) everything was procedural: you click "run" on the program and it starts executing from the first line of code, and runs until completion. An event-driven program never ends! It is constantly waiting to react to user input like clicking a button, or moving your mouse. You write programs by deciding which events you want to respond to and by writing a discrete function to respond to that specific event. As part of its execution that function may run some loops, perform calculations, call other functions, and so on.

Next we cover **variables, user input (including text strings), Boolean expressions and if-statements**. It's quite a blitz through a gamut of fundamental programming concepts. The story to tell is that these concepts are behind features of apps that you are familiar with. Want to keep score in a game? Variables. Want to respond to something the user types? Text input. Want your app to exhibit different behaviors based on certain conditions? Boolean expressions and if-statements. The **Color Sleuth** game/app is an important culminating project that ties together all the concepts learned in this chapter. It is a unique lesson in which the student follows a conversation between two fictional students collaborating to plan, design and write the code for their project. The student follows along in and writes the code to match the plans of the fictional students.

#### Ready for the Create PT?

We think that the end of this chapter represents a minimum point at which students could complete a successful Create performance task. Check out the Performance Task pacing section on page 32 for more details.

### Our Approach to the Content

Our approach to teaching these concepts is somewhat "traditional" in terms of the sequence of concepts and how they build on each other. If you study the lessons you will notice a rough pattern to how we scaffold the learning for each concept which is typically as follows. (1) Introduce the concept in an unplugged or discussion-based way to activate prior knowledge and motivate the students' need to learn the concept. (2) Learn about and practice the code related to that concept. Students read about and work through a series of exercises, which they can do in pairs or solo, to practice using any new code related to the concept, as well as solving and debugging a few problems. (3) Follow a Building an App lesson, which walks students through the construction of an app from scratch. In the lesson students progressively build parts of it, and submit a final version. These apps allow room for some student creativity and indeed students should be encouraged to "make it their own" while still using the underlying concepts.

The concepts covered in this chapter, especially variables, conditional logic and if-statements carry a lot of classic misconceptions. Our lessons often try to lead students into those misconceptions by asking them to debug or problem-solve around them. This means that there is a risk for some students becoming frustrated or confused by these lessons. We've tried to provide a number of supports and resources you might use to help clear up confusion. One key resource is the video related to each concept. The videos are dense enough that we recommend you use them to sense-make after students have been through a programming experience, as well as before. Another resource to be aware of would be the "maps", which are static pages that explain the code for a concept and contain diagrams with other helpful information that are meant to serve as a reference and an introduction to the concept in the first place.

---



# Chapter 2: Programming with Data Structures

## Big Questions

- How are real world phenomena modeled and simulated on a computer?
- How do you write programs to store and retrieve lots of information?
- What are "data structures" in a program and when do you need them?
- How are algorithms evaluated for "speed"?

## Enduring Understandings

- 2.3 Models and simulations use abstraction to generate new understanding and knowledge.
- 3.1 People use computer programs to process information to gain insight and knowledge.
- 4.1 Algorithms are precise sequences of instructions for processes that can be executed by a computer and are implemented using programming languages.
- 5.1 Programs can be developed for creative expression, to satisfy personal curiosity, to create new knowledge, or to solve problems (to help people, organizations, or society).

## Week 5

### Lesson 11: While Loops

Students are introduced to the "while loop" construct by first analyzing a flow chart and then by completing a series of exercises in Code Studio. The "while loop" repeats a block of code based on a boolean condition.

### Lesson 12: Loops and Simulations

Students make a simple computer simulation to model a coin flipping experiment that is possible, but unreasonable, to do by hand. Students write code that uses while loops to repeatedly "flip coins" (random number 0 or 1) until certain conditions are met.

### Lesson 13: Introduction to Arrays

Students learn about arrays in JavaScript as a means of storing lists of information within a program. Students build a simple app, My Favorite Things, which stores and cycles through a list of words describing their favorite things.

## Week 6

### Lesson 14: Building an App: Image Scroller

Students extend the My Favorite Things app to manage and display a collection of images instead of words. Students also learn to make the program respond to keys (left and right arrow) by using the "event" parameter that is created when an event is triggered.

### Lesson 15: Processing Arrays

#### Unplugged | App Lab

In this long lesson, students learn to use for loops to process lists (arrays) of data in a variety of ways to accomplish various tasks like searching for a particular value, or finding the smallest value in a list. Students also reason about linear vs. binary search.



## Lesson 16: Functions with Return Values

Students learn to write functions that calculate and return values, first through an unplugged activity by playing Go Fish, then by practicing in Code Studio, and finally by writing functions that return values in a simple turtle driver app.

### Week 7

## Lesson 17: Building an App: Canvas Painter

Canvas Painter is a culminating project brings together processing arrays, functions with return values, and handling keystroke events. The app allows a user to draw an image while recording in an array every single x,y location the mouse passes over on the canvas. By processing this array in different ways, the image can be redrawn in different styles, like random, spray paint, and sketching.

## Lesson 18: Practice PT - Create Your Own App

Students design an app based off of one they have previously worked on in the programming unit. Students choose the kinds of improvements they wish to make and write responses to reflection questions similar to those they will see on the AP® Create Performance Task.

# Chapter Commentary

## Unit 5 Chapter 2 - What's the story?

This chapter tells the story of the real power of computers, which is to quickly and precisely perform many of computations on data to produce a result. There are two pieces to this puzzle. (1) We need to better understand how to control **iteration** (loops) beyond a simple repeat loop. (2) We need to be able to store and process lists of data rather than single variables.

Knowledge and facility with loops and lists opens an almost infinite number of doors to different types of programs you can write and problems you can solve. The projects and examples in this chapter merely scratch the surface of what's possible. List processing is a core pattern for much of computation. The point in these lessons is for students to see the pattern in action a **few times** to get the gist.

For example, in computer science, writing computer programs to **model and simulate** real world events is a hugely important topic. The idea of using randomness or random sampling over a large number of trials to obtain a numerical result is a foundational practice in computing. We address it briefly here with the coin flipping experiment, in which students write a program to model flipping a coin repeatedly while keeping track of the results in various ways. This type of method is broadly known as the "Monte Carlo method" and could be used in any situation to determine the probabilities of certain outcomes. Monte Carlo methods have been used to model drivers' behavior in traffic, the flow of multiple fluids, business risk models, and so on.

## Our Approach to the Content

The teaching patterns for this chapter are similar to prior lessons in Unit 5 -- (1) introduce and motivate the concept, (2) do some skill-building and practice with the code related to that concept, (3) complete a project.

We want to encourage students to continue working with a partner or "programming buddy" - a person that they can check their work with, clarify instructions, etc. The model we suggest is two students sitting side by side, one with the instructions up on her screen, while the other writes code on hers. Since the projects are usually individual and creative, there is no risk in students helping each other along the way.

Even though the coin flipping experiment seems simplistic, it has the same root elements of more sophisticated models. The main takeaway for students should be this: when some computation is too long or complicated to do by hand with mathematics, if you can think of how to represent or model the thing using the tools of programming such



as variables, loops, and conditions (lists), then you can run a simulation a million times to approximate a result.

It's also worth pointing out a deliberate connection between processing arrays in this chapter and the **Human Machine Language**" problems students worked on in Unit 3, where they designed algorithms and programs to process a list of playing cards. You can appeal to some of those exercises in this work here. There are numerous aspects to using lists, and the concept takes some time to sink in. Doing a linear pass over an array (a loop that starts at the front of a list and does something to or with each element one at a time until it reaches the end) is the most sophisticated programming technique students will encounter in the course and be expected to reason about in an exam situation.





This curriculum is available under a  
Creative Commons License (CC BY-NC-SA 4.0).

If you are interested in licensing Code.org materials for commercial purposes, **contact us**.



# Lesson 1: Introduction to Event-Driven Programming

## Overview

Students will be introduced to a new feature of App Lab: **Design Mode**. Design Mode allows students to easily design the User Interface (UI) of their apps using a drag-and-drop editor. Students learn how to create UI elements they have seen before such as images, text labels and buttons, but they will see many more options for styling these elements with colors, font sizes and so on. Students also learn how to add **event handlers** - code that listens for and responds to user-events. Students also explore some common errors that come up in event-driven programming and will learn some important skills for debugging programs, chief among them being responding to error messages. Students end the lesson by creating the foundation of a simple "chaser game" which they will add onto in the next lesson.

## Purpose

Most modern applications are interactive, responding to when users click buttons, type in a textbox, tilt the screen, swipe between screens, etc. In every instance, the user's action is generating some kind of event and the program responds by running an associated block of code. Programming a modern application is therefore typically an exercise in creating a user interface and then defining what will happen when the user interacts with that interface.

The "event-driven" mindset of programming can take a little getting used to. Often when learning, you write sequential programs that run from start (usually the first line of the program) to the end, or to some point when the program terminates. In an event-driven program your code must always be at the ready to respond to user events, like clicking a button, that may happen at any time, or not at all. More complex event-driven programs require interplay and coordination between so-called "event handlers" - which are functions the programmer writes, but are triggered by the system in response to certain events.

This lesson is a first step toward getting into this mindset. Certain high-level programming languages and environments are designed to make certain tasks easier for a programmer. Being able to design the user interface for an app using a drag-and-drop editor makes designing a stylish product much faster and easier.

## Objectives

**Students will be able to:**

- Use Design Mode to user interface (UI) elements to a screen.
- Create a simple event-driven program by creating user-interface elements with unique IDs and attaching event handlers to them.
- Recognize debugging and responding to error messages as an important step in developing a program.
- Debug simple issues related to event-driven programming

## Links

**Heads Up!** Please make a copy of any documents you plan to share with students.

### For the Students

- **Tutorial - Introduction to Design Mode** - Video ([download](#))
- **Unit 5 on Code Studio**

## Vocabulary

- **Callback function** - a function specified as part of an event listener; it is written by the programmer but called by the system as the result of an event trigger.
- **Event** - An action that causes something to happen.
- **Event-driven program** - a program designed to run blocks of code or functions in response to specified events (e.g. a mouse click)
- **Event handling** - an overarching term for the coding tasks involved in making a program respond to events by triggering functions.
- **Event listener** - a command that can be set up to trigger a function when a particular



App Lab has a way to make the User Interface elements quickly and easily, leaving your brain more free to think about how to write the event handlers.

## Agenda

### Getting Started (10 Minutes)

**What events do familiar apps use to be interactive?**

### Activity (45 Minutes)

### Wrap-up (10 Minutes)

**Share chaser games**

type of event occurs on a particular UI element.

- **UI Elements** - on-screen objects, like buttons, images, text boxes, pull down menus, screens and so on.
- **User Interface** - The visual elements of a program through which a user controls or communicates with the application. Often abbreviated UI.

## Introduced Code

- `onEvent(id, type, function(event)){ ... }`
- `setPosition(id, x, y, width, height)`
- `setSize(id, width, height)`



# Teaching Guide

## Getting Started (10 Minutes)

### What events do familiar apps use to be interactive?

#### Prompt:

- Take out a piece of paper or journal
- Draw a rectangle representing the screen of a mobile device
- Take **one minute** to sketch out what a screen in your favorite app looks like

#### Discussion

Begin the transition to events and event-driven programming by building on existing knowledge of elements and behaviors that are common to most modern applications.

#### Give students a minute to sketch

- Now **make a quick list** of everything on that screen that you can interact with as a user.
- Finally, **write down one action-and-reaction** of the app: one thing you do, and how the app responds.

#### 💡 Discussion:

- Allow students an opportunity to share their sketches and lists with their classmates before asking a few students to share with the entire class.
- Ask students to share their lists of screen elements and how apps respond.
- Likely events will include things like:
  - clicking a button
  - swiping a screen
  - dragging your finger
  - tilting a phone
  - pressing a key, etc.
- Modern apps are interactive because they can respond to this and other forms of user input (i.e., human-generated events).

#### 💡 Teaching Tip

Technical knowledge of how modern applications work is not necessary to run this discussion, and the conversation itself should avoid being overly technical. For now, the point is to get the language of **"events"** out in the open.

- Apps have elements on the screen that you can interact with (i.e. cause user-generated events)
- Apps respond to these **events** in various ways.

You want to be on the lookout for **types of events** we can program with like: mouse click or movement, typing keys on the keyboard, etc. in combination with **types of screen elements** you can perform those actions on like: buttons, menus, text fields, images, etc.

#### 🎤 Remarks

We may not understand all the technical details yet, but it seems clear that most applications we use respond to events of some kind.

Whether we're clicking a button or pressing a key, the computer is sensing **events that we generate** in order to determine how the application should run.

Today, we're going to start exploring how **event-driven programming** makes this possible.

## Activity (45 Minutes)

#### 🖥️ Transition to Code Studio:

#### 🖥️ Code Studio levels

### Unit 5 Lesson 1 Introduction 📄

#### Student Overview



## Introduction to Design Mode

[Teacher Overview](#)[Student Overview](#)

You might consider skipping this video (and coming back to it later) in the interest of time.

[View on Code Studio](#)

- Students should see it at some point but **it is not essential to understanding or completing this lesson.**
- The video provides a general overview of the purpose of Design Mode and a little behind-the-scenes detail of what it's doing.
- You may want to watch or show this video AFTER the lesson to tie things together.
- You may want to ask students to watch it outside of class.

### Levels

[3](#)[4](#)

(click tabs to see student view)

## How onEvent Works

[Student Overview](#)

### Levels

[6](#)[7](#)

(click tabs to see student view)

## Event-Driven Programming Patterns

[Teacher Overview](#)[Student Overview](#)

### Teaching Tips

[View on Code Studio](#)

#### Key Idea:

- There is a pattern to how these programs are constructed and developed in App Lab.

#### Key Behavior:

- Setting up the expectation that programs don't work on the first try. The Run-Test-Debug cycle is part of programming practice. As you get better you learn to write a small amount of code, verify that it works and then move on.
- You can do a lot as a teacher to model this expectation. We call it acting as the "lead learner" in the classroom. In the face of some problem or uncertainty, rather teacher-as-source-of-all-knowledge you model the behavior of a good learner, who says "I don't know, but with some effort and attention to detail, together, I'm sure we can find out."

## Rules About Choosing Good IDs

[Student Overview](#)

### Levels

[10](#)[11](#)

(click tabs to see student view)

## Intro to Debugging and Common Problems

[Teacher Overview](#)[Student Overview](#)



## Teaching Tips

You may want to pause at this point to go over and provide encouragement related to:

1. Key Behavior and Attitude about Debugging
2. Common Types of Errors

Here's an off-the-wall metaphor you might consider trying with your students: **programming and debugging is like getting dressed up to go out.** (stay with me).

 Try This

- You put on some clothes that you think will look good but then you have to look in the mirror, make some adjustments and decisions, maybe even realize you need to go a different direction entirely, and so on -- you are debugging your outfit.
- Writing a program is initially is like throwing on some clothes, and running the program is like looking in the mirror for the first time. You do it to see what it looks like, knowing that you're going to have to make some adjustments.
- But looking in the mirror frequently to see what everything looks like together actually **speeds up the process**. Getting ready to go out, putting on makeup or combing your hair **without** looking in the mirror would not only slow things down, it's foolish.
- The **Run. Test. Debug.** pattern of behavior is part of the programming process, just like using a mirror is part of making yourself presentable.

Understanding that debugging is part of the problem solving process is a **key understanding and behavior** we want to see from students. Many early programmers express frustration when code they write doesn't work the first time, or report that the "computer hates me!". Or that if they write a program that doesn't work or has problems that they are "dumb" or that they'll never get it. This is **exactly the wrong attitude to have**.

Writing a program is not like solving some big problem with blinders on and then checking at the end to see if you were right. It's a process of writing and making adjustments.

### Common Types of Errors

You might point out to students that:

- **Syntax Errors** are the kinds of problems that show errors in the console. In the grand scheme of things **syntax errors are easy problems to solve** because the computer is telling you it can't understand something, you just have to find out what it is.
- **Logic Errors** can be **much harder to solve** because the computer doesn't report anything wrong at all. The program just doesn't do what you think it should or want it to. Tracking down these kinds of errors is much harder, and requires some practice to get used to it. We suggest some techniques in the levels that follow.

## Levels

13

14

15

16

(click tabs to see student view)

## How setPosition and screen dimensions work

Teacher Overview

Student Overview



## Teaching Tip

This might be a good reference to project at the front of the room for students as they work on the last few levels.

### Levels

[18](#)[19](#)[20](#)

(click tabs to see student view)

### How Images Work

[Student Overview](#)

### Finalize Your Chaser Game v.1

[Student Overview](#)

## Wrap-up (10 Minutes)

### Share chaser games

#### Share Applications:

Use a Gallery Walk, Pair-Share, or other strategy to allow students to share their Chaser Games with each other. Encourage students to note design features they would want to include in future applications they create.

#### Remarks

Today we were actually introduced to two tools that will help us build increasingly complex applications. The first was Design Mode, and hopefully it was quickly apparent how powerful this tool is for creating visually appealing and intuitive user interfaces without cluttering up your code. The second tool was `onEvent` which is the general command for all event-handling in App Lab.

Event-driven programs are an important concept in programming, but we've just gotten our feet wet. In the next lesson we'll go further by adding multiple screens, and getting better at debugging.

## Standards Alignment

### CSTA K-12 Computer Science Standards (2011)

- ▶ **CL** - Collaboration
- ▶ **CPP** - Computing Practice & Programming
- ▶ **CT** - Computational Thinking

### Computer Science Principles

- ▶ **1.1** - Creative development can be an essential process for creating computational artifacts.
- ▶ **1.2** - Computing enables people to use creative development processes to create computational artifacts for creative expression or to solve a problem.
- ▶ **5.1** - Programs can be developed for creative expression, to satisfy personal curiosity, to create new knowledge, or to solve problems (to help people, organizations, or society).
- ▶ **5.4** - Programs are developed, maintained, and used by people for different purposes.



If you are interested in licensing Code.org materials for commercial purposes, **contact us**.



# Lesson 2: Multi-Screen Apps

## Overview

Students continue learning about Event Driven programming in this lesson by learning how to add multiple "screens" to an app and adding code to switch between them. More techniques of debugging are presented, namely using `console.log`, a command that allows them to print out text which the user cannot see. It is useful for displaying messages to yourself to figure out what is happening as your program runs. Students will end the lesson by creating an improved version of the "chaser" game which has multiple screens.

## Purpose

As event-driven applications get more complex, it is easy to generate errors in a program, and the need to debug the program will become more prevalent. In some instances, the error will be in the syntax of the program (e.g., a missing semicolon or misspelled function name). In other instances, however, programs will have logical errors which the computer will not catch and so can only be found by testing. Debugging and learning to interpret error messages is a critical step in the process of developing reliable software. Learning about yourself and the types of mistakes you typically make is one aspect of getting good at debugging. Learning how to insert `console.log` statements into your code to display messages that give you insight into what your program is doing and when is also an important, universal technique of program development and debugging.

## Agenda

### Getting Started

#### Recall and Move on

### Activity

#### Instructions for Getting Started and Setup

### Wrap-up

#### Share Chaser/Clicker games

#### Reflection on debugging and error messages

### Extended Learning

### Assessment

## Objectives

### Students will be able to:

- Write a simple event-driven program that has multiple screens.
- Recognize debugging as an important step in developing a program.
- Use `console.log` to debug simple issues related to event-driven programming.

## Preparation

□ Review levels that explain concepts, decide if you would like to demonstrate them or have students read/do on their own.

## Links

**Heads Up!** Please make a copy of any documents you plan to share with students.

### For the Students

- [Unit 5 on Code Studio](#)

## Vocabulary

- **Debugging** - Finding and fixing problems in an algorithm or program.
- **Event-driven program** - a program designed to run blocks of code or functions in response to specified events (e.g. a mouse click)
- **Event handling** - an overarching term for the coding tasks involved in making a program respond to events by triggering functions.

## Introduced Code

- `setScreen(screenId)`
- `console.log`



# Teaching Guide

## Getting Started

### Recall and Move on

Recall or revisit important ideas from the last lesson as necessary.

- Are students comfortable adding buttons, images and text to an app?
- Are students comfortable adding a simple onEvent handler for a button or image?

Address any questions as necessary. Refer students to the instructional pages from the previous lesson that contain diagrams and explanations of how to do these things.

#### **Remarks**

In the last lesson you ended up making a simple "chaser game" that wasn't much of a game.

In this lesson you'll learn improve that app by:

- adding more screens
- and adding a way for the game to end.

Without further ado let's get to it.

## Activity

#### **Transition to Code Studio:**

### Instructions for Getting Started and Setup

- Today students will still work independently but there are a few more problems to solve and mysteries to figure out than in the previous lesson.
- It is recommended that each student have at least one **coding buddy** or thought partner to work through these stages with.
- Students can read instructions together, and ask questions of each other.
- In particular, it's effective to have students do **prediction tasks** with a partner.
- At the end of the lesson it's okay for students to work more independently as they will be touching adding to their chaser game project.

#### **Code Studio levels**

##### Lesson Vocabulary & Resources

 1

(click tabs to see student view)

##### Using Design Mode

 2

 3

(click tabs to see student view)

##### Debugging with Console.log

 4

 5

 6

 7

 8

(click tabs to see student view)

##### Making Multiple Screens

 9

 10

 11

(click tabs to see student view)



(click tabs to see student view)

## Wrap-up

### Share Chaser/Clicker games

Time Permitting it's fun to have students share their work. You can do it a variety of ways.

- Do a gallery walk
- Or have students share their apps by using the Share tools in code studio - via URL or Text Message
- Do small-group demos

### Reflection on debugging and error messages

Discuss

This lesson is one of the first times students will likely have consistently generated and responded to error messages. Students may (incorrectly) view error messages as “bad,” when in reality they are an entirely normal part of programming and extremely useful when debugging code.

In fact, logical error messages, which can be “silent” and don’t generate error messages are much worse, since they are much harder to catch. Use this early moment to normalize getting error messages and needing to debug code.

**Prompt:**

- **“Today was one of the first times we saw error messages in our programs and started thinking about debugging our code. Is it “bad” to generate an error message? Will every error in our programs generate an error? Why might a programmer actually “like” to get an error message?”**

**Discuss:**

Give students an opportunity to share their thoughts, either in small groups or as a class.

Points that might come up:

- Even expert programmers make errors, so **debugging** is a critical step of writing any program.
- Since we can assume that all code will have some errors in it, we’d much prefer the computer to catch those errors for us. Error messages are how the computer gives you a helping hand in writing your program, and often they’ll include helpful information about how you can fix your code.
- Of course, not every error will generate an error message because sometimes we write functional code that does something different than we want. In order to catch these **logical errors**, we’ll need to understand how our code is supposed to run and then test it to make sure that it does.
- In either case, this process of finding and fixing errors in your code is entirely normal and is just as important a skill as writing the code in the first place.

#### **Remarks**

We're making a big deal out of error messages and debugging because they are often hurdles for new learners.

But you just need to have the right attitude about writing code - debugging is part of the process.

You get used to a pattern of:

- Write a little code
- Test it to make sure it does what you think
- Write the next piece

If you do this, the errors you make will tend to be smaller and easier to catch.

## Extended Learning



- Add a "bad guy" to the game. If you click the bad guy instead of the target you lose. Trick: make the bad guy move to a random location **every time the mouse moves** on the screen.
- Make a prediction: Exchange code with a partner and try to figure out what her program does without running it.

## Assessment

If you wish to assess the chaser game, see the teacher notes associated with that level.

### Questions:

1. Which of the following statements about debugging and program errors is FALSE?
  - Error messages help programmers identify problems in their code.
  - Not all errors in a program will generate an error message.
  - Debugging is the process of locating and correcting errors in a program.
  - It is common for programs to contain errors the first time they are written.
  - A program that does not generate any error messages can be assumed to run as intended.
2. Elements in your app are required to have unique IDs. Given what you now know about how event handlers work, why is it important for the IDs of page elements to be unique?

### Teaching Tip

If you didn't before you might consider bringin up this somewhat off-the-wall analogy: **programming and debugging is like getting dressed up to go out.**

- You put on some clothes that you think will look good but then you have to look in the mirror, make some adjustments and decisions, maybe even realize you need to go a different direction entirely, and so on – you are debugging your outfit.
- Writing a program is initially is like throwing on some clothes, and running the program is like looking in the mirror for the first time. You do it to see what it looks like, knowing that you're going to have to make some adjustments.
- But looking in the mirror frequently to see what everything looks like together actually speeds up the process. Getting ready to go out, putting on makeup or combing your hair without looking in the mirror would not only slow things down, it's foolish. The **Run. Test. Debug.** pattern of behavior is part of the programming process, just like using a mirror is part of making yourself presentable.

## Standards Alignment

### CSTA K-12 Computer Science Standards (2011)

- ▶ **CPP** - Computing Practice & Programming
- ▶ **CT** - Computational Thinking

### Computer Science Principles

- ▶ **1.1** - Creative development can be an essential process for creating computational artifacts.
- ▶ **1.2** - Computing enables people to use creative development processes to create computational artifacts for creative expression or to solve a problem.
- ▶ **5.1** - Programs can be developed for creative expression, to satisfy personal curiosity, to create new knowledge, or to solve problems (to help people, organizations, or society).
- ▶ **5.4** - Programs are developed, maintained, and used by people for different purposes.



This curriculum is available under a Creative Commons License (CC BY-NC-SA 4.0).

If you are interested in licensing Code.org materials for commercial purposes, **contact us**.



# Lesson 3: Building an App: Multi-Screen App

## Overview

This lesson gives students time to familiarize themselves with the process of making event-driven apps before we move on to deeper content. They will design and create a (minimum) 4-screen app on a topic of their choosing. There are some other constraints on the project to help guide students in their thinking. Students are also encouraged to do independent work, but alongside a "coding buddy" or "thought partner" to be a help along the way.

**Note:** This activity is **not intended to be a Practice PT** but could be used similarly. The aim is to give an opportunity to get comfortable with Design Mode and the structure of event-driven programming in a creative way. Another goal is to intentionally build in an environment of informal collaboration, even when doing individual work. Suggestions for containing the scope of the project and amount of time allocated to it can be found in the lesson plan.

## Purpose

This lesson is not heavy on new CS content. It is primarily a time to reinforce programming skills in App Lab while quickly prototyping a simple event-driven application. The lesson does, however, fall at the intersection of the Big Ideas of **Creativity and Programming**. The fact that students will share ideas before programming their projects and will provide feedback using a peer rubric also mirrors some of the practices of collaboration that students can employ on the Create Performance Task.

As for the project itself, it probably bears the closest resemblance to creating a "computational artifact" as outlined in the Explore Performance Task -- Creating something to communicate an idea non-textually.

## Agenda

### Getting Started

Introduce the Multi-screen App mini project.

### Activity

Complete the multi-screen app design worksheet and project

Suggested Project timeline

Complete peer review.

### Wrap-up

Incorporate peer feedback

Create PT Prep

## Objectives

**Students will be able to:**

- Develop and design a plan for multi-screen application
- Collaborate with a "thought partner" during the implementation of a project
- Create a multi-screen application in App Lab using simple UI elements and event handling

## Preparation

☐ Print project planning guides (see student documents).

☐ Review the lesson plan to decide how many days of class time you want to use for this mini-project.

☐ Decide how peer review will work (anonymous or not).

## Links

**Heads Up!** Please make a copy of any documents you plan to share with students.

### For the Students

- **Activity Guide - Multi-screen App** - Activity Guide [Make a Copy](#)
- **Unit 5 on Code Studio**

## Vocabulary

- **Event-driven program** - a program designed to run blocks of code or functions in response to specified events (e.g. a mouse click)
- **Event handling** - an overarching term for the coding tasks involved in making a program respond to events by triggering functions.







# Teaching Guide

## Getting Started

### Introduce the Multi-screen App mini project.

#### *Remarks*

Today you will get a chance to make an app of your own design that uses multiple screens and lets you practice using design mode and programming some simple user interactions. We want to spend most of our time working on it, so let's get to it.





- Pair students with a "coding buddy" for this project.
- Students will make a project independently, but have a partner with whom they can get instant and rapid feedback and help.
- See first two levels of Code Studio which you might use as review-and-kickoff to the project.

## Activity

### Complete the multi-screen app design worksheet and project

For a **suggested project timeline** look below the section that shows code studio levels.

#### **Code Studio levels**

- Levels
-  1
-  2
-  3
-  4

#### Student Instructions

[View on Code Studio](#)

## Unit 5: Lesson 3 - Make your own Multi-Screen App

## Background

In the last two lessons we looked at Design Mode and Event-Driven Programming in App Lab. This lesson is all about you getting a chance to use those new skills to make a multi-screen app. Use your creativity and personal interests to make your app unique.

## Vocabulary

#### Review

- **User Interface (UI)** - The "User Interface" or UI of an app refers to how a person (user) interacts with the computer or app.
- **Event-driven program** - a program designed to run blocks of code or functions in response to specified events (e.g. a mouse click)



- **Event handling** - an overarching term for the coding tasks involved in making your app respond to events by triggering functions.

## Lesson

- Choose the theme of your app.
- Complete the Planning Guide.
- Share your plan with a classmate.
- Program your app.
- Give and receive feedback on apps.

## Resources

- Activity Guide: Multi-screen App ([PDF](#) | [DOCX](#))

[Continue](#)

### Student Instructions

[View on Code Studio](#)

# Event-Driven Programming Recap

Before embarking on making your own app from scratch let's recap a few important things:

### Teaching Tip

Perhaps show/review this page as a warm-up just to tie up loose threads from previous lessons.

## 1. Mental Note: UI elements all function basically the same way

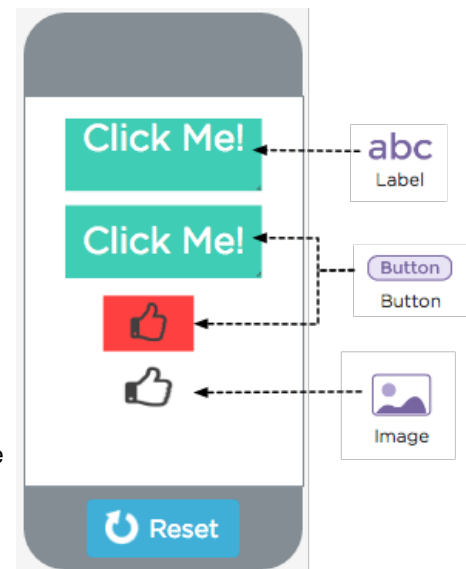
All UI elements **have IDs** and can listen for user events and be used with `onEvent`.

- **Labels** can have a background color, but are designed to be filled with more text.
- **Buttons** have a default styling (green button) that changes slightly in color when you click it. But buttons can also have a background image **and** background color at the same time which is convenient if your image is an icon (red thumbsup).
- **Images** can act a lot like buttons, only they can't have text or a background color.

**Screens** have IDs and you can use `onEvent` with them but they don't quite fit the mould because they have the special property that if you use `onEvent` with a screen, it will capture **every** event of the type you specify regardless of whatever other elements are clicked on.

### What UI element should you use?

There are no particular rules. Use the UI element that makes the most sense to you and is easiest for you use and do what you want. The UI elements actually aren't **exactly** the same, but the basic event-driven



**This example shows how buttons, labels and images can be made to look very similar. They can all behave similarly as well depending**



## 2. There is a Pattern to Developing Event-Driven Programs in App Lab

You will find yourself going through this process over and over again. The point is not that it's boring or repetitive (far from it!), but rather that **when you have an idea** for something to make, you know you can apply this process to get it done.

**Step 1 - Design Mode**

**Step 2 - Add onEvent to Code**

**Step 3 - Write the code for the event handling function**

**Step 4 - Run. Test. Debug.**

**Repeat**

As you learn more, of course, you'll see there are nuances to these steps, but these are the big ones.

**Click continue to see "Tips for Working on Your Own"**

### Student Instructions

[View on Code Studio](#)

## Tips for working on your own

As you are about to embark on your first solo project we thought it would be a good time to give you some tips.

### Tip 1: Have a "coding buddy" and "thought partner"

Working on your own doesn't mean working by yourself. It's very useful to have someone nearby who you can use as a "thought partner". Many professionals work at the same table or desk, even if they are working on completely different projects, because of the benefits of having someone nearby. There's a lot to remember and a lot to try to keep straight, so it's helpful to have someone nearby who provide another perspective.

What Thought Partners Do	How it might sound
Bounce ideas off each other	"Hey, would it be cool if I tried x, y, or z?"
Share insights or discoveries they've made through their own programming	"Whoa! I didn't know it could do that! Check it out!"
Answer each other's questions in the moment	"What's the command to change the location of an object again?"



Help Double-check code and provide a second pair of eyes for debugging

💡 Teaching

"Gah! This is driving me nuts. Can you look at this? What **Teaching Tip?**"

You may want to review the 3 things on this page as a whole class to kickoff to the upcoming project.

Especially true for talking about "coding buddies" which is a form of collaboration.

For this and future projects it's worth pointing out that **even if you're doing independent work, you can have a buddy to help you with technical problems and to bounce ideas off of.** Remind students that:

- It's not a competition!
- Work on your own ideas, be generous when helping others.
- You should **always write your own code** but you can have a friend help you spot problems.

Plagiarism can present a gray area for students here.

"Help" means:

- Helping a friend work through their ideas
- Helping a friend get "unstuck" from a bug of a particular kind.
- Suggesting a strategy for getting something done.
- Pointing out a cool idea

"Help" **does not** mean:

- Writing code for a friend
- Giving your project to a friend to use as a starting point
- Telling your friend what to do

## Tip 2. Persistence Pays off

When you are learning to program, you will inevitably run into problems.

As you get better, this doesn't change :) Only the types of problems do. Like anything else, over time you stop making the same mistakes you made as a novice, and in fact, you don't even think about them as mistakes.

### Remember:

1. No program ever works correctly the first time
2. The whole point is to build something up in small increments
3. You can't break anything. Add code, try it out. Doesn't work? Get rid of it. Try something else. No big deal.
4. Add. Run. Test. Debug.

### Got problems?:

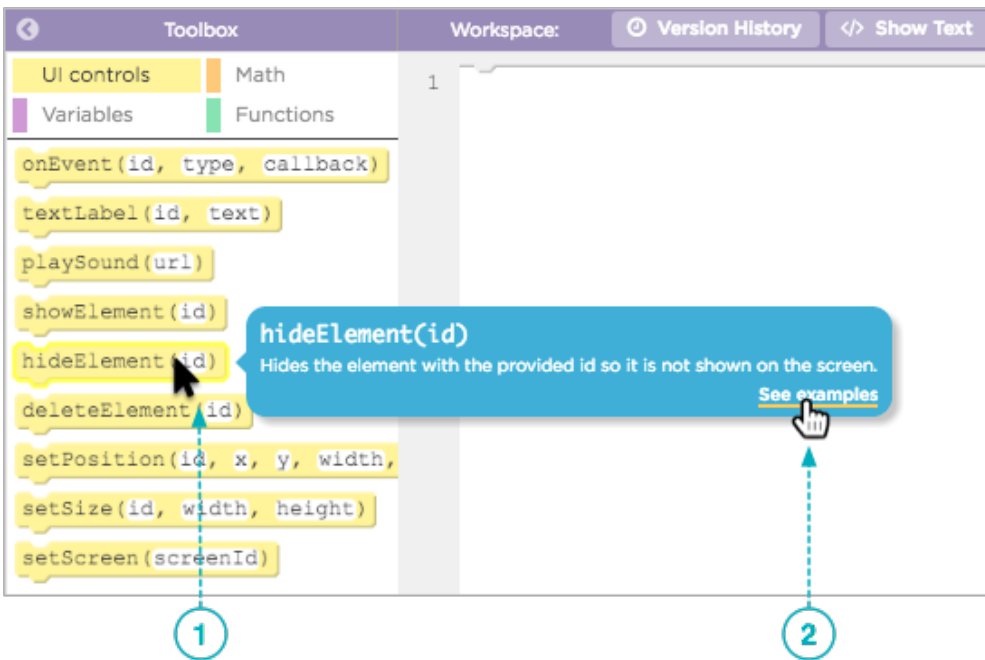
- If you run into a snag where something isn't working stay calm and **work the problem** -- This is where having a program buddy and thought partner really helps.
- **There is a reason** why it's not working, you just have to find it.
- Once you've solved a problem or bug, **you've learned something** and you're less likely to make that same kind of mistake again.

**Stick with it. It pays off!**

## Tip 3. Use online documentation - some new commands

You'll see we've included a more full toolbox of commands for you to use and experiment with. Some of the commands may be new to you, but you can probably figure out how to use them if you **read the documentation**.





1. Hover your mouse over a block and it will show a tool tip with a brief explanation.
2. Click on "see examples" to expand the documentation. It gives more explanation plus code examples that show how it works.

**With those lessons learned...click continue to start making your own app!**

## Student Instructions

[View on Code Studio](#)

# Multi Screen App

You will be creating your own multi-screen app to practice designing user interfaces and writing event-driven programs.

**Look at the Project Guide and Planning Sheets before programming .**

## NOTE: Bigger toolbox

- You may notice that we've included all of the commands you know so far in the coding toolbox **plus a few more**
- Remember you can hover over a command to see documentation for it.
- You can also just try it out to see what it does.

## Requirements Reminder

- Your app must have a purpose
- Your app will have at least **4 screens**.
- Your app should include **text, images, buttons, and sound** .
- There should be **no "getting stuck" on any screen**. It should always be possible to navigate from a screen in your app to some other screen.
- Your program code should follow **good style**.
- Your user interface should be **intuitive to use**.

## Suggested Project timeline



A proposed schedule for doing this project entirely in-class is shown below. See “Teaching Tips” for alternatives.

This project can take anywhere between **one to three days** depending how much time you want to spend in class working on it.

#### 💡 Possible Timeline:

##### Day 1 - Review and Start Planning

##### Distribute: Activity Guide - Multi-screen App - Activity Guide

- Review the project requirements, process, and timeline, review process, and rubric.
- Answer any questions and move onto the planning / sketching stage.

#### 💡 Planning: Students use Planning Guide to sketch out multi-screen app.

- Example provided in Activity Guide.

**Peer Review:** Share app sketch with a classmate to get feedback. Students should focus on giving feedback about

- Connections between pages
- Descriptive IDs
- Design/ Layout

🖥️ **Programming:** Students start programming their apps.

##### Day 2 -- Start/Continue programming your multi-screen app

**Programming:** Work day. Students should continue working on app.

**Goal:** Students' app should be completely functional by the end of Day 2 and ready for debugging tests.

##### Day 3 -- Finalize their app, get feedback

- Focus should be primarily on debugging and making final aesthetic changes.
- Students should go through the rubric themselves first before sharing it with peers.

#### 💡 Complete peer review.

**Peer Review:** Set up peer reviews of students' final apps.

**Wrap-up:** Students will improve one piece of their app, based on feedback.

#### 💡 Teaching Tip

This mini-project is a good candidate for asking students to work on outside of class. For example, you might do this, which would save you the better part of 2 class days:

**Day 1:** Take 10 minutes to simply introduce the project, and move on to the next lesson.

- Do planning/sketching outside of class.
- Do programming outside of class.

**Day 2:** (some days later): Full class day. Do gallery walk and peer review.

#### 💡 Teaching Tip

**Function Before Design** Encourage students to work on getting working connections between screens before focusing on layout. Function before Design should be something students get used to as it's more important. Design can always be improved.

## Wrap-up

### Incorporate peer feedback

#### Improve App:

Give students a chance to respond to the feedback they receive on their app. They should pick at least one piece of feedback to implement in their app. This could be done outside of class, if desired.

### Create PT Prep



#### 💡 Teaching Tip

**Peer Review:** One strategy for peer review, especially if you haven't done any in the class up to this point, is to do a "double-blind" review where both the programmer and the reviewer are anonymous to each other. To do this:

- Have students share their apps via the share link, or bring up on another device.
- Assign each student to do a peer review of one or two other apps, using the rubric provided.
- You'll need some way for students to indicate which app they are reviewing.
- Collect feedback forms and return to original programmer.
- You might also assess students on the quality of their feedback. (Students will remain anonymous to each other, but you'll know who reviewed what.)

## Code Studio levels

- Levels
-  5

## Student Instructions

[View on Code Studio](#)

# AP Practice - Create PT - Process

One component of the **AP Create Performance Task** is describing the development process used for your program.

### 2. Written Responses

2b. Describe the incremental and iterative development process of your program, focusing on two distinct points in that process. Describe the difficulties and/or opportunities you encountered and how they were resolved or incorporated. In your description clearly indicate whether the development described was collaborative or independent. At least one of these points must refer to independent program development. *(Must not exceed 200 words)*

Here's two rows of the scoring guide for this question



Row 2  Developing a Program with a Purpose	RESPONSE 2B	<ul style="list-style-type: none"> <li>Describes or outlines steps used in the incremental and iterative development process to create the entire program.</li> </ul>	<b>Do NOT award a point if any one of the following is true:</b> <ul style="list-style-type: none"> <li>the response does not indicate iterative development;</li> <li>refinement and revision are not connected to feedback, testing, or reflection; or</li> <li>the response only describes the development at two specific points in time.</li> </ul>
Row 3  Developing a Program with a Purpose	RESPONSE 2B	<ul style="list-style-type: none"> <li>Specifically identifies at least two program development difficulties or opportunities. <b>AND</b></li> <li>Describes how the two identified difficulties or opportunities are resolved or incorporated.</li> </ul>	Response earns the point if it identifies two opportunities, or two difficulties, or one opportunity and one difficulty AND describes how each is resolved or incorporated.  <b>Do NOT award a point if any one of the following is true:</b> <ul style="list-style-type: none"> <li>only one distinct difficulty or opportunity in the process is identified and described; or</li> <li>the response does not describe how the difficulties or opportunities were resolved or incorporated.</li> </ul>

## Grade the Response

A student wrote the following response.

**"In developing my program I encountered two major problems. The first one was early in programming when sometimes the apple would disappear from the screen. By debugging my program I was able to recognize that the ranges of my random X values went from 0-3200, not 0-320. I was easily able to correct this issue in my code. A second issue occurred when I realized that classmates using the game often didn't know how to start it. In order to fix this problem I made the button significantly larger and a different color than the background. Afterwards I no longer saw this problem."**

Each row is worth one point that either can or cannot be awarded **Explain why you would or would not award the points for Row 2 and Row 3.**

Hint: Pay particular attention to the last column of the scoring guidelines and the checklists entitled "Do NOT award a point if..."

Have students complete the Create PT prep question that appears at the end of the lesson.

## Assessment

**Rubric:** Use the provided rubric (in the Activity Guide), or one of your own creation, to assess students' submissions.

### Extended Assessment:

If you want to make the project more like a practice performance task you could have students write responses to applicable reflection prompts from the real Performance tasks.

You might modify these slightly for this context, but useful prompts are:

From **Create PT**:

- 2b. "Describe the incremental and iterative development process of your program, focusing on two distinct points in that process. Describe the difficulties and/or opportunities you encountered and how they were resolved or incorporated. In your description clearly indicate whether the development described was collaborative or independent. At least one of these points must refer to independent program development; the second could refer to either collaborative or independent program development. **(Approximately 200 words)**"

From **Explore PT**:

- 2b. "Describe your development process, explicitly identifying the computing tools and techniques you used to create your artifact. Your description must be detailed enough so that a person unfamiliar with those tools and techniques will understand your process. **(Approximately 100 words)**."



# Standards Alignment

## Computer Science Principles

- ▶ **1.1** - Creative development can be an essential process for creating computational artifacts.
- ▶ **1.2** - Computing enables people to use creative development processes to create computational artifacts for creative expression or to solve a problem.
- ▶ **5.1** - Programs can be developed for creative expression, to satisfy personal curiosity, to create new knowledge, or to solve problems (to help people, organizations, or society).
- ▶ **5.4** - Programs are developed, maintained, and used by people for different purposes.



This curriculum is available under a  
Creative Commons License (CC BY-NC-SA 4.0).

If you are interested in licensing Code.org materials for commercial purposes, **contact us**.



# Lesson 4: Controlling Memory with Variables

## Overview

This lesson gets into the basic mechanics of working with variables in programs. The lesson shows students how to create and assign values to variables and navigates through a series of common misconceptions about variables and how they work. Along the way, the lesson tries to build up the student's mental model of how computers and programs work, which is essential for being able to reason about programs.

## Purpose

Developing a good mental model for how variables work in computer programs is absolutely essential to long-term success as a programmer. However, because most students have had years' worth of math classes before taking this course, there are two **major misconceptions** that early students often have about variables. We suggest that you try to avoid relating this material to mathematics at all. Some of the words and symbols are the same, but:

- The = sign in programming is an instruction to store a value in memory, NOT a statement of equality.
- "Variables" in computer programming are just named pieces of memory, NOT unknowns in an equation or symbols for undetermined values.

Thus, lines of code that assign values to variables and expressions that use variables are really instructions to retrieve and store values in memory. And those values change while the program executes. Being able to reason about what's happening in consecutive lines of code like:

```
a = a + b;
```

```
b = a + b;
```

correlates highly with a person's success in programming because you must have a good mental model for program execution and what the computer is doing.

## Agenda

### Getting Started

**Recall patterns in making event-driven apps.**  
**Motivate the need for variables in our programs to make them more useful.**

## Objectives

**Students will be able to:**

- Use variables in a program to store numeric values.
- Store the value returned by a function (randomNumber, promptNum) in a variable for use in a program.
- Debug problems related to variable re-assignment.
- Write arithmetic expressions that involve variables.
- Reason about multi-line segments of code in which variables are re-assigned multiple times.

## Links

**Heads Up!** Please make a copy of any documents you plan to share with students.

### For the Students

- **Tutorial - Introduction to Variables Part 1** - Video ([download](#))
- **Tutorial - Introduction to Variables Part 2** - Video ([download](#))
- **Unit 5 on Code Studio**

## Vocabulary

- **Data Type** - All values in a programming language have a "type" - such as a Number, Boolean, or String - that dictates how the computer will interpret it. For example 7+5 is interpreted differently from "7"+"5"
- **Expression** - Any valid unit of code that resolves to a value.
- **Variable** - A placeholder for a piece of information that can change.

## Introduced Code



## Activity

### App Lab: Controlling Memory with Variables

## Wrap-up

Foreshadow adding variables to apps.

## Extended Learning

- `write(text)`
- `value1 + value2;`
- `num1 - num2;`
- `num1 * num2;`
- `num1 / num2;`
- `randomNumber`
- `var x = ____;`
- `x = ____;`
- `var x = promptNum("Enter a value");`
- `var x = " ____";`



# Teaching Guide

## Getting Started

### Recall patterns in making event-driven apps.

You are now pretty well acquainted with the basic mechanics of making event-driven apps. There is a pattern to writing these programs that you should be used to:

- Add UI elements to the screen.
- Give the UI elements meaningful IDs.
- Add event handlers to those elements.

### Motivate the need for variables in our programs to make them more useful.

#### Moving Forward

However there's a whole bunch of things that we can't do in our apps yet. The next step is to learn how to control the computer's memory to remember things while the program is running.

Most apps keep track of some information that changes and updates as you use the app. For example, a simple game can keep track of your score and the number of lives you have left as you play.

Note that keeping track of data while a program is running is different from remembering things in the long term across multiple uses of the app, things like storing the all-time high score or remembering your user profile.

Most programs need to use memory for even basic processing tasks. App Lab already keeps track of a lot of things for you in memory without you doing anything, like the position and styling of elements on the screen, especially if they are moving around.

But you will want to write programs that keep track of data that's not "built-into" the programming environment. These apps use and control the computer's memory to do this, and learning how to use memory in programs is a powerful skill to have. Today we'll start!

## Activity

### App Lab: Controlling Memory with Variables

#### Transition to Code Studio

#### Transition:

The programming tasks in this lesson acquaint you with basics of working with variables and building up a mental model for how programs use and manage memory. To keep things simple, the output will mostly be simple text displayed to the app screen or debug console. In the next lesson we'll apply what you learn to an app for a simple game.

#### Teaching Tip

The Variables concept video is embedded in one of the early levels in the lesson on Code Studio, but it's recommended that you watch the video as a class so you can make transitional or motivating comments before sending students to work in Code Studio.

#### Code Studio levels

### Unit 5 Lesson 4 Introduction

#### Student Overview



## Basic mechanics of variables

[Student Overview](#)

## Introduction to Variables - Part 1

[Student Overview](#)

### Levels

[4](#)[5](#)[6](#)[7](#)[8](#)*(click tabs to see student view)*

## Controlling Memory - Other ways to assign values

[Student Overview](#)

### Levels

[10](#)[11](#)[12](#)[13](#)[14](#)[15](#)[16](#)*(click tabs to see student view)*

## Introduction to Variables - Part 2

[Student Overview](#)

## The Mental Model for Variables

[Student Overview](#)

### Levels

[19](#)[20](#)[21](#)[22](#)[23](#)[24](#)[25](#)[26](#)[≡ 27](#)*(click tabs to see student view)*

## Wrap-up

### Foreshadow adding variables to apps.

#### Remarks

Now that you've had a fair amount of practice working with the basic mechanics of variables, and learning how to debug your own problems, you're more than ready to start using variables in apps.

This lesson is subtly one of the most important for you as a programmer. Being able to answer questions like the last multiple choice question in the lesson on Code Studio means that you have a good mental model for how programs execute code and how machines work.

Some research has shown that being able to answer questions about simple variable re-assignment correlates highly with doing well in programming overall. So you're on your way!

## Extended Learning

Students can make their own program that prompts the user for some numeric values and then performs an action. The user input values could be used to print a calculation to the screen, or they could be used to control some part of a turtle drawing (such as the number of times to repeat an action).

## Standards Alignment

### CSTA K-12 Computer Science Standards (2011)

- ▶ **CPP** - Computing Practice & Programming
- ▶ **CT** - Computational Thinking



## Computer Science Principles

- **5.2** - People write programs to execute algorithms.



This curriculum is available under a  
Creative Commons License (CC BY-NC-SA 4.0).

If you are interested in licensing Code.org materials for commercial purposes, **contact us**.



# Lesson 5: Building an App: Clicker Game

## Overview

In this lesson, students add variables to two different exemplar apps to keep track of a score, or a count of some number of button clicks. The major topic is **variable scope** and understanding the differences, benefits, and drawbacks, of using global versus local variables. This lesson focuses more on using global variables, since in event-driven apps that's what you need to keep track of data across multiple events.

The very basics of a **simple if statement** are also presented in this lesson, mostly to highlight the difference between the `=` and `==` operators. Finally, students are asked to apply what they've learned about variables, scope, and if statements, to make their own "clicker" game modeled after one of the exemplars they saw during the lesson.

## Purpose

This lesson is mostly a continuation and furthering of our understanding of variables and how they work. There are many, many pitfalls and misconceptions about variables and how to use them in programs for the early learner. Variables are often difficult to learn because they are not visual, they are abstract, and one must have a good mental model for what's happening in the computer and with program instructions, in order to reason about the code and develop one's own solutions to problems.

The topic and concept of variable scope is a big one in any programming language. However, since many languages do it differently, the concept of variable scope isn't listed explicitly as a learning objective in the CSP framework. As a concept, though, variable scoping is a form of abstraction - a programming language lets you create variables with as narrow or broad a scope as you need to program a certain task. As a general practice, you usually want to create variables with the most narrow scope you can for the task at hand, since the other extreme - everything is global - can become unwieldy or hard to manage, and it doesn't promote code reuse.

## Agenda

### Getting Started

Recall basic mechanics and terminology of working with variables

### Activity

App Lab: Building an App - Clicker Game

## Objectives

### Students will be able to:

- Use global variables to track numeric data in an app.
- Give a high-level explanation of what "variable scope" means.
- Debug problems related to variable scoping issues.
- Modify existing programs to add and update variables to track information.
- Create a multi screen "clicker" game from scratch

## Preparation

☐ Decide whether you want to introduce the activity guide at the beginning of the lesson or the end.

☐ Familiarize yourself with the Clicker Game and rubric, decide how to organize peer review.

## Links

**Heads Up!** Please make a copy of any documents you plan to share with students.

### For the Students

- **Activity Guide - The Clicker Game** - Activity Guide [Make a Copy](#)
- **Unit 5 on Code Studio**

## Vocabulary

- **==** - The equality operator (sometimes read: "equal equal") is used to compare two values, and returns a Boolean (true/false). Avoid confusion with the assignment operator "=",
- **Global Variable** - A variable whose scope is "global" to the program, it can be used and updated by any part of the code. Its global scope is typically derived from the variable being declared (created) outside of



## Wrap-up

### Peer Review of Clicker Games

any function, object, or method.

- **If-Statement** - The common programming structure that implements "conditional statements".
- **Local Variable** - A variable with local scope is one that can only be seen, used and updated by code within the same scope. Typically this means the variable was declared (created) inside a function -- includes function parameter variables.
- **Variable Scope** - dictates what portions of the code can "see" or use a variable, typically derived from where the variable was first created. (See Global v. Local)

## Introduced Code

- `setText(id, text)`
- `if( ){ //code }`
- `__ == __`



# Teaching Guide

## Getting Started

### Recall basic mechanics and terminology of working with variables

**Recall** In the previous lesson, we learned about the basic mechanics of working with variables in JavaScript. We developed a mental model for thinking about how values are stored and retrieved from memory and that we should read the “=” sign as “gets” to avoid confusion.

**Moving forward** The whole purpose of learning about variables though is so that our apps can make use of them while a program is running. In this lesson, we’ll see how to do that. So let’s get to it.

## Activity

### App Lab: Building an App - Clicker Game

#### Teacher Note:

- This **Activity Guide - The Clicker Game - Activity Guide** is not strictly needed until the **very end** of the lesson -- it is referred to in level 21
- However, if you think it would provide some motivation, you may want to optionally show this activity guide at the beginning of the lesson.

#### Code Studio levels

##### Lesson Vocabulary & Resources

 1[\(click tabs to see student view\)](#)

##### Clicker Game Demonstration

 2[\(click tabs to see student view\)](#)

##### AppLab Practice

 3 4 5 6[\(click tabs to see student view\)](#)

##### Reflection on Debugging

 7[\(click tabs to see student view\)](#)

##### Variable Scope: Local vs. Global

 8[\(click tabs to see student view\)](#)

##### Debugging Variables

 9 10 11 12[\(click tabs to see student view\)](#)

##### Using Global Variables

 13 14[\(click tabs to see student view\)](#)

##### Simple Decisions with if-statements

 15 16 17 18 19 20[\(click tabs to see student view\)](#)



**(new) AP Practice Response - Choosing an Abstraction***(click tabs to see student view)*

## Wrap-up

### Peer Review of Clicker Games

**Activity Guide - The Clicker Game - Activity Guide** contains a rubric that students can use to evaluate their classmates' apps. It is up to you to determine who should evaluate which programs, how to pair or group students, and the degree of anonymity you wish to maintain.

Peer review should be a useful activity for students in preparation for the Create Performance Task in which they need to give and receive feedback with a partner to improve their programs.

**Transition:** Now that we understand a bit about variables and how to use them in our programs, a whole new world will open to us. First, we will learn that variables can hold **other kinds of data besides numbers**. We'll also learn other ways to **get data from the user** using other UI elements like text input, pull-down menus, and so forth.

## Standards Alignment

### CSTA K-12 Computer Science Standards (2011)

- ▶ **CL** - Collaboration
- ▶ **CPP** - Computing Practice & Programming
- ▶ **CT** - Computational Thinking



This curriculum is available under a Creative Commons License (CC BY-NC-SA 4.0).

If you are interested in licensing Code.org materials for commercial purposes, **contact us**.



# Lesson 6: User Input and Strings

## Overview

In this lesson, students are introduced to the string data type as a way of representing arbitrary sequences of ASCII characters. They will use strings to accept input from a user as they work on mastering two new UI elements, the text input and the text area. Students combine these skills to develop a simple Mad Libs® app.

**Mad Libs® is a trademark of the Penguin Group (USA) LLC., which does not sponsor, authorize or endorse this site.**

## Purpose

Strings are a feature of essentially every programming language, and they allow for variable-length pieces of text to be represented, stored, and manipulated. While a single string can be stored in a variable, it is worth noting that a string will typically use much more memory than a number. Numbers are typically stored in fixed-width 8-, 16-, 32-, or 64-bit chunks. ASCII characters require a single byte and so a string of 100 characters, short by most standards, would require 800 bits in order to be stored in memory. While “typed” programming languages require you to declare the size and type of a variable before you use them, in more dynamic programming languages, including JavaScript, this memory management is abstracted away.

## Agenda

### Getting Started

Explore a Mad Libs app and plan your own

### Activity

App Lab: User Input and Strings

### Wrap-up

## Objectives

**Students will be able to:**

- Identify strings as a unique data type which contains a sequence of ASCII characters.
- Describe characteristics of the string data type.
- Accept string input in a program.
- Manipulate user-generated string input to generate dynamic output.

## Links

**Heads Up!** Please make a copy of any documents you plan to share with students.

### For the Students

- **Unit 5 on Code Studio**
- **Activity Guide - Mad Libs** - Activity Guide [Make a Copy](#)

## Vocabulary

- **Concatenate** - to link together or join. Typically used when joining together text Strings in programming (e.g. "Hello, "+name)
- **String** - Any sequence of characters between quotation marks (ex: "hello", "42", "this is a string!").

## Introduced Code

- `getText(id)`
- `var x = prompt("Enter a value");`
- `str.toUpperCase`
- `str.toLowerCase`



# Teaching Guide

## Getting Started

### Explore a Mad Libs app and plan your own

**Explore:** Students should begin the lesson by moving to the first activity in Code Studio where they will use a Mad Libs app. Over the course of this lesson students will develop skills that will allow them to build their own Mad Libs app by accepting user input as strings. Note: After students move to Code Studio, they should complete the Activity Guide before continuing.

💡 **Distribute:** the **Activity Guide - Mad Libs - Activity Guide**. Students should use this opportunity to decide on what the theme of their Mad Libs app will be, what text they will accept into their app, and how it will be incorporated into its output. The primary guidelines of the project (also included in the Activity Guide) are:

#### 💡 Teaching Tip

Put a time limit (e.g., 5-10 minutes) on this brainstorming session. It is intended to drive interest in and provide context for the coming activities. Students should feel free to change their ideas later in the lesson when they actually build their Mad Libs app.

- The app should be a “how-to” Mad Libs (e.g., “How to take care of your pet ostrich”). Afterwards, you list steps with key components left open for user input. This is primarily to help students quickly conceive of ideas.
- There should be at least 3 steps in their instructions.
- Their app should accept at least 3 pieces of user input.

Before moving into the rest of Code Studio, students should have a rough outline of their project.

Once they have completed their outlines, students should return to Code Studio.

## Activity

### App Lab: User Input and Strings

#### 🖥️ Code Studio levels

#### Unit 5 Lesson 6 Introduction 📄

#### Student Overview

#### Levels

[2](#)[3](#)[4](#)[5](#)[6](#)[7](#)[8](#)[9](#)[10](#)[11](#)[12](#)[13](#)[14](#)[15](#)

(click tabs to see student view)

## Wrap-up

**Share:** Once students have completed their applications they should share their work with their peers, trying one another’s Mad Libs.

## Standards Alignment

CSTA K-12 Computer Science Standards (2011)



- ▶ **CL** - Collaboration
- ▶ **CPP** - Computing Practice & Programming
- ▶ **CT** - Computational Thinking

### **Computer Science Principles**

- ▶ **5.1** - Programs can be developed for creative expression, to satisfy personal curiosity, to create new knowledge, or to solve problems (to help people, organizations, or society).
- ▶ **5.3** - Programming is facilitated by appropriate abstractions.



This curriculum is available under a  
Creative Commons License (CC BY-NC-SA 4.0).

If you are interested in licensing Code.org materials for commercial purposes, **contact us**.



# Lesson 7: If-statements unplugged

## Overview

We take a whole lesson to learn about if statements, what they are, the terminology around them, and what they have to do with "selection" in programs. Students trace simple robot programs on paper to develop a sense of how to read and reason about code with if statements in it. Students also try their hand at writing code by hand to handle a robot situation.

## Purpose

The activities here get right to many common misconceptions about how if-statements work and how programs execute. Students may have a simple common-sense intuition about how if-statements work, but there are several ways you can get your wires crossed when considering how programs actually execute. There are two main issues: 1) how the flow of program execution works, and 2) How complex logical statements are composed and evaluated. In this lesson we just address program flow and tracing execution. We'll look at more complex logical expressions later. Even though Boolean expressions show up in this lesson, we try to avoid using that term until the next lesson. For this lesson it's a condition that is simply true or false.

## Agenda

### Getting Started (5 mins)

When v. If

### If-statements Unplugged (40 mins)

"Will it crash?" Activity

### Wrap Up (20 mins)

What was trickiest?

Algorithms and Creativity

## Objectives

**Students will be able to:**

- Reason about if-statements by tracing pseudocode programs by hand
- Write a short program in pseudocode that uses if statements
- Explain the purpose of if-statements in programs

## Preparation

☐ Decide whether or not to print the "Will it Crash?" Activity Guide for students (it's ~6 pages, but nice to have on paper. There are digital alternatives, though)

☐ Decide how students will review the first two code studio pages - see teaching tips.

☐ Budget time: the main activity is working through the problems in the "will it crash?" activity - keep in mind that the last problem ask students to write code which may take time as well.

## Links

**Heads Up!** Please make a copy of any documents you plan to share with students.

### For the Teacher

- **KEY - Will it Crash?** - Answer Key

### For the Students

- **Will it Crash?** - Activity Guide  
[Make a Copy](#)
- **Annotated Pseudocode: if-statements and Robot** - Resource [Make a Copy](#)
- **Breakdown - If-statements explained** - Resource [Make a Copy](#)
- **Unit 5 on Code Studio**

## Vocabulary

- **Conditionals** - Statements that only run under certain conditions.



- **If-Statement** - The common programming structure that implements "conditional statements".
- **Selection** - A generic term for a type of programming statement (usually an if-statement) that uses a Boolean condition to determine, or select, whether or not to run a certain block of statements.



# Teaching Guide

## Getting Started (5 mins)

### When v. If

- Most of the programs you've written so far have event handlers that get triggered when certain events occur.
- But in the last program - the version of "Apple Grab" - we had a very simple `if` statement that said something like:

```
if(count==20){
    setScreen("gameOver");
}
```

#### Discussion

Distinguish between events ("when") and conditional statements ("if"). In everyday conversation, it is common to interchange the words "when" and "if," as in "If the user presses the button, execute this function."

In programming event-driven apps, "when" should refer to an event and "if" should refer to a program executing some conditional logic – deciding whether to run some code, based on a boolean condition.

- The introduction of "if" introduces an English language issue for us moving forward. Here is an example:

#### Prompt:

**I'm going to read out loud two sentences that describe a program. With a partner discuss what the difference is between them, and decide which one is "right". Here are the two sentences:**

1. **When** the button is clicked add one to the score.
2. **If** the button is clicked add one to the score.

Give pairs a minute to discuss and then ask which one people think is "right." Get a few opinions out, but don't stop here.

#### Let's try another one:

1. **When** the score reaches 20, set the screen to "game over."
2. **If** the score reaches 20, set the screen to "game over."

Give pairs a minute to discuss and then ask which one people think is "right".

#### Discuss

Points to raise during discussion:

- There is no right answer. In **English** both pairs of sentences mean basically the same thing.
- However in **programming**, using the words "if" and "when" map to some expectations about how the underlying code is written.
- Here is the difference:
  - **"When"** is used in reference to an **event** -- **When something happens respond in such and such a way.**
  - **"If"** is used in reference to a **decision** about whether or not to execute a certain piece of code -- **If something is true, then do this, otherwise do that .**
- When describing the behavior of a program events and decisions might get mixed together. For example:
  - **"When the button is clicked, if the score is 20 go to 'game over', otherwise add one to the score".**

#### Teaching Tip

Don't get too hung up on word-parsing "when" v. "if". This distinction is nuanced and in the long run is actually not hugely important. But it can be a distraction early on because of ambiguities in common English language usage - which is why we draw attention to it here.

As students gain more experience with if statements, the difference between events and if statements will likely become more clear and obvious.

For now, the **key idea** is that "if" statements are new entity that let us do things we could not do with event handlers.

#### Transitional Remarks






Today's activity focuses solely on **if statements**.

If the distinction between "when" and "if" is still a little fuzzy, that's okay.

For now, the **key idea** is that if statements are a new entity that let us do things we could not do with event handlers - writing code to make decisions about whether or not to run some other piece of code.

## If-statements Unplugged (40 mins)

### Code Studio levels

- Levels
-  1
-  2
-  3

### Student Instructions

[View on Code Studio](#)

## Unit 5: Lesson 7 - If Statements Unplugged

[View on Code Studio to access answer key\(s\)](#)

## Background

We take a whole lesson to learn about if statements, what they are, the terminology around them, and what they have to do with "Selection" in programs. We trace simple robot programs on paper to develop a sense of how to read and reason about code with if statements in it.

## Lesson

- Introduction to if statements with the AP pseudocode
- A worked example with if statements and the "robot"
- "Will it Crash?" Activity

## Vocabulary

- **if Statement** - The common programming structure that implements "conditional statements".
- **Conditionals** - Another term for if statements -- statements that only run under certain conditions.
- **Selection** - A generic term for a type of programming statement (usually an if-statement) that uses a Boolean condition to determine, or select, whether or not to run a certain block of statements.

## Resources

### In order of usage:

- Resource - **Breakdown - If-statements explained** (embedded in code studio, next page)
- Resource - **Annotated Pseudocode: if-statements and Robot** (embedded in code studio, next pages)
- Activity Guide - Will it Crash? ([PDF](#) | [DOCX](#))

[Continue](#)



# Big-Picture: If-statements

## Word Soup: If-statements, Conditionals, Selection

- **If-statements** exist so that your program can adapt, respond, or make choices about whether or not to execute certain portions of code based on some condition that you check while the program is executing.
- Because it involves checking conditions, these statements are sometimes called **conditional statements**, or sometimes just **conditionals**.
- A conditional statement (if-statement) requires a **conditional expression**, something that is either **true** or **false** and it's what an if-statement uses to decide whether or not to execute a certain portion of code.
- A generic term used by the AP CSP Framework for this is **Selection**. As in: your program can **select** whether or not to run certain blocks of code.

The important part here is to familiarize students with AP Pseudocode for if-statements and the robot commands.

Some of the terminology can be reviewed later by students.

### 💡 Teaching Tip

Review this as a class and: **point out the key idea** have students read the annotations in the pseudocode documentation.

## Key Idea: If-statements are how programs adapt and respond to conditions on the ground.

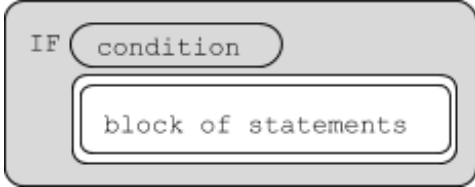
The whole point is to be able to handle cases where you can't know ahead of time whether or when certain conditions will exist in your program. So you have to write code to say something like: "Okay, at this point in the program, if **such and such** is true, then do **this**, otherwise do **that**."

## Practice with the AP Pseudocode

For the activities that follow, we're going to get our feet wet with if-statements using the **AP CS Principles pseudocode** language. To start we're going to use the IF/ELSE structures, but to keep things simple we'll only use the Robot conditional expression `CAN_MOVE (direction)` - which evaluates to **true** or **false**.

Below is the official documentation for Selection and Robot statements along with some extra commentary.



Selection	
<p><b>Text</b>  IF (condition)  {    &lt;block of statements&gt;  }</p> <p><b>Block</b></p> 	<p>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.</p> <p><b>Commentary:</b>  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, <i>just ignore the whole block</i> and pick up executing the code that comes after it.</p>

View as a separate document: **Annotated Pseudocode - If-statements and Robot**

## Student Instructions

[View on Code Studio](#)

# A Worked Example

The following shows a step-by-step, line-by-line execution of a 10-line program with if-statements that uses the AP pseudocode for if-statements and the robot.

The purpose of the example is to show:

- Code executes one line at a time from top to bottom.
- Each if-statement condition is checked in the order of execution
- The conditions of later if-statements may be affected by what did, or didn't happen, in earlier lines of code.



## Teaching

You have a few options for how to review this worked example:

- You may want to review this worked example as an entire class, projecting it on the screen.
- Have students read, individually, or in pairs, then follow up with a partner, or review questions.
- You may want to print - though it is a long-ish document to print, especially since the "will it crash" handout is long as well.

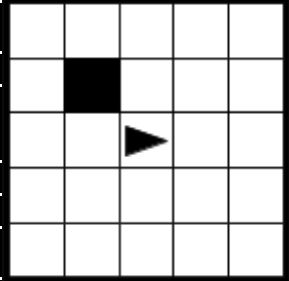
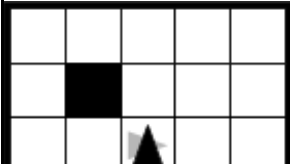
## Teaching Tip

Again, this page is largely meant as reference.

You don't need to belabor the points here, as students will get **much more practice** with the "Will it crash?" activity.

Remind students that this is here if they need to refer back to it when doing the next activity.

**Don't miss the problem** at the very end of the document. Students should try it and you can review as a class to verify that they understand.

Code	Robot Scenario	Commentary
<pre> 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} </pre>		<p><b>Before:</b> The starting scenario before any lines have been executed.</p> <p>Before reading the rest of the page, you might want to try to predict where the robot will end up.</p>
<pre> 1 ROTATE_LEFT () 2 IF (CAN_MOVE   (forward)) 3 { 4   MOVE_FORWARD   () 5 } </pre>		<p><b>Line 1 executes:</b> Robot turns 90 degrees to the left</p>

View the above as a separate document: **Worked Example - If-Statements and Robot**

Before clicking continue:



- Make sure you understand each step of the example
- Have tried out the last one on your own? and compared results with a classmate?

## "Will it crash?" Activity

### Distribute: Will it Crash? - Activity Guide

- Put students in partners
- Review the rules and do the first example together (if necessary)
- Partners should work together to trace and reason about the code.

### Compare results

- Put groups together to review the ending state and position of robots for each scenario.
- If there are disagreements about the end state, have pairs work it out and re-trace the examples.
- If there are common problems, save them, and review in the wrap-up

### Write code for the last problem and exchange

- Partners can work on writing the code together
- When done, have groups exchange code and trace the other team's work to verify correctness or reveal problems.

#### 💡 Teaching Tip

The last problem asks students to write some code by hand, and requires some time and thought. You might consider giving it for homework.

## Wrap Up (20 mins)

### What was trickiest?

If there were common problems that students had trouble with be sure to review those.

#### Prompt:

- Were you tripped up by any of the problems? Which ones? Why?
- What's the difference between a sequence or series of if statements versus an if-else statement?


#### Discussion

Points to raise during discussion:

- If-statments and conditional expressions are huge part of programming and we're going to spend some time digging in with them.
- There are two main issues to concern yourself with when it comes to if-statements and today we've looked a lot at one of them, namely, **program flow and order of execution**.
- For example, one very common misconception, or place where people get tripped up is, in the **difference between a sequence of if-statements, and using an if-else statement**.

## Algorithms and Creativity

### Code Studio levels

- Levels
-  4

### Student Instructions

[View on Code Studio](#)

# Algorithms - Solving Problems

## What is an algorithm?



## Definition

An algorithm is a precise sequence of instructions for a process that completes a task. Algorithms can be executed by a computer and are implemented using programming languages.



By asobuno (Own work) [CC BY-SA 3.0], via Wikimedia Commons



By Clem Rutter, Rochester, Kent. (self) [GFDL or CC BY 3.0], via Wikimedia Commons

## Teaching

### How to use this level

This level has a lot of text. Ways you might use it / incorporate it into your class:

- Assign as reading for students the day before
- Have students stop at this level during the normal progression and read as a group - discuss key points.
- Read and summarize for your students
- Make note of it as a reference for students that explains "algorithms"
- Use in conjunction with a preview of the AP Create Performance Task

# Automating Physical Tasks

## Physical Tasks in Daily Life

Daily life is filled with tasks. Most mornings, for example, you'll need to get dressed, pack your things, and then travel from one place to another. Your day at work or school will be filled with tasks to complete. Even keeping up with friends, relaxing, or going to bed includes some tasks if you look closely.

## Automating Tasks

We want to complete most tasks quickly, easily, and reliably. One important way we do this is by identifying step-by-step processes that we know work well. The steps to tie your shoes or the steps of a recipe are examples of processes we use to help us effectively take care of everyday tasks.

Processes to complete tasks are powerful because not only can humans use them, so can machines. Automation is the use of machines to complete some task with little to no human intervention, and from agriculture to manufacturing to transportation, it has transformed our society, economy, and daily lives.

## Automation Requires Algorithms

At the heart of automation is a well-defined step-by-step process that the machine is completing. A machine to weave



cloth, for example, is built to make stitches in a precise way in a precise number of rows using a precise number of threads. In other words, automating a task means first identifying the process or **algorithm** your machine will complete. Often a human could use that same algorithm to complete a task, but the machine will typically do so more quickly, easily, and reliably.

## Algorithms and Information Tasks

### Information Tasks and Tools

Many tasks don't require physical work, but they do require thinking. For example, you might need to keep track of money, remember birthdays, or schedule activities. At their core these problems have to do with how we organize and make sense of information. Tools like calendars, clocks, and financial records help us complete these information tasks.

### Automating Information Tasks

Just like physical tasks, many information tasks can be completed using algorithms. For example when you learn the steps to add or multiply two numbers, you're really just using an algorithm for addition or multiplication. The recognition that information tasks could be described algorithmically led to the desire to automate these tasks as well, and eventually, to the creation of the computer.

## Algorithms, Programming, and Computer Science

### The Everything Machine

Through history machines to automate information tasks usually did only one thing. A machine could track stars in the sky, or add numbers, but couldn't do both. By comparison, a single modern computer can add numbers, show video, communicate over the Internet, and play music. This is clearly a very different type of machine!

### Everything is Numbers

Many important ideas led to the design of the modern computer. First was the realization that most information can be represented as numbers. In fact, you learned in Units 1 and 2 that text, images, sound and many other pieces of information you can dream up can be represented in some way as binary numbers. This means information problems can be represented in a standardized way.

### Simple Commands

The next important realization is that information processes can be broken down into a common set of very simple commands. For example those steps might be adding or subtracting two numbers, moving information from one place to the next, or comparing two numbers. Even complex information processes like sorting a list of 1,000,000 names or determining if a picture has a cat in it can be represented on some level as a sequence of these simple commands.

### People Write Algorithms for Computers

Together these two ideas allow information tasks to be standardized to a degree that a single machine (a computer) could be designed to complete many of them. In order for this to work a computer is first designed to do this small set of low level commands. Next, and most importantly, the computer is designed to let a human being write out their own sequence of commands to control the machine to complete the task at hand. Said another way, a computer is a machine that's designed for a human to write algorithms for it to run!

## Algorithms and Creativity

### Sequence, Selection, Iteration

Any programming language only provides so many commands. Algorithms are created by combining these instructions in three ways. In fact, using these three you can describe ANY algorithm completed by a computer.



Those three ways are:

- **Sequence:** This is placing commands in an order. When you write a program that runs line by line you are defining the order in which a computer should run the fundamental commands that it understands.
- **Selection:** This is when a computer chooses to run one of two or more sections of code. When you use an if-statement you are making use of selection.
- **Iteration:** This is when a computer repeats a section of code. For example you can do this by using a loop.

### Algorithms, Programming, and Creativity

Even with the limited commands a computer understands and the limited ways you can combine them, there are actually many, conceivably infinite, ways to write a program to complete a task. Some may be more efficient or easier to understand than others, but there is typically no single "right" algorithm to complete a task. There also isn't an "algorithm for writing algorithms". You need to investigate and understand the problem you are trying to solve, and then get creative with how you'll combine the tools the programming language provides you. Computer science is a creative discipline because computers literally require human creativity to do anything at all!

## Algorithms, Unit 5, and the AP Exam

### Algorithms and AP Computer Science Principles

- (1) Algorithms is one of the seven big ideas of AP Computer Science Principles.
- (2) For the AP Create Performance Task you need to...

[identify] a code segment that contains an algorithm you developed...[and]...explain how the algorithm helps achieve the purpose of your program.

**Review:** Together read the summary of algorithms found in the last level of this lesson. Note this is a longer explanation of algorithms.

### **Remarks**

It's likely that solutions to the last problem varied considerably. Point this out as a positive.

- Programming is a creative activity.
- When you are planning a solution to the problem, you are thinking about **algorithms**

### **Prompt:**

- How many different coding solutions to the last problem were there?
- Why are different solutions possible?

### **Discussion**

Points to raise during discussion:

- There are multiple correct solutions
- This is because there are multiple ways to think about the problem
- There are also multiple algorithms for solving it
- Even if you used the same algorithm, the code might be different.
- All of this demonstrates that **programming is a creative activity**.

### **Remarks**

Next we'll look at writing if statements in JavaScript, and also dive into understanding conditional expressions a little more deeply



# Standards Alignment

## Computer Science Principles

- ▶ **4.1** - Algorithms are precise sequences of instructions for processes that can be executed by a computer and are implemented using programming languages.
- ▶ **5.2** - People write programs to execute algorithms.



This curriculum is available under a Creative Commons License (CC BY-NC-SA 4.0).

If you are interested in licensing Code.org materials for commercial purposes, **contact us**.



# Lesson 8: Boolean Expressions and "if" Statements

## Overview

In this lesson, students write `if` and `if-else` statements in JavaScript for the first time. The concepts of conditional execution should carry over from the previous lesson, leaving this lesson to get into the nitty gritty details of writing working code. Students will write code in a series of "toy" problems setup for them in App Lab that require students to do everything from debug common problems, write simple programs that output to the console, or implement the conditional logic into an existing app or game, like "Password Checker" or a simple Dice Game. The lesson ends with a problem requiring nested `if` statements to foreshadow the next lesson.

## Purpose

The main purpose here is **Practice, Practice, Practice**. The lesson asks students to write `if`-statements in a variety of contexts and across a variety of program types and problem solving scenarios.

## Agenda

### Getting Started

When vs. If

Optional: Flow Charts

### Activity

App Lab: Boolean expressions and `if`-statements

### Wrap-up

Compare and Contrast - easy/hard

Nested `if` statements

## Objectives

Students will be able to:

- Write and test conditional expressions using comparison operations
- Given an English description write code (`if` statements) to create desired program logic
- Use the comparison operators (`<`, `>`, `<=`, `>=`, `==`, `!=`) to implement decision logic in a program.
- When given starting code add `if`, `if-else`, or nested `if` statements to express desired program logic

## Preparation

### ☐ Forum

☐ Review student instructions in Code Studio (see below) along with teacher commentary.

☐ (Optional) A copy of (Optional) Flowcharts - Activity Guide

## Links

**Heads Up!** Please make a copy of any documents you plan to share with students.

### For the Students

- (Optional) Flowcharts - Activity Guide
- [Make a Copy](#)
- Unit 5 on Code Studio

## Vocabulary

- **Boolean** - A single value of either `TRUE` or `FALSE`
- **Boolean Expression** - in programming, an expression that evaluates to `True` or `False`.
- **Conditionals** - Statements that only run under certain conditions.
- **If-Statement** - The common programming



structure that implements "conditional statements".

- **Selection** - A generic term for a type of programming statement (usually an if-statement) that uses a Boolean condition to determine, or select, whether or not to run a certain block of statements.

## Introduced Code

- `if( ){ //code }`
- `if ( ){ // if code } else { // else code }`
- `__ == __`
- `__ != __`
- `__ > __`
- `__ >= __`
- `__ < __`
- `__ <= __`



# Teaching Guide

## Getting Started

### When vs. If

#### Remarks

In everyday conversation, it is common to interchange the words “when” and “if,” as in “If the user presses the button, execute this function.” The English language is tricky. **We often say “if” the button is clicked when really we mean “when” a button is clicked.** This can cause confusion because **“if” has a well-defined meaning in programming.**

**How are conditionals (if statements) different from events?**

Here is one way to think about it:

- Events are setup by a programmer, but triggered by the computer at any moment in time.
- If statements are a way a programmer can have her code make a decision during the normal flow of execution to do one thing or another.

As we have already seen in prior lessons, an **if statement** is evaluated when the code reaches a particular line and uses a true/false condition (like a comparison between values e.g., `score == 5`), to decide whether to execute a block of code.

#### Teaching Tip

These are nuanced distinctions and may not be immediately clear to every student. As students gain more experience with **if** statements, the difference between events and **if** statements will likely become more clear. For now, they should at the very least understand there is a difference between the two and begin to get in the habit of asking whether they want to run a block of code based on a user action, a condition, or some combination of the two.

#### Transition

- As we begin to write event-driven programs with if-statements we need to be clear about what we mean, or what we intend our programs to do.
- Sometimes when you say “if” you mean “when” and vice-versa. Just be on the lookout.

## Optional: Flow Charts

Some people find flow-charting a useful exercise for thinking about if-statements.

Here is an **optional activity: (Optional) Flowcharts - Activity Guide** you can do with your students to warm up on paper.

Alternatively, you might revisit this activity **after** students have had some experience writing if-statements to solidify their understanding.

## Activity

### App Lab: Boolean expressions and if-statements

#### Transition to Code Studio

Students will be introduced to conditionals by solving many different types of small puzzles and writing many small programs in different contexts with different kinds of output.

Read the student instructions and teacher commentary for more info.

#### Code Studio levels



## Introduction to Conditionals: Boolean Expressions

Student Overview

## Boolean Expressions and Comparison Operators

Student Overview

### Levels

4

(click tabs to see student view)

## Introduction to Conditionals: if Statements

Student Overview

## How If Statements Work pt 1

Student Overview

### Levels

7

8

9

(click tabs to see student view)

## Introduction to Conditionals: if-else Statements

Student Overview

## How If-Else Statements Work

Student Overview

### Levels

12

13

14

(click tabs to see student view)

## How Dropdown Menus Work

Student Overview

### Levels

16

17

18

(click tabs to see student view)

## Wrap-up

### Compare and Contrast - easy/hard

#### Reflection Prompt:

- "You've now had experience reasoning about if-statements on paper with the "Will it Crash?" activity, and now actually writing if-statements in working code. Compare and Contrast these experiences.

- For "Will it Crash" - what was easy? what was hard?
- For this lessson, writing if-statements - what was easy, what was hard?
- If there was one thing you wish you understood better at this point, what would it be?

#### Discussion

There are of course no right answers to these prompts. But it should be an opportunity for students give voice to new learnings or frustrations.

It's also an opportunity for you get insight about areas where your students are struggling, or things you might need to revisit.



## Nested if statements

### Prompt:

- "The last problem ("it's the weekend") was tricky. What made it hard? How did you end up solving it?"

Let students discuss for a moment and then bring to full class discussion. Points to raise:

- What made it hard was that you needed to check more than one condition at the same time. You needed to say "it's saturday OR sunday". That's more than one condition to check.
- So a solution (using only what we know so far) is **tonest** if-statements.
- **Nesting if statements** is one way to check more than one condition at a time.

### Transition

There are other ways to check more than one condition at a time that we will learn about in the next lesson.

## Standards Alignment

### CSTA K-12 Computer Science Standards (2011)

- ▶ **CL** - Collaboration
- ▶ **CPP** - Computing Practice & Programming
- ▶ **CT** - Computational Thinking

### Computer Science Principles

- ▶ **4.1** - Algorithms are precise sequences of instructions for processes that can be executed by a computer and are implemented using programming languages.
- ▶ **5.5** - Programming uses mathematical and logical concepts.



This curriculum is available under a Creative Commons License (CC BY-NC-SA 4.0).

If you are interested in licensing Code.org materials for commercial purposes, **contact us**.



# Lesson 9: "if-else-if" and Conditional Logic

## Overview

In this lesson, students will be introduced to the boolean (logic) operators NOT, AND, and OR as tools for creating compound boolean conditions in if statements. Students will learn how to more efficiently express complex logic using AND and OR, rather than deeply nested or chained conditionals. Students will work through a worksheet that covers the basics and a few problems with evaluating logical expressions, then write code in App Lab to practice using && and || in if statements. Finally, students will improve the Movie Bot so it can respond to multiple keywords and provide recommendations based on both the genre and rating provided by the user.

## Purpose

Similar to the previous lesson, the primary objective here is **practice, practice, practice!** We want students to get into the exercises and solve many different types of problems in many different types of contexts so they can pick up the patterns of writing, testing and debugging if-statements with more complex conditions.

This lesson introduces both the if-else-if construct and the Boolean operators AND, OR, and NOT. While it may appear that these operators extend the types of boolean conditions we can write, this is not actually the case. Nested and chained conditionals alone can be used to express any possible set of boolean conditions. The addition of these new boolean operators merely helps that expression be more succinct, clear, and elegant. But logic can get tricky, since often the way we say things in English is not the way we need to write them in code.

## Agenda

### Getting Started

Review nested and chained conditionals  
Compound Conditionals worksheet - page 1

### Activity

Transition to Code Studio practice using && and ||

### Wrap-up

Review what makes logic tricky  
Create PT Prep  
Preview "Building an App: Color Sleuth"

### Extended Learning

## Objectives

### Students will be able to:

- Write and test conditional expressions using Boolean operators AND (&&) OR (||) and NOT (!)
- Given an English description write compound conditional expressions to create desired program logic
- Use a "chain" of if-else-if statements to implement desired program logic
- When given starting code add if-else-if statements or compound boolean expression to express desired program logic

## Preparation

☐ Decide whether to use Compound Conditionals worksheet. (Best to use **after** students have learned about if-else-if and Boolean Operators AND, OR and NOT).

☐ **Note:** The first page of the worksheet should be distributed separately.

☐ Review code studio levels and associated teacher's notes.

## Links

**Heads Up!** Please make a copy of any documents you plan to share with students.

### For the Teacher

- **Worksheet KEY - Compound Conditionals**

### For the Students

- (Optional) Compound Conditionals - Worksheet [Make a Copy](#)
- Unit 5 on Code Studio

## Vocabulary



- **Boolean** - A single value of either TRUE or FALSE
- **Boolean Expression** - in programming, an expression that evaluates to True or False.
- **Conditionals** - Statements that only run under certain conditions.
- **If-Statement** - The common programming structure that implements "conditional statements".
- **Selection** - A generic term for a type of programming statement (usually an if-statement) that uses a Boolean condition to determine, or select, whether or not to run a certain block of statements.

## Introduced Code

- `if ( ) { // if code } else { // else code }`
- `__ && __`
- `__ || __`
- `!__`



# Teaching Guide

## Getting Started

### Review nested and chained conditionals

**Goal:** Review nested and chained conditionals and reveal their shortcomings when trying to express more complex logical statements.

### Compound Conditionals worksheet - page 1

💡 **Distribute:** (Just page 1 of) **(Optional) Compound Conditionals - Worksheet** and ask students to work together on the questions on the first sheet. Hold off on distributing the rest of the worksheet, since it shows an example solution to each of the problems from the first page.

**Discuss:** Have students share their answers with their neighbors and compare to see if they had the same solutions. Students can use the following questions to drive their conversations.

- Is my partner's solution correct?
- Is my partner's solution different from my own in any way?
- Are there ways we could improve our solutions?

You may wish to demonstrate possible solutions to each question, but they will also be found later on in that same worksheet.

**Transition:** Nested and chained conditionals are important tools when designing boolean conditions in our programs. In fact, every boolean condition can be expressed in some way using nesting and chained conditionals. That said, often when we write out these solutions they are long or force us to write redundant code. Today we are going to learn some new tools that won't let us write any new conditions, but WILL allow us to write many complex conditions much more clearly.

#### 💡 Teaching Tip

**Reviewing Concepts:** This warm-up activity is an excellent opportunity to review nested and chained conditionals. You may wish to briefly remind students what each of these is prior to the warm-up activity. Students should verify that one another's solutions are valid and make proper use of chained and nested conditionals.

**Pseudocode:** Students will be writing their solutions in pseudocode, which is a useful and important skill. Highlight that their syntax need not be perfect but that their pseudocode should be clear and reflect the actual programming structures they have seen.

## Activity

### Transition to Code Studio practice using && and ||

#### 🖥️ Transition to Code Studio:

Much like the previous lesson students will complete a series of short exercises to write code into "toy" programs to get practice using if-else-if constructs and the logical operators AND, OR, and NOT.

**NOTE:** If you want to break up the lesson into a few parts - the **(Optional) Compound Conditionals - Worksheet** contains many problems and activities that students can do on paper.

- Using it is optional, but you might use it to reinforce concepts (or even introduce them if you like).
- You don't have to use the whole thing. You may want to point students to individual pages for practice with certain things.
- You could use and re-visit it at several points during this lesson as gathering-point activities.

#### 🖥️ Code Studio levels



## Lesson Vocabulary & Resources

1

(click tabs to see student view)

## Introduction to Conditionals: if-else-if Statements

2

(click tabs to see student view)

## How "if-else-if" Works

3

4

5

6

7

(click tabs to see student view)

## Introduction to Conditionals: Compound Boolean Expressions

8

(click tabs to see student view)

## How the Boolean &&, || and ! Operators Work

9

10

11

(click tabs to see student view)

## How Compound Boolean Expressions Work

12

13

14

15

(click tabs to see student view)

## (new) AP Practice Response - Score the Response

16

(click tabs to see student view)

## Wrap-up

### Review what makes logic tricky

Prompt:

**"What's the trickiest logical statement you encountered in this lesson? What made it tricky?"**

- We often use "and" and "or" in English in imprecise ways, or at least in ways that could have multiple meanings. In programming logic, AND and OR have very precise meanings and they don't always map directly to English.

\*"True or False: the Boolean operators AND, OR and NOT, enable us to express boolean conditions that we couldn't before?"

- **False.** Anything that you can express with AND, OR and NOT, **can** be expressed with a chain or nesting of if-else statements.
- Certainly, it allows us to expression complex boolean conditions more succinctly, and makes our code MUCH easier to read. But in terms of program logic, we can do everything with just if-else statements\*

**An example using OR**

- In English, we sometimes use OR in the same way it's used in programming - to mean either or both. "Do you want cream or sugar in your coffee?" But we often use OR to mean exactly one thing or the other, not both. "Is the elevator going up or down?" The programming-logic answer to that question is: yes. Because it is the case that the elevator is either going up or it's going down.



- AND can get really tricky because in English we sometimes use the word “or” to convey a logical AND. For example: In English you might say: “If it’s not Saturday **or** Sunday, then it’s a weekday.” In programming you might express this as:

```
!(day=="Saturday" || day=="Sunday")
```

In other words: "It is not the case that the day is Saturday or Sunday"

But you might also express the same condition in code as:

```
(day != "Saturday" && day != "Sunday")
```

In other words: "It is the case that BOTH the day is not Saturday AND the day is also not Sunday."

### Logic can get tricky

Because logic can get convoluted and tricky, even professionals mess it up. However, as a programmer, you can take steps to make sure you’ve got it right by testing your code thoroughly to make sure you get expected results.

Because the boolean operators essentially take binary values (T/F) as input, you can easily figure out how many possible inputs there are for any complex boolean expression and test them all.

For example if you have a statement like:

```
if (expr1 && expr2 || expr3)
```

there are 3 expressions there, and each can be either **true** or **false**, so there are 8 possible ways to assign true or false to expr1 , expr2 and expr3 -- (TTT, TTF, TFT, TFF, FTT, FTF, FFT, FFF).

You can test all 8 to make sure you get the right outputs.

## Create PT Prep

If you have not already, complete and review the Create PT-style question that appears at the end of the lesson.

## Preview "Building an App: Color Sleuth"

You’ve learned to write conditional statements and boolean expressions in a variety of ways on small programs so far. In the **next lesson**, you’ll have an opportunity to build an entire app from the ground up that will require to you write if statements and come up with your own conditional expressions.

## Extended Learning

**Connection to logic gates in hardware** These AND, OR, and NOT logic operators can be very useful in directing the flow of your programs. They also represent a fundamental part of your computer’s hardware. Your processor uses logic gates such as these to do computations and direct the flow of information. Remember, inside your computer, you have electricity flowing. “true” is indicated by a high voltage and “false” is indicated by a low voltage.

- AND gate: Two wires are attached to one side of an AND gate, and one is attached to the other. If both input wires have a high voltage, the AND gate will give a high voltage to the output wire.
- OR gate: Two wires are attached to one side of an OR gate, and one is attached to the other. If either input wire has a high voltage, the OR gate will give a high voltage to the output wire.
- NOT gate: One wire is attached to one side of a NOT gate, and one is attached to the other. If the input wire has a high voltage, the output wire will have a low voltage and vice versa.

### Collaborative programming

- Form teams of three students.
- Instruct student one to write a description of a real-life situation that requires multiple conditions.



- When finished, the first student passes the description to the second student, who is tasked with drawing the flowchart or pseudocode for the scenario.
- The paper with the description and flowchart or pseudocode is then passed to a third student, who writes code for the event. They may rely upon imaginary functions if necessary (e.g., **is\_raining()**)

## Standards Alignment

### Computer Science Principles

- ▶ **4.1** - Algorithms are precise sequences of instructions for processes that can be executed by a computer and are implemented using programming languages.
- ▶ **5.1** - Programs can be developed for creative expression, to satisfy personal curiosity, to create new knowledge, or to solve problems (to help people, organizations, or society).
- ▶ **5.5** - Programming uses mathematical and logical concepts.



This curriculum is available under a Creative Commons License (CC BY-NC-SA 4.0).

If you are interested in licensing Code.org materials for commercial purposes, **contact us**.



# Lesson 10: Building an App: Color Sleuth

Programming | Conditionals | App Lab

## Overview

This lesson attempts to walk students through the iterative development process of building an app (basically) from scratch that involves the use of `if` statements. Following an imaginary conversation between two characters - Alexis and Michael - students follow the problem solving and program design decisions they make for each step of constructing the app. Along the way they decide when and how to break things down into functions, and of course discuss the logic necessary to make a simple game.

The last step - writing code that executes an end-of-game condition - students must do on their own. How they decide to use `if` statements to end the game will require some creativity. The suggested condition - first to score 10 points - is subtly tricky and can be written many different ways.

At the conclusion of the lesson there are three practice Create PT-style questions as well as resources explaining the connection between this lesson and the actual Create PT. Depending on how you use these materials they can easily add an additional day to this lesson.

## Purpose

The purpose here is for students to see how "experts" would approach writing an app from scratch when all you have to start out with is a sketch on paper of some idea. Research has shown that what novices often need is an expert walk-through to explain the rationale behind certain decisions and to see the kinds of problems they anticipate and solve. There are a few key things that happen in this lesson that we hope students see and take to heart:

- There is no one "correct" way to approach writing a program
- You don't write programs "in order" from top to bottom - you write in pieces and organize the code into sections and functions.
- Start with a small problem to solve - solve it and move to the next one
- Use a buddy or "thought partner" to talk things through
- Sketch out pseudocode on paper to get your thoughts straight
- If you get stuck, there is always something small you can do to make progress

After this lesson students are prepared to complete the AP Create PT. If you have more time in your year you may continue through

## Objectives

**Students will be able to:**

- Write code to implement solutions to problems from pseudocode or description
- Follow the iterative development process of a collaboratively created program
- Develop and write code for conditional expressions to incorporate into an existing program
- Write a large program from scratch when given directions for each step

## Links

**Heads Up!** Please make a copy of any documents you plan to share with students.

**For the Students**

- **Color Sleuth and the AP Create PT** - AP Explanation [Make a Copy](#)
- **Color Sleuth** - Rubric [Make a Copy](#)
- **Unit 5 on Code Studio**

## Vocabulary

- **Boolean Expression** - in programming, an expression that evaluates to True or False.
- **Conditionals** - Statements that only run under certain conditions.
- **If-Statement** - The common programming structure that implements "conditional statements".
- **Selection** - A generic term for a type of programming statement (usually an if-statement) that uses a Boolean condition to determine, or select, whether or not to run a certain block of statements.

## Introduced Code

- `setProperty(id, property, value)`



Unit 5 Chapter 2 before beginning the task. When you decide to begin the task use the materials in the AP Create PT Prep unit to further prepare students. For more details refer to pages 32 and 33 of the **CSP Curriculum Guide - Teal Book**.

## Agenda

### Getting Started

What are you worried about? Where to Start?

### Activity

Transition to Code Studio

### Wrap Up (15-50 mins)

Gallery Walk (Optional)

Review the Epilogue

Connections to the AP Create Performance Task

Review if statements for assessment

- `rgb(r, g, b, a)`



# Teaching Guide

## Getting Started

### Remarks

In this lesson you're going to build an app from scratch, but you'll have guidance and help. The app you'll make will require you to apply **all** of the skills you've learned so far:

- Adding event handlers
- Using variables
- Breaking problems down into functions
- Using if statements

In the lesson, you'll follow the path of two imaginary students - Alexis and Michael - as they walk through each of the problems that have to get solved along the way. They'll do the thinking and problem solving, you'll write the code.

This project will mirror many of the elements of the Create PT which we'll look into more deeply at the end of the project. At the end of this project you should know all the programming skills you'll need to complete that task.

Here is what you're going to make...

## What are you worried about? Where to Start?

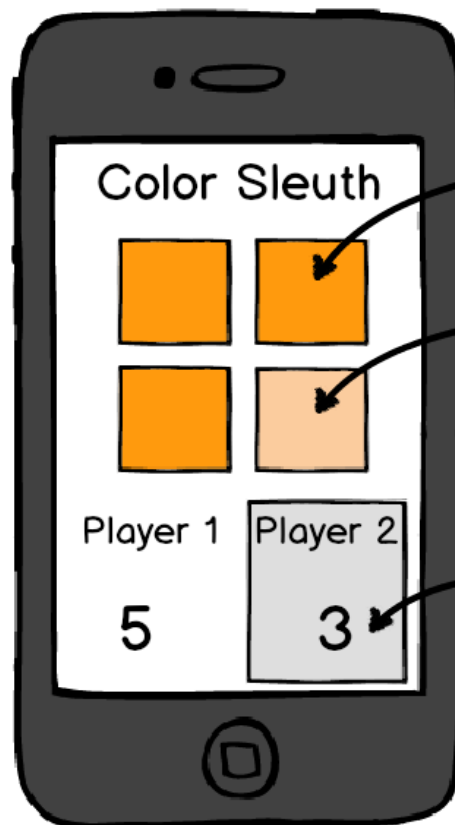
Show this image of the sketch mock-up of the app.

- Students can also find it in code studio in the lesson introduction.

Each player takes a turn trying to click on the square that's slightly different in color.

### Open questions:

1. Scoring?  
+1 if you get it right?  
-1 if wrong?
2. How does it end?  
First to 10?  
First to get 3 wrong?  
What else?



Generate random color and apply to all squares.

1 square is randomly chosen to be a **slightly** different color.

IDEA: use buttons and setProperty function

Do something to indicate whose turn it is.  
IDEA: show/hide a colored box behind the text

With this image on display...

### **Prompt:**

"Looking at just this sketch of the app with its few notes about what it's supposed to do...What are you thinking?...What are you worried about?...Are there elements of this that you're not sure you know how to program?"



With an elbow partner write down two things:

1. Make a quick list of **things you're not sure you know how to program yet, or things you're worried about.**
2. What would be the first thing you'd try to get working? (Assume that all of the design layout is done -- what's the first code you'd write to get things started?)

Let students discuss with their partners for a few minutes.

Open discussion to the group: What are you worried about? Where would you start?

Possibilities for "Worried about:"

- Making random colors with code
- Choosing a random button to make a different color
- Knowing whether the user clicked the "right" button or not
- Switching player turns
- Keeping track of whose turn it is - and the score
- How does the game end?

Possibilities for "Where to start:"

- NOTE: almost anything is possible, encourage students to start with things they know rather than things they're worried about, like:
- Adding event handlers for the button clicks
- Adding a variable (or two) to keep score
- Can they use what they know about `randomNumber` to generate these random colors? -- worth looking into.

#### Discussion

This is an informal way to do the first part of a KWL (Know, **Want-to-know**, Learned) Chart. You can do a formal one for this lesson if you think it would be helpful.

The point of the conversation here is really just to **activate students' thinking** about what they do and don't know.

For things they don't know or they're worried about you can **assure them that techniques for doing those things are revealed in the lesson**

For where they should start **encourage students to start with what they know, rather than what they don't**. A good way to make progress on an app is start by adding things they know first since that will help get them started.

## Activity

### Transition to Code Studio

Students can work individually **or with a partner**

- As usual, we recommend that students at least have a coding buddy - someone they can work with and ask questions of as they work through the exercises.
- It's also reasonable to have students **pair-program** during this lesson, switching off writing the code at each level.

### Code Studio levels

#### Unit 5 Lesson 10 Introduction

[Teacher Overview](#)[Student Overview](#)



## Teaching This Lesson

This lesson looks like it has a lot of reading. The reading should go fairly fast since it's written like a script of two people talking.

You can **think about your pause points** in this lesson, where you might gather everyone to check for understanding, or at least do a check-in with various groups.

- There are a few unavoidably tricky steps in the latter half of the lesson - steps that require students to modify or add to the code in more than one place in order to see the next thing. It's worth verifying that students have successfully completed these before moving on.

You **might also consider reading out loud** some parts of the script - particularly some the trickier decision-making points - to make sure students understand what decision was made and why.

- It's also important to acknowledge along the way what's recapped in the Epilogue - that what we're walking students through here is NOT the **one true correct** way to make this app. Each decision, while well-reasoned, is relatively arbitrary.
- The goal of the script is to provide a model for collaboration, thinking about problems in small pieces, and how a process of iterative development can lead to a robust project.

### Color Sleuth - Planning the App

Student Overview

### How Set Property Works

Student Overview

### Using setProperty

4

5

(click tabs to see student view)

### How to pick a random button

Student Overview

### How to Pick a Random Button

7

(click tabs to see student view)

### How to make a random color

Student Overview

### How to Make a Random Color

9

(click tabs to see student view)

### Functions in Your Color Sleuth App

10

11

(click tabs to see student view)

### U5 color sleuth check correct

Student Overview

### Activating Buttons

13

14

15

(click tabs to see student view)



## Color Sleuth - How to switch player turns

[Student Overview](#)

### How to Switch Player Turns

[17](#)[18](#)[\(click tabs to see student view\)](#)

## Color Sleuth - Keeping score

[Student Overview](#)

### Updating the Score

[20](#)[21](#)[\(click tabs to see student view\)](#)

## Color Sleuth - How does it end?

[Student Overview](#)

## Project: Finish Color Sleuth

[Teacher Overview](#)[Student Overview](#)

This is probably going to be the hardest level because there is the least amount of guidance.

[View on Code Studio](#)

Also, the end game condition can be written **anumber of different ways**. Even for a particular condition like "first to 10" there are several different ways you could go about it. This is intentional in order to spur actual problem solving, and encourage students to talk to each other about possible solutions.

Encourage students to: **Study the pseudocode diagram from the previous page - it has the outline of what you need to add to the code** Don't forget to add a little bit and test with console.log statements \* Ask a friend for help.

Below are 3 possible ways a checkGameOver function could be written - each using a different technique of if-statements that we learned about.

```
// Nested if-statements
function checkGameOver(){
  if(p1Score >= 10 || p2Score >= 10){    // if anyone is over 10 points the game is over
    setScreen("gameOver_screen");
    if(p1Score > p2Score){                // figure out who won and show the label
      showElement("player1Wins_label");
    } else {
      showElement("player2Wins_label");
    }
  }
}

// If-else-if with compound booleans
function checkGameOver(){
  if(p1Score >= 10 && p1Score > p2Score){ // if player 1 is over 10 points and winning
    setScreen("gameOver_screen");       // game over
    showElement("player1Wins_label");
  }
  else if(p2Score >= 10 && p2Score > p1Score){ //otherwise if player 2 is over 10 points and winning
    setScreen("gameOver_screen");       // game over
    showElement("player2Wins_label");
  }
}

// sequential if statements
function checkGameOver(){
  var winnerId = "player1Wins_label"; //make a variable of the label id for whoever is winning right now
  if(p2Score > p1Score){
    winnerId = "player2Wins_label";
  }
}
```



```

if(p1Score >= 10 || p2Score >= 10){ // if either is over 10 points
  setScreen("gameOver_screen"); // then game is over, show the label of winner
  showElement(winnerId);
}
}

```

It's worth pointing out that the solutions give above aren't **quite** fair because it advantages player 1 in the case where the game is neck-and-neck: if player 1 and 2 are tied with 9 points apiece, and player 1 gets to 10 first, the code above will declare her the winner, and player 2 wouldn't have a chance to even the game up.

So, while this would be unexpected from students on a first pass -- you could add another condition to all of these to make sure that it's player 2's turn before declaring anyone the winner. There are a number of ways to implement this condition as well. We'll show one here:

```

function checkGameOver(){
  if(currentPlayer == 2 && (p1Score >= 10 || p2Score >= 10)){
    setScreen("gameOver_screen");
    if(p1Score > p2Score){
      showElement("player1Wins_label");
    } else {
      showElement("player2Wins_label");
    }
  }
}
}

```

## Color Sleuth - Epilogue

[Teacher Overview](#)
[Student Overview](#)

Use this epilogue as the foundation for wrap up. Either have students read it, review it out loud. [View on Code Studio](#)

## (new) AP Practice Responses - Algorithms and Abstraction

 25

 26

 27

(click tabs to see student view)

## Wrap Up (15-50 mins)

### Gallery Walk (Optional)

- In theory students made some of their own personal modifications - you can do a gallery walk to share
- You can also gallery walk to look at and appreciate code for the end-of-game conditions.

### Review the Epilogue

- The last level in code studio contains some text about the app development walk through. Students don't **have** to read it, but you should review its key points about:
- how and why writing code is a creative process
- There is no one correct way to do things
- The realistic parts of Alexis and Michael's conversation



## Connections to the AP Create Performance Task

**Distribute: Color Sleuth and the AP Create PT - AP Explanation** (students can find a link on Level 1 of this lesson)



- This document explains how the elements of this project, and the process, map to each requirement for the AP Create PT.
- Point out to students that **if** they had gone through a similar process as Alexis and Michael that this project basically **meets the requirements of the AP Create Performance Task**. The trick would be provide written responses that properly highlight everything.
- You could optionally have students practice writing responses to the real AP Create Task writing prompts using this document as a guide.

#### Teaching Tip

**Ready for the Create PT:** After completing this lesson students will have the minimum skills they need to complete the AP Create PT. Time permitting you should continue through Unit 5 Chapter 2 and get as far as you can before starting the Create PT - the more programming students have under their belts the better.

When you do opt to begin it, students should use the materials in the **AP Create PT Unit** to prepare. You can find it as a unit in the dropdown menu to assign to your section in the teacher dashboard.

#### Other Create PT Options and Resources:

- There are three questions (bubbles) at the end of this lesson in style of the Create PT. They use the color sleuth project as an example as though it were submitted for the Create PT. Two questions address algorithms and one is a refresher on abstraction.
- Optionally pull out the **AP Digital Portfolio Student Guide - College Board Handout** so that you can have it on hand to review the different components of the Create PT.
- Optionally head over to the **AP Create Task Prep Unit** if you'd like to go more in depth with Create Task review at this point.

## Review if statements for assessment

The stage that follows this one is an assessment the covers if-statements. You can review the kinds of "toy" problems that appear in the AP Assessment, many of which are similar to the kinds of problems students did in the exercises in lessons prior to this one.

That includes the unplugged "will it crash?" exercises. You might look at those again for review.

## Standards Alignment

### Computer Science Principles

- ▶ **1.1** - Creative development can be an essential process for creating computational artifacts.
- ▶ **1.2** - Computing enables people to use creative development processes to create computational artifacts for creative expression or to solve a problem.
- ▶ **3.1** - People use computer programs to process information to gain insight and knowledge.
- ▶ **4.1** - Algorithms are precise sequences of instructions for processes that can be executed by a computer and are implemented using programming languages.
- ▶ **5.1** - Programs can be developed for creative expression, to satisfy personal curiosity, to create new knowledge, or to solve problems (to help people, organizations, or society).
- ▶ **5.5** - Programming uses mathematical and logical concepts.



This curriculum is available under a Creative Commons License (CC BY-NC-SA 4.0).

If you are interested in licensing Code.org materials for commercial purposes, **contact us**.



# Lesson 11: While Loops

## Overview

This lesson demonstrates how a slight manipulation of a conditional statement can allow for the creation of a new and powerful tool in constructing programs, a **while** loop. Students are introduced to a **while** loop by analyzing the flow chart of a conditional statement in which the "true" branch leads back to the original condition. Students design their own flowcharts to represent a real-world situation that could be represented as a **while** loop, and they learn how to recognize common looping structures, most notably infinite loops. Students then move to App Lab, creating a **while** loop that runs exactly some predetermined number of times. While learning about creating **while** loops, students will be introduced to many of the common mistakes early programmers make with **while** loops and will be asked to debug small programs. They finally progress to putting if statements inside a while loop to count the number of times an event occurs **while** repeating the same action. This activity will recall the need for counter variables and foreshadows their further use in the following lesson.

## Purpose

**while** loops are the most primitive type of loop. The for loop, which students used in a very basic form during turtle programming, is just a more specific case of a **while** loop. **while** loops repeat a set of steps until a certain condition is met. Thus, like conditional statements, **while** loops use boolean expressions to determine if they will run and how many times. One of the biggest problems a programmer can run into with a **while** loop is to create an infinite loop. There are a couple different defensive programming strategies introduced in this lesson to help prevent infinite loops.

## Agenda

### Getting Started

Following a looping flowchart

### Activity

(Optional) Flowcharts with while Loops  
App Lab: while loops

### Wrap-up

while loops exit ticket

## Objectives

### Students will be able to:

- Explain that a while loop continues to run while a boolean condition remains true.
- Translate a real-life activity with repeated components into a form that could be represented by a while loop.
- Analyze a while loop to determine if the initial condition will be met, how many times the loop will run, and if the loop will ever terminate.
- Write programs that use while loops in a variety of contexts.

## Preparation

☐ Decide whether to use the flow charts activity or not

☐ Print an (Optional) Flowcharts with While Loops - Activity Guide for each student.

☐ Review Levels in Code Studio

## Links

**Heads Up!** Please make a copy of any documents you plan to share with students.

### For the Teacher

- **Activity Guide KEY - Flowcharts with While Loops** - Answer Key

### For the Students

- (Optional) Flowcharts with While Loops - Activity Guide [Make a Copy](#)
- Unit 5 on Code Studio

## Vocabulary

- **Iterate** - To repeat in order to achieve, or get closer to, a desired goal.
- **while loop** - a programming construct used to repeat a set of commands (loop) as long as (while) a boolean condition is true.



## Introduced Code

- `while( ){ // code }`



# Teaching Guide

## Getting Started

### Following a looping flowchart

#### Display:

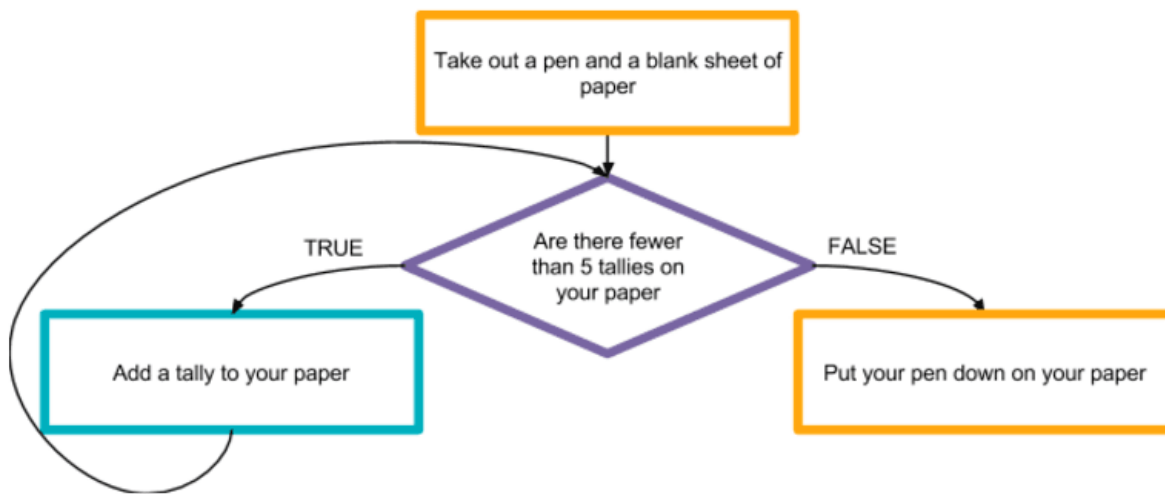
Either by writing it on the board or displaying it on a projector, display the following flowchart. Feel free to substitute your own example.

#### Prompt:

Ask students to follow the "instructions" in the diagram and then wait quietly once they are done.

#### Goal

Introduce the structure of a **while** loop by demonstrating how a conditional statement that "loops" back on itself can be used to repeatedly execute a block of commands.



#### Share:

Have students share their results with one another. They should each have a sheet of paper that contains 5 tally marks. Once they've shared their responses, ask them to discuss the following prompts in small groups:

- How is the flowchart we just saw similar to a normal conditional (or **if** statement)? How is it different? How do these differences change the results of the flowchart?

#### Discuss:

Once students have had an opportunity to share with their groups, open the discussion to the class. The most significant points to draw out are:

- This conditional statement loops back on itself. As a result, the conditional might be evaluated multiple times.
- As a result, the same command is running multiple times.

#### **Transitional Remarks**

The structure we just explored is called a **"while loop."** When you look at the flowchart, you can see that we "loop through" a command as long as, or "while," a condition is true. **while** loops are a way we can easily represent a process that includes many repeated steps.

Once you develop an eye for them, you'll start to notice **while** loops all around you. Many activities that require repeated action are done "while" a condition is true.

## Activity



## (Optional) Flowcharts with while Loops

### Remarks:

- This is an **optional** unplugged activity. You may skip to app lab if you don't think it would be useful to you or your students.
- It may also be something to come back to **after** writing code to reinforce the concepts.

### Distribute: (Optional) Flowcharts with While Loops - Activity Guide to each student.

- Students will be reminded of the components of a flowchart and shown a couple of examples of real-life **while** loops.
- After determining how many times these loops will run, they will develop a real-life **while** loop of their own.

### Share:

Once students have created their own real-life **while** loop, they should exchange with a partner. They should be looking for:

- Whether the **while** loop is properly structured
- What the **while** loop accomplishes
- How many times the **while** loop runs (It might not be possible to know exactly.)

### Discuss:

Discuss the results of this exchange as a class, keeping the primary focus on whether everyone is properly structuring their loops. These early activities are primarily designed to get students familiar with the structure of a **while** loop.

Common misconceptions include:

- Writing the condition on which the **while** loop should stop rather than continue.
- Not including steps in the **while** loop that will make the condition false at some point (i.e., creating an infinite loop)

The final two **while** loops on the worksheet are written with code and ask students to write what output that program would generate. They will likely need to keep track of the values generated by the program on their paper, in addition to the output. Encourage them to do so. The primary ideas foreshadowed here are:

- **Counters:** Variables that count how many times a **while** loop has run, also called **iterators**, can be used to control the number of times a **while** loop runs.

- **Infinite Loops:** The final example in this activity guide produces an "infinite loop." This means that the computer (or person implementing the algorithm) will cycle through a set of commands

forever, because the **while** loop condition is always true. Infinite loops are almost always undesirable, but they can be deceptively easy to create by mistake. Use the last portion of this activity to call out the fact that this is an infinite loop, that it is a very real possibility to make one in a program, and that they will have to be careful when making **while** loops since, as this example shows, it is fairly easy to create an infinite loop which prevents the rest of your program from running.


#### 💡 Teaching Tip

There is no need to get into the details of counters or infinite loops here. Students will see them throughout the levels, but a quick comment pointing out these two ideas may help make connections between this activity and what students will see in their programs.

## App Lab: while loops

Now that students have some background with the structure of **while** loops, they will move to Code Studio where they will program with them.

### Code Studio levels

Unit 5 Lesson 11 Introduction 

Teacher Overview

Student Overview



[View on Code Studio to access answer key\(s\)](#)

[View on Code Studio](#)

## Levels

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

(click tabs to see student view)

## Wrap-up

### while loops exit ticket

#### Discussion

Students can summarize their understanding of **while** loops. Students will have more opportunities to use these skills tomorrow, so this is primarily an opportunity to make sure students have an accurate understanding of what a **while** loop is and how it works.

#### Thinking Prompt:

- "In your own words, describe how a **while** loop works. Explain two things to pay attention to when creating **while** loops. In your response, justify why the name "while loop" accurately describes the behavior of this new programming construct."

#### Discuss:

Students may discuss this question in small groups before sharing as a class. Students' answers may vary but will likely include:

- It repeats a set of commands.
- It continues to run "while" a boolean expression is true.
- It is called a loop because it "loops through" a set of commands.
- They may mention that visually it looks like a loop when shown in a flowchart.
- Things to pay attention to when programming **while** loops:
  - Something inside the **while** loop has to update the variable in the condition so the **while** loop will stop
  - **while** loops may never run if the condition begins as false
  - **while** loops can become infinite if the condition never becomes false
  - Off-by-one errors are common with **while** loops

## Standards Alignment

#### Computer Science Principles

- ▶ **3.1** - People use computer programs to process information to gain insight and knowledge.
- ▶ **4.1** - Algorithms are precise sequences of instructions for processes that can be executed by a computer and are implemented using programming languages.
- ▶ **5.2** - People write programs to execute algorithms.
- ▶ **5.4** - Programs are developed, maintained, and used by people for different purposes.
- ▶ **5.5** - Programming uses mathematical and logical concepts.



This curriculum is available under a Creative Commons License (CC BY-NC-SA 4.0).

If you are interested in licensing Code.org materials for commercial purposes, **contact us**.



# Lesson 12: Loops and Simulations

## Overview

In this lesson, students gain more practice using **while** loops as they develop a simulation that repeatedly flips coins until certain conditions are met. The lesson begins with an unplugged activity in which students flip a coin until they get 5 heads in total, and then again until they get 3 heads in a row. They will then compete to predict the highest outcome in the class for each statistic. This activity motivates the programming component of the lesson in which students develop a program that allows them to simulate this experiment for higher numbers of heads and longer streaks.

## Purpose

The ability to model and simulate real-world phenomena on a computer has changed countless fields. Researchers use simulations to predict the weather, the stock market, or the next viral outbreak. Scientists from all disciplines increasingly rely on computer simulation, rather than real-life experiments, to rapidly test their hypotheses in simulated environments. Programmers might simulate users moving across their sites to ensure they can handle spikes in traffic, and of course videogame and virtual reality technology is built around the ability to simulate some aspects of the real world. The speed and scale at which simulations allow ideas to be tested and refined has had far-reaching impact, and it will only continue to grow in importance as computing power and computational models improve.

## Agenda

### Getting Started

#### Coin Flipping Experiment

### Activity

#### App Lab: Loops and Simulations

### Wrap-up

#### Reflection

### Extended Learning

## Objectives

### Students will be able to:

- Use a while loop in a program to repeatedly call a block of code.
- Use variables, iteration, and conditional logic within a loop to record the results of a repeated process.
- Identify instances where a simulation might be useful to learn more about real-world phenomena.
- Develop a simulation of a simple real-world phenomenon.

## Preparation

☐ Print a **Worksheet - Flipping Coins** for each student.

☐ A coin for every student or pair of students.

## Links

**Heads Up!** Please make a copy of any documents you plan to share with students.

### For the Students

- **Worksheet - Flipping Coins**

[Make a Copy](#)

- **Unit 5 on Code Studio**

## Vocabulary

- **Models and Simulations** - a program which replicates or mimics key features of a real world event in order to investigate its behavior without the cost, time, or danger of running an experiment in real life.



# Teaching Guide

## Getting Started

### Coin Flipping Experiment

#### Instructions:

Flip a coin until you get 5 total heads. Then again until you get 3 heads in a row. Record your results and predict the highest result in the class.

**Distribute: Worksheet - Flipping Coins** and give each student / pair of students a coin.

#### Prompt:

- **"We're going to run two simple experiments. Use your worksheets to keep track of your results (by writing "H" or "T" for each flip) but keep them a secret for now."**
- **Experiment 1:** Groups will flip their coins as many times as it takes in order to get **5 heads total**
- **Experiment 2:** Groups will flip their coins as many times as it takes to get **3 heads in a row**

#### Prompt:

- **"Let's have a little competition. You should have recorded your results for your two experiments. Based on your experiment, predict, among every group in the class the most and fewest flips needed to complete each of the experiments."**

#### Compare Results:

Collect worksheets (or just have students share answers) to determine how accurate guesses were. Perhaps offer small prizes for the group whose guesses were most accurate.

#### **Transitional Remarks**

That was pretty interesting with only 5 total heads or a streak of 3 heads. If we want to run this experiments for higher numbers of heads or longer streaks of heads however, we'll quickly find that it's tedious to do once, let alone many times. Luckily we know now that we can use loops to repeatedly perform commands, so we're going to **simulate** these larger experiments instead.

#### **Goal**

Run a simple experiment by hand that it would be unmanageable to run on a larger scale, thus motivating the need to simulate it on a computer.

## Activity

### App Lab: Loops and Simulations

- Develop a simulation in App Lab that allows you conduct the experiments from the warm-up with many more coins.
- Repeat the warm-up activity using the simulation and update hypotheses as a result.

#### **Code Studio levels**

Unit 5 Lesson 12 Introduction 

Teacher Overview

Student Overview



The series of problems and tasks in this lesson progressively build up a series of experiments that simulate coin flipping.

[View on Code Studio](#)

We ask students to **make a hypothesis**, then experiment with code, revise the hypothesis and so on, around some question. Below are some guidelines about what students should find.

**Insights:** The following insights should arise from this experiment:

- **Total Heads:** When trying to flip 5 heads, it is quite possible that it will only take 5 flips, but it may also easily take 15 or 20. Relative to the number of heads you are looking for, this is a massive range! When you are trying to flip 10,000 heads the likely range is typically between 19,000 and 21,000, and typically much closer. As you are looking for more flips, the relative width of the likely window shrinks.
- **Longest Streak:** When trying to find a streak of 3 heads it will typically take between 3 (it's always possible!) and 20 flips, though of course it may take longer. Even slightly longer streaks of heads, however, will rapidly increase the average time it takes to find that streak. Looking for a streak of 12 heads might occasionally happen in fewer than 100 flips, but it can also easily take tens of thousands. As the length of the streak increases, the number of flips it takes to find that streak grows rapidly.

### How Much Math is Necessary?

This lesson might seem to naturally lend itself to a more detailed discussion of the mathematical properties of random experiments. Flipping coins is, after all, perhaps the most classic example of a random experiment.

The goal of this lesson **is not** for students to walk away knowing the precise mathematical relationship between the number of coins flipped and the number of heads observed or the longest streak of heads. Instead they are supposed to **appreciate that questions that might be impossible or hard to address by hand are possible to examine by using computer simulation**. Just as developing a mathematical model is one way to address a problem, so too is developing a simulation.

**There's no need to dive deep into the mathematics** this lesson touches on, but students should be able to describe the patterns they observed while running their simulations, and use those observations to justify new hypotheses.

## Make a Hypothesis

[Teacher Overview](#)

[Student Overview](#)

[View on Code Studio](#)

### Pause Point - Make a Hypothesis

- Use this level to describe the fact that students will be making a simulation of flipping coins.
- Ask them to make a prediction for the outcomes of the experiments they will run.

Make sure students make this prediction before moving on.

- **Students should record their predictions in the space provided on their worksheets.**

## Levels

[3](#)

[4](#)

[5](#)

[6](#)

(click tabs to see student view)

## Update your Hypothesis - Part 1

[Student Overview](#)

## Levels

[8](#)

[9](#)

[10](#)

(click tabs to see student view)



## Wrap-up

### Reflection

#### Prompt

- **"Update your hypothesis based on the results of your simulation and predict the outcomes of an even larger experiment using the new knowledge you have gained."**
  - On the second part of their worksheets, students are asked to extend their hypotheses to try to predict how long it might take to flip 10,000,000 heads or find a streak of 20 heads. Even for a computer, these could take a great deal of time to run, but luckily students should have developed intuitions about these problems based on their earlier simulations.
  - Their actual predictions are less important than whether they demonstrate having reflected on the outcomes of the earlier simulation.

#### Discuss:

- Once students have finished writing their predictions for these problems, they should present their predictions and their reasons for making them with their classmates.
- Discuss whether and how the results of their earlier simulation impacted their new hypotheses.

#### Conclusion

Not all problems are as easy to simulate as a coin flip of course, and we've even seen how some problems we can simulate still take a very long time to run.

Simulations are an increasingly important tool for a variety of disciplines. Weather and traffic predictions are based on computer models that simulate weather patterns or people moving through a city. Scientific research, whether in physics, chemistry, or biology, increasingly uses simulations to develop new hypotheses and test ideas before spending the time and money to run a live experiment.

Before you use most of your favorite websites and apps, they will be tested by simulating high levels of traffic moving across the server. Simulations take advantage of computers' amazing speed and ability to run repeated tasks, as we've seen through our exploration of the while loop, in order to help us learn more about the world around us.

As computers get ever faster and models improve, we are able to answer old questions more quickly and start asking new ones.

## Extended Learning

- Extend this activity to new statistics, rolling dice, etc. Use these both as opportunities to practice programming and to develop the habit of using simulations to refine hypotheses.
  - How long does it take for the number of heads and tails flipped to be equal?
  - Longest streak where each roll is greater than or equal to the last
  - Longest streak made of only five (any five) of the six faces on the die (i.e., not equal to the other)

## Standards Alignment



- ▶ **2.3** - Models and simulations use abstraction to generate new understanding and knowledge.
- ▶ **4.1** - Algorithms are precise sequences of instructions for processes that can be executed by a computer and are implemented using programming languages.
- ▶ **5.1** - Programs can be developed for creative expression, to satisfy personal curiosity, to create new knowledge, or to solve problems (to help people, organizations, or society).



This curriculum is available under a  
Creative Commons License (CC BY-NC-SA 4.0).

If you are interested in licensing Code.org materials for commercial purposes, **contact us**.



# Lesson 13: Introduction to Arrays

## Overview

This lesson introduces arrays as a means of storing lists of information within a program. The class begins by highlighting the difficulties that arise when trying to store lists of information in a variable. Students then watch a short video introducing arrays and a subset of the operations that can be performed with them. Students will work in Code Studio for the remainder of the class as they practice using arrays in their programs. At the conclusion of the sequence, students build a simple app which can be used to store and cycle through a list of their favorite things. In the next lesson, students will continue working with a version of this app that can display images and not just text strings.

## Purpose

Some sort of list data structure is a component of almost all programming languages. A list allows large amounts of information to be easily referenced and passed around a program, and the use of a numeric index allows individual items in a list to be accessed. Historically a list would have literally been a single contiguous chunk of memory and the index or address was used to know how far into that chunk a relevant piece of information was stored. In many modern languages, however, it is more likely that the items in an array are stored at many locations on your computer's hard drive, and the index is only useful to help the programmer identify different components. In this way, a JavaScript array is actually another example of abstraction. We know that it is holding a list of related information, but we don't need to think about the actual implementation details.

## Agenda

### Getting Started

**Prompt:** What makes lists useful in everyday life?

**Transition:** Variables are a bad way to store lists.

### Activity

**App Lab:** Introduction to Arrays

### Wrap-up

**Reflection:** When to use a variable and when to use an array

### Extended Learning

## Objectives

### Students will be able to:

- Identify an array as a data structure used to store lists of information in programs.
- Create arrays and access information stored within them using an index.
- Manipulate an array using the append, insert, and remove operations.
- Account for the fact that JavaScript arrays are zero-indexed when using them in a program.

## Links

**Heads Up!** Please make a copy of any documents you plan to share with students.

### For the Students

- [Unit 5 on Code Studio](#)

## Vocabulary

- **Array** - A data structure in JavaScript used to represent a list.
- **List** - A generic term for a programming data structure that holds multiple items.

## Introduced Code

- `list.length`
- `insertItem(list, index, item)`
- `var list = ["a", "b", "d"];`
- `var x = [1, 2, 3, 4];`
- `appendItem(list, item)`
- `removeItem(list, index)`



# Teaching Guide

## Getting Started

### Prompt: What makes lists useful in everyday life?

#### Thinking Prompt:

- "Today we're going to start looking at how we can use lists in programs, but before we dive into that, let's think about why we would want to in the first place. What are the benefits of creating lists? Why is it helpful to keep information in lists?"

#### Goal

Demonstrate that lists are a desirable feature of a programming language and that using variables to store lists is cumbersome or impossible. Motivate the need for arrays.

#### Discuss:

Students may discuss in small groups or you can just ask for ideas from the class. Potential answers include:

- Lists help us organize information.
- Lists help us collect all the relevant information in one place.
- Lists show that a lot of ideas are related.
- Lists help us order or prioritize ideas.
- Lists help us think about the big picture.

### Transition: Variables are a bad way to store lists.

#### Transitional Remarks

There are a lot of benefits to keeping lists of information in real life. Since we use programming to solve a lot of similar problems, we would like to keep lists of information in our programs, too.

Right now, the only way we know how to store information in our programs is with a variable, but each variable can only store a single piece of information.

Today we'll be learning about a new programming construct that will allow us to hold as many pieces of information as we want within a single list.

## Activity

### App Lab: Introduction to Arrays

This set of levels looks long, but all the problems are relatively short and small.

You might consider watching the first video as a whole class before diving in.

#### Code Studio levels

##### Unit 5 Lesson 13 Introduction

[Student Overview](#)

##### Introduction to Lists - Part 1

[Student Overview](#)

#### Levels

[3](#)[4](#)

(click tabs to see student view)



## Introduction to Lists - Part 2

[Student Overview](#)

### Levels

[6](#)[7](#)*(click tabs to see student view)*

## Introduction to Lists - Part 3

[Student Overview](#)

### Levels

[9](#)[10](#)[11](#)[12](#)[13](#)[14](#)[15](#)*(click tabs to see student view)*

## Introduction to Lists - Part 4

[Student Overview](#)

### Levels

[17](#)[18](#)[19](#)[20](#)[21](#)[22](#)[23](#)[24](#)[25](#)[26](#)[27](#)[28](#)[29](#)[30](#)[31](#)*(click tabs to see student view)*

## Wrap-up

### Reflection: When to use a variable and when to use an array

**Goal:** Students now know how to store information in both variables and arrays. Help students synthesize their new knowledge by trying to develop a rule for when to use a variable vs. an array. This is also just a useful way to assess students' understanding of arrays at the conclusion of the lesson.

#### Thinking Prompt (Also in Code Studio)

1. Your app needs to store the following information. Decide whether you would use an **array** or a **variable** to store it?
  1. All the messages a user has sent
  2. The highest score a user has ever reached on the app
  3. A username and password to unlock the app
2. In general, when do you think you should store information in an array, and when should you use a variable?

**Discuss:** You can use these questions as an exit ticket, but it is probably even more useful to discuss as a class. Use this discussion to possibly identify and address any misconceptions about what an array is and how it stores information. Here are some key points to pull out:

- Variables store single pieces of information, while arrays store many.
- An array can grow in size to accommodate more information.
- Arrays are slightly more complex to use than variables. If you are only going to be storing a small and fixed amount of information, it is probably appropriate to use multiple variables.

#### Conclusion:

#### Remarks

We are going to keep exploring arrays in the coming lessons. Keep your eye out for some of the distinctions we just discussed, and keep thinking about how you might want to use arrays in applications of your own.

## Extended Learning



One of the last levels in App Lab challenges students to keep developing the app. If you wish, you may also have students submit these projects.

## Standards Alignment

### Computer Science Principles

- ▶ **1.1** - Creative development can be an essential process for creating computational artifacts.
- ▶ **5.1** - Programs can be developed for creative expression, to satisfy personal curiosity, to create new knowledge, or to solve problems (to help people, organizations, or society).
- ▶ **5.3** - Programming is facilitated by appropriate abstractions.
- ▶ **5.5** - Programming uses mathematical and logical concepts.



This curriculum is available under a  
Creative Commons License (CC BY-NC-SA 4.0).

If you are interested in licensing Code.org materials for commercial purposes, **contact us**.



# Lesson 14: Building an App: Image Scroller

## Overview

Students will extend the **My Favorite Things** app they built in the previous lesson so that it now manages and displays a collection of images and responds to key events. Students are introduced to the practice of refactoring code in order to keep programs consistent and remove redundancies when adding new functionality. As part of learning to use key events, students are shown that event handlers pass a parameter which contains additional information about the event. This lesson also serves as further practice at using arrays in programs.

## Purpose

Most applications you use are not based on static pieces of code. Instead the software will be continuously updated both to correct errors and introduce new pieces of functionality. If later improvements are anticipated it is generally possible to develop programs in a way that easily incorporates new functionality. At other times it is necessary to make larger changes to the way a program operates in order to incorporate new features while maintaining existing functionality. Refactoring code in this way can be a tedious and challenging endeavor, but it helps ensure that the final product is consistent and easy to maintain. If software is not kept in a logical, consistent, and succinct form, then it will only get harder to keep introducing new features, increasing the likelihood of errors.

## Agenda

### Getting Started

#### Refactoring and re-writing code

### Activity

#### App Lab: Building an App - Image Scroller

### Wrap-up

#### Reflection: When to refactor

## Objectives

### Students will be able to:

- Use an array to maintain a collection of data in a program.
- Create apps that allow user interaction through key events.
- Refactor code in order to appropriately incorporate new functionality while maintaining readability and consistency.

## Vocabulary

- **Key Event** - in JavaScript an event triggered by pressing or releasing a key on the keyboard. For example: "keyup" and "keydown" are event types you can specify. Use event.key - from the "event" parameter of the onEvent callback function - to figure out which key was pressed.

## Introduced Code

- `onEvent(id, type, function(event)){ ... }`
- `setImageUrl(id, url);`
- `playSound(url)`



# Teaching Guide

## Getting Started

### Refactoring and re-writing code

Thinking Prompt:

- "When we want to add new functionality to our programs, we'll of course have to write new code. Sometimes, when we add new code to an existing program, we'll also have to make changes to the original components of our program. Why might this be the case?"

 Goal

Students should reflect on why adding new functionality to their programs might mean they need to make changes to the old code they wrote as well.

Discuss:

Students may discuss in small groups or you can just ask for ideas from the class. Potential answers include

- The old code and the new code contradict one another.
- The old code and the new code may have redundant components.
- Incorporating the new code may help us find better ways to write the old code.

#### *Transitional Remarks*

Writing software is an iterative process. We continuously update and improve our ideas as we learn new techniques, debug our software, or identify new features we'd like to add to our code. While our code will constantly be changing, we'd like it to remain organized, consistent, and readable. As a result, when we add new code to our programs, we may sometimes need to change the way we originally approached a problem.

Today we're going to be further extending our **My Favorite Things** app, and seeing how this process plays out in practice.

## Activity

### App Lab: Building an App - Image Scroller

#### Code Studio levels

Unit 5 Lesson 14 Introduction 

Teacher Overview

Student Overview



## Notes about this lesson

There are two major things happening in this lesson

1. Using the `event` parameter from `onEvent` to determine which key was pressed.
2. Modifying the "My Favority Things" Apps made in the last lesson

### Helping Students with Incomplete "My Favorite Things" Apps :

It is quite possible that some of your students will not have succeeded in creating the text version of the **My Favorite Things** app. Since this lesson asks students to extend that project, it might be a challenge for those students to participate. Some strategies are below.

- Let students continue working on the text version of their app. While this lesson introduces new event types, it falls in a sequence focused on arrays. Students can still participate in discussions about refactoring code and will be exposed to the new key events.
- Provide students the **exemplar version** of the text-based **My Favorite Things** app. They can Remix the project and work on it in Free Play mode (i.e., outside of the curriculum). This way, they have a clean starting point and focus on the content of the lesson. If they wish, students can copy the code into Code Studio and will only need to create the UI elements themselves.
  - **My Favorite Things Exemplar**

### Levels

2

3

4

5

6

7

8

9

(click tabs to see student view)

## Project - Final Image Scroller

### Student Overview

### Levels

11

(click tabs to see student view)

## Wrap-up

### Reflection: When to refactor

#### Thinking Prompt:

- "In today's activity, we needed to make some changes to our programs in order to incorporate new functionality. Sometimes this meant we needed to make changes to our old code as well."

- Why might you want to change or refactor old code?
- Is it necessarily a bad thing to refactor code?
- What steps can we take to avoid refactoring code too frequently?

#### Discuss:

You can use these questions as an exit ticket, but it is probably even more useful to discuss as a class. Use this discussion to identify and address misconceptions about what refactoring code is and why we would want to do it. Here are some key points to pull out:

- Refactoring is the process of changing the way we wrote old code in order to keep programs consistent and

#### Goal

Reflect on the process of making improvements to existing code, how to do it well, and how to avoid having to do it too frequently.



readable while incorporating new functionality.

- It is possible that refactoring code will not change the user's experience but will make the program easier to read and maintain.
- Refactoring is a useful process, but it can be time consuming and challenging. We'd ideally not refactor code very often but it is sometimes necessary.
- Good planning and design can help avoid refactoring. Good use of functions and an organized program means that at the very least we limit areas that need to be changed.

## Standards Alignment

### Computer Science Principles

- ▶ **5.1** - Programs can be developed for creative expression, to satisfy personal curiosity, to create new knowledge, or to solve problems (to help people, organizations, or society).
- ▶ **5.2** - People write programs to execute algorithms.
- ▶ **5.3** - Programming is facilitated by appropriate abstractions.
- ▶ **5.4** - Programs are developed, maintained, and used by people for different purposes.
- ▶ **5.5** - Programming uses mathematical and logical concepts.



This curriculum is available under a  
Creative Commons License (CC BY-NC-SA 4.0).

If you are interested in licensing Code.org materials for commercial purposes, **contact us**.



# Lesson 15: Processing Arrays

Unplugged | App Lab

## Overview

This lesson will probably take two days to complete. It introduces students to algorithms that process lists of data. The students will do two unplugged activities related to algorithms and program some of them themselves in App Lab. The **for** loop is re-introduced to implement these algorithms because it's straightforward to use to process all the elements of a list. The lesson begins with an unplugged activity in which students write an algorithm to find the minimum value in a hand of cards. Students then move to Code Studio to write programs that use loops and arrays. Students are shown how to use a **for** loop to visit every element in an array. Students use this pattern to process an array in increasingly complex ways. At the end of the progression, students will write functions which process arrays to find or alter information, including finding the minimum value - a problem they worked on in the unplugged activity. Finally, an unplugged activity has students reason about linear vs. binary search and attempt to write pseudocode for a binary search.

## Purpose

There are many situations where we want to repeat a section of code a predetermined number of times. Although this can be done with a **while** loop by maintaining a variable to keep track of how many times the loop has executed, there are many small pieces to keep track of. The **for** loop consolidates all of those pieces - counter variable, incrementing, and boolean condition - onto one line. One of the most common uses of **for** loops in programming is to process arrays. **for** loops allow programmers to easily step through all the elements in an array. This basic pattern is at the core of many algorithms used to process a list of items. Whether you are looking to find a name in a list, find the closest store to your current location, or compute the total money in your account based on past transactions, a loop will probably be used at some point to access all the elements in a list.

## Agenda

**Getting Started (15 minutes)**

**Recall: Minimum Card Algorithm**

**Teacher Reference: Min Card Sample Algorithms**

**Activity (30 minutes)**

**App Lab: Processing Arrays**

**Activity 2**

## Objectives

**Students will be able to:**

- Use a **for** loop in a program to implement an algorithm that processes all elements of an array.
- Write code that implements a linear search on an unsorted array of numbers.
- Write code to find the minimum value in an unsorted list of numbers.
- Explain how binary search is more efficient than linear search but can only be used on sorted lists.

## Preparation

☐ Print **Activity Guide - Minimum Card Algorithm - Activity Guide**

☐ Print **Activity Guide - Card Search Algorithm - Activity Guide** for each student.

☐ Playing cards (or pieces of paper with numbers on one side).

## Links

**Heads Up!** Please make a copy of any documents you plan to share with students.

**For the Students**

- **Activity Guide - Minimum Card Algorithm** - Activity Guide

**Make a Copy** ▾

- **Activity Guide - Card Search Algorithm** - Activity Guide

**Make a Copy** ▾

- **Unit 5 on Code Studio**

## Vocabulary

- **for loop** - A typical looping construct designed to make it easy to repeat a section of code using a counter variable. The **for** loop combines the creation of a



### **Unplugged Activity: Card Search Algorithm**

#### **Wrap-up (10 minutes)**

**Reflection: Processing sorted arrays and binary search**

variable, a boolean looping condition, and an update to the variable in one statement.

## **Introduced Code**

- `for(var i=0; i<4; i++){ //code }`
- `function myFunction(n){ //code }`



# Teaching Guide

## Getting Started (15 minutes)

### Recall: Minimum Card Algorithm

Introduce students to thinking about processing lists of information by recalling the **FindMin** problem they wrote an algorithm for in **Unit 3 Lesson 2**

In particular, introduce the common pattern of using a loop to visit every element in a list, rather than the jump command.

#### Opening:

#### **Remarks**

Remember in a lesson a while back when we wrote algorithms for playing cards using the "Human Machine Language"?

Notice how a row of cards is kind of like a list.

Today we're going to begin to write code to process lists of data. Processing large lists of data is one of the most powerful things computer programs can do. Many of the most important algorithms in computer science have their roots in processing lists of data.

So as a warm-up today, let's think back to algorithms that process lists with a short activity.

#### **Teaching Tip**

Students might want help with language as they write out their algorithms. In particular, they might recognize that trying to clearly articulate which hands to use to pick up which cards is challenging. It's good if they recognize this. Here are some suggestions you can make:

- You may refer to the "first" and "last" cards in the row as part of your instructions.
- You may also give an instruction to move a hand some number of cards (or positions) to the left or right.
- You can give an instruction to put a card down on the table in one of the open positions, or put it back where it was originally picked up from.

Here are two examples of algorithms students might write. These are not the "correct answers" per se - there are many ways students might go about it - but they should give you the gist of what you might be looking for.

#### **Distribute: Activity Guide - Minimum Card Algorithm - Activity Guide**

#### **Setup:**

- Put students in pairs or small groups.
- Students should:
  - Read the instructions (or you might want to read out loud as a class).
  - Write their algorithm out on paper and test it out with each other (or possibly other groups).
- Test it out with other groups, or demonstrate one.

### Teacher Reference: Min Card Sample Algorithms

For your reference here are some plain English Algorithms

#### **SAMPLE ALGORITHM 1** (using a numbered list of instructions):

1. Put your left hand on the first card in the row and your right hand on the card next to it.
2. Pick up the card your left hand is on.
3. Pick up the card your right hand is on.
4. IF the card in your right hand is less than the card in your left hand,
5.     THEN swap the cards (so the smaller one is in your left hand).
6. Put the card in your right hand back down on the table.
7. IF there is another card in the row to the right of your right hand,
8.     THEN move your right hand one position to the right, and go back and repeat step 3 (with your right hand now on a new card).
9. OTHERWISE: say "I found it!" and hold the card in your left hand up in the air.

Since we have learned about loops in the course, your students might write pseudocode with a loop construct in it.

#### **SAMPLE ALGORITHM 2** (using a loop construct):



Pick up the first card in your left hand.  
FOR EACH card IN the row of cards (ALTERNATIVE: WHILE there are more cards in the row)  
    Pick up the next card with your right hand.  
    IF the card in your right hand is less than the card in your left hand,  
        THEN swap the cards (so the smaller one is in your left hand).  
    Put the (larger) card in your right hand back down on the table.  
(after the loop) Say "I found it!" and hold the card in your left hand up in the air.

### **Transitional Remarks**

The same kind of thinking that went into designing this algorithms can be applied to making working code as well.  
Don't confuse thinking about the algorithm with actually writing code.  
Today you'll get some practice writing code with loops and if-statements to process a list - skills that will help you write your own algorithms for lists.

## Activity (30 minutes)

### App Lab: Processing Arrays

#### **Code Studio levels**

##### Unit 5 Lesson 15 Introduction

Student Overview

##### Processing Lists with Loops

Student Overview

#### Levels

3

4

5

6

7

8

9

10

11

12

13

14

15

(click tabs to see student view)



## Activity 2

### Unplugged Activity: Card Search Algorithm

In the lesson students programmed linear search (scan all the values in the list from beginning to end until you find what you're looking for). "Binary search" uses a different algorithm, that is faster, but requires that the list be in sorted order ahead of time - linear search will work for any list. Demonstrate why this algorithm can only be performed on sorted arrays and justify the fact that it is faster.

#### **Distribute: Activity Guide - Card Search Algorithm - Activity Guide**

**Note:** The wrap-up for this whole lesson focuses primarily on the outcomes from the Card Search activity.

#### Teaching Tip

In terms of pacing, the unplugged **Activity Guide - Card Search Algorithm - Activity Guide** could be done on a second day of class. It's also likely that students will not finish all of the Code Studio levels for the lesson by the end of the first day, but you do not have to wait for every student to complete every level to run it. It's reasonable to be done at any point after students have gotten halfway through the Code Studio levels (basically, after they've written code for linear search).

You might elect to use the next activity as a getting started activity on day 2 if most students are halfway through.

## Wrap-up (10 minutes)



# Reflection: Processing sorted arrays and binary search

## Remarks

When you talk about how “long” or how much “time” an algorithm takes to run, time is usually a measure of the number of operations a computer needs to perform to complete the task. You can measure the amount of time it takes to run an algorithm on a clock, but it’s often not a useful measure, because the speed of the computer hardware obscures whether the algorithm is good or not.

There are several essential knowledge statements from the framework that directly tie to information about algorithms, efficiency and linear vs. binary search, and which we’ll use in the wrap-up.

## Goal

The only algorithms the CSP framework mentioned by name are “linear search” and “binary search.” Students should be able to reason about an algorithm’s “efficiency.” Students should understand the connection (and differences) between designing an algorithm and actually writing (implementing) the algorithm in code.

## Activity:

Give each pair of students one of the 5 statements (D,E,F,G,H) listed below, which are taken directly from the CSP Framework under “4.2.4 Evaluate algorithms analytically and empirically for efficiency, correctness, and clarity. [P4]”

- Ask the pair to come up with a brief (60 second) explanation of that statement and relate it to something they experienced as part of this lesson.
- Give students 3 minutes to think and discuss.
- Do a whip-around, or put pairs together, or group by statement, and
  - Have each pair read the statement out loud.
  - Given their explanation of what it means.

**4.2.4D Different correct algorithms for the same problem can have different efficiencies. Both linear search and binary search solve the same problem, but they have different efficiencies.**

**4.2.4E Sometimes more efficient algorithms are more complex. Binary search is more efficient than linear search, but even though it might be easy to understand at a high level, it is much more challenging to write code for.**

**4.2.4F Finding an efficient algorithm for a problem can help solve larger instances of the problem. The algorithms we wrote work for any size input.**

**4.2.4G Efficiency includes both execution time and memory usage. Execution “time” here means number of operations that need to be performed in the worst case.**

**4.2.4H Linear search can be used when searching for an item in any list; binary search can be used only when the list is sorted. Emphasis should be placed on the fact that binary search only works when the list is sorted. It’s a fact often forgotten.**

## Standards Alignment

### Computer Science Principles

- ▶ **1.2** - Computing enables people to use creative development processes to create computational artifacts for creative expression or to solve a problem.
- ▶ **4.1** - Algorithms are precise sequences of instructions for processes that can be executed by a computer and are implemented using programming languages.
- ▶ **4.2** - Algorithms can solve many but not all computational problems.
- ▶ **5.1** - Programs can be developed for creative expression, to satisfy personal curiosity, to create new knowledge, or to solve problems (to help people, organizations, or society).
- ▶ **5.3** - Programming is facilitated by appropriate abstractions.
- ▶ **5.5** - Programming uses mathematical and logical concepts.





This curriculum is available under a  
Creative Commons License (CC BY-NC-SA 4.0).

If you are interested in licensing Code.org materials for commercial purposes, **contact us**.



# Lesson 16: Functions with Return Values

## Overview

In this lesson students are introduced to the **return** command and learn to write their own functions that return values. Students first complete a simple unplugged activity based on the game **Go Fish** to introduce the concept of a return value. They will then complete a short sequence of exercises in Code Studio, which introduces preferred patterns for writing functions that return values. At the end of the sequence, students write and use functions that return values in a simple turtle driver app.

## Purpose

The ability to return values is closely tied to the concept of scope. All variables declared within a function are in local scope and so will be removed once the end of the function is reached. As a result any useful information generated during that function will be lost. One solution to this problem is storing the value in a global variable, but this is generally considered bad programming practice. Global variables can be accessed by many functions and so reasoning about their logic requires considering the logic of all of those functions. Return values are a way to move information out of the local scope of a function without using a global variable. As a result a function call can be treated as if it were the type of data that a function returns, and it is up to the programmer to determine if or how it will be used.

## Agenda

### Getting Started (20 minutes)

#### Unplugged Activity: Return Values with Go Fish

### Activity (25 minutes)

#### App Lab: Functions with Return Values

### Wrap-up (5 minutes)

#### Exit ticket: Function with Returns vs. Functions without Returns

## Objectives

### Students will be able to:

- Use the return command to design functions.
- Identify instances when a function with a return value can be used to contain frequently used computations within a program.
- Design functions that return values to perform frequently needed computations within a program.

## Links

**Heads Up!** Please make a copy of any documents you plan to share with students.

### For the Students

- **Activity Guide - Return Values with Go Fish** - Activity Guide [Make a Copy](#)
- **Unit 5 on Code Studio**

## Vocabulary

- **Return Value** - A value sent back by a function to the place in the code where the function was called from - typically asking for value (e.g. `getText(id)`) or the result of a calculation or computation of some kind. Most programming languages have many built-in functions that return values, but you can also write your own.

## Introduced Code

- `return`



# Teaching Guide

## Getting Started (20 minutes)

### Unplugged Activity: Return Values with Go Fish

Opening:

#### **Remarks**

Today we are going to look at how to write our own functions with return values. We are going to explore this idea by playing the classic card game Go Fish.

#### **Distribute: Activity Guide - Return Values with Go Fish - Activity Guide**

- Break students into groups of 4 with a set of cards.
- Give each student a copy of the worksheet.
- Each group should play a couple rounds of Go Fish with their team.
  - They do not need to finish the game to get the point here.
- Students should complete the worksheet together as a group.

#### **Share:**

Have students share their algorithms for the Responder.

- The main goal here is for students to talk about the parameters for the function, the algorithm used in the function and, most important, the information that needs to be returned at the end of the function.

#### **Discussion Prompt:**

- **"Why do we need to return some information from the Responder to the Asker?"**

The main thing to draw out:

- Once the asker has gained the information, he uses it to continue computing information.
- The asker can not easily gain the information without the help of the responder, as he doesn't have access to the cards.

#### **Transitional Remarks**

As we saw playing Go Fish, we often need to ask for information and receive an answer to be able to make decisions. We have seen a few different functions that do something like this, such as **randomNumber**, **getText**, and **includes**. Up until now, though, we have never been able to create our own functions that return information. Today we are going to learn how to write functions with return values.

#### **Goal**

Introduce the idea of a function with a return value as a process in which a question is asked, something computes an answer and gives the answer back to the asking location. We are looking to draw out the need for some way to share the information between the two parts of the program.

## Activity (25 minutes)

### App Lab: Functions with Return Values





#### Teaching Tip

Students' algorithms will vary. An example of what they might create is: `function responder(desiredCard)`

- `var gaveCard = false`
- for each card in responder's hand
  - if card is equal to desiredCard
    - `gaveCard = true`
    - `RETURN card`
- if gaveCard is false
  - `RETURN "Go Fish"`

Although technically this first example is not how you would write this with code, as the return statement would cause you to leave the function, it gets across the main understanding we are working towards. Therefore, it is a completely correct way for students to be thinking at this point. You should not feel the need to correct this understanding before students work on the levels, but if you are wondering about more correct versions of this algorithm, check out below. A more correct version of this algorithm would be: `function responder(desiredCard)`

- `var cardsToGive = []`
- for each card in responder's hand
  - if card is equal to desiredCard
    - add card to cardsToGive list
- if cardsToGive length is 0
  - `RETURN "Go Fish"`
- else
  - `RETURN cardsToGive`

The above algorithm is good, as it will return at the end of the computation and therefore could be something you could use to translate into code. However, it is usually best practice to always return the same type of information. "Go Fish" is a string, whereas cardsToGive is an array.

Even better would be:

`function responder(desiredCard)`

- `var cardsToGive = []`
- for each card in responder's hand
  - if card is equal to desiredCard
    - add card to cardsToGive list
- `RETURN cardsToGive`

## Code Studio levels

### Unit 5 Lesson 16 Introduction

[Teacher Overview](#)[Student Overview](#)[View on Code Studio to access answer key\(s\)](#)[View on Code Studio](#)



# Using Output from Functions: The return Command

## Student Overview

### Levels

[3](#)[4](#)[5](#)[6](#)[7](#)[8](#)[\(click tabs to see student view\)](#)

## Wrap-up (5 minutes)

### Exit ticket: Function with Returns vs. Functions without Returns

#### Remarks

Return values are a useful way to move useful information generated inside of a function to the rest of your program. There is another way we can do this. If we write our function so that it stores its output within a global variable, then that information will be accessible when the function terminates, since the global variable will stay in scope.

#### Goal

Highlight the benefits of using a function that returns a value, in particular the fact that it makes it much easier to reason about a program.

#### Prompt:

- **"Why would we prefer to write a function that returns a value to using the strategy shown above? How might return values make our function more generally useful? How might they make our code easier to reason about?"**

#### Discuss:

Ask students to share their responses to this question. They may need to be reminded about the concept of variable scope. The primary points to pull out are:

- A function that saves output in a global variable must specifically reference that variable by name. Since that name is "hard-coded," your function can only ever save information in that variable. If we wish for our functions to be generally useful, we should be able to decide how to use the output a function generates.
- A global variable is accessible by all functions. As a result, it can be difficult to determine every location in your program that modifies this variable. Reasoning about how its value will change over the course of the program is much harder, as is debugging unexpected behavior. Using a return value limits the effects of a function to the local variables of the function and the place where the function was called.

## Standards Alignment

### Computer Science Principles

- ▶ **1.1** - Creative development can be an essential process for creating computational artifacts.
- ▶ **1.2** - Computing enables people to use creative development processes to create computational artifacts for creative expression or to solve a problem.
- ▶ **2.2** - Multiple levels of abstraction are used to write programs or create other computational artifacts
- ▶ **4.1** - Algorithms are precise sequences of instructions for processes that can be executed by a computer and are implemented using programming languages.
- ▶ **5.1** - Programs can be developed for creative expression, to satisfy personal curiosity, to create new knowledge, or to solve problems (to help people, organizations, or society).
- ▶ **5.3** - Programming is facilitated by appropriate abstractions.
- ▶ **5.5** - Programming uses mathematical and logical concepts.





This curriculum is available under a  
Creative Commons License (CC BY-NC-SA 4.0).

If you are interested in licensing Code.org materials for commercial purposes, **contact us**.



# Lesson 17: Building an App: Canvas Painter

## Overview

Students continue to practice working with arrays and are introduced to a new user interface element, the canvas. The canvas includes commands for drawing simple geometric shapes (circles, rectangles, lines) and also triggers mouse and key events like any other user interface element. Over the course of the lesson, students combine these features to make an app that allows a user to draw an image while recording every dot drawn on the canvas in an array. By processing this array in different ways, the app will allow students to redraw their image in different styles, like random, spray paint, and sketching. Along the way, students use their knowledge of functions with return values to make code which is easy to manage and reuse.

## Purpose

The study of computing is in many ways the study of information and the automation of processes to transmit, transform, and learn from that information. The combination of list data structures, like the JavaScript array, and loops allows for large amounts of information to be generated, maintained, and then transformed in useful and interesting ways. The patterns used to perform these processes are frequently quite similar, even across disciplines. By recognizing when and how to use arrays and loops to store and process information, a programmer can quickly solve problems and create things at a scale unimaginable without the power of computing.

## Agenda

### Getting Started

#### Introduction to Activity

### Activity

#### App Lab: Building an App - Canvas Painter

### Wrap-up

**Share any additional features added to your app.  
Brainstorm other effects that could be created.**

## Objectives

### Students will be able to:

- Programmatically control the canvas element in response to user interactions.
- Maintain a dynamically generated array through the running of a program in order to record and reuse user input.
- Use nested loops within a program to repeat a command on the same array index multiple times.
- Perform variable arithmetic within an array index to access items in an array by their relative position.

## Links

**Heads Up!** Please make a copy of any documents you plan to share with students.

### For the Students

- [Unit 5 on Code Studio](#)

## Vocabulary

- **Canvas** - a user interface element to use in HTML/JavaScript which acts as a digital canvas, allowing the programmatic drawing and manipulation of pixels, basic shapes, figures and images.
- **Key Event** - in JavaScript an event triggered by pressing or releasing a key on the keyboard. For example: "keyup" and "keydown" are event types you can specify. Use event.key - from the "event" parameter of the onEvent callback function - to figure out which key was pressed.

## Introduced Code

- setActiveCanvas
- line
- circle



- `setStrokeColor`
- `setFillColor`
- `clearCanvas`



# Teaching Guide

## Getting Started

### Introduction to Activity

#### Remarks

Today we are going to be building a drawing app in App Lab. Along the way, we'll be introduced to a couple new ideas and concepts, but for the most part, we will be combining old skills. At this point, you all know most of the core concepts of programming, and so as we move forward, we'll spend more time thinking about interesting ways to combine them. With that in mind, let's get into Code Studio and start building our next app!

#### Goal

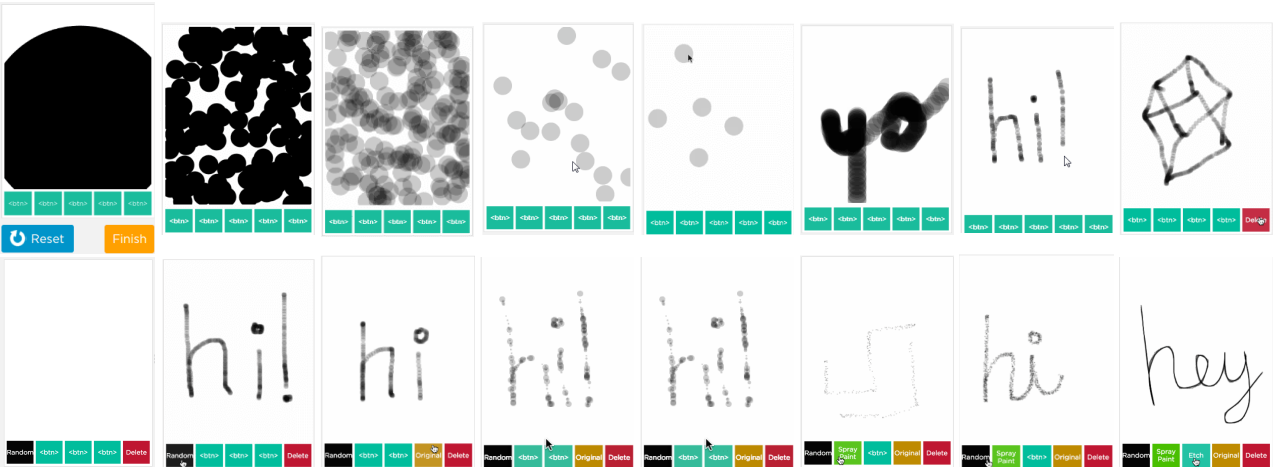
This lesson requires a fair amount of programming and combines most of the programming constructs students have learned up to this point. Briefly let students know the aim of the lesson and then move on to programming the app.

## Activity

### App Lab: Building an App - Canvas Painter

The images below show the progression of what students create throughout the lesson. It's worth just glancing over them to get a sense of what you'll be seeing.

The final product is an app that lets you draw something by dragging the mouse around the screen. Afterward, you can apply different effects to the drawing. The code saves all of the mouse coordinates in an array. The effects are created by looping over all of the coordinates to effectively re-draw the image.



### Code Studio levels

Unit 5 Lesson 17 Introduction

Student Overview

Levels

02

03

04

05

06

07

08

09

10

11

12

13

14

15

16

17

18

19

(click tabs to see student view)

CSPU5\_U3 - Canvas - freePlay

Student Overview



# Wrap-up

## Share any additional features added to your app.

**Goal:** If students have had time to brainstorm or create additional features in their drawing apps, give them an opportunity to share. Brainstorm other ways that this stored data could be processed, and the types of effects that could be produced as a result. Some students may wish to extend this project on the Practice Create Performance Task they will complete in the next lesson.

## Brainstorm other effects that could be created.

### Prompt:

- **"We've seen a few ways to process our array of events over the course of this lesson, but there are many other effects we could produce. How else could we use the information we stored in our array? What other effects do you think we could make?"**

### Discuss:

Either in groups or as a class, students should share the ideas they brainstormed. Ask students to describe how they would actually process the array to develop the effects, ideally by referencing specific the programming constructs they would need.

### **Remarks**

Processing lists of information is a very powerful ability. We've just brainstormed many different ways we could process lists of points in a drawing app, but those same skills could be used to process lists of transactions, images, messages sent through an app, or anything else that is stored in a list. Keep an eye out for other instances where we can use list processing to create new features in your programs.

# Standards Alignment

## Computer Science Principles

- ▶ **1.2** - Computing enables people to use creative development processes to create computational artifacts for creative expression or to solve a problem.
- ▶ **1.3** - Computing can extend traditional forms of human expression and experience.
- ▶ **2.2** - Multiple levels of abstraction are used to write programs or create other computational artifacts
- ▶ **4.1** - Algorithms are precise sequences of instructions for processes that can be executed by a computer and are implemented using programming languages.
- ▶ **5.1** - Programs can be developed for creative expression, to satisfy personal curiosity, to create new knowledge, or to solve problems (to help people, organizations, or society).
- ▶ **5.3** - Programming is facilitated by appropriate abstractions.
- ▶ **5.4** - Programs are developed, maintained, and used by people for different purposes.
- ▶ **5.5** - Programming uses mathematical and logical concepts.



This curriculum is available under a  
Creative Commons License (CC BY-NC-SA 4.0).

If you are interested in licensing Code.org materials for commercial purposes, **contact us**.



# Lesson 18: Practice PT - Create Your Own App

## Overview

**Note - We recommend you skip this lesson. It has not been updated to match the 2018 Create PT Scoring Guidelines and its contents are now covered in the AP Create PT Prep Unit.**

**This lesson has been marked for deprecation. Never fear, many of the original contents (including an updated version of the Grumpy Cat exemplar) are now included in the AP Create PT Prep unit. Since some contents of this lesson may already have been used by students to create projects it will not be removed during the 2017-2018 school year.**

**For any question about this please write in to [support@code.org](mailto:support@code.org).**

**Best, CSP Curriculum Team**

To conclude their introduction to programming, students will design an app based off of one they have previously worked on in the programming unit. Students will choose the kinds of improvements they wish to make to a past project in order to show their ability to make abstractions and implement complex algorithms. The project concludes with reflection questions similar to those students will see on the AP® Create Performance Task. Students can either complete the project individually or with a partner. Every student will need a collaborative partner with whom they will give and receive feedback.

**Note:** This is NOT the official AP Performance Task that will be submitted as part of the Advanced Placement exam; it is a practice activity intended to prepare students for some portions of their individual performance at a later time.

## Purpose

A skill that programmers must develop is the ability to plan and execute a project from idea all the way through shipping of a product. Some of the best apps are new ideas brought on by the past work of a programmer themselves or other programmers. In order to execute these new ideas programmers must identify the programming structures needed to implement their idea and create a project plan. Often there are deadlines on projects which require programmers to make choices about the top features which need

## Objectives

**Students will be able to:**

- Complete reflection questions in a format similar to those on the AP performance tasks.
- Collaborate to give and receive feedback on program implementation to improve program functionality.
- Update existing code to add new functionality to a program.
- Create a video demonstrating the functionality of a program.

## Links

**Heads Up!** Please make a copy of any documents you plan to share with students.

### For the Students

- **Practice PT Overview and Rubric - Improve Your App** [Make a Copy](#) ▾
- **Practice PT Planning Guide - Improve Your App** [Make a Copy](#) ▾
- **Unit 5 on Code Studio**



to be in a release of a new product. Finally, programmers must be able to express to others the work they have done to create their app.

**AP® is a trademark registered and/or owned by the College Board, which was not involved in the production of, and does not endorse, this curriculum.**

## **Agenda**

### **Getting Started**

**Brainstorm: Programming Projects and Concepts So Far**

### **Activity**

**Review Code Studio Levels**

**Students identify target App and major components that must be programmed.**

**Students individually program major components.**

**Work with classmates to give and receive feedback.**

**Students complete project reflection questions and create video.**

### **Wrap-up**

**Submit and potentially present submissions.**

### **Extended Learning**

### **Assessment**



# Teaching Guide

## Getting Started

### Brainstorm: Programming Projects and Concepts So Far

- 🔗 **Goal:** Recall the programming projects, both large and small, done in the unit. Review how they can be used to frame the coming project as a practice for the Create Performance Task.

#### 🎤 Remarks

For the project we are beginning today, you are going to create a project of your choice built on past work. Let's make a list of all the past projects you have worked on and the programming concepts you have learned.

**Brainstorm:** Divide a piece of paper in half the long way. On the left side of the piece of paper, list all the programming projects you have done so far in this unit. On the right side of the piece of paper, list all the programming concepts you have learned so far.

**Share Out:** Have students share what they wrote and compile a class list of programming projects and programming concepts. Congratulate students on coming this far! That's a lot of things they have learned!

## Activity

### Review Code Studio Levels

There is not much here. A student introduction, and place for them to create and submit their project.

#### 💡 Teaching Tip

Here is a pretty extensive list of the things students should come up with.

Programming Projects (apps)	Programming Concepts
<ul style="list-style-type: none"><li>• Digital Scene</li><li>• Chaser Game</li><li>• Multi-screen App</li><li>• Clicker Game</li><li>• Mad Libs</li><li>• Secret Number</li><li>• Dice Game</li><li>• Digital Assistant</li><li>• Coin Flipping Simulation</li><li>• Word/Image Scroller</li><li>• Drawing App</li></ul>	<ul style="list-style-type: none"><li>• Turtle Commands</li><li>• Functions<ul style="list-style-type: none"><li>◦ Functions with Return Values</li></ul></li><li>• Parameters</li><li>• Looping<ul style="list-style-type: none"><li>◦ While</li><li>◦ For</li></ul></li><li>• Random Numbers</li><li>• Commenting</li><li>• Debugging<ul style="list-style-type: none"><li>◦ Debug Console</li><li>◦ Debug Commands</li><li>◦ Console.log</li></ul></li><li>• Events</li><li>• UI Elements<ul style="list-style-type: none"><li>◦ Buttons, Text Labels, Dropdowns, Images, Screens, Sounds, etc.</li><li>◦ Prompt and PromptNum</li></ul></li><li>• Variables</li><li>• Conditionals (If, Else if, Else)<ul style="list-style-type: none"><li>◦ Boolean Expressions</li><li>◦ Boolean Operators (&amp;&amp;, &amp;&amp;, &amp;&amp;)</li></ul></li></ul>

#### 🖥️ Code Studio levels

### Lesson Vocabulary & Resources

1

(click tabs to see student view)

### Practice Performance Task: Create

2

(click tabs to see student view)



# Students identify target App and major components that must be programmed.

## **Remarks**

Now that we've jogged your memory...for our final project of the unit you will use one of the projects we've done already as a point of inspiration to make something new. You may build on and add features to an app you wrote before. You may also write something completely new that you are inspired to create.

## **Distribute: Practice PT Planning Guide - Improve Your App**

This planning guide should help students think about how to plan and execute the project. The planning guide contains a link to **Practice PT Planning Guide - Improve Your App** for students as well. Students should begin reviewing the project guidelines and getting down to work. This project will take some time, as it has new elements, such as a video, and it asks students to create PDF documents of their write-ups.

A proposed schedule of the steps of this project is included below, as well as more thorough explanations of how to conduct the various stages.

### **Day 1**

- Review the project guidelines and the rubric.
- Assign students to collaborative partners.
- Have students brainstorm and complete App Design Guide guide in Practice PT Planning Guide - Improve Your App.

## **Students individually program major components.**

### **Day 2**

- Students begin work on programming projects.
- Add at least one or two new features/components to the app.

## **Work with classmates to give and receive feedback.**

### **Day 3**

- Students give and receive feedback with collaborative partner.
- Students pick two pieces of feedback to act on and improve in their program.
- Continue working on program.

### **Day 4**

- Students finalize their first implementation of the program.
- Students begin their reflection questions and/or video.

## **Students complete project reflection questions and create video.**

### **Day 5**

- Students complete their reflection questions and/or video.
- Students submit their projects.

## **Teaching Tip**

**Complete Project Planning Guide:** Students should use the **Practice PT Planning Guide - Improve Your App** to develop an overview description of their target app. The first thing students should do as part of planning is to...

**Read Requirements:** Read through the guidelines of the project together and address any high-level questions about the aims of the project. Students will have a chance to review the requirements once they start planning.

**Assign Collaborative Partners:** On the real Create Performance Task, teachers are not supposed to give much help to students. Instead students are supposed to work with a collaborative partner. Assign each student a specific person as her collaborative partner.

**Note:** Students can work with a partner to create an app together. They should probably still consult with someone outside of the partnership who does not know the details of their project. This will help with the feedback process.

## **Teaching Tip**

If students are having difficulty developing their project plan, encourage them to talk with their collaboration partner. Develop the expectation that prior to asking you for help, students will have consulted one another.

**Program:** Students should work individually to program their app or portion of their app. While they are responsible for writing their own code for the project, they may still consult with the other members of their class, especially their collaborative partner.



## 💡 Wrap-up

### Submit and potentially present submissions.

**Self-assess:** It can be a useful exercise to have students briefly assess themselves using the rubric they were provided at the beginning of the project. Ask them to identify points where they could improve, and remind them that this rubric is very similar in structure to the one that will be used on the actual AP Performance Tasks they will see later in the year. **Presentation (Optional):** If time allows, students may wish to have an opportunity to share their final apps with one another. Consider options like creating a “Digital Gallery” by posting all links to a shared document.

**Presentation (Optional):** If time allows, students may wish to have an opportunity to share their final

💡 apps with one another. Consider options like creating a “Digital Gallery” by posting all links to a shared document.

## Extended Learning

- Locate the most recent Performance Task Descriptions:

<http://media.collegeboard.com/digitalServices/pdf/ap/ap-computer-science-principles-performance-assessment.pdf>

- Locate the most recent Performance Task Rubrics: <http://www.csprinciples.org/home/about-the-project>

## Assessment

**Rubric:** Use the provided rubric (in **Practice PT Overview and Rubric - Improve Your App** ), or one of your own creation, to assess students' submissions.

### 💡 Teaching Tip

If students work in partners, they will need some way to combine their code. Possible solutions are:

- emailing links to their individual code \*creating a shared document / spreadsheet into which students can paste links Groups only need to create one program which contains all of their work. Individual group members can then “Remix” this project or just copy the code by using a link.

**Peer Consultation:** After students have finished implementing a draft of their program, they should meet with their collaborative partner, present their work so far, and provide feedback regarding their progress. They should complete the Feedback Guide. Other potential questions to address: **Is there anything that's particularly clever or gives you ideas for your own project?** Do you agree with the choices your partner has made? Is there anything missing?

### 💡 Teaching Tip

**Reflection Questions:** Students will complete the reflection questions included in **Practice PT Planning Guide - Improve Your App**.

**Video Creation:** Students will create a video to demonstrate the functionality of their program. The video should not be longer than 1 minute. It does not need sound.

#### **Video Creation - Suggested Tools:**

- Many of the short program clips of programs running throughout the curriculum were created using LICECAP, which is an easy way to create gifs of things happening on your computer - <http://www.cockos.com/licecap/>
- QuickTime - You can do a screen recording with QuickTime as well. Can use QuickTime on Mac or Windows.





#### Teaching Tip

**Adding Code Segments To PDF:** In order to add pictures of segments of their code, students may need to take screenshots. Below are the shortcuts for a couple different platforms for taking screenshots (a picture of part or all of the computer screen).

Chromebook	
Mac	Command + Shift + 4
Windows	Print Screen Button

**Adding Shapes:** One way you can add shapes to a picture is by using the drawing feature of Google Docs. Click Insert -> Drawing. Then add the image you want to put the shape on by clicking on . Then pick the shape you need from the dropdown .

Many PDF viewers also have the ability to add simple shapes to a document. If neither of those options seems to be working for students, they can always print a copy, draw on the shapes, and scan it back in.

**Project Submission:** Students will submit their projects, but they will need instructions on how to submit them as there are several different files. For the real performance task, all documents will have to be combined into a single PDF file. They need to hand in:

- A copy of their Planning Document
- A copy of their Feedback Guide where they got feedback
- A separate PDF with their write-up
- A video demo of their code
- A copy of their code. (Note: Although they can submit their code directly through Code Studio, they will not be able to put the ovals and rectangles required for the Performance Task. Students should practice copying their code and adding those shape components to a PDF. )

**Final Submissions:** Make a determination of how best students can submit final work. On the actual Performance Tasks, students will be required to submit all of their documents in a single PDF document, but it

#### Teaching Tip

Feel free to exclude the wrap-up activities in the interest of time. Neither is an essential portion of the Performance Tasks and they are included only to provide a more natural conclusion to the project within your class.

To make grading easier, you might have students anonymously score projects according to the rubric. Both the scorer and score should be anonymous.



# Standards Alignment

## Computer Science Principles

- ▶ **1.1** - Creative development can be an essential process for creating computational artifacts.
- ▶ **1.2** - Computing enables people to use creative development processes to create computational artifacts for creative expression or to solve a problem.
- ▶ **2.2** - Multiple levels of abstraction are used to write programs or create other computational artifacts
- ▶ **4.1** - Algorithms are precise sequences of instructions for processes that can be executed by a computer and are implemented using programming languages.
- ▶ **5.1** - Programs can be developed for creative expression, to satisfy personal curiosity, to create new knowledge, or to solve problems (to help people, organizations, or society).
- ▶ **5.4** - Programs are developed, maintained, and used by people for different purposes.
- ▶ **5.5** - Programming uses mathematical and logical concepts.



This curriculum is available under a Creative Commons License (CC BY-NC-SA 4.0).

If you are interested in licensing Code.org materials for commercial purposes, **contact us**.