

# **An Introduction to Programming**

A Pencil Code Teacher's Manual



By Deepa Muralidhar  
and David Bau

## Dedication

This book is dedicated to my mother - *Late Smt. Rajam Krishnan* and my mother - in -law *Smt. Malathi Raman* for instilling grit and resilience while shaping me, and to the educational team at Google which has done so much inspiring work in bringing computer science to students.

## Table of Contents

1. Introduction to the Pencil Code environment
    - 1.1 An introduction to the environment
  2. Lines and Points
    - 2.0 Chapter Introduction
    - 2.1 Lesson Plans
    - 2.2 Resources
  3. Input / Output Statements
    - 3.0 Chapter Introduction
    - 3.1 Lesson Plans
    - 3.2 Resources
  4. Loops
    - 4.0 Chapter Introduction
    - 4.1 Lesson Plans
    - 4.2 Resources
  5. Functions
    - 5.0 Chapter Introduction
    - 5.1 Lesson Plans
  6. Selection Statements
    - 6.0 Chapter Introduction
    - 6.1 Lesson Plans
    - 6.2 Resources
  7. Learning a Second Language: JavaScript
    - 7.0 Chapter Introduction
    - 7.1 Lesson Plans
  8. Introducing Lists and One-Dimensional Arrays
    - 8.0 Chapter Introduction
    - 8.1 Lesson Plans
  9. Nested Loops
    - 9.0 Chapter Introduction
    - 9.1 Lesson Plans
    - 9.2 Resources
  10. Recursion
    - 10.0 Chapter Introduction
    - 10.1 Lesson Plans
  11. Building a Website Using HTML, CSS
    - 11.0 Chapter Introduction
    - 11.1 Lesson Plans
  12. Traversing Data Using JQuery
    - 12.0 Chapter Introduction
    - 12.1 Lesson Plans
- Appendix A: Pencil Code - Recommended coding standards  
Appendix B: Links to the list of programs used in the manual  
Appendix C: Pacing Guide (1 semester)

## Foreword

I saw Pencil Code in 2015 SIGCSE. I realized that it was a tool that would be an excellent fit to teach programming in the high school classroom.

Students find block programming languages a non-intimidating way to start programming; however, blocks can eventually come in the way of productive learning. Pencil Code has a very nifty ability to switch between blocks and text code. When it comes to trying try a new concept or to understand the structure of a program to spot a bug, block code is very useful. But when it is time to write a quick program to solve a problem, text coding can be the better approach. I could see that the ability to smoothly transition between block and text mode could reduce the frustration and intimidation that students encounter while learning to program.

David and I have been working for the last 10 months to put together a teacher's manual to guide teachers and students to be able to use Pencil Code effectively. This manual is a reflection of our work as David as a computer scientist at Google and MIT and my 16 years of experience as a high school teacher.

All this work would not be possible if not for the contributions of many outstanding dedicated individuals.

First of all we would like to thank Google for the financial contributions in making this Pencil Code teacher's manual a reality. In particular Chris Stephenson for her vision and direction in making this manual a useful resource for the teachers and students, and Maggie Johnson and Steve Vinter for continuing support of Pencil Code and other computer science education efforts.

We would also like acknowledge the people who put Pencil Code together. Pencil Code is an open source project with many contributors. David's son Anthony Bau created the block-mode editor (Droplet). Pencil Code was developed by testing it in Citizen Schools classrooms taught by Google engineers Ethan Apter, Yana Malysheva and James Synge. And Pencil Code has been the beneficiary of many open-source contributions, in particular from students supported by Google Summer of Code.

We would like to thank the Programming for Novices course at Middlebury College, created by Professor Amy Briggs for providing us with relevant well illustrated examples to use in our manual.

We would like to acknowledge Visa Thiagarajan, Subject Expert Teacher, BASIS Independent Silicon Valley at for her ideas on some of the examples illustrated in the manual.

We would like to acknowledge Pradyumna Bhattar, IT Analyst, Bank of America for his ideas on how to illustrate algorithm development in the most simplistic manner.

And we would like to thank Googlers Phil Wagner and Matt Dawson and our other reviewers for their valuable input and recommendations. Any remaining errors in these pages our our own.

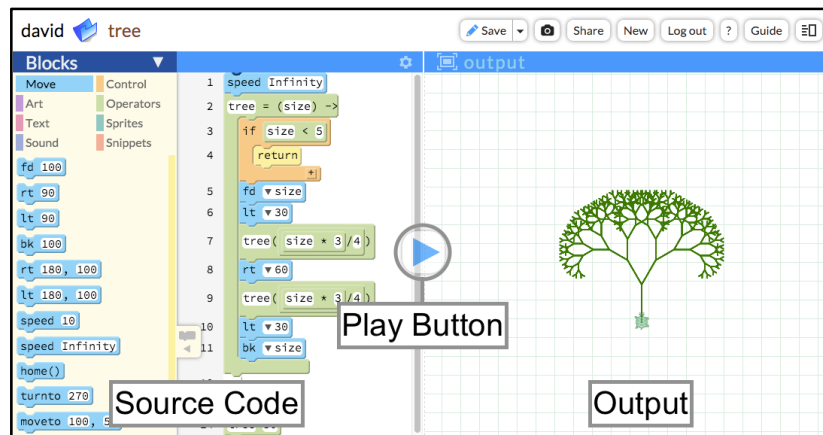
- Deepa Muralidhar and David Bau

# Chapter 1: Introduction to the Pencil Code Environment

In this manual we will show how to use Pencil Code to explore programming. Pencil Code is a free programming tool available at pencilcode.net. Pencil Code was developed by Google engineer David Bau together with his son Anthony Bau, with open-source contributions from many others.

## Two Ways of Looking at a Program

There are two ways of viewing a program. A computer user sees a program by looking at its **output**. A programmer works with a program's **source code**. In Pencil Code, the screen is split into two halves, with the source code on the left and the output on the right. You run programs by pressing the “play” button in the middle.



*The Pencil Code interface splits the screen, showing source code on the left and output on the right. The play button in the center runs the code and produces output.*

## Languages and Libraries in Pencil Code

Pencil Code supports code in both blocks and text using mainstream web programming languages and useful libraries including:

- HTML, the standard HyperText Markup Language for the web.
- JavaScript, the standard programming language of web browsers.
- CSS, the standard Cascading Style Sheet language for visual styles on the web.
- jQuery, a popular library that simplifies programming interactive websites.
- CoffeeScript, a language that lets you do more with less typing than JavaScript.
- jQuery-turtle, which extends jQuery to simplify graphics and animation for novices.

With Pencil Code, students can use these technologies to create web applications with global visibility and impact while exploring fundamental concepts in computational thinking.

We will discuss all these topics in later chapters of this Pencil Code teachers' manual.

## A Web Page is a Program

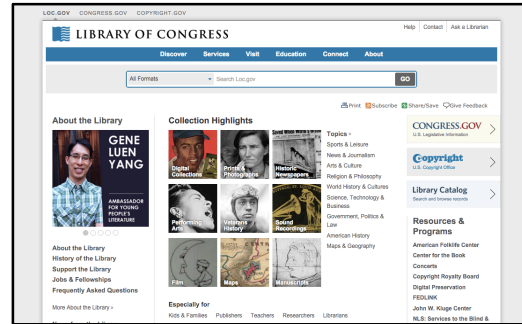
Every web page is a program, and has both source code and output.

The source code is sent to your computer when you request a web page. It may contain a combination of different languages, like HTML and JavaScript. The output is what you see when your browser interprets the source code.

```

11 <!DOCTYPE html>
12
13 <html lang="en" class="no-js" prefix="loc: http://loc.gov/#">
14 <head>
15
16 <meta name="description"
17   content="The Library of Congress is the nation's oldest federal cultural institution,
18   Congress. It is also the largest library in the world, with more than 120 million items. The
19   recordings, motion pictures, photographs, maps, and manuscripts." />
20
21 <meta name="dc:identifier"
22   content="http://www.loc.gov/" />
23
24 <meta rel="canonical"
25   href="http://loc.gov/" />
26
27
28 <meta charset="utf-8">
29 <meta name="viewport" content="width=device-width,initial-scale=1"/>
30 <meta http-equiv="X-UA-Compatible" content="IE=edge">
31 <meta name="version" content="$Revision: 31067 $"/>
32 <meta name="sevalidate:01" content="SC9F9D99599AD3JF55BD95CJAS9BD91"/>
33 <link title="schema(DC)" rel="schema:dc" href="http://purl.org/dc/elements/1.1/" />
34 <meta name="dc:language" content="eng" />
35 <meta name="dc:source" content="Library of Congress, Washington, D.C. 20540 USA" />
36
37 <meta property="fb:admins" content="Libraryofcongress"/>
38 <meta property="og:site_name" content="The Library of Congress"/>
39 <meta property="twitter:site" content="Libraryofcongress"/>
40 <meta property="og:type" content="article" />
  
```

Source code may include languages such as HTML, JavaScript, and CSS. See this example in your browser at [view-source:http://www.loc.gov/](http://www.loc.gov/).



Output is the result of your browser interpreting the source code.

Encourage your students to explore the source code of different websites by looking for hidden messages contained in the webpage sources. Go to [www.ebay.com](http://www.ebay.com), [www.flickr.com](http://www.flickr.com), or [www.mozilla.org](http://www.mozilla.org). To view source, press **Ctrl-U**. (On a Mac, the keyboard command is **Command-Option-U**, and on Safari, “view source” needs to be enabled first using Advanced Preferences.)

## Every Pencil Code Program is a Web Page

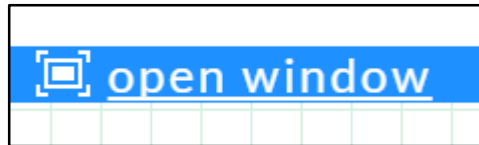
In Pencil Code, every program is a web page. At the top the editor are a few details to help control how a page is published. In the upper right are buttons for saving and sharing the program, as well as buttons for managing your website and getting help. The upper left shows the name. Rename a program by clicking on the name in brown and editing it.



The name sets the URL web address for the program, as shown in the examples in the table below.

Account name	Project name	Output URL
coolsite	first	<a href="https://coolsite.pencilcode.net/home/first">https://coolsite.pencilcode.net/home/first</a>
david	example/posterize	<a href="https://david.pencilcode.net/home/example/posterize">https://david.pencilcode.net/home/example/posterize</a>

Hovering over the “output” icon in the blue bar on the upper right will provide an “open window” link that opens a new tab showing just the output of the program as it would appear to users visiting the webpage. It does not show code. (This link is available only after logging and saving a program.) It is valuable to try running your programs full-screen, and from there use Ctrl-U to “view source” on your own webpage.



*Clicking the open window button will run the program in full-screen mode, without showing source code.*

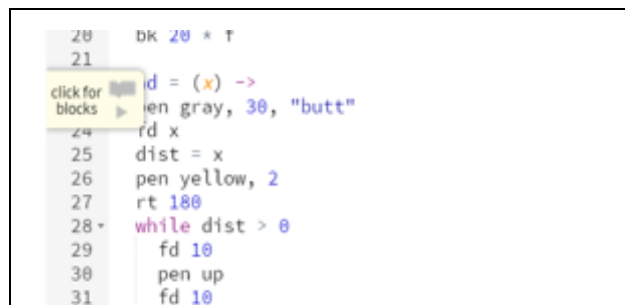
In Pencil Code, the full-screen output URLs have the word “/home/” in them. These addresses can be linked, emailed, or embedded anywhere. Changing the “/home/” to “/edit/” will make a URL to show the Pencil Code editing UI, revealing the source code for any program on Pencil Code.

## What is a Programming Language?

Pencil Code allows the programmer to use “block-mode” to drag and drop blocks to design a program. The blocks in Pencil Code are a direct representation of an underlying text language: CoffeeScript, JavaScript, or HTML. Although the blocks look different from text code, they are just a visual way to view and edit instructions in a programming language.



*Viewing Source Code in Block Mode*



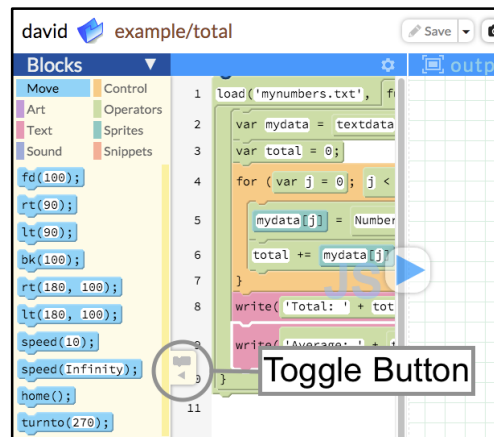
*Viewing Source Code as Text*

A programming language is any language that is precise enough for a machine to interpret, while also being understandable by people. The words “run this over and over” in English mean the same thing as the Intel Pentium opcode “111010111111110”. But the English words are too ambiguous for a computer to follow, and the machine opcodes are too obscure for a person to read. A programming language is written in readable words, but in a way that follows precise patterns, called a **syntax**, that a computer can follow precisely.

When viewing JavaScript or CoffeeScript in block-mode in Pencil Code, the syntax of the programming language is shown through the block structure. For example, when words are part of different commands, they are shown as different blocks. When one command is under the control of another, the blocks show the commands nested within one another.

## Switching Between Blocks and Text

In Pencil Code, block-mode and text-mode are perfectly equivalent in power and expressiveness. Blocks are a just a visual view of the syntax of JavaScript, CoffeeScript, or HTML, and students can switch between blocks and text freely.



*This yellow tab with a gray arrow is a toggle button that switches between text and blocks.*

The toggle button on the yellow tab on the lower-left edge of the editing area lets the programmer switch from block to text and from text to block-mode. Hover on the tab to see the tab expand to a button that says either “click for blocks” or “click for text”.

## When to Use Blocks

When should a student use blocks or text?

The best time to use blocks is when a student is learning a new function or command. Blocks are organized on the palette with the right syntax to use and shapes that snap together correctly. They make it easy to try a new idea because you only need to recognize a block to use it.

The best time to use text is after a student knows functions and commands well enough that to type them from memory. Once students become familiar with the parts of the language they need for a project, they will find that typing can be faster and more fluid than dragging blocks.

In Pencil Code, the blocks contain code that exactly matches the syntax for the language being used. For example, when using CoffeeScript, the block to move the turtle forward by 100 pixels will read “fd 100”, which is exactly the same code to type in text mode. If students modify or add code using text mode, they can switch back to block mode to see how their code looks as blocks.

Students should feel free to work in either blocks or text, clicking the button to switch at any time. In early sections we assume students will be working with mostly blocks. As students become more familiar with the syntax of a language by remembering the syntax within the blocks, they will often want to type code directly as text, switching to blocks when trying something new, or when trying to understand work that they typed. Most students will naturally move from blocks to text as they become familiar with the functions and commands in the language they are using.



## Beginning with CoffeeScript

Pencil Code supports both JavaScript and CoffeeScript natively, but the default language in Pencil Code is CoffeeScript, and we recommend students start with CoffeeScript.

CoffeeScript is a professional language that is used by many tech companies including Github and Dropbox. Its power and speed are equivalent to JavaScript. CoffeeScript, however, has a simpler syntax (similar to Python) that uses meaningful indents and less boilerplate punctuation. The simpler syntax requires less typing when students make the first tricky leap to a text language. It also clarifies the code for concepts such as functions, nesting, loops, input, and arrays.

```
pen(red);
for (var i = 0; i < 20; i++) {
  fd(i);
  if (i < 10) {
    rt(90);
  }
  dot(blue);
}
```

*JavaScript is the standard programming language of the web, but the punctuation in the language can be overwhelming to a novice.*

```
pen red
for i in [0...20]
  fd i
  if i < 10
    rt 90
  dot blue
```

*CoffeeScript is a popular language used by professionals to abbreviate JavaScript. It requires less punctuation than JavaScript.*

In this manual we will use CoffeeScript up to Chapter 7, after which we will introduce JavaScript.

## Comments

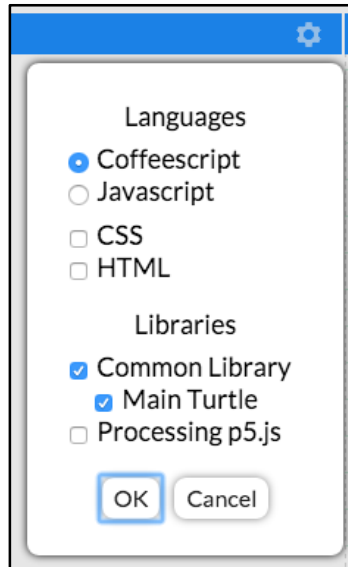
A note on comments: to create a comment in the code in block, first create an empty block by pressing the “Enter” key and then type in the block starting with the # sign. The comment block now looks something like this:

```
1 # A comment: this program says hello.
2 say 'hello'
```

The # symbol is the CoffeeScript comment symbol. To create a comment in JavaScript, use “//”.

## Switching Languages - CoffeeScript, JavaScript, HTML, and CSS

To switch from CoffeeScript to JavaScript, click on the "gear icon" in the blue bar. From this box, choose between the two scripting languages and optionally add panes for either or both of the layout languages HTML and CSS. You can also enable or disable the main turtle here, which is helpful if you are making a program that does not use the turtle.

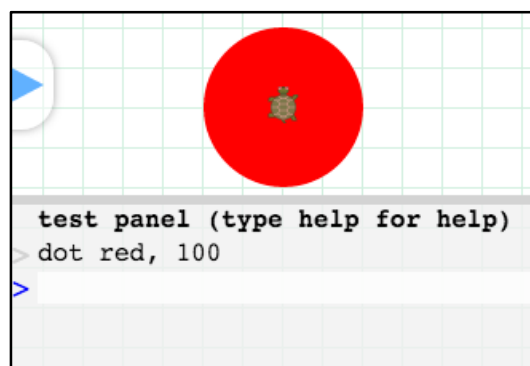


*The gear button opens a settings panel that allows switching languages and libraries.*

Settings in this panel will be remembered next time you create a new program. To switch settings again, just use the gear button again.

## The Test Panel

One way to explore commands is by typing them in first to try them. The test panel in the lower-right side of the screen lets you type individual commands in CoffeeScript or JavaScript. For example, if you type "dot red, 100" and press enter, the command will be run right away so you can see what it does.



*The test panel. If the test panel is not visible, it may be necessary to open it by dragging the gray divider.*

Type a command by itself in the test panel, without any arguments, to get a bit of help about it. (See below for an example of getting help about “label”.) If the test panel is too small to see the help, the dark gray bar at the top of the panel can be dragged to increase it or decrease it.

```
test panel (type help for help)
> label
label(text) Labels the current position with HTML: label 'remember'
label(text, styles, labelsite) Optional position specifies 'top',
'bottom', 'left', 'right', and optional styles is a size or CSS object:
label 'big', { color: red, fontSize: 100 }, 'bottom'
```

You can also click the examples highlighted in yellow in the help text to try them out right away.

### Debugging Using the Test Panel

You can use the test panel to debug the variables in a program. For example, try running a program in the left-hand panel that reads “x = 42” as follows (if using blocks, find the variable assignment operator under the “Operators” panel).

```
1 x = 42
2
```

Then type “x” in the test panel and press enter to see the value of x. If the test panel says “x is not defined,” it means that the program has not run yet - just press the “play” button, and then interact with the program after it has run to see the value of x is 42.

There is a special “debug” command that can be used to produce output directly to the test panel without interfering with the main part of the web page (find the debug block under the “Text” panel). Try creating a program that reads “debug ‘hello’” as follows. The word “hello” needs to be in quotes.

```
1 debug 'hello'
```

When you run it, the test panel will say hello! (Debug is an abbreviation for the “console.debug” command often used by web programmers, which will also work the same way.)

### The Pencil Code Library

An experienced programmer may ask “what functions are available to a Pencil Code program?” About 100 of the functions that can be used in Pencil Code are listed on the block palette but Pencil Code provides a large open-ended library of functions that goes far beyond what is listed in the palette. Basically, anything that a web page you can do in Pencil Code.

Only a small fraction of these functions will be discussed in this teacher’s manual, but armed with the names of the libraries below, you can find many examples and tutorials on the Internet with code that can be used in Pencil Code. The libraries available to every Pencil Code program include:

1. **The Web Document Object Model (DOM).** Standardized by international committee, these functions are available to every page on the Internet.
2. **jQuery.** The most widespread web page library on the Internet, used by most popular websites. We will introduce the workings of the jQuery library in Chapter 11.
3. **jQuery-turtle.** The turtle library for Pencil Code is an extension to jQuery. It provides all the simple-to-use functions that we will take advantage of in the first part of this manual. Most of the functions on the block palette are from this turtle library.
4. **socket.io.** This is a real-time communications library that enables immediate communication between browsers..

## Exploring the Vast Library Beyond Turtle Functions

Although the web programming world has too many features to cover in a single manual, all the objects available to a Pencil Code program can all be explored using the test panel. For example, type “location” to view the DOM “location” object, and they expand it by clicking on the triangle. The test panel shows that “location” contains many functions and pieces of data including “href”: a program could use this with the variable `location.href`.

```
test panel (type help for help)
> location
▼Location
  replace: function () { [native code] }
  assign: function () { [native code] }
  hash: ""
  search: ""
  pathname: "/home/first"
  port: ""
  hostname: "pencilcode.net"
  host: "pencilcode.net"
  protocol: "http:"
  origin: "http://pencilcode.net"
  href: "http://pencilcode.net/home/first"
  ancestorOrigins: DOMStringList{0: "http://pencilcode.net"}
  reload: function reload() { [native code] }
```

Web programming functions are widespread enough that there are pages on the Internet about almost every one of them. A Google search for “location.href” will bring up excellent pages that explain it.

## How to Use This Book

### Goals and Standards:

This teacher’s manual is designed to help students learn the basics of programming. It is intended to assist a teacher in teaching an *Introduction to Programming* one semester course.

This manual shows how to take students step-by-step through the Pencil Code environment and start writing simple programs. The chapters are organized around the fundamental programming constructs, starting with basic concepts and then moving on to more advanced concepts. The manual also shows how to transition students from block coding to text coding: programming for beginning students can be intimidating, and starting with block-mode can reduce the level of intimidation. While the focus is on learning programming, many of the programs are aimed at problem solving.

The content for this teacher’s manual is based around the CSTA K-12 Standards. Every chapter consists of a section that does a crosswalk of the lesson plans to specific standards of the framework. The lessons are based on programs designed by David and Deepa, many of which are available on [guide.pencilcode.net](http://guide.pencilcode.net).

## **How should this book be read?**

This manual is intended primarily for teachers who would like to teach programming using Pencil Code. There are several sections in each chapter to help the teacher in each topic. The [key concepts](#) section give the teacher a quick technical overview of the topic. The [key terms](#) identify important words / terms that are used in the chapter. Finally the [lesson plans](#) guide the teacher through each program. A teacher new to teaching programming can follow the lessons and teach the students as suggested in the manual. The more experienced teacher can use the programs and use the lesson plans as suggestions on how to teach the various concepts. Every program represents an idea on how to solve problem and how the programming construct that is be taught can be used solve it. Teachers are encouraged to use the lesson plans they find useful in the classroom and modify them to fit the needs of their students.

There is a suggested pacing for each lesson in the [Suggested timeline](#) section. Note that each chapter has several lesson plans spanning over a couple of class periods. There is a separate pacing guide (Appendix C) that gives the sequence and pacing on how the material provided by the manual should be used.

While this is intended to be a teacher's manual, an advanced student can peruse through the various programs and use them as resources to funnel their creativity as they create their projects on Pencil Code.

## **What grade level students is the material in the book appropriate for?**

This manual is intended for a high school, an introduction to programming course. Students 9<sup>th</sup>, 10<sup>th</sup> and possibly 11<sup>th</sup> graders would benefit from taking this course. An advanced 8<sup>th</sup> grade student could take this course. A typical math pre-requisite of pre-algebra would be sufficient to take this course.

# Chapter 2: Lines And Points

## 2.0.1 Objectives

In these lessons, we introduce straight-line programs that use turtle graphics to create visual output. A straight line program runs a series of directions in the same order each time the program is run. Students will learn how to plan, create, and debug a sequence.

## 2.0.2 Topic Outline

- 2.0 Chapter Introduction
  - 2.0.1 Objectives
  - 2.0.2 Topic Outlines
  - 2.0.3 Key Terms
  - 2.0.4 Key Concepts
- 2.1 Lesson Plans
  - 2.1.1 Suggested Timeline
  - 2.1.2 CSTA Standards
  - 2.1.3 Lesson Plan I on using the Move Blocks
  - 2.1.4 Lesson Plan II on using the Art Blocks
  - 2.1.5 Lesson Plan III on using the Arcs
  - 2.1.6 Lesson Plan IV on using the Assignment operator
- 2.2 Resources
  - 2.2.1 Videos
  - 2.2.2 Useful links
  - 2.2.3 Additional exercises

## 2.0.3 Key Terms

Sequencing	Algorithms
Bugs	Pen
Cartesian geometry	Deterministic
Turtle geometry	Trace

## 2.0.4 Key Concepts

A **program** defines a **sequence** of actions for a computer to take.

- **Straight line programs** run a sequence of actions from top to bottom without making choices. These simple programs are **deterministic**: they always take the same actions in the same order every time they run, and the sequence of actions can be read directly by reading the program.
- Even a deterministic program can have **bugs**. A bug is any behavior the user or the programmer does not want, for example, a program that draws a different shape than the one you want.
- To **debug** a program, it is helpful to **trace** (carefully follow) the programs steps as they run. Each step is called a different **state** of the program.

The programs we have used so far created graphics on the screen. There are two types of commands for creating graphics that we have used:

- **Turtle geometry**, which draws lines, angles, and other shapes by controlling the direction and movement of a screen object.

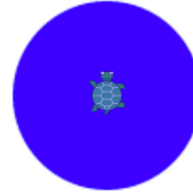
- **Cartesian geometry**, which draws lines or other shapes by using (x, y) coordinates to navigate the screen. For example, the `moveto` command moves the turtle using Cartesian geometry.

## Drawing at a Point

The simplest drawing is a single a dot or a box at the current location of the turtle.

```
dot blue, 100
```

The `dot` command draws a colored circle of a specified size directly at the location of the turtle.

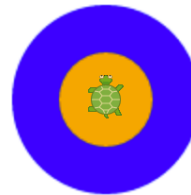


The number is the diameter in pixels. (In the case of a box, the number is the side length.) There are about 100 pixels in an inch (about 40 pixels in a centimeter), with the exact scale depending on the device being used. Many colors are available for drawing: there are 140 standard CSS color names including common names like "red" and uncommon ones like "gainsboro." A full table of color names, together with as a list of useful function names, can be found on the Pencil Code one-page reference sheet at <http://reference.pencilcode.net/>.

Drawing a dot or a box does not move the turtle. If a second dot is drawn, that dot is drawn at the same location as the first dot. Order matters: the second dot will cover the first one, and if it is larger, it can completely hide the first dot.

```
dot blue, 100
dot orange, 50
```

Order is important: drawing a second dot will draw it on top of the first one.



## Motion and Lines

The turtle can move forward and backward in a straight line using the `fd` and `bk` commands. A row of three dots can be created by moving the turtle between each dot.

```
dot pink, 25
fd 25
dot pink, 25
fd 25
dot pink, 25
```

The turtle moves forward using `fd`.



The turtle can also draw with a pen as it moves using the `pen` command. The pen has a color and thickness, chosen the same way the color and diameter of a dot are chosen. Once the pen is chosen, it will draw the path everywhere the turtle goes. Use `pen off` to turn the pen off again.

```
pen purple, 10
fd 25
pen off
fd 25
dot aqua, 25
```



The turtle creates a line using pen.

## Turning and Angles

Pivot the turtle to the right by using `rt`, and left using `lt`. These commands turn in units of degrees.

```
pen red, 5
lt 90
fd 100
rt 90
fd 100
rt 30
fd 100
```



Turning and making angles using `rt` and `lt`.  
Notice that small turns create obtuse angles.

Notice that a 30 degree turn creates a 120 degree angle! When the turtle changes direction by only a small amount, the angle created is very large. A mathematician would say that the amount of change in turtle direction (30 degrees) is the exterior angle measure, whereas the angle you get (120 degrees) is the interior angle measure.

To create a thin acute angle, the turtle must turn sharply and change its direction by more than 90 degrees. A 180 degree turn is the sharpest turn possible, turning the turtle around backwards.

## Debugging with Dots and Arrows

When working with a complicated program that creates a drawing, it can be helpful to add a dot before or after a line of code being investigated. The dot itself will not move the turtle, so it is useful for recording where the turtle is located when the program runs that line of code. There is also an arrow drawing command which can be used to draw the current direction of the turtle without moving the turtle.

```
bk 100
pen red, 5
lt 90
fd 100
dot blue, 25
rt 90
fd 100
arrow blue, 50
rt 30
fd 100
```



Using a blue dot and arrow to help debug the execution of code.



Adding extra output to record the state of the program at a given line of code is the most common debugging technique used in all sorts of programmers.

For example, if one angle in a drawing is not correct, the first step of the solution is to find the specific line of code responsible for that angle. Adding dots and arrows help to identify what the turtle was doing when the program arrived at a specific step, and can help to narrow the problem. Once the problematic line is found and fixed, the extra dots and arrows can be removed.

## Using Other Images

It is possible to change the turtle to any image on the internet. To output a “dog” image, try using the “wear” block:

```
wear 'dog'
```

*The wear command outputs an image by changing the appearance of the turtle to an image from the internet*

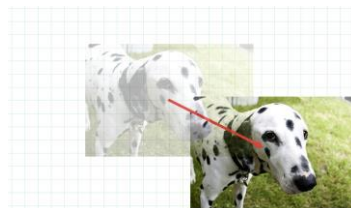


The wear command changes the turtle appearance to any image URL that your browser can load. When you use a short name such as “dog,” Pencil Code loads the image using special image URLs starting with <http://pencilcode.net/img> that find an image using a creative-commons image search. These URLs showing freely reusable images matching the name after the /img, such as showing a mountain for <http://pencilcode.net/img/mountain>. If you ask for an image starting with t- such as 't-dog', it will provide an image with some transparency.

The image can be moved by moving the turtle. For example, use the following to move the turtle to a point 200 pixels to the right and 100 pixels above the origin:

```
moveto 200, -100
```

*The moveto command moves the image to a location using Cartesian coordinates.*



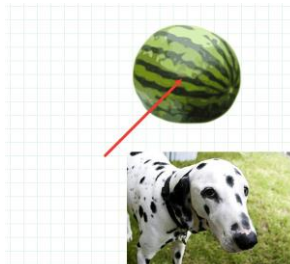
The moveto command is different from the turtle motion commands such as fd and rt, because it is an absolute motion, locating a point in Cartesian coordinates, whereas the turtle motion commands are relative motions, making motions relative to the current location and direction of the turtle.

## Moving a Second Image Using a Variable

To create a second image on the screen, use the “img” command. It can be moved (or manipulated in any way that a turtle can) by using a variable, and using dot notation:

```
w = img 't-watermelon'  
w.moveto 150, 150
```

*This code creates a new image showing a watermelon with transparency, then using the variable w, moves it to the location 150, 150.*



The code above introduces two of the most important concepts in programming: it assigns a variable, `w`, using the “=” operator, and it directs commands to it using the dot notation “.”.

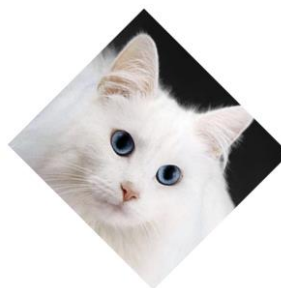
A variable is a name defined by a program to represent some object or data, and can be chosen to be any word that is memorable for the programmer. For example, the variable “`w`” above was chosen to represent an image of a watermelon. Another sensible name might have been “`wm`” or “`melon`” or simply “`watermelon`”.

The “=” operator in `w = img 't-watermelon'` is slightly different from the “=” used in math class. It does not mean that `w` is known to be equal to the image. It is an assignment. The “=” assigns the meaning of the variable `w` to refer to the image of the watermelon. If, prior to the assignment, `w` had some other meaning, then that old meaning is discarded after the assignment.

The “.” operator in `w.moveto 150, 150` directs the `w` object to execute the `moveto` function, instead of telling the turtle to move. Images can be moved like turtles, so “.” operator can be used together with any turtle function. In the example below, `c` is a variable for a cat image, and `c.rt 45` tilts it right 45 degrees.

```
c = img 'cat'
c.moveto 0, 0
c.rt 45
```

*This code uses the variable `c` for a cat image, then moves the cat to the origin, then tilts the cat right by 45 degrees.*



### 2.1.1 Suggested Timeline: 1 55-minute class period

Instructional Day	Topic
1 Day	Lesson Plan I
1 Day	Lesson Plan II
2 Days	Lesson Plan III & IV


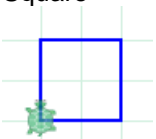
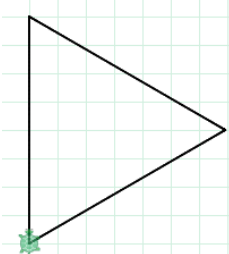
### 2.1.2 Standards

CSTA Standards	CSTA Strand	CSTA Learning Objectives Covered
Level 3 A (Grades 9 – 12)	Computational Thinking (CT)	Explain how sequencing, selection, iteration and recursion are building blocks of algorithms.
Level 3 A (Grades 9 – 12)	Computing Practice & Programming (CPP)	Apply analysis, design, and implementation techniques to solve problems.
Level 3 A (Grades 9 – 12)	CPP	Use Application Program Interfaces (APIs) and libraries to facilitate programming solutions.

### 2.1.3 Lesson Plan I




This lesson will give students an overview of Pencil Code and the Move block palette.

Note: Make sure you are in block mode. Type in the code (switch to block-mode if needed) and click the play arrow to demonstrate the results.

Content details	Teaching Suggestions	Time
<p>Use the resources and the narrative in Chapter 1 as guide.</p>	<p>Give an overview of Pencil Code.</p>	<p>Demonstration: 10 minutes</p>
<div data-bbox="207 443 418 569" style="border: 1px solid black; padding: 5px; margin-bottom: 10px;"> <p><u>Line</u> pen red fd 50 rt 90</p> </div> <hr/> <div data-bbox="207 632 418 940" style="border: 1px solid black; padding: 5px; margin-bottom: 10px;"> <p><u>Square</u> pen blue fd 50 rt 90 fd 50 rt 90 fd 50 rt 90 fd 50 rt 90</p> </div> <hr/> <p>Code:</p> <div data-bbox="207 1003 375 1255" style="border: 1px solid black; padding: 5px;"> <p><u>Triangle</u> pen black fd 200 rt 120 fd 200 rt 120 fd 200 rt 120</p> </div>	<p>Demonstrate Line,</p>  <p>Square</p>  <p>and Triangle (Move block). <u>Output</u></p>  <p>pen from the Art Block actually draws the pattern on the grid. Different colors can be picked from the pen option.</p>	<p>Demonstration: 10 minutes.</p>
<p>Encourage creativity by asking students to explore the different colors and thickness of the lines of the pen.</p>	<p>Students will work on their own to create their lines, square and triangle.</p>	<p>Student Practice: 15 minutes</p>
<p>Students who are unable to complete this work in class can finish it home as homework.</p>	<p>Students will start experimenting with House and lighthouse</p>	<p>Student Practice: 20 minutes</p>

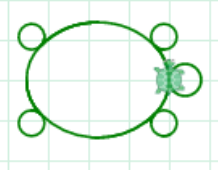
### 2.1.4 Lesson Plan II

This lesson introduces the block palette **Art**.

Content Details	Teaching Suggestions	Time
<p>Code:</p> <pre> Dot Row rt 90 dot lightgray fd 30 dot gray fd 30 dot() fd 30  Smiley speed 10 dot yellow, 160 fd 20 rt 90 fd 25 dot black, 20 bk 50 dot black, 20 bk 5 rt 90 fd 40 pen black, 7 lt 30 lt 120, 35 ht() </pre>	<p>Demonstrate <a href="#">Dot Row</a> and <a href="#">Smiley</a> (Art Block)</p> <p>Show the use of the speed block</p> <p><u>Output</u></p>  <p><u>Output</u></p>  <p>Note: Take your time as you demonstrate the smiley face. Ask the students to help you locate the position of the black eye.</p> <p>What does function ht() (last line in the <a href="#">Smiley</a> code), do?</p> <p>Explain that sequencing is a key computational thinking practice.</p>	<p>Demonstration: 15 minutes</p>
<p>Design your own... Encourage students to experiment with Dot diameter, pen color, etc.</p>	<p>Have the students will design their own versions of the Smiley face and Dot Row.</p>	<p>Student Practice: 10 minutes</p>
	<p>Students will work on creating a BullsEye artifact. Here is the code (solution)</p> <pre> speed 2 x = 20 dot black, x*5 dot white, x*4 dot black, x*3 dot white, x*2 </pre> <p>Notes:</p> <ol style="list-style-type: none"> <li>1. Encourage students to make it of various sizes and colors.</li> <li>2. Walk around the class and express satisfaction on demonstrations of personal expression.</li> </ol>	<p>Student Practice: 15 minutes</p>

### 2.1.5 Lesson Plan III

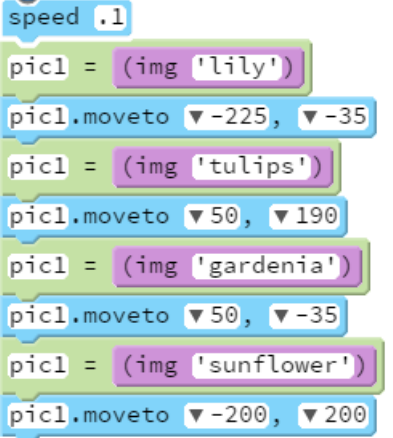

This lesson introduces the block palette Art and Move.

Content Details	Teaching Suggestions	Time
<p>Using this link  <a href="http://gym.pencilcode.net/draw/#/draw/filled.html">http://gym.pencilcode.net/draw/#/draw/filled.html</a>            Do the following:            Change the color            Change the angles and radius in the rt and lt commands.            Watch how the figure changes shape.</p>	<p>Have the students experiment with the crescent.             Encourage them to make modifications that allow for artistic expression as well as mathematical manipulations.</p>	<p>Student Practice:            10 minutes</p>
<p><u>Code:</u></p> <pre>Turtle speed 100 pen green rt 360, 10 lt 45, 30 rt 360, 8 lt 90, 50 rt 360, 8 lt 90, 30 rt 360, 8 lt 90, 50 rt 360, 8 lt 45, 30</pre> <p><u>Output</u></p> 	<p>Demonstrate the Turtle program.            Explain how angles work using this tool:  <a href="http://guide.pencilcode.net/edit/explainer/turns">http://guide.pencilcode.net/edit/explainer/turns</a>             Explain the rt (degrees) block using CoffeeScript: rt pivots right by degrees.             Explain how arcs work with rt (dg, rad) block, which turns with a turning radius  <a href="http://guide.pencilcode.net/home/explainer/curves">http://guide.pencilcode.net/home/explainer/curves</a>             Lt block does the same in the counter-clockwise direction.            Note: The code shown here is in text-mode. Encourage students to switch between block and text to “look under the hood” whenever they code.</p>	<p>Student Practice:            20 minutes</p>
<p><a href="http://activity.pencilcode.net/home/worksheet/flower.html">http://activity.pencilcode.net/home/worksheet/flower.html</a></p>	<p>Students can now implement the drawing of the turtle on the grid.             Print and hand out paper copies of the two worksheets (Flower and Car. Ask them to complete and share the exercise with you before end of class.             You could also use this assignment as a filler until the end of class, a warm-up activity, or a homework assignment.             You could offer the students a completion grade when they share the completed assignment with you.</p>	<p>Student Practice:            30 minutes</p>

### 2.1.5 Lesson Plan IV

This lesson the idea of using the img-bot to create interesting scenes and give students an opportunity for creative expression.

**Teaching Notes:** There are two concepts that have to be taught. First, the assignment operation. Pencil Code allows you to create a variable and assign anything including images. Next, using the img-bot to find a fun image on the internet the student uses the 'moveTo' block to move to a specific spot.

Content Details	Teaching Suggestions	Time
<p><b>Code:( Text Mode)</b></p> <pre> speed .1 pic1 = (img 'lily') pic1.moveto -225, -35 pic1 = (img 'tulips') pic1.moveto 50, 190 pic1 = (img 'gardenia') pic1.moveto 50, -35 pic1 = (img 'sunflower') pic1.moveto -200, 200 </pre> <p><b>Block-Mode:</b></p> 	<p>Copy / paste the program from the left-column into Pencil Code editor.</p> <p>Explain the function of img – i.e. it searches the internet and finds the first image that matches the word in quotes and displays it.</p> <p>Explain the '=' assignment statement and the '.' notation. (refer to Key concepts.)</p> <p>Explain that the image is assigned to the variable pic1. Now pic1 can be moved to a location as specified in the moveTo block. The Speed block helps give the animation effect.</p> <p>Demonstrate to students that by trial and error to find the right location on the screen to get the collage effect.</p> <p>Now ask students to create their own collage. They can explore locations, images and animation effects to produce their own unique artifact.</p> <p>The program code can be found <a href="#">here</a>.</p> <p>A good end of project activity is a reflection exercise. Ask students to write in about 200 words the process of creating a collage and their expression of creativity incorporated in the collage they have created.</p> <p><b>Output</b></p> 	<p>Demonstration Time: 15 minutes Practice Time: 30 minutes</p>

## 2.2 Resources

### Videos:

Lines: <https://www.youtube.com/watch?v=edN07wcbj2w>

Arcs & Angles: <https://www.youtube.com/watch?v=xUTPb0ozy8M>

### Useful links:

<http://gym.pencilcode.net>

Tutorial of angles: <http://pencilcode.net/material/measuring.pdf>

Tutorial of arcs: <http://pencilcode.net/material/arcs.pdf>

Book: [book.pencilcode.net](http://book.pencilcode.net)

### Additional exercises:

Exercises – Add turtle Tail to turtle

Understand the use of 'Move' by making this stick figure:

[http://activity.pencilcode.net/home/worksheet/stick\\_figure.html](http://activity.pencilcode.net/home/worksheet/stick_figure.html)

## Chapter 3: Input / Output

### 3.0.1 Objective

Modern computers use a rich variety of forms of input and output. In this unit, students will explore output of images, text, speech, and music, and they will explore input of mouse clicks, buttons, text, voice, and keypresses. Although programs that combine input and output can be created with just a few lines of code, these simple programs can be among the most interesting and engaging for students. Any form of input can be attached to any form of output, so creating connections between input and output provides a large range of creative possibility.

### 3.0.2 Topic Outline

- 3.0 Chapter Introduction
  - 3.0.1 Objectives
  - 3.0.2 Topic Outlines
  - 3.0.3 Key Terms
  - 3.0.4 Key Concepts
- 3.1 Lesson Plans
  - 3.1.1 Suggested Timeline
  - 3.1.2 CSTA Standards
  - 3.1.3 Lesson Plan I on using the Text and Sound block
  - 3.1.4 Lesson Plan II on using the Button block
  - 3.1.5 Lesson Plan III on using the Click block and the /img bot
  - 3.1.6 Lesson Plan IV on in class lab activity.
- 3.2 Resources
  - 3.2.1 Important links

### 3.0.3 Key terms

Input	Output
Human Computer Interaction (HCI)	Event Object
Event Handler	Event Binding Function
Say, Play	Await
Spiral assignment	Assignment statements
Variables	

### 3.0.4 Key Concepts

Computers are most interesting when used to interact with the world.

- **Input** brings data into the computer, e.g., when you type on a keyboard.
- **Output** sends data out of the computer, e.g., when you see things on the screen.

Together, input and output are sometimes called I/O. There are many types of I/O including human interfaces, network interfaces, storage interfaces, and robotic interfaces. There is a lot of commonality in how a computer program deals with all these types of input and output, regardless of whether the interaction is with a person, a file, or some other device. User can learn important I/O techniques just by learning how to create user interfaces.



## Common Forms of Human Computer Interaction

This section focuses on **human-computer interaction (HCI)**. When creating a user interface in a Web application, programmers deal mainly with keyboard and mouse input, and with screen and audio output. Here are some examples:

	Input	Output
Graphical	Mouse, keypress	Graphics
Text-Oriented	Keyboard input	HTML
Audio	Microphone	Music, Speech

## An Overview of I/O Concepts to Teach

Introduction to input and output:

- Output of graphical images, as seen in Chapter 2
- Simple input of mouse clicks
- Combining input and output

Expanding to different types of input and output:

- How to output text
- How to output speech, and music
- How to input from keys and buttons
- How to input text and speech

Special input strategies:

- Using CoffeeScript “await” to wait for input
- Using “pressed” to poll for input

## Events for Mouse Click Input

In a graphical environment, the simplest way to collect input is to listen to **events**. An event is an object created by the system that represents a single unit of input. For example, every time the mouse is clicked, an event object is created representing the click. The event object has properties representing details of the input such as the position, time, and which mouse button was clicked.

Event e

e.type = 'click'	The kind of event
e.x = -195	X position of click
e.y = 40	Y position of click
e.which = 1	Which mouse button
e.timestamp = 1454775914487	Number representing the time of the click.



*An event object has properties representing details of the input, such as its position and time, and which button was used. Not all properties of the click event are shown here.*

*Clicking the mouse creates an invisible event object.*

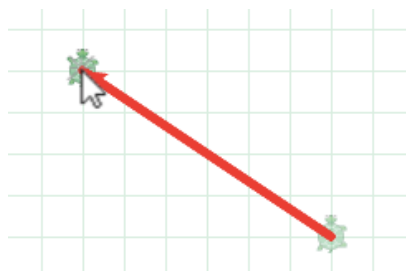
Event objects are created by the system whenever user input occurs. A program can respond to events by creating event handlers, explained next.

## Creating Event Handlers

An event handler is a piece of code that runs to process an input event. It looks something like this:

```
click (e) ->
  moveto e.x, e.y
```

*An event handler to process a click event.  
Each time a mouse click occurs, the handler runs  
and moves the turtle to the location of the click.*



There are three key parts of the code in the set up for an event handler.

The (e) is the **event object** parameter. When the input happens, the event object (containing the location of the click on the screen) is made available in the variable e. The variable name can be chosen to be any convenient name. It is conventional to use the name “e”, or “event” for an event object.

All together, the (e) -> moveto e.x, e.y is the **event handler** function, which is the code to run when the event happens. Any number of lines of code can be indented after the arrow, and they will all be part of the same event handler. (An event handler happens to be a **function**, which we will talk more about in Chapter 5.)

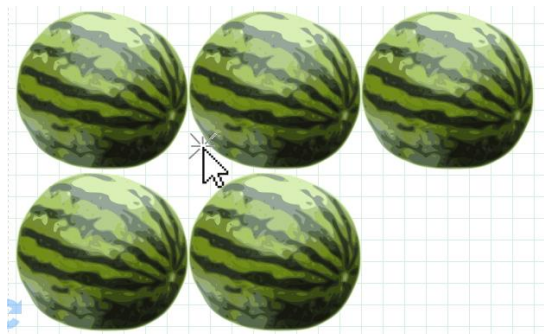
The click command is an **event binding function** that means “listen to clicks”. It is a command that connects the event handler to the system so that the handler is triggered when there is a click.

## Combining Input and Output

The magic of input and output lies in creating new effects by combining them. For example, a new image can be created for each click with this:

```
click (e) ->
  img 't-watermelon'
```

*Combining input and output by creating  
a new image within a click event handler.  
A watermelon is drawn for each click.*



As students learn different types of input and output, it is helpful to have them try combining input and output in different ways. Have students try the following:

<pre>click (e) -&gt;   w = img 't-watermelon'   w.moveto e.x, e.y</pre>	In addition to making an image, move it to the clicked location.
<pre>w = img 't-watermelon' click (e) -&gt;   w.moveto e.x, e.y</pre>	Move only a single watermelon image instead of making a new one for each click.
<pre>pen purple click (e) -&gt;</pre>	Draw a line between clicks.

```
moveto e.x, e.y
```

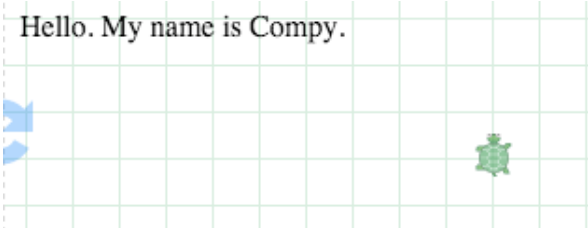
Students can create a simple drawing program using just click events. They can do even more if they combine different kinds of input and output.

### Output of Text

To write text output on the screen, use the write command like this:

```
write 'Hello. My name is Compy.'
```

*Writing a line of text output.  
The write command writes text from top to bottom, not at the the turtle.*



When text is written to the screen using “write”, it appears from top to bottom on the screen, under all the written text so far (not, for example, where the turtle is). The “img” command also puts new images at the end of all written text and images so far.

Just like img, it is possible to use a variable to remember a text object and move it on the screen:

```
t = write 'Hello'  
t.moveto 50, 100  
t.rt 180  
t.grow 2
```

*By using a variable with written text,  
text can be moved, turned, and grown.*



To create text on the screen at the location of the turtle, the “label” command can be used:

```
label 'Turtle was here'
```

*The label command makes text  
at the location of the turtle.*



Labels can also be moved in the same way as text with write and images, by using a variable.

### Output of Speech and Music

The screen is not the only output device on a computer! The computer can also output using audio. There are two interesting ways to do this: using speech or using tones.

The say command utters speech audibly.

```
say 'Hello. My name is compy.'
```

*The say command utters speech aloud.*



To hear a program that uses speech, the browser needs to support speech synthesis. Chrome, Safari, and Opera do, and browser support for speech standards may increase over time. The webpage <http://caniuse.com/#feat=speech-synthesis> lists current browser versions that support speech.

The `play` command plays a song using ABC notation, which represents each musical note with the letter that musicians use for the note.

```
play 'EDCDEEEzDDDzEEE'
```

*The play command uses ABC notation to play musical notes.*



In ABC notation, the letters A-G are used for notes. Uppercase is an octave higher, and the letter z rests silently for a beat.

There are many other things that can be done with ABC notation (which you can read about by searching for “ABC notation” on Google). For example, put a number “2” or fraction “1/2” after a note to change the number of beats of that note, or put a “^” or “\_” before a note to make it “sharp” or “flat”, or a comma after a note to make it an octave lower.

The `play` command will sequence notes and wait its turn before beginning a song, but sometimes in an interactive program, it is useful to play a note right away (without sequencing). To play a tone right away without sequencing, use the “`tone`” command:

```
tone 'C'
```

*Use tone to make a sound immediately.*



Together, these are all the tools needed to make the computer say something or play a song or a tone or write or utter a word when you click the mouse. Have students experiment with the different types of output to create different types of interactions. Students should experiment to understand the difference in timing between using “`play`” and “`tone`” when responding to multiple mouse clicks.

## Input from Keyboard and Buttons

The mouse button is only one of the buttons a computer has: a typical computer will have another 100 or so buttons on a keyboard!

An event handler can be used to collect input from those buttons using two other event binding functions: `keydown` and `keyup`.

```
keydown 'A', ->  
tone 'C'
```

*Pressing the A on the keyboard sounds a C.*



The program above will sound the C tone whenever the user presses down on the “A” key.

The comma after the key name is necessary. The comma is used because `keydown` is an event binding function that is using two arguments instead of one: the first argument is the name of the key, and the second argument is the event handler. Like any other command with two arguments, a comma must be used between the arguments.

There is also a `keyup` event binding function. For example, to silence the C note when the user lets go of the key, use this trick for sounding a zero-duration C when you release the A key:

```
keydown 'A', ->
  tone 'C', 0
```

*Releasing the A on the keyboard silences the C.*



Many keydown and keyup event handlers can be combined create a whole piano or to create other effects. For example, it is possible to create event handlers to attach turtle movements to specific keys and make a system for steering the turtle around. Notice that the letter keys have obvious names, but there are also names for the arrow keys: you can listen to the “up” arrow by saying keydown 'up', and similarly for “down”, “left”, and “right”.

```
keydown 'up', ->
  fd 100
```

*The up arrow key moves the turtle forward.*



An alternative to using physical keyboard keys is to use on-screen buttons. The “button” command is used for this:

```
button 'forward', ->
  fd 100
```

*An on-screen button labelled “forward” moves the turtle forward.*



The advantage of on-screen buttons is that the user can see exactly what controls are available. With good labels, they are self-explanatory. The disadvantage is that they take space on the screen.

## Input of Text and Speech

When collecting text input from a user, listening to a single keypress at a time can be done, but it is very inconvenient! That is why user interfaces use text input elements for entering text. The input element is a box that shows text, and when it has focus, all keypresses automatically turn into text in the box.

To use a text input box in Pencil Code, use the read command, like this:

```
read 'Your name?', (n) ->
  write 'Hello, ' + n
```

*The handler is triggered after text is entered and submitted.*

Your name?

As with click or button or keydown, the read command calls an event handler after the user has finished providing input. There are a few differences between read and click:

- Instead of waiting for a single small action, read waits for a whole series of keystrokes and then finishes when the user presses “Enter” (or clicks submit).
- The event handler for read is called just once. After the input, the input box goes away.
- Instead of binding the variable to an event object that has properties like x and y, read sets it to the text value that was input (n in the example above).

To enter a number, consider the special variant `readnum`, which constrains the input to just digits.

```
readnum 'Your age?', (n) ->
write 'Next year you will be ' + (n + 1)
```

Your age?

*readnum constrains input to a number.*

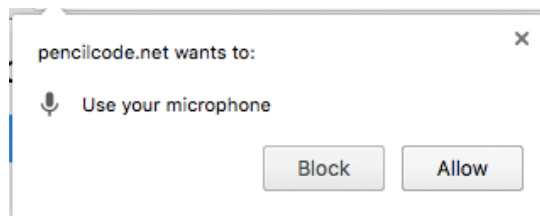
A keyboard is not the only way to enter text. Another option is to use voice input, which can be done using “listen”. That function works just like “read”:

```
listen 'Say something', (t) ->
say 'You said: ' + t
```

Say something

*listen accepts spoken voice input.*

A few tips for working with voice: Currently voice recognition and speech synthesis work only on Chrome. Before a webpage attempts turns on the microphone on Chrome, it must obtain the user’s permission. If using the https (secure) version of Pencil Code, Chrome will remember the permission after it is given first time so it does not need to ask every time.



*When a page listens to the microphone, the browser asks for permission.  
If the page is loaded over https, the permission is remembered.*

## Using CoffeeScript `await` to Wait for Input

Sometimes a program needs to read a sequence of inputs. To do this, chain the event binding functions inside one another like this:

```
readnum 'Right triangle side a?', (a) ->
readnum 'Right triangle side b?', (b) ->
c = sqrt(a*a + b*b)
write 'The hypotenuse is', c
```

Right triangle side a? 12  
Right triangle side b? 5  
The hypotenuse is 13

*Using a sequence of input by  
chaining event handlers.*

This nesting makes the program look more complicated than it is, and make it difficult to use a loop.

The version of CoffeeScript used in Pencil Code has a pair of keywords “`await`” and “`defer`” that can help in this situation by putting a program on hold while waiting for an event to occur. You put the word “`await`” before the command that you want to pause, and “`defer`” in the place of an event handler along with any variables that would have been event handler parameters:

```
await readnum 'Right triangle side a?', defer a
await readnum 'Right triangle side b?', defer b
c = sqrt(a*a + b*b)
write 'The hypotenuse is', c
```

Await and defer have a subtle relationship with function calls, so before putting await inside a user-defined function, find understand the Web pages about “Iced CofeeScript” (if using await inside a function, that function should also return its results using callbacks).

However, await is very straightforward and useful when used together with loops. Here is an example:

```
await readnum 'How many numbers to average?', defer count
total = 0
for j in [1..count]
  await readnum 'Enter #' + j, defer val
  total += val
write 'The total is ' + total
write 'The average is ' + (total / count)
```

This style of code is called “blocking i/o”, because the program blocks (stops) its progress while waiting for an input or output to occur. Blocking i/o is the traditional way to teach Python or C input/output, but it is very different from the way UI events are typically handled in JavaScript or Java GUIs, where input is done using event handlers. Iced CoffeeScript’s await allows teaching both styles in the same system, and even in the same program.

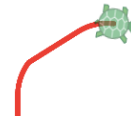
The await keyword is so useful that a version of it is on track to be added to a future version of the JavaScript standard. However, it is not in the language right now, so you cannot use await in JavaScript today. Instead, you must use function definitions (see Chapter 5) to achieve similar effects.

## Polling Keyboard State Using pressed

So far we have seen two styles of input: “event handling”, and “blocking i/o.” A third style of input, called **polling**, is often used in video games and real-time systems such as robots and you can also try it with Pencil Code. A program using polling repeatedly checks the input state (of the keyboard) by asking a question such as “is the key pressed down right now?”

Here is how Pencil Code does polling (usually in combination with the “forever” command).

```
forever ->
  if pressed 'W'
    fd 2
  if pressed 'D'
    rt 2
```



*Inside a forever block, the function pressed can be used to poll the keyboard state.*

The “pressed” command is the polling command. It is true if a key is pressed and it is false if the key is not pressed. The “if” can be used decide whether to take an action based on the state of a key. With “pressed,” it is even possible to support “chording”, that is, making a program that responds to two keys pressed at the same time. Students can experiment with this effect in the program above.

Polling is an advanced technique and there are several subtleties for getting it to work correctly that are handled by the “forever” command. A “forever” loop differs from a traditional loop in several ways.

Inside a “forever” loop, the speed of turtles is automatically set to Infinity to avoid animation delays. Also, a “forever” loop will also automatically put a short delay between each repetition so that you can see the effects of the repetition over time. You can change the framerate of the “forever” loop by adding an extra number argument. For example, “forever 10” will do 10 frames per second.

```

forever 10, ->
  if pressed 'space'
    fd 1
    rt 1

```



*The frequency of a forever block, adjusted to 10 repetitions per second.*

## Combining Ideas

This unit on input and output covers a lot of powerful concepts, but the real power comes from finding creative new ways to combine input and output. By combining graphics, text, and audio, students can create applications such as calculators, games, conversational assistants, interactive drawing programs, or musical instruments.

Each application may require a particular i/o model. The most common models are: event-based i/o, blocking i/o, and polling. Sometimes the same application can be built in a different way using a different i/o model so it is worth having students experiment with more than one model to learn how they work.

### 3.1.1 Suggested Timeline: 1 55-minute class period

Instructional Day	Topic
1 Days	Lesson Plan I: Text and Sound Blocks
1 Day	Lesson Plan II: Use of Buttons and Click (e) controls to show input
1 Day	Lesson Plan III- Demonstrate the use of the /img bot
2 Day	Lesson Plan IV Lab Activity – choose between a shape bot or paint bot

### 3.1.2 Standards

CSTA Standards	CSTA Strand	CSTA Learning Objectives Covered
Level 3 A (Grades 9 – 12)	Computational Thinking (CT)	Describe how computation shares features with art and music by translating human intention into an artifact.
Level 3 A (Grades 9 – 12)	Computers and Communication Devices (CD)	Describe the principal components of computer organization (e.g., input, output, processing, and storage).
Level 3 A (Grades 9 – 12)	CD	Compare various forms of input and output.

### 3.1.3 Lesson Plan I

This lesson focuses on using the Text, Sound and Control block palettes. Click on the Text, Sound and Operators block to show students that input/output statement commands are located under these palettes. Read and type the code as shown below and demonstrate the output to the students.

Note: Make sure you are in block mode. Type in the code (switch to block-mode if needed) and click the play arrow to demonstrate the results.

Content details	Teaching Suggestions	Time
Demonstrate write and say (Text & Sound block). <pre> Write Say write 'Hi' write 'Hello.' write 'Can you say hello world?' </pre>	These are the output statements. 	Demonstration 10 minutes



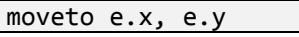



Content details	Teaching Suggestions	Time
<pre>say 'Hello World!'</pre>	Type in the code and click the play arrow to demonstrate the results.	
<p>Displaying expressions</p> <pre>name = 'David Bau' write 'Good to meet you ' + name</pre> <p>Output</p> <pre>Good to meet you David Bau!</pre>	Show how write can show the value of a variable or an expression.	Demonstration 10 minutes
<p>The question bot is a simple program that asks questions and displays responses in an intelligent manner.</p> <p>Code:</p> <pre># questionBot # short interview with await..defer  await read 'What is your name?', defer name await read ('What is your favorite food, ' + name) + '?', defer food await read ("Sounds tasty. What's so good about " + food) + ', ' + name + '?', defer response write 'Fair. I might just go try me some ' + food + 'now. Nice chat!'</pre> <p>Output</p> <pre>What is your name? PencilCode What is your favorite food, PencilCode? Lead Sounds tasty. What's so good about Lead, PencilCode? It is sturdy Fair. I might just go try me some Lead now. Nice chat!</pre>	<p>Input Statements: Demonstrate Await - Read, using Question Bot (Text Block)</p> <p>Await waits till an input is received. It then stores the input to the variable declared next to defer.</p>	Demonstration 15 minutes
<p>Code:</p> <pre># Question Bot using numbers write 'Hello. Can you tell me your name please?' await read 'Your name?', defer name write 'Hi ' + name await readnum 'Can you tell me your age, ' + name, defer age write 'Hi ' + (name + ('. I have noted your age ' + age))</pre>	<p>Demonstrate Await – ReadNum using Question Bot (Text Block).</p> <p>Output</p> <pre>Hello. Can you tell me your name Your name? QBot Hi QBot Can you tell me your age, QBot 2 Hi QBot. I have noted your age 2</pre>	Demonstration 15 minutes
<p>Students can now work on their version of Question Bot.</p>	<p>Encourage students to express their own individuality and creativity and experiment with using “Say” in places where “Write” is used. What happens?</p>	Student Practice 15 minutes
<p>Look at exercises</p>	<p>Using the Art, Move, Text and Sound block</p>	Student Practice 15 minutes

### 3.1.4 Lesson Plan II

This lesson discusses the use of Buttons: the use of button clicks as input.

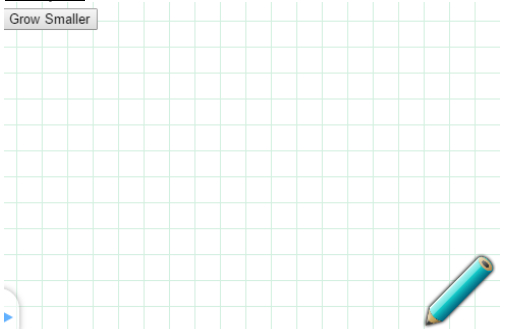
Note: Make sure you are in block mode. Type in the code (switch to block-mode if needed) and click the play arrow to demonstrate the results.

Content details	Teaching Suggestions	Time
<p>Code</p> <pre>button 'Press to see a BullsEye', -&gt;   x = 18   dot black, x * 5   dot red, x * 4   dot black, x * 3   dot orange, x * 2</pre>	<p>Demonstrate <a href="#">Button</a> ('Click')</p> <p>The Button option lets the user label the button and runs the code that is within the block.</p> <p>Output</p> 	<p>Demonstration 20 minutes</p>
<p>Code:</p> <pre>keydown 'a', -&gt;   x = 18   dot black, x * 5   dot red, x * 4   dot black, x * 3   dot orange, x * 2</pre>	<p>Demonstrate <a href="#">Keydown</a></p> <p>The Keydown waits for the 'a' key pressed to execute the code within the block.</p> <p>Output:</p> 	
<p>Code:</p> <pre>click (e) -&gt;   moveto e.x, e.y   x = 18   dot black, x * 5   dot red, x * 4   dot black, x * 3   dot orange, x * 2</pre>	<p>Demonstrate <a href="#">Click</a></p> <p>The click will wait for a mouse click and then execute the code within the block. The e variable represents the click event, so</p>  <p>moves to the location of the click.</p> <p>Output</p> 	
<p>Finish the lab exercise that was started the previous class period..</p>		<p>Student Practice: Use the remainder of class period and homework if needed.</p>

### 3.1.5 Lesson Plan III

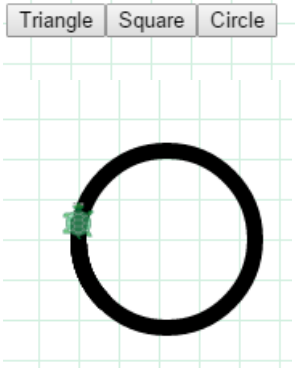
This lesson plan introduces the Buttons and the Click (e) capability along with wear and img blocks which display images from the internet. The wear and img blocks are available under the Art panel.

Note: Make sure you are in block mode. Type in the code (switch to block-mode if needed) and click the play arrow to demonstrate the results.

Content details	Teaching Suggestions	Time
<p><b>Code:</b></p> <pre>wear 't-pencil' button 'Grow Smaller', -&gt;   jumpxy 30, 20   grow 0.5 button 'Grow Larger', -&gt;   grow 2.0</pre> <p><b>Output</b></p> 	<p>Demonstrate how the wear block works. Open the <a href="#">ImgBot</a> program.</p> <p>Point out the use of Button and Click (e) from the previous lesson plans.</p> <p>Explain how wear and img work (refer to key concepts if necessary). Substitute other values for pencil and show the kinds of images that result.</p> <p>Encourage students to play with the wear and grow blocks.</p>	<p>Demonstration: 20 minutes</p> <p>Student activity: 25 minutes.</p>

### 3.1.6 Lesson Plan IV

This lesson plan provides instructions for designing the Shape Bot. Students Design a simple program that draws geometric shapes such as a square, triangle, circle, etc. The program first asks the user for a shape. It asks from the user to provide details such the number of sides, length of sides, and the radius of the circle, etc.

Content details	Teaching Suggestions	Time
<p>Code:</p> <pre> speed 100 pen black, 10 button 'Triangle', -&gt;   await read 'How long are the sides?', defer side   cs()   fd side   rt 120   fd side   rt 120   fd side   rt 120 button 'Square', -&gt;   cs()   await read 'How long are the sides?', defer side   fd side   rt 90   fd side   rt 90   fd side   rt 90   fd side   rt 90 button 'Circle', -&gt;   cs()   await read 'How long is the radius', defer radius   rt 360, radius   fd 10 </pre>	<p>Give the lab program to the students. Encourage them to experiment and improve the design of the program. After students have worked on it pull up the <a href="#">Shapes Bot</a> program and start walking the students through the program. Encourage students to come up and demonstrate their work.</p> <p><u>Output</u></p> 	<p>Student activity 55 minutes.</p> <p>Demonstration 20 minutes</p>
<p>Encourage students to explore and understand their inclinations and strengths in programming by giving various assignments to accomplish the same purpose. For example, consider a simulation of paint splatter drawn as a collection of colored dots. This can be called the Paint Splatter Bot.</p>		

### 3.2 Resources

**Important Links:**

<http://gym.pencilcode.net>

Book: [book.pencilcode.net](http://book.pencilcode.net)

# Chapter 4: Loops

## 4.0.1 Objectives

Repetition is a fundamental programming tool. This chapter introduces three types of loops, which are the basic code building blocks used to repeat actions in a program. At the end of this unit, students should be able to reason about the number of repetitions and terminating conditions of a loop, and they should be able to apply for, while, and forever loops in their programs.

## 4.0.2 Topic Outline

- 4.0 Chapter Introduction
  - 4.0.1 Objectives
  - 4.0.2 Topic Outlines
  - 4.0.3 Key Terms
  - 4.0.4 Key Concepts
- 4.1 Lesson Plans
  - 4.1.1 Suggested Timeline
  - 4.1.2 CSTA Standards
  - 4.1.3 Lesson Plan I on using While and For Loop in the Control block
  - 4.1.4 Lesson Plan II on using the ability to switch between text-mode and block-mode.
  - 4.1.5 Lesson Plan III on using the 'for... each' loop
  - 4.1.6 Lesson Plan IV on creation of the Question Bot and the idea of a spiral assignment.
  - 4.1.7 Lesson Plan V on using a 'Forever' loop
- 4.2 Resources
  - 4.2.1 Lab extension

## 4.0.3 Key Terms

Control statements: repetition and iteration	Fixed and variable repetitive statements
Terminating condition	Increments
Infinite loops	For loop
While loop	Start value or beginning condition

## 4.0.4 Key Concepts

**Iteration** allows a short program to represent a long series of steps by including repeated sequences.

A part of a program that repeats commands is called a **loop**.

Every loop has two parts:

- The **condition** that controls how many times to repeat the loop.
- The **body**, which is a block of code that is repeated as long as the loop is running.

A loop that never stops repeating is called an **infinite loop**. An infinite loop will prevent a program from ever finishing, so usually a program with an infinite loop is not desirable. Within a browser, an infinite loop will even prevent a program from ever responding to mouse clicks, so Pencil Code will try to detect and interrupt programs with infinite loops.

To avoid an infinite loop, the condition that controls how many times the loop is repeated must be written correctly. There are two main ways to make a looping condition in CoffeeScript:

- `for` repeats a block of code one for each item in an **iterated list**, and
- `while` repeats a block of code as long as the **loop condition** remains true.

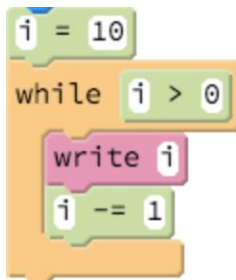
## Using Loop Blocks

The loop structure is located under the Control palette in the block. There are three types of loop structures available:

`for` loop: Loops over a set of program statements for a fixed number of times in fixed increments. The following loop writes the word "Hello" three times.

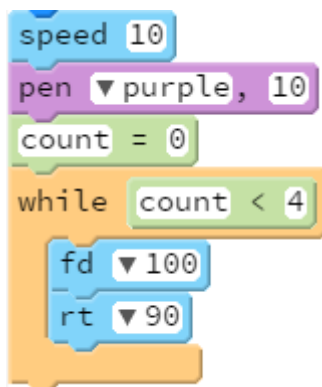


`while`: Loops over a set of program statements as long as the evaluating condition returns true. The actual number of iterations is not known until execution time.

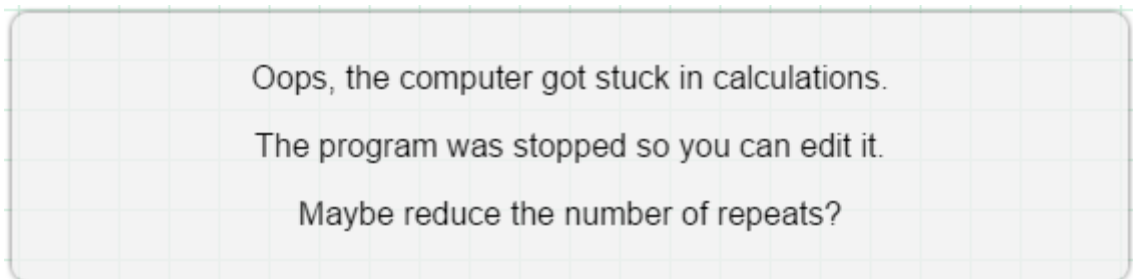


It is possible that, due to a logical error, the loop may be set to iterate through the block of code an infinite number of times. This is very common with beginning programmers. Pencil Code prevents this by generating an error and halting program execution. Internally, Pencil Code keeps a timer when inside a loop. If a program remains stuck inside the loop for several seconds without processing input, Pencil Code assumes the loop is stuck and interrupts execution.

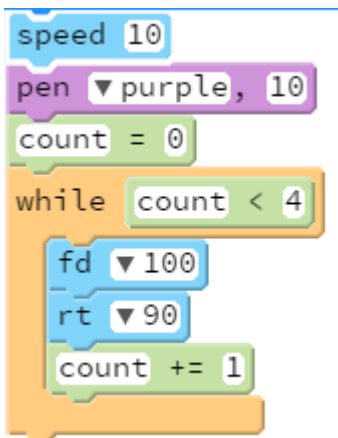
Please see below for an example:



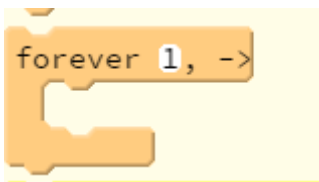
Since count will never reach 4, Pencil Code generates an error after drawing the square.



The fixed code looks like this:



forever: An ordinary while loop that loops forever will cause the browser to **hang**, which means it freezes up the page so the user is unable to interact with it. To repeat a process forever, a program can use a forever block which implements a loop by pausing briefly after every iteration to allow the browser to process input. This pause makes the forever loop different from the for and while loops above. The forever loop can be used to repeat something indefinitely without freezing the entire browser. The number after forever controls the length of the pause: it is the number of times to repeat every second.



## Using for Loops in Text

If students have not yet experimented with writing programs in text, this section is an excellent time to suggest that they try out text coding. The examples below will all be shown in text. Several of the ideas in loops are more convenient to work with in text-mode than in block-mode. For example, it is easier to switch a for loop into a while loop or add or remove an iterator variable in a for loop in text-mode than in block-mode.

There are several kinds of for loops, but they all work in the same general way, this is, each kind of for loop repeats code once for each item in an iterated list. It is easiest to switch between these forms of the for loop in text-mode.

<pre>for [1..3]   write 'hello'</pre>	<pre>for x in [0...3]   write 'hello', x</pre>	<pre>for x in ['alice', 'bob', 'carol']   write 'hello', x</pre>
hello hello hello	hello 0 hello 1 hello 2	hello alice hello bob hello carol

All of these loops repeat their bodies three times because each of them iterates over a list with three elements. The loops that use a variable “for x in” assign successive values for x each time around the loop, and the loop body can use that variable to create slightly different output each time.

Here is some information about list notation. (Chapter 8 discusses arrays in greater detail.) A CoffeeScript range array is enclosed in brackets, and [1..3] with two dots between integers indicating that the list of integers starts at 1 and ends at (and including) 3.

To indicate a list that does not include the last item, use three dots [0...3]. This three-dot form is particularly useful because (unlike the two-dot form) it can express an empty array such as [3...3] (a list with no items). An array can also be written explicitly by separating elements using commas or by writing each element on its own line.

Loops using for terminate automatically when they get to the end of their list, so it is more difficult to make the mistake of creating an infinite loop using a CoffeeScript for. The only way to do so is to create an infinite list to iterate!

## Using while loops

A loop made with while checks a loop condition. If the loop condition is true, the loop runs the code in the body of the loop then repeats the process, doing another check of the loop condition at the start of each repetition. Since the loop condition must be false for the loop to terminate, it is very easy to mistakenly create an infinite loop by writing a condition that never becomes false.

<pre>j = 0 while j &lt; 3   j += 1   write 'checking', j write 'finished'</pre>	<pre>roll = -1 while roll isnt 1   roll = random(6)   wrote 'got', j write 'finished'</pre>	<pre>countdown = 3.5 while countdown isnt 0   countdown -= 1   write countdown write 'finished'</pre>
checking 1 checking 2 checking 3 finished	got 4 got 0 got 5 got 5 got 1 finished	2.5 1.5 0.5 -0.5 -1.5 .... (an infinite list of numbers) Oops!

The first two examples terminate, and the third is an example of a buggy program with an infinite loop. In all these cases, the programmer has included three things in the looping program:

- A clear **starting state** (for example, `j = 0`, `flip = -1`, or `countdown = 3.5`).
- A **loop condition** that indicates that the loop should continue repeating. In the programs above, the loop condition repeats the loop when `j < 3`, `flip isnt 1`, or `j isnt 0`.
- A **state change** that eventually leads to the loop condition becoming false. In the programs above, the programs change `j`, `flip`, and `countdown`.



All these programs proceed using **variable assignments** that involve single-equals-assignment operators.

- An assignment such as `j += 1` increases the value of `j` by one.
- An assignment such as `roll = -1` or `roll = random(6)` replaces the value of the variable `roll`.
- An assignment such as `countdown -= 1` decreases the value of `countdown` by one. Notice that the minus before the equals sign means “subtract from this variable and set the value.” It is equivalent to `countdown = countdown - 1`.

In these programs, the successive variable assignments change the variables involved in the loop condition. In the first two programs, the assignments eventually lead to the loop condition being false. However, in the third program, the assignments decrease a non-integer countdown by one and they never hit exactly zero, so the loop condition never becomes false. You could fix this loop by changing the loop condition, the starting state, or the decrease amount.

## Choosing Between Text-Mode and Block-Mode

In Pencil Code, block code and text code are perfectly equivalent (anything that can be done in one representation of the code can be done in the other) and you can switch between the two modes at any time. The choice between working in text or blocks is a matter of personal productivity.

Blocks are particularly helpful for those who are new to a language. Blocks facilitate the use of correct syntax and help programmers recognize patterns of allowable code. However, using blocks makes it difficult to find choices that are not provided on the palette.

Programming text lets programmers enter a program as fast as they can type. They are not limited by the choices on a palette. Programming in text, however, requires the programmer to remember and follow the syntax rules and it is very easy to create text programs that do not run due to syntax errors.

Although block languages have been improving, professional programmers still write their programs using text because one they are familiar with the programming language, they can work more quickly with text than with blocks.

## Finding and Fixing Syntax Errors

Writing text programs is a skill that takes some time to develop. There are two strategies for learning syntax:

1. Observe and copy examples of correct syntax. In Pencil Code, block is useful for this. If students do not recall how a specific syntax works, they can always flip to block to try out something, and then switch back to text once they are confident.
2. Pay attention to syntax errors reported by the computer and fix them right away, before too many syntax errors are created. In Pencil Code, syntax errors are highlighted in text mode with an “x” to the left of the code, like this:

```
1 write 'Hello!'
x2 write 'Can't get this to work!'
3 write 'Can you?'
```

The red “x” will sometimes appear when there is an unfinished line of code. However, if the line of code is finished, students should pay attention to the red “x” and try to fix it right away. It is much easier to fix a single syntax error in a program than fixing a program with many syntax errors.

This code snippet is an example of “mismatched quotes”. Block mode automatically corrects bare apostrophes by prefixing them with “\” to tell Pencil Code that to include an apostrophe and not to end the string. In text mode, the programmer needs to type this backslash before the apostrophe explicitly.

Because, by definition, a syntax error is a part of the program that the computer does not understand, the computer sometimes puts the “x” in the wrong place. The error in the program might not be spotted until more lines of code are added. The error may be on a line above the “x” and it might have been caused by a problem other than the one the computer indicates.

## Common Syntax Errors

Here are some examples of common syntax errors in CoffeeScript. One way to learn the syntax of a programming language is to do a bit of practice fixing examples of syntax errors like this.

Explanation	Example with an error	Fixed example
Mismatched quotes	<code>write('can't get this to work')</code>	<code>write('can\'t get this to work')</code>
Using word-processor-style “smart quotes”	<code>write('quotes must be straight')</code>	<code>write('quotes must be straight')</code>
Mismatched parentheses	<code>write(1+(2*(x - 1))</code>	<code>write(1+(2*(x - 1)))</code>
Missing comma between arguments	<code>moveto 100 200</code> ↑	<code>moveto 100, 200</code>
Misaligned indenting creating incorrect scoping	<code>if pressed('X')</code> <code>fd 100</code> <code>rt 90</code> <code>^^^^</code>	<code>if pressed('X')</code> <code>fd 100</code> <code>rt 90</code>
Mixing up different brackets	<code>for x in (1..10)</code> <code>write x</code>	<code>for x in [1..10]</code> <code>write x</code>

Programming language syntax is very sensitive to punctuation and, in some cases, spacing. People are usually good at spotting a missing word or a misspelling but it takes practice to pay attention to the punctuation and spaces in a program.

## Common Runtime Errors

Here are examples of the most common runtime errors. A program that uses the language syntax correctly may still have runtime errors by referring to a name, variable, or function that has not been defined by the time it is run.

Explanation	Example with an error	Fixed example
Missing quotes	<code>write hello</code>	<code>write 'hello'</code>
Variable used before definition	<code>write x * 7</code> <code>x = 10</code>	<code>x = 10</code> <code>write x * 7</code>
Misspelled function name	<code>wrte 'hello'</code>	<code>write 'hello'</code>

### 4.1.1 Suggested Timeline: 1 55-minute class period

Instructional Day	Topic
2 Days	Lesson Plan I: For Loops and descending for loops. Practice with for loops
1 Day	Lesson Plan II: Question Bot
1 Day	Lesson Plan III & IV: While loops and forever loops. Practice with while loops
1 Day	Lesson Plan V: Tracing a loop with values


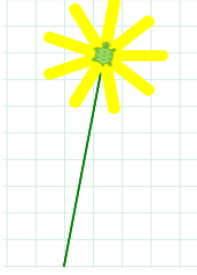
### 4.1.2 Standards

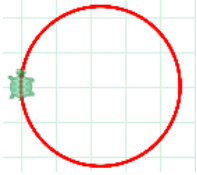

CSTA Standards	CSTA Strand	Learning Objectives Covered
Level 3 A (Grades 9 – 12)	Computational Thinking (CT)	Explain how sequencing, selection, iteration and recursion are building blocks of algorithms.
Level 3 A (Grades 9 – 12)	Computing Practice & Programming (CPP)	Use Application Program Interfaces (APIs) and libraries to facilitate programming solutions.

### 4.1.3 Lesson Plan I

This lesson introduces the 'For Loop' using CoffeeScript. It includes both the ascending and descending variants.

Note: Make sure you are in block mode. Type in the code (switch to block-mode if needed) and click the play arrow to demonstrate the results.

Content details	Teaching Suggestions	Time
<p>Dandelion: Code</p> <pre>speed 20 pen green rt 10 fd 200 pen yellow, 10 for x in [1..9]   fd 50   bk 50   rt 360 / 9</pre> <p>Gold Star</p>  <p>Loop 360</p>	<p>Demonstrate for and while loop (Control panel). Type the code as shown and click play to generate the dandelion. Point to the students how x takes values from 1 to 9 to draw 9 petals. Explain that rt moves the turtle by an angle for every iteration. Give students a chance to try this out.</p> <p><u>Output</u></p>  <p>You could also have the students experiment with Gold Star and Loop 360 as they are good examples for reinforcing the same concept. Click on the link below for the source code for these two examples.</p> <p><a href="http://guide.pencilcode.net/edit/loops/">http://guide.pencilcode.net/edit/loops/</a></p>	<p>Demonstration: 20 minutes</p> <p>Student Practice:</p>

Content details	Teaching Suggestions	Time
		30 minutes
<p><u>Spiral: Code</u></p> <pre>pen purple, 10 for x in [50...1]by -1   rt 30, x</pre> <p><u>Output</u></p> 	<p>Once students understand the for loop ascending, introduce the for loop variant for descending loops. The loop decreases from maximum value to minimum value in regular negative intervals. For example:</p> <pre>for x in [50..1] by -1</pre> <p>Explain how the angle of 50 is decremented by 1 for every iteration of the loop. Click on the link below for the source code for this example.</p> <p><a href="http://teachersguide.pencilcode.net/edit/chapter4/spiral">http://teachersguide.pencilcode.net/edit/chapter4/spiral</a></p>	Demonstration: 10 minutes

#### 4.1.4 Lesson Plan II

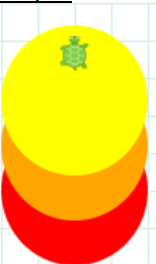
Pencil Code allows programmers to move easily between block- mode and text-mode. This lesson introduces programming using text-mode.

Content details	Teaching Suggestions	Time												
<p><u>Code:</u></p> <pre>for x in [0...3]   write 'hello', x</pre> <p><u>Output:</u> hello 0 hello 1 hello 2</p>	<p>While in the Pencil Code environment, switch from block-mode to text-mode. Type the program as shown in the left-most column.</p> <p>Show students the three dots in the loop and explain that the variable x takes on the values 0, 1, 2 and 3 and for each value of x, the word “hello” is printed and the value of x is displayed.</p>	<p>Demonstration: 20 minutes</p> <p>Student practice: 80 minutes</p>												
<table border="1"> <thead> <tr> <th>Iteration #</th> <th>Value of x</th> <th>write x</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>0</td> <td>0</td> </tr> <tr> <td>2</td> <td>1</td> <td>1</td> </tr> <tr> <td>3</td> <td>2</td> <td>2</td> </tr> </tbody> </table>	Iteration #	Value of x	write x	1	0	0	2	1	1	3	2	2	<p>Introduce the concept of tracing variables.</p> <p>Have students draw a table and trace the values x takes as shown in the left column.</p>	
Iteration #	Value of x	write x												
1	0	0												
2	1	1												
3	2	2												

Content details	Teaching Suggestions	Time																								
<p><b>Code:</b></p> <pre>count = 1 for x in [1..5]   type count + ': ' x   count += 1</pre> <p><b>Output:</b></p> <pre>1:1  2:2  3:3  4:4  5:5</pre> <table border="1"> <thead> <tr> <th>Iteration #</th> <th>Value of x</th> <th>type count</th> <th>Count +1</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>1</td> <td>1</td> <td>2</td> </tr> <tr> <td>2</td> <td>2</td> <td>2</td> <td>3</td> </tr> <tr> <td>3</td> <td>3</td> <td>3</td> <td>4</td> </tr> <tr> <td>4</td> <td>4</td> <td>4</td> <td>5</td> </tr> <tr> <td>5</td> <td>5</td> <td>5</td> <td>6</td> </tr> </tbody> </table>	Iteration #	Value of x	type count	Count +1	1	1	1	2	2	2	2	3	3	3	3	4	4	4	4	5	5	5	5	6	<p>Demonstrate a simple math operation such as</p> <pre>count = count + 1</pre> <p>then type the code as shown.</p> <p>Ask students to trace the code. The tracing values are also shown in the left-most column.</p> <p>Once the students have worked on it show the solution given here.</p>	
Iteration #	Value of x	type count	Count +1																							
1	1	1	2																							
2	2	2	3																							
3	3	3	4																							
4	4	4	5																							
5	5	5	6																							
<p>As a way of giving students additional practice, provide the following for loops problems and ask them to trace the values.</p> <pre>for x in [0..5]   type count + ': ' x   count += 1 for x in [5..1]   type count + ': ' x   count += 1 for x in [0..0]   type count + ': ' x   count += 1</pre>																										

#### 4.1.4 Lesson Plan III

This lesson introduces different for loop that iterates over a collection of data.

Content details	Teaching Suggestions	Time
<p><b>Code</b></p> <pre>for color in [red, orange, yellow]   dot color, 100   fd 30</pre> <p><b>Output</b></p> 	<p>Demonstrate this <a href="#">program</a> while explaining to that this is a different kind of loop.</p> <p>The variable 'color' goes through the collection of colors and executes the 'dot' command with that value of color.</p> <p>The execution of the code traces how 'dot' command takes various values.</p> <p>Encourage students to use variations of this loop to produce interesting output.</p>	<p>Demonstration: 15 minutes Student Practice: 20 minutes</p>


### 4.1.5 Lesson Plan IV

This lesson introduces the first large programming assignment and the idea of iterating on a program over time, in this case the Question Bot program.

Teaching Suggestions	Time
<p>Spiral Assignment: Ask students to use the Question Bot program they created in the previous chapter.</p> <p>Give the students a chance to explore and experiment. Put them in groups and challenge them to develop the most creative Question Bot they can. If they choose to add constructs that have not been covered, allow them to do this but adjust the time accordingly and be quick to move on to the next topic when your class is ready.</p>	<p>Student practice: 55 minutes</p>

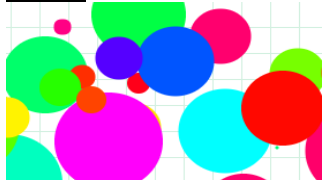
### 4.1.6 Lesson Plan V:

This lesson introduces the While Loop

Content details	Teaching Suggestions	Time
<p>Spiral using While Loop</p> <p><u>Code</u></p> <pre>pen purple, 10 x = 50 while x &gt; 0   rt 30, x   x = x-1</pre> <p><u>Output</u></p> 	<p><b>While Loops:</b> Type the code as shown for creating a while loop. Explain that a while loop is written differently than a 'for loop' but it produces the same output. This helps students understand that there are more than one solution to a problem.</p> <p>Here <a href="#">is another example</a> of while loop that uses input statements in the code and solves a problem that cannot be solved with a 'for loop'. The program waits for an input from the user to determine if the code within the loop should be executed one more time.</p>	<p>Demonstration: 25 minutes</p> <p>Students Practice: All students to us the remainder of the class period and the entire next class period.</p> <p>Note: Gauge how well the students have understood the content. If feasible, have them also complete lesson plan VI.</p>

### 4.1.7 Lesson Plan VI

This lesson introduces the 'Forever' loop[MU1].

Content details	Teaching Suggestions	Time
<p><u>Code</u></p> <pre>forever 1, -&gt;   dot (random color), random   100   moveto (random position),   random position</pre>	<p>Forever (x) loop: Demonstrate the use of this loop using <a href="#">this code</a>. (Confetti)</p> <p><u>Output</u></p>  <p>Ask students for ideas on where it would be appropriate to use the forever loop.</p>	<p>Demonstration: 20 minutes</p> <p>Students Practice: Allow the remainder of the class period and the next full class period.</p>

## 4.2 Resources

Book: [book.pencilcode.net](http://book.pencilcode.net)

# Chapter 5: Functions

## 5.0.0 Objectives

Functions are the most important concept in programming because they allow programmers to break down programs into smaller subprograms. Yet functions have another important purpose: they allow programmers to set off code to be run later, and then control when precisely when that code runs.

Students may not see the value in creating subprograms at first, since their programs are small. Therefore, we suggest starting with a focus on the use of functions to control “when code runs” by beginning with functions attached to buttons.

Named functions, function calls, and functions with parameters can be introduced next as a powerful way to generalize the idea. At the end students should learn to apply the “DRY” principle, using functions to abstract common sequences of code by creating their own commands.

## 5.0.1 Topic Outline

- 5.0 Chapter Introduction
  - 5.0.1 Objectives
  - 5.0.2 Topic Outlines
  - 5.0.3 Key Terms
  - 5.0.4 Key Concepts
- 5.1 Lesson Plans
  - 5.1.1 Suggested Timeline
  - 5.1.2 CSTA Standards
  - 5.1.3 Teaching notes
  - 5.1.4 Lesson Plan I on calling functions using buttons.
  - 5.1.5 Lesson Plan II on creating functions using buttons
  - 5.1.6 Lesson Plan III on re-using function code.
  - 5.1.7 Lesson Plan IV on passing parameters in functions.

## 5.0.2 Key Terms

Parameters	Variable
Abstraction	Program execution
Modularity	Arguments
Reusability	Event handlers
DRY: Don't Repeat Yourself	Callbacks

## 5.0.3 Key Concepts

### What Are Functions?

A **function** is a program within a program. Functions allow programmers to divide up code, just like authors use paragraphs to divide up an essay. Mathematicians also use functions to divide up formulas into simple rules for calculating values. In computer science, however, functions are used for more than just dividing a program into formulas:

1. **Functions allow reuse of code.** Once a function is defined, its code can be used many places in a program without writing the individual lines of code again.
2. **Functions control when and how code runs.** When code is put in a function, it is not run right away, but later, when and if the function is called.



The key is for understanding functions in computer code is to understand that code defined in a function does not execute immediately. Code within a function runs when the program executes a **function call**. This means that functions can be used to **reuse** the same code multiple times, and they can be used to **defer** execution of code to some future time.

### How Are Functions Written?

In CoffeeScript, a **function call** is written by putting arguments after a function name, in one of two ways:

fd 100	Use a space after the function name fd to call it, passing it the argument 100
fd(100)	Use parentheses and no space fd to call it with arguments in parentheses.

A function call may have no arguments, but then it requires parentheses, such as hide(). We have previously used many calls to built-in functions like fd and hide, but custom programmer-defined functions are called in exactly the same way.

In CoffeeScript, a **function definition** begins with an arrow -> (typed as two characters, minus-angle) pointing from parameters to the body of the function. We may put the body of the function on a series of lines after the arrow if the lines are indented.

(x) -> x * x	An unnamed function that takes any value x and returns x * x.
sq = (x) -> return x * x	The same function, this time named "sq", and typed differently using indenting and "return".
exclaim = -> write 'hey!' write 'yo!'	A function named "exclaim" that has no parameters and writes two messages to the screen.

### How Can a Function Compute the Answer to a Question?

Functions are used by combining function definitions with function calls. A function can compute the answer to a question in CoffeeScript like this:

```

1  sq = (x) -> x * x
2  write "My favorite square numbers"
3  write sq(8)
4  write sq(3)

```

*The function sq calculates the square of a number and is used twice in this program.*

Line 1 has a function definition, defining sq as the function (x) -> x \* x. This function has one input **parameter** listed in parentheses (x) before the arrow. After the arrow ->, the **body** of the function calculates x \* x.

The body of the function is a piece of code that is not run right away! It makes sense that it does not run yet, because the program does not yet know what value to use for x. The parameter x is a kind of **variable**: its value varies depending on the situation, and we will not know what value to use for x until later when there is a function call.

On line 3, `sq(8)` is the function call. The number 8 is the function argument. This is the specific value to be assigned to the parameter (`x`). When running line 3, the program immediately executes the `sq` function by jumping up to the body of the function `sq` on line 1, setting `x` temporarily to 8, and then computing `x*x`, which is 64. When this is done, it returns 64 back to line 3, and the number 64 is written.

The program then proceeds to line 4, which calls `sq` again. This time `x` is assigned to 3, and `x * x` is returned as 9.

The flow of this kind of program may seem simple, but functions are so fundamental that it is important to thoroughly understand the sequencing of function calls and return.

Notice that each time `sq` is called, `x` can have a different value. We say that `x` has a different meaning for every **invocation** of the function. Because of this, `x` is called a **local** variable - it has no meaning outside the invocation of the function.

### How Can Functions Control When Code Is Run?

A function is an object that can be called any time (whenever needed). For example:

```
myfunc = -> write 'ouch!'
button 'click me', myfunc
```

*The function `myfunc` does not run immediately, but only when the button is pressed.*

Here `myfunc` is the function `-> write 'ouch!'` that requires no arguments, and that writes a message each time it is called. Notice that, as usual, the function is not run when it is defined: no “ouch!” is written to the screen when the program is first run. But whenever we click the button, the function is called and we see “ouch!”

In this example, we have not written the function call! Instead, the built-in function `button` sets up its own function call to be done whenever the button is clicked. A function that is given for the purpose of getting a call back is called a **callback** function, and since our callback is called whenever an event occurs, it is sometimes also called an **event handler**.

In CoffeeScript, we can make a function without ever naming it - an **anonymous function**:

```
button 'click me', -> write 'ouch'
write 'ready?'
```

*Functions containing code to run later can be created even without ever giving them a name.*

Here, again, the function `-> write 'ouch!'` is passed as the second argument to the `button` function. However, unlike the previous example, we have not given a name to the function `-> write 'ouch!'`. We just define it inline where we need to pass it to `button`. Although anonymous functions sound mysterious, they are commonly used for creating event handlers.

When anonymous event handlers are indented and passed directly as callbacks, the code makes it clear that the function body is the code to run whenever a specific event occurs.

```
button 'go forward', ->
  fd 100
  dot red
button 'go backward', ->
  bk 100
  dot blue
```

*Event handlers (from Chapter 3) are functions.*

It is worth considering why commas are needed in the code above: the arrow and two indented lines of code after each comma form an anonymous function that is passed as the second argument to `button`. Although this code is simple to write, it contains several very important concepts, and we suggest that students experiment with writing the code for event handlers in different ways using named functions and anonymous functions.

### When Would a Programmer Define a Function?

Functions are useful whenever we have code that we want to **reuse** in several places in a program or (the same idea looked at in another way) whenever we have code whose execution we want to **defer** to some future call.

One principle that guides the use of functions in these situations is called **DRY**: “Don’t Repeat Yourself.” If you find that you are writing similar code in two or more different places in your program, you should define a function whose body contains that code exactly once; and then use function calls to reuse that same function in different places.

To aid in DRY, programmers routinely call functions from within the body of other functions and they usually define functions with several parameters to allow their functionality to be customized to fit different situations. Advanced programmers often customize functions by calling other functions passed as parameters - that is how callbacks are created. If done carefully, functions can even be called from within themselves - that is called **recursion**. (Chapter 10)

Because they make it possible to organize and arrange the code of a program in both simple and complicated situations, functions are the most powerful and fundamental concept in programming.

#### 5.1.1 Suggested Time-line: 1 55-minute class period

Instructional Day	Topic
2 Days	Lesson Plan I & II: Use of buttons to explain the purpose of functions.
1 Day	Lesson Plan III: Teach how functions help with reusability of code.
1 Day	Lesson Plan IV: Teach parameter passing in functions.

#### 5.1.2 Standards

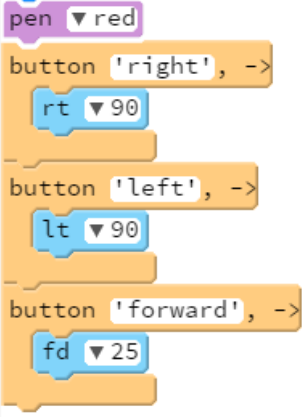

CSTA Standards	CSTA Strand	CSTA Learning Objectives Covered
Level 3 A (Grades 9 – 12)	Computational Thinking (CT)	Use predefined functions and parameters, classes and methods to divide a complex problem into simpler parts.
Level 3 A (Grades 9 – 12)	Computing Practice & Programming (CPP)	Apply analysis, design, and implementation techniques to solve problems.
Level 3 A (Grades 9 – 12)	CPP	Use Application Program Interfaces (APIs) and libraries to facilitate programming solutions.
Level 3 B (Grades 9 – 12)	CT	Decompose a problem by defining new functions and classes.
Level 3 B (Grades 9 – 12)	CPP	Use tools of abstraction to decompose a large-scale computational problem (e.g. procedural abstraction, object-oriented design, functional design).
Level 3 B (Grades 9 – 12)	CT	Discuss the value of abstraction to manage problem complexity.

### 5.1.3 Teaching Notes:

This is the first topic that will bring significant programming challenges to the beginning programming student. In many languages, when a student tries to break into modules of reusable code, parameter passing, returning the correct values with appropriate data types and calling the modules result in compile errors. Students will resist modularity so as to avoid compile errors. Pencil code in block-mode will help avoid some of the errors.

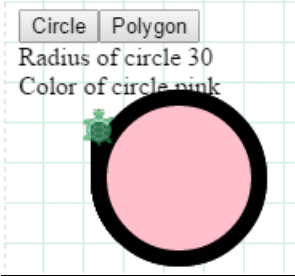
### 5.1.4 Lesson Plan I

This lesson demonstrates that functions must to be called to be executed.

Content details	Teaching Suggestions	Time
<p>Code:</p>  <pre data-bbox="203 1060 738 1276">pen red button 'right', -&gt;   rt 90 button 'left', -&gt;   lt 90 button 'forward', -&gt;   fd 25</pre>	<p>In this lesson, the buttons call the functions. Every button press is a function being called.</p> <p>Demonstrate the press of a button to the students. Show the code for every button press.</p> <p>Provide this link: <a href="http://teachersguide.pencilcode.net/edit/functions/remotecomtrol">http://teachersguide.pencilcode.net/edit/functions/remotecomtrol</a> for the students to play with the buttons and to enable them to create patterns.</p> <p>Output</p> 	<p>Demonstration: 15 minutes</p> <p>Student Practice: 20 minutes</p>

## 5.1.5 Lesson Plan II

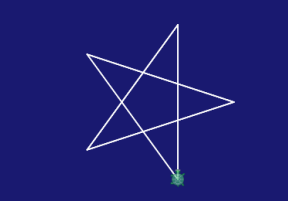

This lesson introduces the idea of functions by creating buttons to call functions.

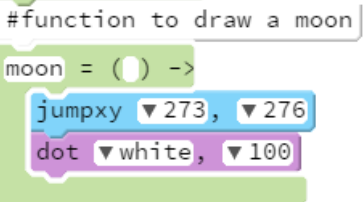
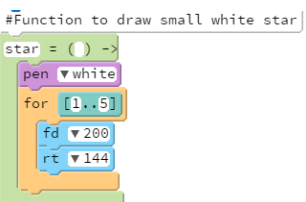
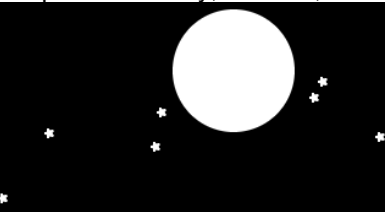
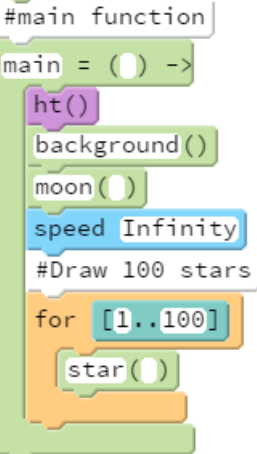
Content details	Teaching Suggestions	Time
<p><b>Code for the Circle button</b></p> <pre># Asks user for radius and color for a circle button 'Circle', -&gt;   await read 'Radius of circle', defer radius   await read 'Color of circle', defer color   jump to -340, -240   circle(radius, color )  # Call to the circle function from the main program # Asks user for radius and color for a circle button 'Circle', -&gt;   await read 'Radius of circle', defer radius   await read 'Color of circle', defer color   circle(radius, color )  speed 100 pen black, 10 pen black, 10 # Asks user for radius and color for a circle button 'Circle', -&gt;   await read 'Radius of circle', defer radius   await read 'Color of circle', defer color   circle(radius, color ) # Asks user for sides and color in a polygon button 'Polygon', -&gt;   await read 'Number of sides', defer sides   await read 'length of each side', defer length   await read 'Color of polygon', defer color   polygon(sides, length, color) circle = (radius, color) -&gt;   rt 360, 50   fd radius   fill color # draws a polygon. Asks user for # of sides, length polygon = (sides, length, color) -&gt;   pen color, 10   for [1..sides]     fd length     rt 360/sides   fill color</pre>	<p>Pull up the Shapes Bot program. <a href="http://teachersguide.pencilcode.net/edit/functions/ShapeBot">http://teachersguide.pencilcode.net/edit/functions/ShapeBot</a></p> <p>Demonstrate the action of the buttons by clicking on the Circle and Polygon button.</p> <p>Next, show the code to the students. Explain that under the Circle button is the code that draws the circle to the specifications created. Also show the main program from which the circle and polygon functions are called. (Screenshots on the left column are for circle.)</p> <p>Encourage students to improve the program by adding their own shapes (Triangle, Star, etc.).</p> <p>Teaching Tip: For now do not focus on the parameters being passed. Just ask to students to accept it as is. We will address it in a lesson plan later.</p> <p><u>Output for the Circle Button:</u></p> 	<p>Demonstration: 20 minutes</p> <p>Student Practice: Until the end of class.</p> <p>Add to the program additional code to draw shapes and buttons.</p>

### 5.1.6 Lesson Plan III

This lesson introduces the use of various palates and blocks to create reusable code snippets.

Content details	Teaching Suggestions	Time
<p><u>Code:</u></p> <pre>tee = -&gt; fd 50 rt 90 bk 25 fd 50 pen green tee() pen gold tee() pen black tee()</pre>	<p>Introduce the idea that a function can be called several times to create something unique or to solve a problem.</p> <p>Type the code and demonstrate that a function can be called repeatedly.</p> <p>Note: The Tee program code can be found at <a href="http://book.pencilcode.net">book.pencilcode.net</a></p> <p><u>Output:</u></p>	<p>Demonstration: 20 minutes.</p>
<p><u>Code</u> Step I</p> <pre>#Function to draw generic black background background = () -&gt;   dot midnightblue, 1500</pre>	<p>Introduce the idea that one small function can be written and another program can call it.</p> <p>Step I: Write the program to draw the background: <code>background ()</code></p>	<p>Demonstration Time: 30 minutes (all steps included)</p>
<p>Step II</p> <pre>#Function to draw small white star white star star = (x) -&gt;   pen white   for [1..5]     fd x     rt 144</pre>	<p>Step II: Write the program to draw the star: <code>Star()</code>.</p> <pre>#Function to draw small white star</pre>	

Content details	Teaching Suggestions	Time
<p>Code: Step III</p> <pre>#function to call all the other functions main = () -&gt;   background()   speed Infinity   star()</pre>	<p>Step III: Write a main program that calls both of these programs: main().</p> <pre>#function to call all the other funct main = () -&gt;   background()   speed Infinity   star()</pre>	
<p>Code: Step IV Blue Sky, 1 Star</p> <pre>main()</pre>	<p>Step IV: Call main ()</p> <p>Explain to students that main () does not know how the star is created. The code for the program can be found here:</p> <p><a href="http://teachersguide.pencilcode.net/edit/chapter5/starsInTheSky_I">http://teachersguide.pencilcode.net/edit/chapter5/starsInTheSky_I</a></p> 	
<p>Black Sky, 25 stars</p> <pre>main()</pre> <pre>main() #Function to draw small white star star = () -&gt;   pen white   for [1..5]     fd 200     rt 144</pre> <pre>#Function to draw generic black background background = () -&gt;   dot black, 1500 #main function main = () -&gt;   ht()   background()   speed Infinity   #Draw 100 stars   for [1..25]     randomX = (random [- 400..400])     randomY = (random [- 400..400])     jumpto randomX, randomY     star() # run everything</pre>	<p>Explain and demonstrate to students that star () can be used in different programs.</p> <p>Code the program and show students how the star () and background () functions are reused.</p> <p><a href="http://teachersguide.pencilcode.net/edit/Chapter5/StarsInTheSkyII">http://teachersguide.pencilcode.net/edit/Chapter5/StarsInTheSkyII</a></p> <pre>#main function main = () -&gt;   ht()   background()   speed Infinity   #Draw 100 stars   for [1..100]     randomX = (random [-400..400])     randomY = (random [-400..400])     jumpto randomX, randomY     star()</pre>  <pre>#Function to draw small white star star = () -&gt;   pen white   for [1..5]     fd 200     rt 144</pre>	<p>Demonstration: 15 minutes</p>





Content details	Teaching Suggestions	Time
<p>Code</p> <pre>#function to draw a moon moon = () -&gt;   jumpxy 273, 276   dot white, 100 #Function to draw generic black background background = () -&gt;   dot black, 1500</pre>  <pre>#Function to draw small white star star = () -&gt;   pen white   for [1..5]     fd 200     rt 144</pre>  <pre>#main function main = () -&gt;   ht()   background()   moon()   speed Infinity   for [1..50]     randomY = (random [- 400..400])     randomX = (random [- 400..400])     jumpto randomX, randomY     star()</pre> <pre># run everything main()</pre>	<p>Demonstrate that it is now easy to add newer functionality to the program (for example adding a moon).</p> <p>Copy the code shown on the left column to demonstrate modularity.</p> <p>Explain that if the star or the moon does not display correctly, it is easier to find the bug in the program because the functionality (behavior) is isolated within the function.</p> <p>Demonstrate this by modifying the position of the moon (jumpxy – values).</p> <p>Or, modify the fd- value in the star function. Change it to something very small such as 6 (code and output shown).</p> <p>Note: The main () function remains the same. (Not shown) View the code for the program here: <a href="http://teachersguide.pencilcode.net/edit/Chapter5/StarsInTheSkyIII">http://teachersguide.pencilcode.net/edit/Chapter5/StarsInTheSkyIII</a></p> <p>Output: Black Sky, 1 Moon, 100 Stars</p>  	<p>Demonstration: 15 minutes</p> <p>Student Practice: 45 minutes</p>



### 5.1.7 Lesson Plan IV

This lesson explores functions with parameters. Programmers design functions with more variables to make their programs reusable for various applications. The values for the variables are obtained from the user and passed to the function as parameters.

Ex. Circle (radius, color) – radius

Content Details	Teaching Suggestions	Time
<p><b>Code</b></p> <pre>#Moon with parameters moon = (x, y, size, color) -&gt;   jumpto x, y   dot color, size</pre> <pre>moon = (x,y,size,color) -&gt;   jumpto x, y   dot color, size moon(230,230,100,blue)</pre> <pre>#Function to draw small white star star = (x) -&gt;   pen white   for [1..5]     fd x     rt 144</pre> <pre>#Function to draw generic blue background background = () -&gt;   dot midnightblue, 1500</pre> <pre>#function to call all the other functions main = () -&gt;   background()   speed Infinity   star(15)</pre> <pre>main()</pre>	<p>Pull up the <a href="#">moon</a> example. Ask the students how they would change the size of the moon. The moon program (see code) can now take parameters on the size and position. And depending on what the values are the position and size of the moon on the sky will be different. Explain to that in moon (230,230...), x will take value 230 and y will take value 230.</p> <p>Teaching suggestion. Ask students to pull up the star program and add parameters (<a href="#">code shown</a>). You can also code along with the students and on your screen.</p> <pre>moon (230,230,100,blue,)</pre>  <pre>moon (230,230,25,black,)</pre>  <pre>star 55</pre>  <pre>star 15</pre>  <pre>#function to call all the other functions main = () -&gt;   background()   speed Infinity   star(15) main()</pre>	<p>Demonstration: 30 minutes</p>
<p>Iterative Development Cycle: Students need a great deal of practice writing programs with functions. Train them to write small chunks of code and test it, and then add small changes and test again, repeating this process until the program behaves as desired. The Pencil Code environment provides the output grid which gives instant feedback. Student Practice:120 minutes</p>		

# Chapter 6: Conditional Statements

## 6.0.1 Objectives

Students typically find conditional statements (also known as selection, or decision statements) easy to understand when compared to other constructs. The main area of confusion involved with conditionals occurs when students begin using Boolean combinations: for example, the word “and” as it is used casually in English can have a different for the formal word “and” in Boolean logic. At the end of this unit, students should be able to apply conditionals, creating Boolean expressions with comparisons, and they should be able to reason correctly about the use of the Boolean operators “and” “or” and “not.”

## 6.0.2 Topic Outline

- 6.0 Chapter Introduction
  - 6.0.1 Objectives
  - 6.0.2 Topic Outlines
  - 6.0.3 Key Terms
  - 6.0.4 Key Concepts
- 6.1 Lesson Plans
  - 6.1.1 Suggested Timeline
  - 6.1.2 CSTA Standards
  - 6.1.3 Lesson Plan I on using the control block- If, If... Else.
  - 6.1.4 Lesson Plan II on re-visiting the spiral assignment on the Question Bot.
  - 6.1.5 Lesson Plan III on using Boolean expressions.
  - 6.1.6 Lesson Plan IV to designing a hi-lo game and the race car game.
- 6.2 Resources
  - 6.2.1 Additional exercises.

## 6.0.3 Key Terms

Boolean Values: true / false	If then else, If then else if else
Simple & Complex expressions	match functionality
AND / OR – Operators	
is (comparison), isnt, < ,>	
Numerical comparisons	
String comparisons	

## 6.0.4 Key Concepts

### Controlling Code Using Conditions

The word `if` can be used to put a block of code under the control of a condition, so it only runs when the condition is true.

Below is an example that uses `if` to control turtle motion by testing keyboard presses.

The indented code `fd 2` only runs when the condition `pressed('W')` is true, that is, when the user is pressing the W key. Similarly, the two lines of code `rt 2; dot blue, 5` only run when the user is pressing the D key.

If neither key is pressed, neither block of indented code is run. If both keys are pressed, both blocks are run.

```
forever ->
  if pressed('W')
    fd 2
  if pressed('D')
    rt 2
  dot blue, 5
```

*Testing key presses using if.*

### Using “else” For the Other Alternative

The “else” keyword allows you to program a second action to take when the “if” does not happen. The second block of code will run when the condition is false.

```
forever ->
  if pressed('W')
    fd 2
  else
    rt 2
```

*Providing two alternatives using if/else.*

This program moves the turtle forward when W is pressed. When W is not pressed, it spins the turtle.

### Chaining “else if” for Multiple Alternatives

When there are three or more actions, if and else can be chained.

```
forever ->
  if pressed('W')
    fd 2
  else if pressed('S')
    bk 2
  else
    rt 2
```

*Chaining if, else if, and else for three alternatives.*

This code moves forward if W is pressed and backward if S is pressed. It spins right if nothing is pressed. A chained if/else only chooses the first condition that is true, so if both W and S are pressed at the same time, this program will just do “fd 2” and not move backward.

### Using “and”, “or” and “not” to Combine Boolean Expressions

The words and, or, and not are **Boolean operators** that can be used to combine conditions. For example, the following program uses “and” and “not”. It draws a blue ring and then only moves the turtle forward if W is pressed and the turtle is not already touching the blue ring.

```

dot blue, 500
dot white, 400
forever ->
  if pressed('W') and not touches('blue')
    fd 2
  else
    rt 2

```

*Combining tests using and and not.*

Although Boolean operators usually work in the same way as when reading them as English words, it is important to understand exactly how they work as mathematical operators because it is easy to get unexpected effects.

### Confusing “and” With “or”

Consider a program that where “up” and “W” keys need to work equivalently, both working in the same way to move the turtle forward. We might be tempted to use the “and” combiner to capture both cases with a single “if” like this:

**WRONG:**

```

forever ->
  if pressed('up') and pressed('W')
    fd 2

```

*Incorrectly using “and” to combine two alternatives.*

This code, however, will not generate the desired effect! To understand why, we need to understand how and and or operate on truth values.

### Boolean Values and Boolean Tables

The words and, or and not are Boolean operations that combine “true” and “false” values (similar to the arithmetic rules you get when using “+”, “\*” to combine regular numbers). Just as we can learn about addition and multiplication by creating addition and multiplication tables, we can understand and and or by writing **truth tables**. Here are two truth tables related to the program above:

pressed('up') <b>and</b> pressed('W')	pressed 'up' = false	pressed 'up' = true
pressed 'W' = false	false	false
pressed 'W' = true	false	true

pressed('up') <b>or</b> pressed('W')	pressed 'up' = false	pressed 'up' = true
pressed 'W' = false	false	true
pressed 'W' = true	true	true

The conjunction and combines two Boolean values and creates “true” only when both of the values are true. For example, `pressed('up') and pressed('w')` is true only when both the up and W keys are pressed at the same time.

The disjunction or combines two Boolean values and creates “true” when either or both of the values are true. For example, `pressed('up') or pressed('w')` is true when just the W key is pressed, or just the up key is pressed, or both. This is what we want for our program.

To fix the program, the and should be switched to or.

### Testing Numbers Using Comparison Operators

Boolean expressions can be used to test the properties of numbers. Most of the comparison operators you would see in math class work in a programming language, but they may be written with slightly different punctuation. For example, “is less than or equal to” is written `<=`. Here is a summary of some common Boolean tests for numbers:

Expression	Description	What if x = 0?	What if x = 3?	What if x = 6?
<code>x is 3</code>	x is equal to 3	false	true	false
<code>x isnt 3</code>	x is not equal to 3	true	false	true
<code>x &lt; 3</code>	x is less than 3.	true	false	false
<code>x &lt;= 3</code>	x is less than or equal to 3	true	true	false
<code>x &gt; 3</code>	x is greater than 3	false	false	true
<code>x &gt;= 3</code>	x is greater than or equal to 3	false	true	true
<code>0 &lt; x &lt;= 6</code>	x is greater than 0 and less than or equal to 6	false	true	true
<code>x % 2 is 1</code>	x is odd (because it has remainder 1 when divided by 2)	false	true	false
<code>x % 3 is 0</code>	x is divisible by 3	false	true	true

### Confusing “or” with Comparisons

Numerical comparisons can be combined with Boolean operators. For example, `(x > 6 and x isnt 9)` means that x is a number greater than 6 other than 9, and `(x is 5 or x is 11)` means that x is either 5 or 11. It is important, though, to remember that the word “or” operates on truth values and not on numbers, so the version of the program on the left does not do produce the desired result.

<p><b>WRONG:</b></p> <pre>await readnum 'How many items?', defer n if n is 1 or 2 write 'Come to the speedy checkout.'</pre>	<p><b>RIGHT:</b></p> <pre>await readnum 'How many items?', defer n if n is 1 or n is 2 write 'Come to the speedy checkout.'</pre>
--	---

The program on the left incorrectly results in the speedy checkout line no matter what number you enter.

To understand why, remember that `or` operates on truth values, so when you say `"or 2"`, it begins with the question `"is 2 true or false?"` By convention, any number that is not zero is treated as `"true"`, so `"or 2"` makes the expression always true regardless of the value of `num`. On the other hand, the program on the right produced the desired result: `"or n is 2"` only makes the expression true when the number is 2.

Another way to think about the difference is with precedence of operators. The word `"or"` has lower precedence than the word `"is"`, so the expression on the left reads like this: `((n is 1) or 2)` and the expression on the right reads like this: `((n is 1) or (n is 2))`.

## Testing Strings Using Pattern Matching

Text strings can also be tested to create Boolean values. It is common to test strings by comparing them exactly (looking at their length) or by testing if the string matches a pattern using the `"match"` method. Pattern matching can be used to determine if a string contains a particular pattern of letters within it.

The following table shows several examples.

Expression	Description	"appear"	"pear"	"peachy"
<code>x is "pear"</code>	x is exactly equal to the string "pear"	false	true	false
<code>x.length is 6</code>	x has exactly 6 characters	true	false	true
<code>x.match(/pp/)</code>	x contains the substring "pp".	true	false	false
<code>x.match(/pea/)</code>	x contains the substring "pea"	true	true	true
<code>x.match(/PEA/)</code>	x contains the substring "PEA"	false	false	false
<code>x.match(/Pea/i)</code>	x contains the substring "Pea", ignoring case	true	true	true
<code>x.match(/^pea/)</code>	x contains "pea" at the start of the string	false	true	true
<code>x.match(/ear\$/)</code>	x contains "ear" at the end of the string	true	true	false
<code>x.match(/a(p ch)/)</code>	x contains "a" followed by either "p" or "ch"	true	false	true
<code>x.match(/ap*e/)</code>	x contains "a", then zero or more "p", then "e"	true	false	false

The patterns used between the `"/"` symbols are called **regular expressions**.

A regular expression can be used to test whether a string contains a fixed pattern, for example whether it contains the letters `"pp"`. Normally regular expressions are case-sensitive, so `"PEA"` does not match `"pea"`, but putting an `"i"` after the regular expression makes it case-insensitive.

Regular expression patterns have several powerful features. For example, in a regular expression, `"^"` matches the beginning of the string, `"$"` matches the end of the string, `"(one|other)"` is used to match alternatives, and `"*"` allows a sub-pattern to be repeated zero or more times.

Although the types of patterns shown above are enough for most situations, regular expressions have several more features. There are many excellent resources about regular expressions on the Internet if you search for `"regular expression lessons"`. When exploring, it is important to know that the symbols used in regular expression patterns are standardized, and the same pattern language is used in JavaScript, CoffeeScript, Python, Perl, Java, C# and other languages.

### 6.1.1 Suggested Timeline: 1 55-minute class period

Instructional Day	Topic
2 Days	Lesson Plan I: If, If Then Else statements using fun visual elements
1 Day	Lesson Plan II: Question Bot & Lesson Plan III Complex If Statements
1 Day	Lesson Plan IV: Pair programming for the HiLo Game
1 Day	Lesson Plan V: Race Car Track game

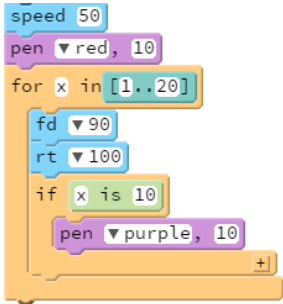
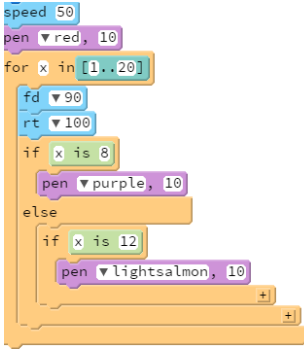
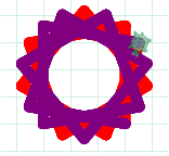
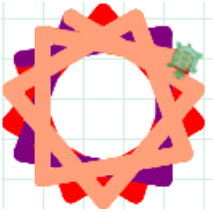
### 6.1.2 Standards

CSTA Standards	CSTA Strand	CSTA Learning Objectives Covered
Level 3 A (Grades 9 – 12)	Computational Thinking (CT)	Explain how sequencing, selection, iteration and recursion are building blocks of algorithms.
Level 3 A (Grades 9 – 12)	CT	Explain the program execution process.
Level 3 A (Grades 9 – 12)	CT	Describe how mathematical and statistical functions, sets, and logic are used in computation.

### 6.1.3 Lesson Plan I

This lesson focuses on the Control Block, If statements and If Else statements. The lesson should take about 30 to 40 minutes of a class period, providing time for students to code programs similar to the ones modeled.

Content details	Teaching Suggestions	Time
<p><u>Code</u></p> <pre>x = random [1..3] write x if x is 1 or x is 2   write 'Today is your lucky day!' else   if `` is ``     write 'Stay low. Let everything happen tomorrow'   else     write 'I cannot see your future.' # Demonstrate complex IF Statements</pre> <p><u>Code</u></p>	<p>Explain the key concept of the evaluation of a Boolean expression.</p> <p>Show the Control Block and the 'IF' statement.</p> <p>Type the <a href="#">code</a> shown on the left column. (Note: You will need speakers for this program. You can use the write block instead of say block.)</p> <p>Teaching Tips: You can extend this lesson further by adding a loop around the entire code for the tune to be played for a fixed period (e.g. 5 times). Explain that the program gives an output based on the value the variable to which "Day" is set.</p>	<p>Demonstration: 20 minutes</p>
<p><u>Code</u></p>	<p>Give another example using patterns. Demonstrate how the conditional can impact</p>	<p>Demonstration: 20 minutes</p>

Content details	Teaching Suggestions	Time
<pre> speed 50 pen red, 10 for x in [1..20]   fd 90   rt 100   if x is 8     pen purple, 10   else     if x is 12       pen lightsalmon, 10 </pre>  	<p>the color in which the pattern is drawn.</p> <p>Teaching Tips: Change the values within the conditional to show how the pattern changes.</p> <p>Add another if statement to show another color. (Point out the use of a nested if statement.)</p> <p>Add an 'if' statement to change the speed. Here is the copy of the program: <a href="http://teachersguide.pencilcode.net/edit/cha-pter6/pattern">http://teachersguide.pencilcode.net/edit/cha-pter6/pattern</a></p>  <p>Now students can start writing their own programs. Students are expected to write both the programs that were demonstrated. Students should complete both programs by the end of the class period (15 minutes of class time).</p> 	<p>Student Practice: 15 minutes</p>
<p><u>Code</u></p> <pre> speed 10 answer = 'yes' hide() while answer is 'yes'   diceRoll = (random 6)   label String.fromCharCode(9856 + diceRoll), 100   say 'Rolling dice now!'   if diceRoll is 6     say 'You made a 6!! Roll again!'     answer = no   else     write diceRoll     say 'Too Bad! Want to </pre>	<p>Add a few fun elements to the program to show that how using a loop and a conditional increases a program's power.</p> <p>Here is the <a href="#">code</a> to simulate the roll of a die. (This is a starter program for a Yahtzee game. The Additional Exercises section provides the specifications for Yahtzee.)</p> <p>Demonstrate and walk students through the code. Point out the use of random numbers, how the assignment of an exit condition lets the loop exit eventually, and the If... Else block.</p>	<p>Demonstration: 55 minutes</p>



Content details	Teaching Suggestions	Time
<pre>try again?'   await read 'Roll again?', defer answer say 'Good bye!'</pre>		

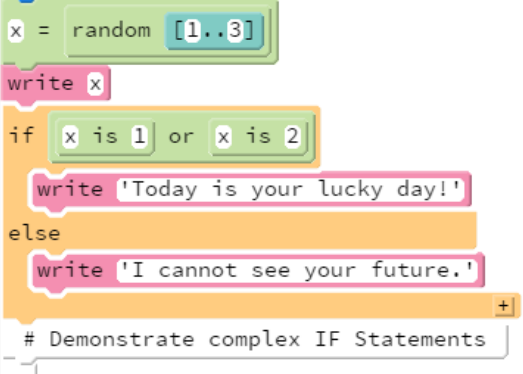
### 6.1.4 Lesson Plan II

This lesson revisits the Spiral assignment Question Bot introduced in the Chapters 3, 4, and 5. Students will extend this program using all of the previously-learned concepts and add conditional statements.

Content details	Teaching Suggestions	Time
<p>Code:</p> <pre># chatbot # CS1004 example chatbot using loop and variable prompts  write 'Bob: Hi! My name is Bob.' await read 'Bob: What\'s your name?', defer name write 'Bob: Hello ' + name done = false while not done   prompt = name + ' can you guess who I am?' + ':'   await read prompt, defer q   if (q.match /quit give up/)     write 'Bob: OK, nice talking to you!'     done = true   else if (q.match /bot/)     write 'Bob: Close... But I am a human, of course.'   else     write 'Bob: Good guesswork!'</pre>	<p>Pull up the Question Bot (Loops version) program. Ask students how its functionality can be expanded.</p> <p>Even though the program asks four people's names, it does not "remember" them?</p> <p>Making the lesson more interesting by asking students for a good question. Walk through the code and emphasize the power of conditionals.</p> <p>This <a href="#">example</a> also introduces the operator <code>x.match ()</code> functionality. In block-mode it looks like this:</p>	<p>Demonstration: 20 minutes</p>

### 6.1.5 Lesson Plan III

This lesson plan demonstrates the use of complex if statements, specifically, combining multiple Boolean expressions.

Content details	Teaching Suggestions	Time
<p>Code</p> <pre>x = random [1..3] write x if x is 1 or x is 2   write 'Today is your lucky day!' else   write 'I cannot see your future.' # Demonstrate complex IF Statements</pre>  <p>The image shows the same code as above but rendered as Scratch-style code blocks. The blocks are: a green 'x = random [1..3]' block, a pink 'write x' block, a green 'if x is 1 or x is 2' block containing a pink 'write 'Today is your lucky day!'' block, an orange 'else' block containing a pink 'write 'I cannot see your future.'', and a grey comment block '# Demonstrate complex IF Statements'.</p>	<p>Explain the key concept of the evaluation of multiple Boolean expressions. Explain how an AND / OR operator combines two expressions and evaluates them.</p> <p>The example provided here is a simple program that acts like a “Genie” and predicts the day based on the random number generated.</p> <p>Teaching Tip: Explain how an OR operator works. Explain using the example that the ‘if’ statement will get executed even if one of the expressions is true.</p> <p>Ask the students to copy the <a href="#">code</a> provided and test the program.</p>	<p>Demonstration: 20 minutes</p>
<p>Code:</p> <pre>x = random [1..3] write x if x is 1 or x is 2   write 'Today is your lucky day!' else   if `` is ``     write 'Stay low. Let everything happen tomorrow'   else     write 'I cannot see your future.' # Demonstrate complex IF Statements</pre>	<p>Now explain in-depth using the modified program as show in the left column.</p> <p>Teaching Tip: Explain how an AND operator works. Explain using the example that the ‘if’ statement will get executed IF AND ONLY IF both expressions are true.</p> <p>Ask students to copy <a href="#">the code</a> provided and test the program.</p> <p>Give students 10 minutes to experiment with if statements and modifications of the code. Let them test other programmers’ code and see what kind of day the genie predicts for them!</p>	<p>Demonstration: 20 minutes.</p> <p>Student Practice: 10 minutes</p>

## 6.1.6 Lesson Plan IV

Students will now work with a partner to design a Hi-Lo Guessing game. The Additional Exercises section provides the problem description. Students will watch a short video on pair programming before beginning the exercise. The lesson plan below focuses on how to design a project with a partner. Students will take about half a class period to design the program. They will have the rest of the class period and one more class period or homework to complete the assignment. (Note: Homework may not result in true collaborative work because it increases the temptation to find answers on the Internet and from people outside of the team.)

Content details	Teaching Suggestions	Time
<a href="https://www.youtube.com/watch?v=vgkahOzFH2Q">https://www.youtube.com/watch?v=vgkahOzFH2Q</a>	Have students watch the video on pair programming.	Student Practice: 5 minutes
<p>Split students into groups. The References section provides resources on forming successful collaborative groups.</p> <p>Using the “Rally Robin” co-operative learning® structure, have the students write the pseudocode for the HiLo game. Their pseudocode should indicate:</p> <ol style="list-style-type: none"> <li>i. Variables used</li> <li>ii. Control structures needed</li> <li>iii. Input / Output statements</li> </ol> <p>Student Practice: 20 minutes</p>		
<p>Rally Robin instructions: On a piece of paper, take turns writing instructions on how to design the guessing game.</p> <p>Each instruction should be a numerical bullet (e.g. 1,2,...) Student 1 writes the first instruction. Student 2 writes the second instruction.</p> <p>Keep repeating this process until all of the instructions have been recorded. Next, go back and take turns revising the instructions until both partners are satisfied.</p> <p>Submit your work your teacher for grading. Here is a sample <a href="#">Hi-Lo</a> program. Student Practice: 20 minutes</p>		
<p>On your computer, type the program with your partner using the pair programming methods presented in the pair programming video. Student Practice: 40 minutes</p>		

## 6.1.7 Lesson Plan V

This interactive lesson involves having the students design a racecar game. We recommend that students type this in text using CoffeeScript. Encourage the students to toggle between text-mode and block-mode to venture out of their comfort zone and increase their confidence. Follow the activity Turtle Race track as provided in this activity sheet

Content details	Teaching Suggestions	Time
<a href="http://activity.pencilcode.net/home/worksheets/race.html">http://activity.pencilcode.net/home/worksheets/race.html</a>	Teaching Tip: Have all the students in the class start on the worksheet at the same time. Print out the worksheet and give it to the students. This will eliminate the temptation to copy-paste the code. Follow the link to complete steps 1 and 2. Instruct students to type the text and stay in text-mode.	Student Practice: 10 mins
<a href="http://activity.pencilcode.net/home/worksheets/race.html">http://activity.pencilcode.net/home/worksheets/race.html</a>	Ask students to answer Questions 1 and 2 verbally in the classroom.	Student Practice: 15 mins
<a href="http://activity.pencilcode.net/home/worksheets/race.html">http://activity.pencilcode.net/home/worksheets/race.html</a>	Now let the students venture out at their own pace as they respond to Challenges 1 through 4. Students can stay in text-mode or switch back to block-mode. If they do switch to block-mode, encourage them to toggle to text and see how their code looks.  The solution code is given in CoffeeScript. Teaching Tip: If the students type the text, encourage them to indent their code to convey intent. All good programming practices (Refer Appendix A) should be followed.	Student Practice: 20 mins
<p>The race car game activity can be completed by students.</p> <ol style="list-style-type: none"> <li>1. <a href="http://activity.pencilcode.net/home/worksheets/race.html">http://activity.pencilcode.net/home/worksheets/race.html</a> - the basic game</li> <li>2. <a href="http://activity.pencilcode.net/home/worksheets/race-car.html">http://activity.pencilcode.net/home/worksheets/race-car.html</a> - making the car look like a car, and this emphasizes the idea of an “object” whose behavior your control</li> <li>3. <a href="http://activity.pencilcode.net/home/worksheets/race-two.html">http://activity.pencilcode.net/home/worksheets/race-two.html</a> - here you introduce a “second object” - two instances - and now you can controls them separately</li> <li>4. <a href="http://activity.pencilcode.net/home/worksheets/race-track.html">http://activity.pencilcode.net/home/worksheets/race-track.html</a> - this is a review of drawing, but used for a very different purpose - to create a track shape</li> <li>5. <a href="http://activity.pencilcode.net/home/worksheets/race-speed.html">http://activity.pencilcode.net/home/worksheets/race-speed.html</a> - this is an introduction to variables, used to keep track of how fast each car goes.</li> <li>6. <a href="http://activity.pencilcode.net/home/worksheets/race-time.html">http://activity.pencilcode.net/home/worksheets/race-time.html</a> - this is another use of variables, this time to keep track of how much time has passed</li> <li>7. <a href="http://activity.pencilcode.net/home/worksheets/race-menu.html">http://activity.pencilcode.net/home/worksheets/race-menu.html</a> - this is an example of use of functions to divide your program into subprograms.</li> </ol>		

## 6.2 Resources

### Additional Exercises:

#### Yahtzee

Design a modified version of Yahtzee where you roll three dice and the score is calculated based on which of the following categories are satisfied. The game ends after each user has had three turns. The user who has the largest point value at the end of three turns is the winner. The three categories are:

- i. Three of a kind – 5 points
- ii. Two of a kind – 10 points
- iii. Yahtzee – You Win! Game over!

#### Hi-Lo- Guessing Game

Design a simple game where the computer generates a random number and the user must guess the number. The computer helps the user by giving responses such as “too high or too low”. The user has a fixed number of tries to guess the right answer. Once the allowed number of tries are completed, the game displays the computer-generated number and says “Game Over”. You can get up to extra five creativity points if you have added a new feature to the game.

# Chapter 7: Learning A Second Language: JavaScript

## Block Mode → CoffeeScript → JavaScript

### 7.0.1 Objectives

The same fundamental programming concepts apply across different programming languages.

Exposure to a second programming language allows students to understand that the programming concepts they learn as beginners are the same concepts used by professionals. This unit introduces JavaScript, which is a very close cousin of CoffeeScript, and one of the most widely used languages used by professionals today. In this unit, students will use blocks to learn JavaScript syntax, and to see how the syntax of JavaScript and CoffeeScript have many similarities, and they will transition from blocks to programming directly in JavaScript text code.

### 7.0.2 Topic Outline

- 7.0 Chapter Introduction
  - 7.0.1 Objectives
  - 7.0.2 Topic Outlines
  - 7.0.3 Key Terms
  - 7.0.4 Key Concepts
- 7.1 Lesson Plans
  - 7.1.1 Suggested Timeline
  - 7.1.2 CSTA Standards
  - 7.1.3 Teaching Notes
  - 7.1.4 Lesson Plan I to use blocks to code in JavaScript mode.
  - 7.1.5 Lesson Plan II to recreate the Broken Scene program demonstrating the iterative development model.

### 7.0.3 Key Terms:

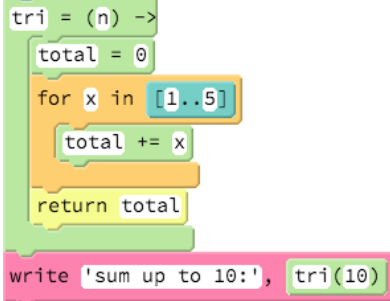
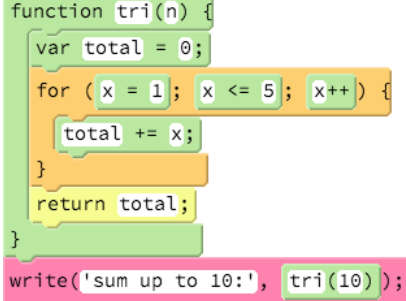
Websites	Block-mode
JavaScript	Text-mode
Scripting Language	Settings box in Pencil Code
Programming Language	

### 7.0.4 Key Concepts

#### Languages in Pencil Code

Every programming language has its own **syntax**, that is, has a specific set of patterns of words and punctuation allowed in the language. As shown in previous chapters, using a block editor view gives students a view of the syntax for a language.

For a sense for how JavaScript differs from CoffeeScript, see the two function definitions below, one written in CoffeeScript, and the other in JavaScript. Both text code and blocks are shown.

<p>CoffeeScript Text</p> <pre> tri = (n) -&gt;   total = 0   for x in [1..5]     total += x   return total write 'sum up to 10:', tri(10) </pre>	<p>CoffeeScript Blocks</p>  <p>The image shows the CoffeeScript code from the text block rendered as a series of colored blocks. The function definition 'tri = (n) -&gt;' is in a light green block. The initialization 'total = 0' is in a light green block. The loop 'for x in [1..5]' is in an orange block. The loop body 'total += x' is in a light green block. The return statement 'return total' is in a light green block. The final write statement 'write 'sum up to 10:', tri(10)' is in a pink block.</p>
<p>JavaScript Text</p> <pre> function tri(n) {   var total = 0;   for (x = 1; x &lt;= 5; x++) {     total += x;   }   return total; } write('sum up to 10:', tri(10)); </pre>	<p>JavaScript Blocks</p>  <p>The image shows the JavaScript code from the text block rendered as a series of colored blocks. The function definition 'function tri(n) {' is in a light green block. The variable declaration 'var total = 0;' is in a light green block. The loop 'for (x = 1; x &lt;= 5; x++) {' is in an orange block. The loop body 'total += x;' is in a light green block. The closing brace of the loop '}' is in a light green block. The return statement 'return total;' is in a light green block. The closing brace of the function '}' is in a light green block. The final write statement 'write('sum up to 10:', tri(10));' is in a pink block.</p>

Although looking at the text makes the languages seem very different, looking at the blocks reveals that JavaScript and CoffeeScript are closely related languages. Generally, JavaScript requires more punctuation, but the structure of most code is essentially identical between the two languages.

### Choosing between JavaScript and CoffeeScript

In Pencil Code, a project can be switched between JavaScript and CoffeeScript by clicking on the “gear” button in the blue bar. When you choose JavaScript, it is run directly by your browser. When you choose CoffeeScript, the CoffeeScript compiler compiles the program into JavaScript before it is run.

CoffeeScript and JavaScript are closely related. They operate on the same objects and they have the same level of speed and power when they run. CoffeeScript was designed after JavaScript, so the CoffeeScript syntax has several advantages:

- CoffeeScript syntax requires less punctuation than JavaScript, so it is easier to type without syntax errors.
- CoffeeScript directly supports more programming concepts; for example, it has syntax for classes and await.
- CoffeeScript uses meaningful indents, which means it is impossible to hide nesting mistakes with deceptive indenting.
- CoffeeScript avoids common mistakes in JavaScript such as approximate-zero-equality and accidental global variables.

The default language in Pencil Code is CoffeeScript, because it is easier to learn how to read and write text code in CoffeeScript. Why would a programmer choose to program in JavaScript? Because JavaScript has three significant advantages:

- JavaScript is an official standard designed by an international committee, and it runs in Web browsers without translation.

- The community of programmers who know JavaScript is larger than the CoffeeScript community.
- Many people are working on improving future versions of JavaScript, and they are aware of the good things in CoffeeScript.

JavaScript continues to evolve, and future versions of JavaScript will incorporate some of the innovations in CoffeeScript. So even for JavaScript aficionados, it is worth knowing CoffeeScript because many of its ideas represent the future of JavaScript.

And for fans of CoffeeScript, it is worth knowing JavaScript because there are benefits its larger community.

### Differences between CoffeeScript and JavaScript

JavaScript and CoffeeScript are closely related languages, and by understanding where some additional punctuation is required, you can translate directly from one to the other. Here is a summary of a few differences between CoffeeScript and JavaScript, showing how equivalent code would be written in each language. We discuss these in detail below.

CoffeeScript	JavaScript
<code>fd 100</code>	<code>fd(100);</code>
<code>if pressed('up') and not pressed('shift')   tone 440</code>	<code>if (pressed('up') &amp;&amp; !pressed('shift')) {   tone(440); }</code>
<code>for x in [0...10]   write x</code>	<code>for (var x = 0; x &lt; 10; ++x) {   write(x); }</code>
<code>for c in [red, orange, yellow]   dot c, 100   fd 100</code>	<code>var colors = [red, orange, yellow]; for (var j = 0; j &lt; colors.length; ++j) {   var c = colors[j];   dot(c, 100);   fd(100); }</code>
<code>button 'doorbell', -&gt;   write 'ding dong'   play 'C'</code>	<code>button('doorbell', function() {   write('ding dong');   play('C'); });</code>
<code>sq = (x) -&gt; x * x write sq(5)</code>	<code>function sq(x) {   return x * x; } write(sq(5));</code>

### Parentheses and Semicolons

JavaScript requires parentheses after a function name in order to run a function call. These same parentheses are optional in CoffeeScript, but are required in JavaScript.

CoffeeScript	JavaScript
<code>dot red, 100</code>	<code>dot(red, 100);</code>



JavaScript also recommends a semicolon at the end of every complete statement (including function calls, return statements, and break and continue statements). Not every line of code, however, is a complete statement. For example, the first line of an `if` is not a complete statement and should not be separated from the body of the `if` by a semicolon.

### Punctuation for Boolean Expressions

JavaScript also requires parentheses after the words `if`, `while`, and `switch` to surround the tested expression. CoffeeScript, does not require these parentheses.

CoffeeScript	JavaScript
<code>if pressed('X') and not pressed('Z') tone 440</code>	<code>if (pressed('X') &amp;&amp; !pressed('Z')) { tone(440); }</code>

The words `and`, `or`, `not`, `is` and `isnt` in CoffeeScript are not supported in JavaScript. Instead, JavaScript uses punctuation for these Boolean expressions. The corresponding punctuation has exactly the same meaning as the spelled-out words in CoffeeScript.

CoffeeScript	<code>x and y</code>	<code>x or y</code>	<code>not x</code>	<code>x is y</code>	<code>x isnt y</code>
JavaScript	<code>x &amp;&amp; y</code>	<code>x    y</code>	<code>!x</code>	<code>x === y</code>	<code>x !== y</code>

CoffeeScript allows the JavaScript punctuation for all these operators, but when programming in CoffeeScript, it is conventional to use the spelled-out English words to improve readability.

### Indents versus Curly Braces

JavaScript uses curly braces to indicate nesting. While indents are allowed, they do not mean anything to the JavaScript interpreter. It important to use curly braces and make sure they match the indenting.

<b>CoffeeScript</b>	<code>if x &lt; 0 write 'x is negative' write 'This is inside the if' write 'This is outside the if'</code>
<b>JavaScript</b> Equivalent, but bad style	<code>if (x &lt; 0) { write('x is negative'); write('This is inside the if'); } write('This is outside the if');</code>
<b>JavaScript</b> Good style	<code>if (x &lt; 0) { write('x is negative'); write('This is inside the if'); } write('This is outside the if');</code>

Some beginners, when discovering that JavaScript is not sensitive to indenting, will write JavaScript code with no indenting at all. This is a terrible idea, because it leads to confusing code like the example above. Good code is not just functional, but readable.

If the curly braces are omitted in JavaScript, a control flow statement such as `if`, `while`, or `for` will only apply to the single line of code after the condition. The red line in the JavaScript below will print “This is not in the if!” regardless of the value of `x`. The correct way to write the equivalent JavaScript is with curly braces, as in the last row.

<b>CoffeeScript</b>	<pre>if x == 0   write 'x is zero'   write 'This is inside the if'</pre>
<b>JavaScript</b> Not equivalent!	<pre>if (x === 0)   write('x is zero');   write('This is not in the if!');</pre>
<b>JavaScript</b> Correct; always include braces	<pre>if (x === 0) {   write('is zero');   write('This is inside the if'); }</pre>

Never omit the curly braces in JavaScript! The JavaScript interpreter will not pay attention to indenting, so omitting curly braces is an invitation to have errors like the one above.

It is important to help students understand that good style in a JavaScript program requires consistent use of curly braces and indenting. Programs should always be written with curly braces after conditionals and loops and indents must match with curly braces so that other programmers can read and understand them. The number of indents should match the number of nested curly braces.

## Understanding the Three-Clause for Loop

The current version of JavaScript does not have an array-based for loop like CoffeeScript. Instead, JavaScript supports a “three part for loop”. This type of for loop is just an abbreviation for three lines of a while loop on a single line. The following three are equivalent:

<b>CoffeeScript</b>	<pre>for x in [0...10]   write x</pre>
<b>JavaScript</b> using while	<pre>var x = 0; while (x &lt; 10) {   write(x); }</pre>
<b>JavaScript</b> using for	<pre>for (var x = 0; x &lt; 10; ++x) {   write(x); }</pre>

The three-part for loop in the last row of the table contains three statements in the parentheses.

1. The loop initializer, such as `var x = 0` here. This statement is run once, before the loop begins.
2. The loop condition, here `x < 10`. This statement is run before each repetition of the loop, including before the first one. If it is ever false, the loop skips to the end and stops running.
3. The loop incrementer, here `++x`. This statement runs after the execution of each iteration of the loop, right before going back to test the condition again.

Although there can be 10 or more pieces of punctuation on the line of a three-part “for” loop, they usually follow the same pattern, counting up from zero to some number.

## Declaring Functions Using the Word “function”

Functions in JavaScript are always written out with the special word “function,” and functions that return a value must contain an explicit return statement (whereas in CoffeeScript, the last value computed is automatically the value returned). Here is a named function declaration in JavaScript:

<b>CoffeeScript</b>	<pre>square = (x) -&gt;   write 'the input was ' + x   return x * x</pre>
<b>JavaScript</b>	<pre>function square(x) {   write('the input was ' + x);   return x * x; }</pre>

In CoffeeScript, functions must be defined before they are called while in JavaScript named functions can be declared out-of-order. JavaScript named function declarations are implicitly bound to their names before any other code runs.

Unnamed functions such as function callbacks from buttons or inputs, still use the word “function” but without the name, like this:

<b>CoffeeScript</b>	<pre>button 'get started', -&gt;   write 'starting now!'</pre>
<b>JavaScript</b>	<pre>button('get started', function() {   write('starting now!'); });</pre>

When an anonymous function is defined inline and passed as the last argument to an event binding function such as “button”, the program ends up including a curious series of punctuation at the end (a closing curly brace, a closing smooth parentheses, and a semicolon). This sequence of punctuation is very common in JavaScript code.

## Waiting on Input Without “await”

Unlike CoffeeScript, the current version of JavaScript does not support the “await” keyword or concept. That means that if you wish to write a program that waits on input, you must arrange function callbacks to produced the desired effect.

Arranging any sequence of execution is possible, but it requires planning.

The table below contains a simple program to total up numbers. The program on the right is written JavaScript using only function definitions while the program on the left uses the equivalent CoffeeScript with “await/defer”.

Getting a loop effect requires the programmer to define a function that sets up a callback that causes its own execution again. This is a form of looping through recursion. (Recursion will be discuss in more detail in Chapter 9.)

CoffeeScript	JavaScript
<pre>await readnum 'How many nums?', defer n total = 0 for j in [1..n]   await readnum 'Enter #' + j, defer x   total += x write 'Average is ' + (total / n)</pre>	<pre>readnum('How many nums?', function(n) {   var total = 0;   var j = 1;   nextnumber();   function nextnumber() {     readnum('Enter #' + j, function() {       total += x;       j += 1;       if (j === n) {         write 'Average is ' + (total / n);       } else {         nextnumber();       }     });   } });</pre>

The people that oversee the ongoing design of JavaScript are aware that the code on the right is more difficult to write than the code on the left, and the standards committee is considering adding “await” to a future version of JavaScript. Until that happens, the code on the right is what is needed to write this type of program in JavaScript.

## Common Syntax Pitfalls in JavaScript

JavaScript has more punctuation than CoffeeScript, so there are additional syntax errors to watch out for when coding in JavaScript in text-mode. Here are a few forms to watch out for:

Issue	Code with syntax error	Fixed code
Mismatched curly braces	<pre>click(function() {   write('clicked'); });</pre>	<pre>click(function() {   write('clicked'); });</pre>
Misspelling a keyword	<pre>funtion s(x) { return x * x; }</pre>	<pre>function s(x) { return x * x; }</pre>
Missing parentheses after “if”	<pre>if x &lt; 2 {   write('try again'); }</pre>	<pre>if (x &lt; 2) {   write('try again'); }</pre>
Missing clauses after “for” (JavaScript)	<pre>for (var j = 0; j &lt; 10) {   write(j); }</pre>	<pre>for (var j = 0; j &lt; 10; ++j) {   write(j); }</pre>
Mismatched curly braces, hidden by deceptive indenting (JavaScript)	<pre>for (var j = 0; j &lt; 10; ++j) {   if (j &gt; 8) {     write('and finally');   }   write(j); }</pre>	<pre>for (var j = 0; j &lt; 10; ++j) {   if (j &gt; 8) {     write('and finally');   }   write(j); }</pre>

When getting used to JavaScript syntax, it is helpful for the student to get some practice reading JavaScript punctuation so that they can quickly identify errors such as the ones above.

### 7.1.1 Suggested Time-line: 1 55-minute class period

Instructional Day	Topic
1 Day	Lesson Plan I: Creating a Random spiral program using blocks and JavaScript
2 Day	Lesson Plan II: Iterative Development Cycle

### 7.1.2 Standards

CSTA Standards	CSTA Strand	CSTA Learning Objectives Covered
Level 3 A (Grades 9 – 12)	Computational Thinking (CT)	Describe a variety of programming languages available to solve problems and develop systems.
Level 3 A (Grades 9 – 12)	Computing Practice & Programming (CPP)	Apply analysis, design, and implementation techniques to solve problems (e.g., use one or more software lifecycle models).
Level 3 B (Grades 9 – 12)	CPP	Classify programming languages based on their level and application domain
Level 3 A (Grades 9 – 12)	CPP	Use various debugging and testing methods to ensure program correctness (e.g., test cases, unit testing, white box, black box, integration testing)

### 7.1.3 Teaching Notes

Mastering these concepts and practices requires student to make several transitions. Students are required to move from using blocks to text. In addition, they will now need to begin programming in JavaScript which is harder programming language to program in as compared to CoffeeScript. Here are a series of steps that will facilitate this new learning.

Step 1: Have the students to choose a program that they have already coded and analyzed in block-mode and text-mode.

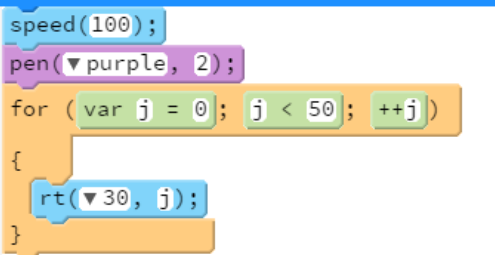

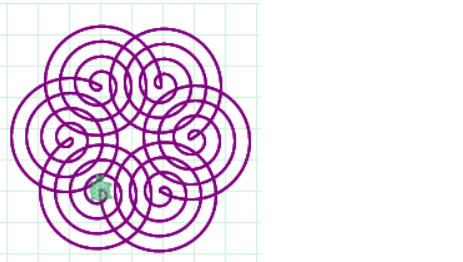
Step 2: Have the students begin pulling the blocks and arranging them (copying from an existing lab) and then have them switch to text-mode.

Step 3: Ask the students to add a new construct to the program that adds a small feature by typing it in the text editor and have them run the program.

Step 4: Encourage the students to keep adding small constructs and watching how the program responds. Encourage students to switch to block-mode and add a block if that would make them more adventurous and ready to try modifying the program; then switch back to text mode to become familiar with text syntax.

### 7.1.4 Lesson Plan I

Note: For this lesson, make sure you are in block mode. Type in the code (switch to block-mode if needed) and click the play arrow to demonstrate the results.

Content details	Teaching Suggestions	Time
<p>Code:</p> <pre> speed(100); pen(purple, 2); for (var j = 0; j &lt; 50; ++j) {   rt(30, j); } </pre> 	<p>Take the spiral program used the Loops chapter. Or this <a href="#">program</a>.</p> <p>Ask the students to use block-mode with settings in JavaScript mode.</p> 	<p>Demonstration: 10 minutes</p>
<p>Code:</p> <pre> speed(100); for(var i=0;i&lt;6;i++) {   pen(random(color), 2);   for (var j = 0; j &lt; 50; ++j)   {     rt(30, j);   } } </pre> 	<p>Switch to text-mode.</p> <p>Add an external loop to build 3 spirals.</p> <p>Teaching Tip: Change the number of iterations to 6, 8, and 12 and show how the patterns change.</p> <p>Talk about indentation, especially the opening and closing of curly braces. Program code <a href="#">here</a>.</p>	<p>Demonstration: 10 minutes</p>

## 7.1.5 Lesson Plan II

This lesson uses the Broken Sample Scene program. Having already worked on functions, students should now be ready to read code, isolate small snippets of code, and make the entire program work. This lesson helps students understand the iterative development cycle, that is, the process of adding small pieces of code to a large program.

Content Details	Teaching Suggestions	Time
<p><b>Code</b></p> <pre data-bbox="219 499 722 871">function mouse() {   dot(gray, 50);   fd(15);   rt(45);   box(gray, 30);   rt(135);   label('..',30);   fd(30);   pen(black, 5);   rt(100,30); }</pre> <pre data-bbox="219 934 722 1281">// MAIN PROGRAM STARTS HERE speed(10); dot(cyan, 1000); bk(1000); box(green, 2000); moveto(250, -70); mouse(); /*moveto (-300, -150); turnto (90); road (600); */</pre>	<p>Give the students the <a href="#">broken sample scene program</a>. Remind students to be in JavaScript mode and to stay in text-mode.</p> <p>Ask the students to run the program and fix the bugs in the code.</p> <p>Encourage the students to stay in text-mode while fixing the bugs. Provide the students with <a href="#">the movie</a> to demonstrate the running of the program.</p> <p>Teaching tips: Teach students to debug by adding one function at a time.</p> <p>Step 1: Only have the code to create the background, a call to the user-defined mouse function and the code for the mouse function. Run the code and have it work as desired.</p> <p>Step 2: Add the road () function. Show students how to comment code that is not being used so that it is available for future inclusion. (See sample code in the left column.)</p> <p>Step 3: Students can now keep un-commenting code and adding pieces to get the entire scene to work.</p>	<p>Demonstration: 30 minutes Student Practice: 1 class period.</p>

# Chapter 8: Introducing One-Dimensional Arrays

## 8.0.1 Objectives

One-dimensional arrays (also known as lists) are the fundamental data structure that allows a program to store many elements of data, using a linear arrangement. In this unit, students will learn how to create and traverse arrays, and how to add, remove, insert and search for elements in an array. Using Pencil Code, students will explore building arrays using data loaded from the internet, and how to create visualizations using data in an array.

## 8.0.2 Topic Outline

- 8.0 Chapter Introduction
  - 8.0.1 Objectives
  - 8.0.2 Topic Outlines
  - 8.0.3 Key Terms
  - 8.0.4 Key Concepts
- 8.1 Lesson Plans
  - 8.1.1 Suggested Timeline
  - 8.1.2 CSTA Standards
  - 8.1.3 Teaching Suggestions
  - 8.1.4 Lesson Plan I on using 'for... each' loop over arrays
  - 8.1.5 Lesson Plan II on traversing arrays – text-mode.
  - 8.1.6 Lesson Plan III on a demonstrating how an array behaves.
  - 8.1.7 Lesson Plan IV using arrays creating a bar graphs with an array of data.
  - 8.1.8 Lesson Plan V using arrays creating a pie chart with an array of data.
  - 8.1.9 Lesson Plan VI using arrays to search for an element in an array of data.

## 8.0.3 Key Terms

Linear data structures	Index starts at 0
Graphing charts	Size of an array - syntax
Range errors	
Index	
Content of an array	

## 8.0.4 Key Concepts

CoffeeScript and JavaScript differ in their support for iterating over arrays, so the presentation of arrays is slightly different depending on the language used. The key concepts in this unit are provided in two ways: once in CoffeeScript, and alternately in JavaScript; teachers may decide to present the concepts with one language or the other.

### Understanding Arrays in CoffeeScript

**Data structures** are objects that a computer program uses to organize more than one piece of information at a time. We have already seen data structures together with the “for” loop. (Note that creating this program requires working in text- mode. Try flipping to text-mode and editing “for” statement to look just like this.)

```
for x in [red, orange, yellow]
  dot x, 100
  fd 50
```



The object in square brackets [red, orange yellow] is a data structure called an **array**. The for loop iterates over the array, repeating the indented code while setting x to each one of the values in the array in turn.

The subtle thing about this code is that the array is its own object. The array can be put in its own variable and be used by name instead. For example, the program below is equivalent to the one above:

```
mycolors = [red, orange, yellow]
for x in mycolors
  dot x, 100
  fd 50
```

In this version, the variable mycolors contains the array of three colors. The array can be used for iterating a loop, but the same array can also be used in other ways.

### Basic Access: Array Indexing and Length

An array is called a **container** because it is a single object that contains other objects. Every array has a number of **elements** in a well-defined order, and every array has a **length**, which counts the number of elements in the array. Individual elements and the length can be fetched from them (they can always flip back to block-mode after typing the array parts in text.)

write mycolors.length	This prints "3", since there are three elements in the array.
write mycolors[0]	This prints "red", since the first element is red.
write mycolors[1]	This prints "orange".
write mycolors[2]	This prints "yellow".

The use of square brackets after the array number is called **indexing**, and the number inside the brackets is the **index** of a specific element in the array. Indexing can also be used to change an element of an array. For example, the following code changes the first color in the array:

```
mycolors[0] = blue
```

Changing one element has no effect on the other elements of the array.

### Zero-Based Indexing

Arrays in CoffeeScript and JavaScript and most other modern programming languages are zero-indexed, which means the first element of the array has index 0 (instead of 1). This also means that the last element of the array has index equal to length - 1.

Since people usually count starting at one, zero-indexing may seem counterintuitive at first. Starting at zero is a language design choice, and some older programming languages such as Fortran do use one-indexing. Many programmers experienced with both zero and one-based indexing contend that zero-based indexing is slightly clearer because it allows programmers to interpret the index as the distance the element would have to be moved to bring it to the beginning of the array. Since the starting element is already at the beginning, its distance, and its index, should be zero. (Another, more mathematical argument for zero-based indexing can be found on the Web by searching for Edger Dijkstra's discussion of zero-indexing.)

## Empty Arrays and Looping with Zero-Based Indexes

To create a loop that uses zero-based indexes in CoffeeScript, use the three-dot range form `[0..N]`, as shown on the second line of the following program:

```
mycolors = [red, orange, yellow]
for j in [0..mycolors.length]
  write element '#' + j + ' is ' + mycolors[j]
```

The three-dot form of a number range is called a “half closed range” which tries to start counting at the first number but which definitely omits the last number, so `[0..3]` counts `[0, 1, 2]`, exactly as needed for zero-based indexing.

The three-dot range has another advantage over the two-dot form: it works correctly when the length is zero! When counting to zero, instead of ranging from `[1..0]`, which would count backwards over two numbers, it ranges from `[0..0]`, which sets up to start at zero and yet omits the zero because it uses three dots. The result is an empty sequence, which means the loop will not be entered at all: exactly what we want for an empty array.

An empty array is a perfectly good and useful array! We can create an empty array by writing the following:

```
favoritecolors = []
write 'the length is ' + favoritecolors.length
```

The length of an empty array, of course, is zero.

## Making a Graph Using an Array

Arrays can contain any type of element such as numbers or strings. **Visualization** is a useful type of program that creates a graph using numbers in an array. Here is an example.

```
data = [2, 10, 3, 7]
rt 90
for j in [0..data.length]
  jumpto 0, 25 * j
  pen red, 20, 'butt'
  fd data[j] * 20
  label data[j], 'right'
hide()
```



This program uses `[0..data.length]` to count up the index `j` from 0 to 3, then it uses `data[j]` to read one element out of the array at a time. These numbers are used with turtle functions to draw the bar graph.

This program uses some additional arguments to the `pen` and `label` function to control formatting precisely. The `pen` is given the `'butt'` option, which is a graphics term that requests a flat squared-off line ending rather than a rounded line. And the `label` is given the `'right'` option, which places the label to the right of the position instead of directly on the turtle.

## Creating Arrays from Strings and Files

Arrays are wonderfully powerful because a single array object can contain many thousands or millions of elements. However, to do this in a practical way, the array needs to be loaded from a data file outside of the program.

To try the next experiment, create a file inside Pencil Code and change its name to “mydata.txt”. As soon as you give it a name ending with “.txt”, Pencil Code will know it is not a regular program, and it will expect a plain text data file. Save a series of numbers in the file with no spaces, just separated by commas, like this:

```
96,73,93,95,85,89,85,99,79,75,89,82,90,85,84,85,88,95,78,96,91,93
```

Any numbers can be used; perhaps this is a series of test scores.

Here is an example program that loads data from a file into an array and calculates basic statistics with it.

```
await load 'mydata.txt', defer textdata
mydata = textdata.split(',')
total = 0
for j in [0..mydata.length]
  mydata[j] = Number(mydata[j])
  total += mydata[j]
write 'Total: ' + total
write 'Average: ' + total / mydata.length
```

### Three Steps for Loading a File into an Array

As illustrated in the program above, loading an array from a file takes two or three steps.

1. **Load** a file as a single large string of text data: (await load 'mydata.txt', defer textdata)
2. **Split** the file into an array of smaller strings, one for each element of data: (mydata = textdata.split(','))
3. (If the data are numbers) **Convert** the strings to numbers: (mydata[j] = Number(mydata[j]))

The function `load` loads a file URL from the Internet and calls a callback function with the content of the file as a single string. The program here loads a short filename “mydata.txt” that will be located in the same Pencil Code directory that the program is running. By using a full URL starting with “http://”, however, Pencil Code can load any data file from the Internet. Note that `load` is a form of input (from the network instead of from the user), and it works just like the “`read`” function from the I/O chapter. Here we combine `load` with `await` to put the program on hold while waiting for the callback.

The function `split` divides a string of text into an array of strings by dividing it up at a **delimiter** character. The delimiter can be any letter or pattern. For example, to split a file with one entry per line, split using `\n` (backslash n is the code for the “newline” character that appears at the end of a line in a text file).

The function `Number` converts a string to a number. To avoid having “96” + “73” result in the answer “9673”, the loaded strings must be converted to numbers before doing arithmetic with them.

### Search: Splitting a Document to an Array, then Joining It Again

Here is another example program that finds words in a file. It uses the `split` function with a special pattern to split the file at all word boundaries and uses the `join` function to join the array back together as one big string to print.

To prepare data for this program, save a file called “document.txt” containing any amount of text such as a paragraph copied from Wikipedia or a public-domain book.

```
await load 'document.txt', defer textdata
words = textdata.split(/\b/)
await read 'A word to search for?', defer q
for j in [0...words.length]
  if words[j] is q
    words[j] = '<mark>' + q + '</mark>'
write words.join('')
```

This program does four things with the array:

1. It creates the array using `split(/\b/)`. The special pattern `/\b/` splits the string at every word boundary.
2. It examines each word using the test `words[j] is q`, to try to find a match.
3. For matching words, it adds a formatting code using an HTML tag. `words[j] = '<mark>' + q + '</mark>'`
4. The `words.join('')` function then joins all the elements of the array back together in a single string for writing.

The result looks something like this:

```
A word to search for? who
There once lived, in a sequestered part of the county of Devonshire, one
Mr. Godfrey Nickleby: a worthy gentleman, who, taking it into his head
rather late in life that he must get married, and not being young enough
or rich enough to aspire to the hand of a lady of fortune, had wedded an
old flame out of mere attachment, who in her turn had taken him for the
same reason. Thus two people who cannot afford to play cards for
money, sometimes sit down to a quiet game for love.
```

Arrays make it possible to write programs that work with big data such as large volumes of numbers or text.

## ALTERNATE DISCUSSION USING JAVASCRIPT

### Understanding Arrays in JavaScript

**Data structures** are objects that a computer program uses to organize more than one piece of information at a time. For example, in the code below, the object `mycolors` is an array that contains three colors.

```
var mycolors = [red, orange yellow];
write(a[0]);
write(a[2]);
```

When running the program, it prints the first color and the last one: “red” and “yellow”.

### Basic Access: Array Indexing and Length

An array is called a **container** because it is a single object that contains other objects.

Every array has a number of **elements** in a well-defined order and every array has a **length**, which counts the number of elements in the array. Individual elements and the length can be fetched from an array as shown below.

<code>write(mycolors.length);</code>	This prints “3”, since there are three elements in the array.
<code>write(mycolors[0]);</code>	This prints “red”, since the first element is red
<code>write(mycolors[1]);</code>	This prints “orange”
<code>write(mycolors[2]);</code>	This prints “yellow”

(Note again that using array syntax in Pencil Code requires students to code in text-mode; but they can always flip back to block-mode after typing the array parts in text.)

The use of square brackets after the array number is called **indexing** and the number inside the brackets is the **index** of a specific element in the array. Indexing can also be used to change an element of an array. For example, the first color can be changed as follows:

```
mycolors[0] = blue;
```

Changing one element has no effect on the other elements of the array.

### Zero-Based Indexing

Arrays in CoffeeScript and JavaScript and most other modern programming languages are zero-indexed, which means the first element of the array has index 0 (instead of 1). That also means that the last element of the array has index equal to `length - 1`.

Since people usually count starting at one, zero-indexing may seem counterintuitive at first. Starting at zero is a language design choice and some older programming languages such as Fortran do use one-indexing. Many programmers experienced with both zero and one-based indexing contend that zero-based indexing is slightly clearer because it allows programmers to interpret the index as the distance the element would have to be moved to bring it to the beginning of the array. Since the starting element is already at the beginning, its distance, and its index, should be zero. (Another, more mathematical argument for zero-based indexing can be found on the Web if by searching for Edger Dijkstra’s discussion of zero-indexing.)

### Empty Arrays and Looping with Zero-Based Indexes

The conventional form of a JavaScript for loop works equally well for empty and nonempty arrays. The code below will repeat the loop zero times if `mycolors` is changed to be an empty array.

```
mycolors = [red, orange, yellow];
for (var j = 0; j < mycolors.length; ++j) {
  write element #' + j + ' is ' + mycolors[j];
}
```

An empty array is a perfectly good and useful array. An empty array can be created as shown below.

```
favoritecolors = [];
write('the length is ' + favoritecolors.length);
```

The length of an empty array is zero.

## Making a Graph Using an Array

Arrays can contain any type of element such as numbers or strings. One very useful type of program is a **visualization** that creates a graph using numbers in an array. Here is an example.

```
data = [2, 10, 3, 7];
rt(90);
for (var j = 0; j < data.length; ++j) {
  jumpto(0, 25 * j);
  pen(red, 20, 'butt');
  fd(data[j] * 20);
  label(data[j], 'right');
}
hide();
```



This program the index  $j$  from 0 to 3, then it uses `data[j]` to read one element out of the array at a time. These numbers are used with turtle functions to draw the bar graph.

This program uses some additional arguments to the `pen` and `label` function to control formatting precisely. The `pen` is given the `'butt'` option, which is a graphics term that requests a flat squared-off line ending rather than a rounded line. The `label` is given the `'right'` option, which places the label to the right of the position instead of directly on the turtle.

## Creating Arrays from Strings and Files

Arrays are wonderfully powerful because a single array object can contain many thousands or millions of elements. However, to do this in a practical way, the array needs to be loaded from a data file from outside the program.

To try the next experiment, create a file inside Pencil Code and change its name to "mydata.txt". As soon as you give it a name ending with ".txt", Pencil Code will know it is not a regular program and will expect a plain text data file. Save a series of numbers in the file with no spaces, just separated by commas as follows.

```
96,73,93,95,85,89,85,99,79,75,89,82,90,85,84,85,88,95,78,96,91,93
```

Any numbers can be used; perhaps this is a series of test scores.

Here is an example program that loads data from a file into an array and calculates basic statistics with it.

```
load('mydata.txt', function (textdata) {
  var mydata = textdata.split(',');
  var total = 0;
  for (var j; j < mydata.length; ++j) {
    mydata[j] = Number(mydata[j]);
    total += mydata[j];
  }
  write('Total: ' + total);
  write('Average: ' + total / mydata.length);
});
```

## Three Steps for Loading a File into an Array

As illustrated in the program above, loading an array from a file takes two or three steps.

1. **Load** a file as a single large string of text data: (await load 'mydata.txt', defer textdata)
2. **Split** the file into an array of smaller strings, one for each element of data: (mydata = textdata.split(','))
3. (If the data are numbers) **Convert** the strings to numbers: (mydata[j] = Number(mydata[j]))

The function `load` loads a file URL from the Internet and calls a callback function with the content of the file as a single string. The program here loads a short filename “mydata.txt” that will be located in the same Pencil Code directory that the program is running. By using a full URL starting with “http://”, however, Pencil Code can load any data file from the Internet. Note that `load` is a form of input (from the network instead of from the user), and it works just like the “`read`” function from the I/O chapter. Here we combine `load` with `await` to put the program on hold while waiting for the callback.

The function `split` divides a string of text into an array of strings by dividing it up at a **delimiter** character. The delimiter can be any letter or pattern. For example, to split a file with one entry per line, split using `\n` (backslash n is the code for the “newline” character that appears at the end of a line in a text file).

The function `Number` converts a string to a number. To avoid having “96” + “73” result in the answer “9673”, the loaded strings must be converted to numbers before doing arithmetic with them.

### Search: Splitting a Document to an Array, then Joining it Again

Here is another example program that finds words in a file. It uses the `split` function with a special pattern to split the file at all word boundaries, and it uses the `join` function to join the array back together as one big string to print.

To prepare data for this program, save a file called “document.txt” containing any amount of text, for example a paragraph copied from Wikipedia or a public-domain book.

```
load('document.txt', function (textdata) {
  var words = textdata.split(/\b/);
  read('A word to search for?', function(e) {
    for (var j = 0; j < words.length; ++j) {
      if (words[j] == q) {
        words[j] = '<mark>' + q + '</mark>';
      }
    }
    write(words.join(''));
  });
});
```

This program does four things with the array:

1. It creates the array using `split(/\b/)`. The special pattern `\b/` splits the string at every word boundary.
2. It examines each word using the test `words[j] is q`, to try to find a match.
3. For matching words, it adds a formatting code using an HTML tag. `words[j] = '<mark>' + q + '</mark>'`.
4. Then the `words.join("")` function joins all the elements of the array back together in a single string for writing.

The result looks something like this:

```
A word to search for? who
There once lived, in a sequestered part of the county of Devonshire, one
```

Mr. Godfrey Nickleby: a worthy gentleman, **who**, taking it into his head rather late in life that he must get married, and not being young enough or rich enough to aspire to the hand of a lady of fortune, had wedded an old flame out of mere attachment, **who** in her turn had taken him for the same reason. Thus two people **who** cannot afford to play cards for money, sometimes sit down to a quiet game for love.

Arrays make it possible to write programs that work with big data such as large volumes of numbers or text.

### 8.1.1 Possible Timeline: 1 (55-minute class period)

Instructional Day	Topic
2 Day	Lesson Plan I
1 Day	Lesson Plan II
1 Day	Lesson Plan III
1 Day	Lesson Plan IV & V
1 Day	Lesson Plan VI

### 8.1.2 Standards

CSTA Standards	CSTA Strand	CSTA Learning Objectives Covered
Level 3 A (Grades 9 – 12)	Computational Thinking (CT)	Explain how sequence, selection, iteration, and recursion are building blocks of algorithms.
Level 3 A (Grades 9 – 12)	CT	Compare techniques for analyzing massive data collections.
Level 3 A (Grades 9 – 12)	Collaboration (CL)	Describe techniques for locating and collecting small and large-scale data sets.
Level 3 B (Grades 9 – 12)	CL	Deploy various data collection techniques for different types of problems.
Level 3 B (Grades 9 – 12)	CT	Compare and contrast simple data structures and their uses (e.g., arrays and lists).
Level 3 B (Grades 9 – 12)	Computers and Communication Devices (CD)	Discuss the impact of modifications on the functionality of application programs.

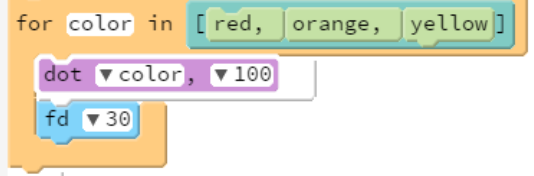
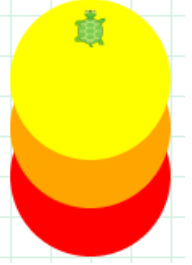
### 8.1.3 Teaching Suggestions

Encourage students to toggle between text-mode and block-mode to constantly extend themselves beyond their comfort level. The lesson plan provides suggestions to toggle between modes. It is best to that the students use block-mode when trying they are trying to understand the overall flow of logic. At other times, though, it is more convenient to type out various keywords in text since the block-mode may not have all the blocks that are needed. This is important because typing in text enable access to the richer library available in JavaScript.



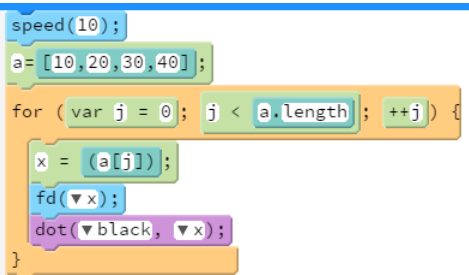
### 8.1.4 Lesson Plan I

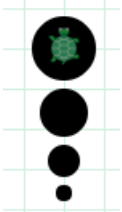

This lesson introduces arrays in CoffeeScript.

Content details	Teaching Suggestions	Time
<p><u>Code:</u></p>  <p><u>Code in text:</u></p> <pre>for color in [red, orange, yellow]   dot color, 100   fd 30</pre> <p><u>Output:</u></p> 	<p>Pull up the <a href="#">program</a> “rainbow Colors” and demonstrate the running of the program.</p> <p>Point out that this was also used in Chapter 4, Lesson Plan III.</p> <p>Explain that color is a variable that is traversing an array of colors red, orange and yellow.</p> <p>Between the [ ] brackets are the various colors that are stored in the data structure arrays. The command ‘dot’ takes values in the arrays and draws the dot of the value of the color.</p> <p>Encourage students to play with various colors.</p> <p><u>Variant:</u> Instead of color being the changing value the program could access an array of dot sizes. The for loop would look like this:</p> <pre>for x in [10,20,30,40]   dot red,x   fd 30</pre>	<p>Demonstration: 15 minutes</p> <p>Student Practice: 20 minutes.</p>

### 8.1.5 Lesson Plan II

This first introduction to arrays shows students how to traverse simple arrays with a list of known elements. The program draws an element of the size as shown in the array. Type the code as shown (in block and text-mode) or pull up the code and walk the students through the code and explain the concepts to them.

Content details	Teaching Suggestions	Time
<p><u>Code: Blocks</u></p> 	<p>Type the <a href="#">code</a> (or pull it up on the projector) and walk the students through the code.</p> <p>Show how the array is declared.</p> <p>Explain length gives the size of the array.</p> <p>Explain why an array starts a position 0 and that the loop traverses through the array.</p> <p>Explain that x represents the value in the</p>	<p>Demonstration: 30 minutes</p>

<p><u>Code: Text</u></p> <pre> speed(10); a=[10,20,30,40]; for (var j = 0; j &lt; a.length; ++j) {   x = (a[j]);   fd(50);   dot(black, x); } </pre> <p>Array values: 10,20,30,40</p> 	<p>array at that position. So when <math>j = 0</math>, <math>x = 10</math>. <math>j=3</math>, <math>x= 40</math>. Each value of <math>x</math> the turtle moves forward by that many blocks and the size of the dot drawn is also dependent on that value of the <math>x</math>.</p> <p>Teaching Tip: A fun exercise would be to have the students change the values in the array so that they are not sequential. This will enable students to watch the size change depending on the position of the index in the array. Ask them to use big numbers such as 10, 50, 15, and 30 and watch the dot go bigger and smaller alternatively.</p> <p>Array values: 10, 50, 15, 39</p> 	
---	---	--

### 8.1.6 Lesson Plan III

This lesson plan shows how an element is added or removed from an array using the stack concept of Last In First Out (LIFO). This demo program can be used to show the students how elements in an array get added and removed. Use the pop feature to demonstrate that, even if the element is removed, the space created by the array for that element is unclaimed. Distribute the program among the students and let them experiment with the program to understand array behavior and stack properties.

Content details	Teaching Suggestions	Time
<p><u>Code</u></p> <pre> var stack = []; pen(green, 25); speed(Infinity); button('F', function() {   fd(10); }); button('R', function() {   rt(30); }); button('Push', function() {   dot(crimson, 50);   var record = {     xy: getxy(),     dir: direction()   }; } </pre>	<p>Use this <a href="#">program</a> to demonstrate basic array functionality.</p> <p>Use the F button to move the turtle forward.</p> <p>Use the Push to create a red dot on the screen, representing an element added to the array.</p> <p>Use the Pop to create the pink dot. This lightens the red dot. This example can be used to represent an element being removed from the array.</p> <p>Use the combination of the 'F' button and</p>	<p>Demonstration: 20 minutes.</p> <p>Student Practice: 55 minutes</p>

```

    stack.push(record);
  });
  button('Pop', function() {
    if (!stack.length) {
      home();
      return;
    }
    var record = stack.pop();
    jumpto(record.xy);
    turnto(record.dir);
    dot(pink,50);
  });

```

### Code: Blocks

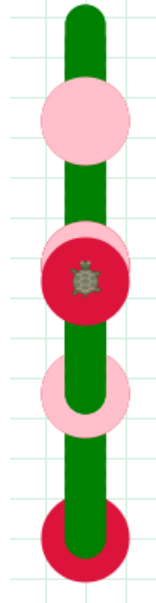
```

stack = [];
pen(▼green, 25);
speed(Infinity);
button('F', function() {
  fd(▼10);
});
button('Push', function() {
  dot(▼crimson, ▼50);
  stack.push[ getxy() ];
});
button('Pop', function() {
  if ( ! stack.length ) {
    home();
    return;
  }
  //[xy, b] = stack.pop();
  jumpto(▼xy);
  turnto(▼b);
  dot(▼pink, ▼50);
});

```

the 'Push' button to demonstrate that the elements in the array can be placed anywhere in the array structure as long as there is an integer index position to hold the element.

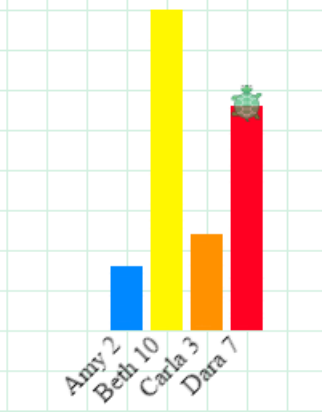
### Output:



Extension Activity: Ask students to create their own versions of the two programs. Give them copies of both programs so they can tinker with them. Encourage students to create their own versions of the 1<sup>st</sup> program (you can choose to give program 1 as a lab activity for them to design on their own.) The 2<sup>nd</sup> program is intended for demonstration purposes only. You can, however, ask the students to tinker with it to help them better understand how the program works. **(55 minutes).**

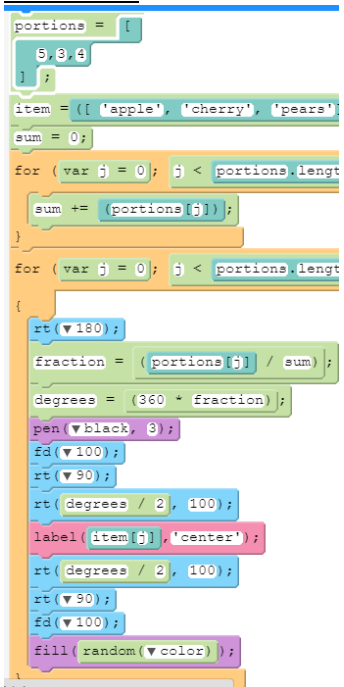
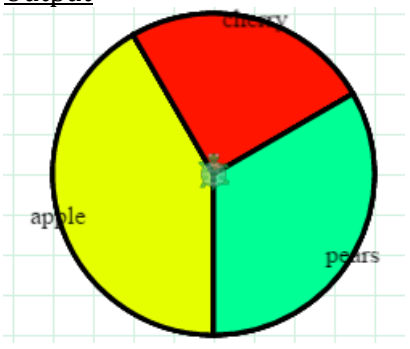
### 8.1.7 Lesson Plan IV

This lesson plan demonstrates the use of arrays to hold data. Students will learn to traverse arrays that hold data to generate data graphs. This lesson shows how a bar graph can be drawn from the two arrays that are in the program. The lesson leverages the feature of Pencil Code to be able to generate visual feedback during execution.

Content details	Teaching Suggestions	Time
<p><b>Output:</b></p>  <p><b>Text Code:</b></p> <pre>data = [2, 10, 3, 7]; labels = ['Amy', 'Beth', 'Carla', 'Dara']; for (var k = 0; k &lt; data.length; ++k) {   jumpto(25 * k, 0);   lt(45);   label(labels[k] + ' ' + data[k], 'bottom left rotated');   rt(45);   pen(random(color), 20, 'butt');   fd(data[k] * 20); }</pre> <p><b>Block Code:</b></p> <pre>data = [2, 10, 3, 7]; labels = ['Amy', 'Beth', 'Carla', 'Dara']; for (var k = 0; k &lt; data.length; ++k) {   jumpto(25 * k, 0);   lt(45);   label(labels[k] + ' ' + data[k], 'bottom left rotated');   rt(45);   pen(random(color), 20, 'butt');   fd(data[k] * 20); }</pre>	<p>Walk the students through the <a href="#">program code</a>.</p> <p>Step 1: Run the program and demonstrate the output.</p> <p>Step 2: Point the two 1-D arrays with the data in it.</p> <p>Step 3: Labels forms the x-axis data. Data forms the y-axis data and determine the length of the bars.</p> <p>Step 4: The random color generator decides the color on the graphs.</p> <p>Step 5: Show the loop in a program. Point to students how the loop traverses the array. (data.length)</p> <p>Step 6: Show students the movement of the turtle on the y-axis to a constant value * the value in the data array.</p> <pre>fd(data[k] * 20);</pre> <p>Step 7: Show students the movement of the turtle on the x-axis to a constant value * the value in the data array.</p> <pre>jumpto(25 * k, 0); lt(45);</pre> <pre>jumpto(25 * k, 0); lt(45);</pre>	<p>Demonstration: 30 minutes</p> <p>Students Practice: 25 minutes</p>
<p><b>Extension Activity:</b> Ask students to create different arrays with different types of data values and use the code provided to create the bar graph. <b>(55 minutes)</b></p>		

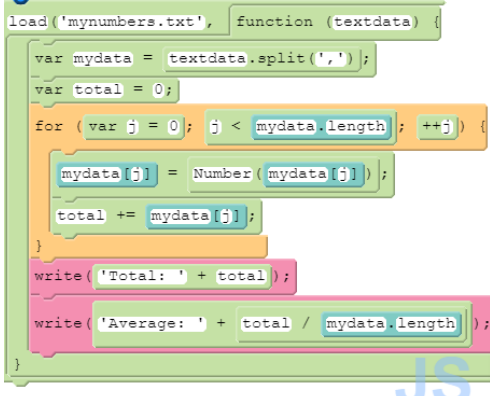
## 8.1.8 Lesson Plan V

In this lesson the students will create a Pie Chart using simple data stored in a one-dimensional array. This lesson is similar to the bar graph one. The data graph drawn from an array is different.

Content details	Teaching Suggestions	Time
<p><b>Text Code:</b></p> <pre>portions = [ 5,3,4]; item =(['apple', 'cherry', 'pears']); sum = 0; for (var j = 0; j &lt; portions.length; ++j) {   sum += (portions[j]); } for (var j = 0; j &lt; portions.length; ++j) {   rt(180);   fraction = (portions[j] / sum);   degrees = (360 * fraction);   pen(black, 3);   fd(100);   rt(90);   rt(degrees / 2, 100);   label(item[j], 'center');   rt(degrees / 2, 100);   rt(90);   fd(100);   fill(random(color)); } </pre> <p><b>Block Code:</b></p> 	<p>Step 1: Run the <a href="#">program</a> to demonstrate the creation of a pie chart.</p> <p>Step 2: Change values in the array to show how the changes are reflected in the resulting graph.</p> <p>Step 3: Walk through the code to show how the pie chart is computed.</p> <p>Step 4: Formula to calculate the fractions</p> $\text{fraction} = \left( \frac{\text{portions}[j]}{\text{sum}} \right);$ $\text{fraction} = (\text{portions}[j] / \text{sum});$ <p>Step 5: Formula to calculate the degrees.</p> $\text{degrees} = (360 * \text{fraction});$ $\text{degrees} = (360 * \text{fraction});$ <p><b>Output</b></p> 	<p>Demonstration: 30 minutes</p> <p>Student Practice: 1 class period</p>
<p><b>Extension Activity:</b> As students to create different arrays with different types of data values and use the code provided to create various pie charts depicting different types of data. <b>(55 minutes)</b></p>		

### 8.1.9 Lesson Plan VI

The lesson plan illustrates how to search for an element in a text file. The lesson plan has two parts. The first part involves showing students how to load (open) a file and read it. The second part has a traversal code that searches for an element in the file.

Content details	Teaching Suggestions	Time
<p><b>Code: Text</b></p> <pre>load('mynumbers.txt', function (textdata) {   var mydata = textdata.split(',');   var total = 0;   for (var j = 0; j &lt; mydata.length; ++j) {     mydata[j] = Number(mydata[j]);     total += mydata[j];   }   write('Total: ' + total);   write('Average: ' + total / mydata.length); });</pre>	<p>Step 1: Pull up the <a href="#">SearchingNumbers</a> program.</p> <p>Step 2: Remember to stay in JavaScript mode.</p> <p>Step 3: Run the program and demonstrate the output.</p> <p>Step 4: Walk through the code.</p> <p>Step 5: Explain file-opening. Refer to the key concepts when needed to reiterate the syntax.</p>	<p>Demonstration: 30 minutes</p> <p>Students Practice: 60 minutes</p>
<p><b>Code: Blocks</b></p>  <p><b>Output:</b></p> <pre>Total: 15 Average: 3</pre>	<p>Step 6: The data file (“mynumbers.txt”) is to be located in the same directory as that of the program. Students can create their own data files by creating a new file in Pencil Code and copy/pasting data into it and then clicking the ‘Save’ button.</p> <p>Step 7: Now explain how the data is stored in an array and the program searches for a match and when it finds it the count is increased.</p> <p>Step 8: Trace the code with the ‘for loop’ and explain the data that is traversed.</p> <p>Note: Students can trace the values of mydata[j], j and total and share it with the class.</p>	

Code: Text

```
load('mydata.txt', function
(textdata) {
  var words =
textdata.split(/\b/);
  read('A word to search for?',
function(q) {
  for (var j = 0; j <
words.length; j++) {
    if (words[j] == q) {
      words[j] = '<mark>' + q +
'</mark>';
    }
  }
  write(words.join(''));
});
});
```

Code: Block

```
load('mynumbers.txt', function (textdata) {
  var mydata = textdata.split(',');
  var total = 0;
  for (var j = 0; j < mydata.length; ++j) {
    mydata[j] = Number(mydata[j]);
    total += mydata[j];
  }
  write('Total: ' + total);
  write('Average: ' + total / mydata.length);
}
```

Step 1: Pull up SearchingText [program](#).

Step 2: Run the program

Step 3: Ask the students to walk through the code and explain it to you.

Note: Both programs use the split () function. For a good explanation on split and how to use it, see: [http://www.w3schools.com/jsref/jsref\\_split.asp](http://www.w3schools.com/jsref/jsref_split.asp)

Demonstration: 30 minutes.

Student Practice: 35 minutes

Output:

A word to search for? Betty  
Betty, bought, some butter. But, the butter was bitter. &  
better. But the bitter butter, made the better butter bitte

# Chapter 9: Nested Loops

## 9.0.1 Objectives

The chapter has two goals. The first goal is to help introduce the concept of nested loops and show students how to build them. The lessons teach nested loops using ASCII art. The second goal is to introduce the concept of a two-dimensional (2D) output grid. The most natural way to traverse a 2D grid is using nested loops. This makes nested loops a powerful construct for students to learn and master.

## 9.0.2 Topic Outline


- 9.0 Chapter Introduction
  - 9.0.1 Objectives
  - 9.0.2 Topic Outlines
  - 9.0.3 Key Terms
  - 9.0.4 Key Concepts
- 9.1 Lesson Plans
  - 9.1.1 Teaching Suggestions
  - 9.1.2 Suggested Timeline
  - 9.1.3 CSTA Standards
  - 9.1.4 Lesson Plan I on using understanding nested loops
  - 9.1.5 Lesson Plan II on traversing 2D grids to create ASCII Art.

## 9.0.3 Key Terms

Index	Tables
Matrix	ASCII Art
Nesting	Tracing
Animation	Inner and outer loop

## 9.0.4 Key Concepts

When loops are **nested** within loops, a program can create a repetition of repetitions. Below is an example.

<pre>for (var j = 0; j &lt; 5; ++j) {   for (var k = 0; k &lt; 8; ++k) {     typebox(random(color));   }   typeline(); }</pre>	<p>Output:</p> 
--	---

*An example of using a nested loop.*

## Inner Loops and Outer Loops

When nesting loops, the two loops play very different roles:

- The **outer loop** starts first and finishes last. It starts once and finishes once. In the example above, the outer loop uses the variable `j` to repeat its body five times, running the inner loop, then doing a `typeline()` exactly 5 times, once at the end of each row of the output.



- The **inner loop** starts last and finishes first. It can start repeatedly and finish repeatedly. Here, the inner starts looping 5 times, each time starting k at 0 and counting to 8, to do typebox 8 times. By the time the program is done, the inner loop will have repeated  $5 \times 8 = 40$  times.

Inner loops spin around quickly, finishing a complete loop for each single iteration of the outer loop. Outer loops spin around slowly.

The hands of a clock work like inner and outer loops. For example, the second-hand of a clock acts like an inner loop and the minute-hand acts like an outer loop. For every single click of a minute hand, the second hand spins all the way around, clicking through all 60 seconds. By the time a minute hand clicks through one whole hour, the second hand has spun through  $60 \times 60 = 3600$  seconds.

Odometers and calendars work the same way.

### Dependent Inner Loops

Sometimes, the way the inner loop works differs depending upon which step of the outer loop is running. For example, to create a program that prints all the dates in a year, you can use a nested loop for the days of the month, but the number of times the inner loop runs depends upon the outer loop, since each month has a different number of days.

```
lengths = [31, 27, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31];
for (var month = 0; month < 12; month++) {
  for (var day = 0; day < lengths[month]; day++) {
    write((month + 1) + "/" + (day + 1));
  }
}
```

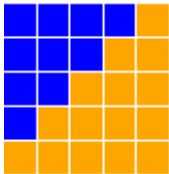
*A nested loop that prints every date in a year.*

Here the inner loop repeats a different number of times, depending on which month it is on, because its ending condition is `day < lengths[month]`.

For months, we charted out a dependent inner loop by just listing the days in each month in an array, but the rule for coming up with a dependent inner loop can require more generalization: a programmer needs to look at the specifics, and try to generalize to a rule.

### Generalizing a Rule for Inner Loops

Suppose we want to create a program that can create patterns like the program below.

<pre>N = 5 for (var row = 0; row &lt; N; row++) {   for (var b = 0; b &lt; ??; b++) {     typebox(blue);   }   for (var j = 0; j &lt; ??; j++) {     typebox(orange);   }   typeline(); }</pre>	<p>Desired output:</p> 
---	--

*Planning a dependent inner loop: how do we choose the loop bounds in red?*

The output we want is a 5x5 arrangement of colored boxes but we want to **generalize** our program so that it can make a similar pattern of any size. Since each line has a number of blue boxes followed by a

number of orange boxes, we can plan to have a nested loop like the program on the right, with one outer loop for each line, and two inner loops, one for the blue boxes, and one for the orange boxes.

Determining the values to put in for the question marks requires generalization. You can do this by looking at the specific output needed and finding patterns. Here is a table summarizing each row of the shape.

Row number	How many blue boxes	How many orange boxes	A formula for blue	A formula for orange
row = 0	4	1	$4 = N - 1 - \text{row}$	$1 = \text{row} + 1$
row = 1	3	2	$3 = N - 1 - \text{row}$	$2 = \text{row} + 1$
row = 2	2	3	$2 = N - 1 - \text{row}$	$3 = \text{row} + 1$
row = 3	1	4	$1 = N - 1 - \text{row}$	$4 = \text{row} + 1$
row = 4	0	5	$0 = N - 1 - \text{row}$	$5 = \text{row} + 1$

You can fill in the first three columns of the table by looking at the example shape and counting. But to generalize, you need to find rules that work in the last two columns. This requires finding a rule that relates the number of boxes on a row to the row number. For example, the number of orange boxes on each line is always one more than the row number. So a formula for the number of orange boxes that works on every line is  $\text{row} + 1$ .

For the blue boxes, the number decreases by one each time the row number increases by one, so the rule should involve subtraction. In this example, a rule that works is  $4 - \text{row}$ . But this is not quite generalized yet. Imagine a larger pattern with 10x10 boxes and  $N = 10$ . In this case the number of blue boxes on row #0 would not be 4; it would be 9! So a fully generalized formula that works for every row of every size  $N$  would be  $N - 1 - \text{row}$ .

Filling in the formulas so that the two inner loop conditions are  $b < \text{row} + 1$  and  $j < N - 1 - \text{row}$  completes the program so that it works for any  $N$ .

Nested loops can be challenging to program correctly because of the relationship between inner loops and outer loops. A good strategy for designing nested loops is to carefully chart out the behavior you want in a single example, and then find a way to generalize it.

### 9.1.1 Teaching Suggestions

The following lesson plans are structured so that the students master the idea of nesting loops. Encourage the students to trace through the values and then introduce the idea of 2D grids. Ensure that Pencil Code is in JavaScript mode. As much as possible encourage students to stay in text-mode and type in their code. They can switch to block-mode (when needed) and drag and drop the blocks into the program.

### 9.1.2 Possible Timeline: 1 55-minute class period


Instructional Day	Topic
1 Day	Lesson Plan I
2 Day	Lesson Plan II

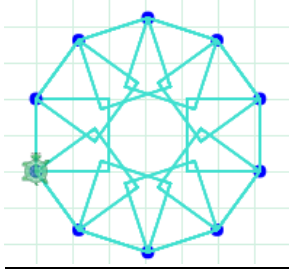
### 9.1.3 Standards

CSTA Standards	CSTA Strand	CSTA Learning Objectives Covered
Level 3 B (Grades 9 – 12)	Collaboration (CL)	Use project collaboration tools, version control systems, and Integrated Development Environments (IDEs) while working on a collaborative software project.
Level 3 A (Grades 9 – 12)	Computational Thinking (CT)	Explain how sequence, selection, iteration, and recursion are building blocks of algorithms.

### 9.1.4 Lesson Plan I

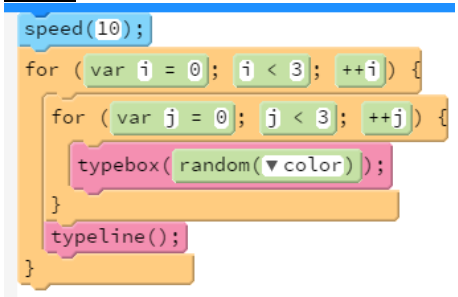

This lesson introduces nested loops. There are two programs. There are a two different teaching tools that have been demonstrated here. The first one is the idea of showing the logical path that the program takes. Tracing the code. Run the video to show the students how the loop execution takes place. The second concept is to ask simple questions on code modifications to elaborate what the role of each line of code is. It should be possible to take the two ideas and use them elsewhere to teach some other concepts.

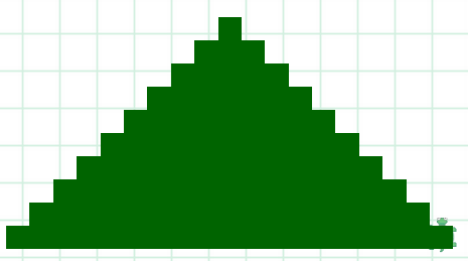

Content details	Teaching Suggestions	Time
<p>Code:</p> <pre>speed(100); for (var i = 0; i &lt; 10; ++i) {   for (var j = 0; j &lt; 19; ++j) {     typebox(purple);   }   typeline(); }</pre> <p>Output</p> 	<p>Use the <a href="#">code</a> that is provided in the left-most column.</p> <p>Switch to text-mode.</p> <p>Open the program and demonstrate how a nested loop works.</p> <p>Run the program and show how, for every one iteration of the outer loop, the inner loop completes all iterations.</p>	<p>Demonstration: 30 minutes</p>
<p>Code:</p> <pre>speed(1); pen(purple, 1); for (var i = 0; i &lt; 10; ++i) {   dot(blue, 10);   for (var j = 0; j &lt; 4; ++j)   {</pre>	<p>Have students pull up the decorated nest <a href="#">program</a>.</p> <p>Encourage students to stay in text-mode. As they watch the program on their monitors, ask students to answer the following questions. (Questions can be projected, written on a whiteboard, or handed</p>	<p>Demonstration: 30 minutes</p>

Content details	Teaching Suggestions	Time
<pre> fd(50); rt(90); } lt(36); bk(50); } </pre> <p><u>Output:</u></p> 	<p>as a worksheet to be filled in.)</p> <ol style="list-style-type: none"> <li>What does the inner loop create?</li> <li>How many blue dots are created?</li> <li>Which loop are the command lt and bk a part of?</li> <li>How many times do they get executed?</li> <li>What happens in the inner loop iterations are increased?</li> </ol> <p>Give students about five minutes to answer the questions and then, as a large group discussion, have students share their answers. (Alternatively, this can be given as homework or warm-up for the next day)</p>	
<p>As an extension to the decorated nest program, ask the students to modify the 'for' loop, for example by using additional fd and rt instructions to create interesting patterns.</p> <p>Teaching Tip: Award extra points for the project voted most creative by the class. Student Practice: 55 minutes</p>		

### 9.1.5 Lesson Plan II

This lesson focuses on building a simple 2D output using nested loops. It is fun and colorful and is great stepping stone to creating ASCII and text art.

Content details	Teaching Suggestions	Time
<p><u>Code:</u></p>  <pre> speed(10); for (var i = 0; i &lt; 3; ++i) {   for (var j = 0; j &lt; 3; ++j) {     typebox(random(color));   }   typeline(); } </pre>	<p>Give students <a href="#">the program</a> that in the left-most column. Have them toggle between text and block to understand the logic.</p> <p>Ask them modify the program using possible variations such as:</p> <ul style="list-style-type: none"> <li>Number of iterations of i and j</li> <li>Color – no variations of color</li> </ul> <p>Ask them what happens if the number of iterations of j and i are not the same?</p> <p><u>Output</u></p> 	<p>Demonstration: 20 minutes</p>

Content details	Teaching Suggestions	Time
<p>Code:</p> <pre> speed(500); for (var i = 0; i &lt; 10; ++i) {   for (var j = 0; j &lt; 9-i; ++j)   {     typebox(transparent);   }   for (var j = 0; j &lt; 1+i*2; ++j) {     typebox(darkgreen);   }   typeline(); } </pre>	<p>Have students use the <a href="#">code</a> in the left-most column to design a triangle.</p> <p>Next, show the students how using <code>typebox()</code> to color a cell <code>typeline()</code> enables them to create ASCII / TEXT art designs.</p> <p>(Students can toggle between text and text-mode if needed.)</p> <p><u>Output</u></p> 	<p>Demonstration: 30 minutes</p>
<p>Code</p> <pre> speed(500); dot(deepskyblue, 5000); moveto(6, 70); for (var i = 0; i &lt; 10; ++i) {   for (var j = 0; j &lt; 9-i; ++j)   {     typebox(transparent);   }   for (var j = 0; j &lt; 1+i*2; ++j) {     typebox(darkgreen);   }   typeline(); } jumpxy(4,250); for (var i = 0; i &lt; 3; ++i) {   for (var j = 0; j &lt; 12; ++j)   {     if (j &lt; 7) {       typebox(transparent);     } else {       typebox(brown);     }   }   typeline(); } </pre>	<p>Using the two art shapes created in this lesson, have the students create a fun scene using ASCII art such as the tree shown in the left-most column.</p> <p>Here is a sample <a href="#">program</a>.</p> <p><u>Output:</u></p> 	<p>Demonstration: 55 minutes</p> <p>Student Practice: 100 minutes</p>

## 9.2 Resources

### Additional Exercises:

Using the Text Art assignment, ask students to work collaboratively to create a long-term project. The end result should be an interesting scene using at least two objects built in TEXT Art. Allow the students to pick other objects from the library from the functions chapter. Require each student to create at least one TEXT art individually and require them to submit it for grading as an early deliverable. (You can choose if this grade is part of the overall grade or a separate one.)

# Chapter 10: Recursion

## 10.0.1 Objectives

The chapter provides a brief introduction to recursion, which is the practice of using a function that calls itself. Students will learn what recursion is and how to read recursive code. Students will learn the key components of a recursive program and, importantly, they will learn that any recursive program must have a means of exiting the function via a base case.

## 10.0.2 Topic Outline

- 10.0 Chapter Introduction
  - 10.0.1 Objectives
  - 10.0.2 Topic Outlines
  - 10.0.3 Key Terms
  - 10.0.4 Key Concepts
- 10.1 Lesson Plans
  - 10.1.1 Teaching Suggestions
  - 10.1.2 Suggested Timeline
  - 10.1.3 CSTA Standards
  - 10.1.4 Lesson Plan I on demonstrating how recursive functions work.
  - 10.1.5 Lesson Plan II on demonstrating how a recursive stack works.

## 10.0.3 Key Terms

Recursive functions	Base case
Terminating condition	Infinite recursion

## 10.0.4 Key Concepts

A **recursive** function is a function that calls itself. Recursive functions are useful for working with **self-similar problems**. There are two ways of thinking of recursion:

- Recursion **reduces** a problem to smaller, similar, problems that can be solved more easily than the larger problem.
- Recursion **expands** computation by repeating a smaller, similar procedure as part of carrying out a larger procedure.

A recursive function has a conditional for dealing with two different cases:

- The **recursive case**. In this case, the function calls itself to solve a smaller problem then use that solution to solve the complete problem.
- The **base case**. In this case, function recognizes the simplest situations and completes the computation without calling itself.

Both cases are important. Since working out the recursive case usually takes a lot of thinking, programmers often forget about the simple base case. When that happens, a recursive function will call itself repeatedly in an infinite recursion and the function will never complete (until a debugger interrupts it).

The best way to understand recursion is with examples.

## A Triangular Spiral Example

Previously we have created computer algorithms by thinking about the first steps first, but in recursion, the planning is done from the end first. Imagine that most of the problem has already been solved. With recursion, the programmer asks: once almost everything is done, how would the very last step be solved? A recursive algorithm is built by starting by just thinking about just this last step.

Suppose a recursive program needs to measure a triangular spiral where each side is 10 units longer than the previous side, with the first side 10, then the next side 20, then the next side 30, and so on. How long is a spiral with N sides?

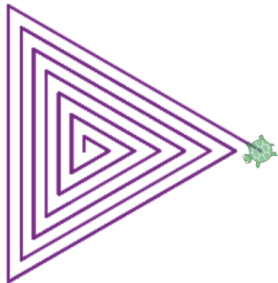
Solving this problem using recursion requires two cases.

- The **recursive case**. To make a spiral with N sides, imagine that the program can already solve the problem for size N - 1. Call the answer `spiral(N - 1)`. Since the last side has length  $10 * n$ , we therefore know that  $\text{spiral}(N) = \text{spiral}(N - 1) + (10 * N)$ .
- The **base case**. Some N is needed which does not require on self-reference. It is convenient to do this when N is zero, as  $\text{spiral}(0) = 0$ .

To write this in code, use an “if” conditional.

```
function spiral(N) {
  if (N > 0) {
    // The recursive case.
    var len = spiral(N - 1); // Assume we can do a smaller spiral.
    fd(10 * N);             // Now draw just the last leg of the big spiral.
    rt(120);
    return len + (10 * N); // Return the whole length.
  } else {
    return 0;              // The base case is zero: zero length.
  }
}
pen(purple);
spiral(20);
```

Output



*A recursive program to measure a triangular spiral.  
The spiral function calls itself and then adds just the last leg.*

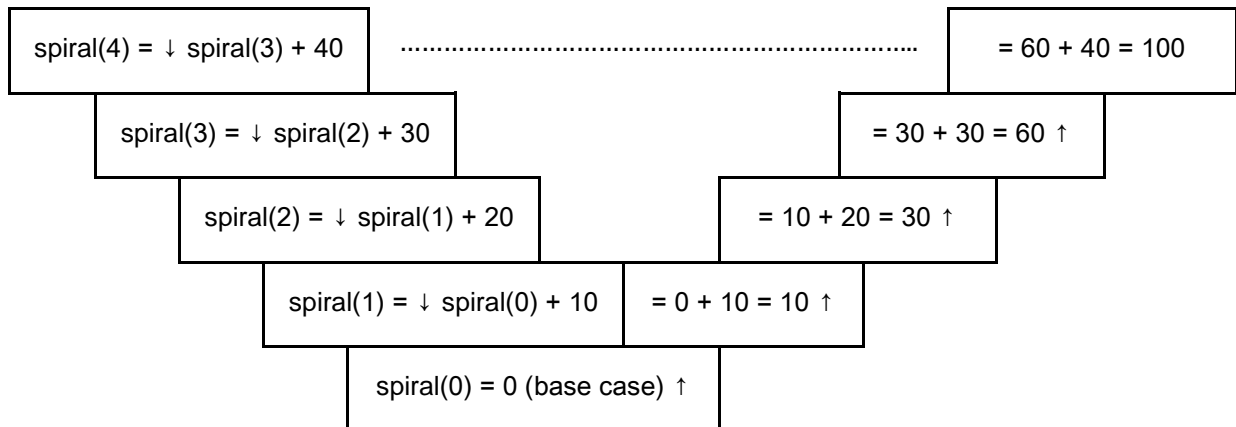
In this code, `spiral(20)` makes a call to `spiral(19)`, makes a call to `spiral(18)`, and so on. How can this possibly work? The computer does not actually have the answer to `spiral(19)` before running `spiral(20)`!

## Tracing Out Recursion on a Grid

Every time a function is called, its parameters can take on a different meaning, and so a computer can even call `spiral(3)` even if the invocation of `spiral(4)` is still not completed. The variable N means



something different in each invocation and the “layers” do not interfere with each other. The computation layers in the diagram below show how this works.



Each row is a single level of the recursion, showing how it calls the next level of the recursion, with the base case at the bottom. Moving from top to bottom shows how each layer of recursion reduces the problem to a smaller problem.

Reading from left to right reveals how the computation proceeds over time: `spiral(4)` does a function call to `spiral(3)`, which does a function call to `spiral(2)`, and so on until the base case returns 0, which allows `spiral(1)` to complete and return 10, which allows `spiral(2)` to complete and return 30, which allows `spiral(3)` to complete and return 60, which allows `spiral(4)` to complete and return 100.

### A Fractal Tree Example

Recursion can be used to create self-similar patterns. If a recursive function calls itself twice, then a branching effect can be created, where a single function call expands to 2, 4, 8, etc, calls. In the example below, the recursive function `tree` takes two inputs, a turtle object `t` and a branch length `x`.

<pre>function tree(t, x) {   t.fd(x);          // 1. Draw a line   if (x &lt; 10) {     return;        // 2. Base case   }   var r = t.copy(); // 3. Copy the turtle   t.lt(30);   tree(t, x * 0.7); // 4. Left mini-tree   r.rt(30);   tree(r, x * 0.7); // 5. Right mini-tree } pen(brown); tree(turtle, 100);</pre>	
--	--

An explanation of the portions of this program.

1. The only drawing done directly is to draw one line of length `x` by moving forward `t.fd(x)`.
2. Then the base case is handled: if the line was shorter than 10, it returns with no further action.
3. A branch will be made by making a copy of `t` called `r`. This will handle the right side.
4. The original turtle `t` will handle the left side by turning left 30 degrees and drawing another tree. The tree on the left will begin with a branch that is smaller than `x` by multiplying by 0.7.

- The new turtle r will handle the right side by turning right 30 degrees and drawing another tree.

The tree in the example above is self-similar because a large tree is made up of smaller trees. Many shapes in nature have the same kind of self-similarity, and recursive programs like this can be used to create organic, natural shapes.

### 10.1.1 Teaching Suggestions

Students often find recursion intimidating, so teachers often need to revisit this concept several times to make sure the students truly understand it. In an introductory programming class, students should at least be exposed to the idea of recursive functions and understand how recursive functions work. It helps a great deal if they can see recursion in action. It can be helpful to ask students to solve a couple of recursive problems using paper and pencil to predict the output. At this point we do not recommend that students create their own recursive solution. You can use the two Pencil Code examples provided in this lesson as a tool explain and demonstrate recursion. It is best to use multiple strategies to teach recursion.

### 10.1.2 Possible Timeline: 1 55-minute class period

Instructional Day	Topic
1 Day	Lesson Plan I
2 Day	Lesson Plan II

### 10.1.3 Standards

CSTA Standards	CSTA Strand	CSTA Learning Objectives Covered
Level 3 A (Grades 9 – 12)	Computational Thinking (CT)	Explain how sequence, selection, iteration, and recursion are building blocks of algorithms.

### 10.1.4 Lesson Plan I

This lesson shows the recursion process. The demonstration should take about 30 minutes. After the demonstration, give the students a copy of the program and let them play experiment with to expand their understanding of how the recursive functions work.

Teaching Notes:

Invariably this is the hardest topic in programming for beginner programmers. The visual feedback that Pencil Code provides helps with understanding the concepts better. Below is a suggested two-fold approach.

First step: demonstrate the program

Next: break it down into pieces and explain the concept using smaller snippets of code.

Demonstrate the [program](#) on the projector (5 minutes)

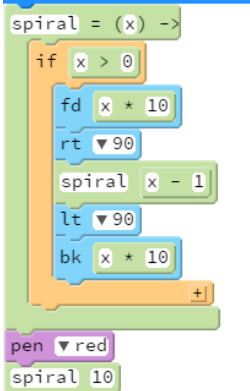
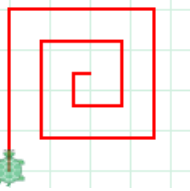
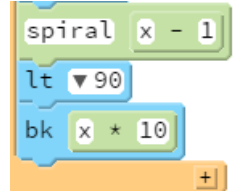
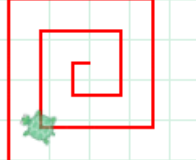
Code	Block	Output
<pre> movexy -101, 250 pen black, 1 speed 2 kolam = (x) -&gt;   if x &gt; 13     for [1..4]       rt 90       fd x       fill random color       lt 90     kolam(x/2) kolam(160) jumpto 98, 51 kolam(160) ht() </pre>		

Demonstrate the concept taking small snippets of code

Content details	Teaching Suggestions	Time
<p>Code</p> <pre> kolam = (x) -&gt;   if x &gt; 13     for [1..4]       rt 90       fd x       fill random color       lt 90     kolam(x/2) </pre>	<p><b>Step 1:</b> The rectangles keep getting smaller at each call to the function.</p> <p><b>Teaching Tip:</b> Project this code and emphasize that the value of x keeps getting smaller.</p>	<p>Demonstration: 15 minutes</p>
<p>Code:</p> <pre> kolam(160) jumpto 98, 51 kolam(160) </pre> <pre> kolam(160) jumpto 98, 51 kolam(160) </pre>	<p><b>Step 2:</b> The main program calls the function twice and generates two sets of three squares.</p>	
<p>Code:</p> <pre> if x &gt; 13 </pre> <pre> if x &gt; 13 </pre>	<p><b>Step 3:</b> There is a condition that is checked before the function is called again.</p> <p><b>Step 4:</b> When the condition fails, the program exits out of the recursive cycle and the control is returned to the main program</p>	

## 10.1.5 Lesson Plan II

This lesson plans demonstrates the recursive process and shows how the compiler stacks the commands that are not yet executed. This should take about 20 minutes to demonstrate and explain.

Content details	Teaching Suggestions	Time
<p>Code:</p> <pre> spiral = (x) -&gt;   if x &gt; 0     fd x * 10     rt 90     spiral x - 1     lt 90     bk x * 10   pen red   spiral 10           </pre> 	<p>Demonstrate the <a href="#">program</a> and explain the recursive function.</p> <p>Point out the recursive formula and the condition that helps end the recursion.</p> <p>Emphasize the importance of the conditional statement.</p> <p><u>Output</u></p> 	<p>Demonstration: 25 minutes</p>
<p><u>Code Snippet:</u></p> <pre> spiral x - 1   lt 90   bk x * 10           </pre> 	<p>When the recursive function is called, the two statements after the recursive call are stacked.</p> <p>Once the recursive calls ends the two statements are popped out of the stack. This is illustrated by the turtle tracing the spiral in reverse order.</p> <p><u>Output</u></p> 	
<p>Encourage students to play execute various recursive programs that are available in the resources list. Students should be encouraged to change values and see the effect of the changes.</p>		<p>Student Practice: 55 minutes</p>

# Chapter 11: Building a Website Using HTML - CSS

## 11.0.1 Objectives

Pencil code offers a very easy drag and drop way to create HTML files. This section introduces the structure of a simple HTML page. The lessons guide students to create basic pages using block-mode and to transition to text-mode as their familiarity and knowledge increase.

## 11.0.2 Topic Outline

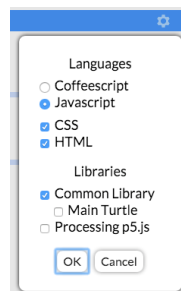
- 11.0 Chapter Introduction
- 11.0.1 Objectives
- 11.0.2 Topic Outlines
- 11.0.3 Key Terms
- 11.0.4 Key Concepts
  
- 11.1 Lesson Plans
- 11.1.1 Teaching Suggestions
- 11.1.2 Suggested Timeline
- 11.1.3 CSTA Standards
- 11.1.4 Lesson Plan I on building a basic HTML page
- 11.1.5 Lesson Plan II on building a basic CSS page
- 11.1.6 Lesson Plan III on writing a JavaScript program embedded in a HTML page.
- 11.1.7 Lesson Plan IV on writing a JavaScript program to create a slideshow.

## 11.0.3 Key Terms

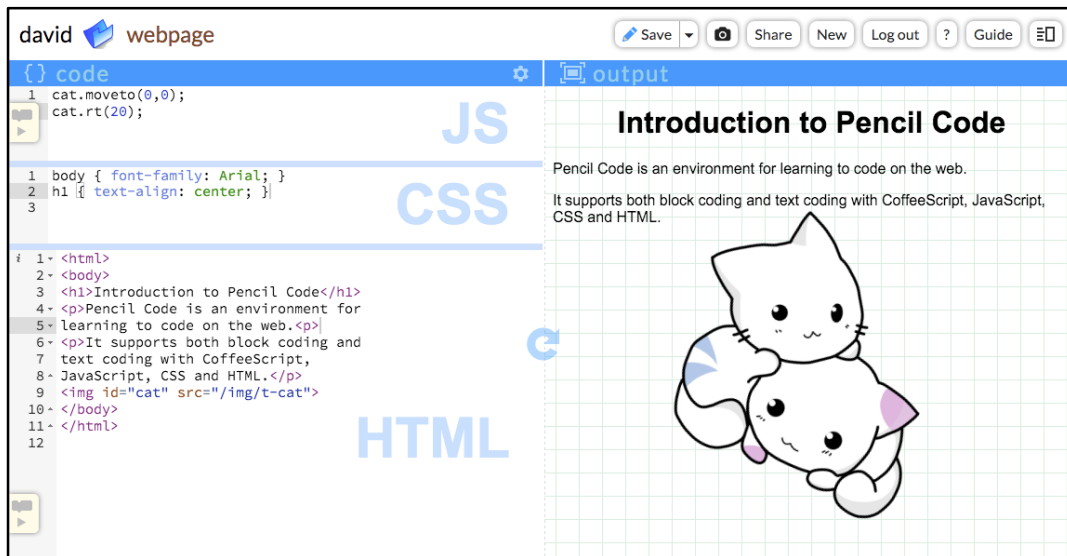
Cascading Style Sheet (CSS)	HTML
url	Hyper-text
Server - client	protocol
https vs. http (encrypted vs. unencrypted)	tags
<script> - tag	

## 11.0.4 Key Concepts

Pencil Code supports coding webpages that combine HTML, CSS, and JavaScript or CoffeeScript. To enable HTML and CSS editing, click the “gear” icon and select those languages.



Enabling HTML and CSS will split the editing area into multiple panes:



*Editing HTML and CSS and JavaScript in the same project*

The code for the screenshot above is shown below. It has an HTML page with several elements, two CSS rules, and two lines of JavaScript code:

JavaScript	<pre>cat.moveto(0,0); cat.rt(20);</pre>
CSS	<pre>body { font-family: Arial; } h1 { text-align: center; }</pre>
HTML	<pre>&lt;html&gt; &lt;body&gt; &lt;h1&gt;Introduction to Pencil Code&lt;/h1&gt; &lt;p&gt;Pencil Code is an environment for learning to code on the web.&lt;p&gt; &lt;p&gt;It supports both block coding and text coding with CoffeeScript, JavaScript, CSS and HTML.&lt;/p&gt; &lt;img id="cat" src="/img/t-cat"&gt; &lt;/body&gt; &lt;/html&gt;</pre>

All three of the languages in this example interact with each other.

- The HTML includes `<body>`, `<h1>`, and `<img id="cat">` elements.
- The CSS has a `body` rule and an `h1` rule, setting the visual styles for those HTML elements.
- The JavaScript refers to the `cat` image, using turtle functions to move it and turn it.

This editor can be used to teach web and internet concepts. Because the Pencil Code library is also provided, all the examples from previous chapters continue to work. Coding can be done directly on a webpage, allowing students to create animations and interactivity at the same time as learning about HTML and CSS, URLs, and the Internet.

## The Internet

The Internet is a global network of connected computers acting as clients, routers, and servers. Some of the most useful data sent on the Internet are webpages, and when some people talk about the Internet, they are really referring to the Web.

**Servers** run programs that wait at the network, ready to answer requests that are sent by other computers. For example, the computer known as “[www.un.org](http://www.un.org)” is a server at the United Nations that waits for requests for webpages, and when it gets a request, it sends back the requested webpage to whichever computer sent the request.

**Clients** run programs that make requests by creating and sending messages to servers. For example, when you use your Web browser on your phone to visit [www.un.org](http://www.un.org), your phone is acting as a client. It sends a small message with a request to the computer at the UN, and after a moment, it gets back a message containing a webpage.

**Routers** run programs that forward messages between other computers. It would be impractical to run a wire (or a direct radio signal) from your client (and every other client in the world) to the UN. So, instead, the computer sends messages to a nearby computer with instructions to pass it on in the right direction. The message is passed on from computer to computer until it arrives at the correct server. Routers are the silent delivery workers of the Internet. You do not normally need to know the dozen or so routers that may sit between your client and a server, but if you have noticed a computer on your network known by the number “192.168.1.1” or “10.0.0.1”, those computers are probably routers.

## The World Wide Web

**HTML** is the HyperText Markup Language, the main language in which pages on the web are written.

A **hypertext document** is a text document that embeds links to other resources on the Internet.

A **web address** is a precise name (written as a URL) that locates one specific document or file on the Internet.

A **webpage** is a hypertext document with a Web address.

If one were to draw a picture of a few linked webpages with a dot for each webpage and a line for each link between them it would look like a Web of connections. The World Wide Web (the Web) is the name we give to the many billions of linked webpages around the world.

The pages that make up the Web are served by millions of different servers around the world, each hosting numerous webpages. To understand how a webpage is found, it is important to understand the URL.

## The Three Main Parts of a URL

The Web address of a webpage is written in a precise way called a **URL** (a **Uniform Resource Locator**). A URL looks like this:

`https://www.un.org/en/index.HTML`

The first part “https:” is the **protocol**, which determines the message formats used when communicating with the server. There are two main protocols on today’s Web. **HTTP** (Hyper Text Transfer Protocol) transmits requests and responses without any encryption so that any router can read the messages. **HTTPS** (HTTP Secure) encrypts the messages so that only the client and the server can read them.

The second part “//[www.un.org](http://www.un.org)” after the double slash is the server. Requests for this URL will be sent to the server [www.un.org](http://www.un.org).

The third part “/en/index.HTML” after the server name is the **path** of the webpage that will be requested within the server. Web servers often organize paths using “/” to divide directory names from the files within them, but Web servers are free to organize their webpages using any paths they choose. For example, Google serves an almost infinite variety of webpages using URLs such as “<https://www.google.com/search?q=un>”. Change the last part of the path from “un” to any other word to ask Google for a page of search results.

## Pencil Code URLs

Pencil Code makes a URL for every program saved by a student:

<http://newbie.pencilcode.net/home/myprogram>

Here the protocol is HTTP unencrypted. (Pencil Code also supports HTTPS.)

The server is “newbie.pencilcode.net”, a server created by the student. All students who save work on Pencil Code gets their own **virtual server**. (It is called “virtual” because there is not actually a new computer for each student: Pencil Code just runs server software using a new name and reuses a physical computer that is shared with other students.)

The path is “/home/myprogram”, which is “/home/”, followed by the name chosen when saving the file. On Pencil Code, the top-level directory name in the path is special. If you use “/home/”, it will serve the raw webpage, just like an ordinary Web server. If you change the directory name to “/edit/”, it will serve a special editing webpage that lets you see the source code and log in to edit the page.

## Tags in HTML

**HTML** (Hyper Text Markup Language) is the language used to write a webpage. A simple HTML document looks like this:

```
<!doctype HTML>
<HTML>
  <body style="background:wheat">
    <h1>My Page</h1>
    <p>This is <em>my</em> page.</p>
    <p></p>
  </body>
</HTML>
```

A standard HTML document begins with <!doctype HTML> followed by three types of information.

**Tags** in angles like <body> designate special locations in the document. The **tag name** is the first word of the tag.

**Attributes** of a tag, also written within the angle of the tag, such as title="little fox" or style="background:wheat".

**Text content** such as “My Page” which does not appear inside angles.

Most of the tags are paired: a **begin tag** like <body> matches **end tag** with a slash like </body>, but there are a few special **self-closing tags** such as <img> that are not paired. A tag and its pair (if any) together



are called an **element**, and elements may be **nested** within each other. For example, within the document above, the `<img>` element is nested within a `<p>` element, nested within `<body>`, which is nested within the `<HTML>`. The `<em>` element is also nested within a different `<p>` element.

HTML has about 100 types of elements and 100 or so attributes that let programmers do a great variety of things in a document. The best way to learn about HTML is to make webpages while trying out different elements described in one of the many online resources available on the Internet. Here is a list of a few particularly useful elements.

Element	Purpose	Type
<code>&lt;HTML&gt;...&lt;/HTML&gt;</code>	The element enclosing an entire HTML document.	section
<code>&lt;body&gt;...&lt;/body&gt;</code>	The visible contents of the document (invisible metadata goes in another similar section <code>&lt;head&gt;...&lt;/head&gt;</code> ).	section
<code>&lt;p&gt;...&lt;/p&gt;</code>	A paragraph.	group
<code>&lt;h1&gt;...&lt;/h1&gt;</code>	A big heading (headings get smaller down to h6).	group
<code>&lt;style&gt;...&lt;/style&gt;</code>	A Cascading Style Sheet for defining visual styles.	code
<code>&lt;script&gt;...&lt;/script&gt;</code>	A script in JavaScript or another programming language.	code
<code>&lt;em&gt;...&lt;/em&gt;</code>	An emphasized phrase (italicized in your browser).	text
<code>&lt;a href="url"&gt;...&lt;/a&gt;</code>	A hyperlink that leads to a page located by the href url.	text
<code>&lt;img src="url"&gt;</code>	An image that is loaded from the url in the src attribute.	embedding
<code>&lt;iframe src="url"&gt;&lt;/iframe&gt;</code>	A subframe containing a nested page loaded from a url.	embedding

The last three elements have attributes href and src whose values are URLs that link to other files or webpages. On a webpage, there is a short way to write URLs that is important to understand.

### Relative URLs

URLs that in webpages can be written in an abbreviated form called **relative URLs**, which omit portions of the URL and assume they take on the same values as the current Web address. Ordinary complete URLs contain a protocol, server name, and path and are called **absolute URLs**. For example, suppose the current URL is <http://newbie.pencilcode.net/home/myproject/welcome.html>. Here are examples of relative URLs.

Relative URL	Relative to <code>http://newbie.pencilcode.net/home/myproject/welcome.html</code>
<code>friends.html</code>	<code>http://newbie.pencilcode.net/home/myproject/friends.html</code>
<code>css/style.css</code>	<code>http://newbie.pencilcode.net/home/myproject/css/style.css</code>
<code>/home/index.html</code>	<code>http://newbie.pencilcode.net/home/index.html</code>
<code>/img/happyfox</code>	<code>http://newbie.pencilcode.net/img/happyfox</code>
<code>//un.org/fr/index.html</code>	<code>http://un.org/fr/index.html</code>
<code>https://google.com/</code>	<code>https://google.com/</code>

The basic rule is this: an absolute URL always starts with a protocol name such as `http:` or `https:`. If a relative URL starts with two slashes, it inherits the current protocol but replaces everything after that. If a relative URL starts with one slash, it replaces everything after the current server name. If a relative URL does not start with a slash, it replaces everything after the last slash in the current URL.

Absolute URLs can be used anywhere a URL is required, but relative URLs can be much faster to type. Relative URLs also have the advantage in that if a directory of webpages is moved together between servers or directories, URLs that make relative references within the directory will continue to work.

### The Style Attribute and the `<style>` Element

Browsers come with default visual styling for every HTML element. For example, the `<em>...</em>` element indicates a phrase that should be emphasized, and browsers will use italic font-style for that text by default. But what if you wish to use a normal font-style and underline the text instead? Visual styles can be overridden by using the style attribute on any visible element, as follows:

```
<em style="font-style:normal;text-decoration:underline">something</em>
```

The value of the style attribute is a **style declaration block**, which can list any number of **style declarations** separated by semicolons. Each style declaration has a **style property** followed by a colon and a value. There are about 100 standard style properties and there are many Internet resources that list them and give examples of how they work.

If you wish to apply the same style declaration block to every `<em>` element in the document, you can create a `<style>` element containing **CSS** (Cascading Style Sheet) rules, such as:

```
<style>
em {
  font-style:normal;
  text-decoration:underline;
}
</style>
```

CSS is a powerful language that provides ways to combine and generalize style rules. We will not talk much about it here but there are excellent resources available on the Internet detailing beautiful effects that can be created with CSS.

### The <script> Element

When combining HTML and JavaScript code, the JavaScript is embedded in a <script> element. Pencil Code provides separate editing panels for separating editing HTML and JavaScript for a webpage, but when you view them together, Pencil Code puts the JavaScript inside the HTML page by adding a <script> element.

The <script> Element also can also be used with other languages. When the CoffeeScript language is loaded, the type attribute may be set so that <script type="text/coffeescript">...</script> contains CoffeeScript.

### 11.1.1 Teaching suggestions

The first two lesson plans detail how to build an HTML page using Pencil Code. There are blocks available for the overall structure that a student can drag and help them students build the HTML page. This chapter differs from others in that each lesson plans is a continuation from the previous lesson plan.

### 11.1.2 Possible Timeline: 1 55-minute class period

Instructional Day	Topic
2 Days	Lesson Plan I & II:HTML webpage design
1 Day	Lesson Plan III: HTML – JavaScript embedded program
1 Day	Lesson Plan IV: HTML – JavaScript using arrays (JavaScript embedded in the webpage)

### 11.1.3 Standards

CSTA Standards	CSTA Strand	CSTA Learning Objectives Covered
Level 3 B (Grades 9 – 12)	CL	Use project collaboration tools, version control systems, and Integrated Development Environments (IDEs) while working on a collaborative software project.
Level 3 A (Grades 9 – 12)	CPP	Use advanced tools to create digital computing innovations and artifacts (e.g., web design, animation, video, etc.).
Level 3 A (Grades 9 – 12)	CPP	Create and organize webpages through the use of a variety of web programming design tools.


### 11.1.4 Lesson Plan I

Build a one-page HTML page using basic blocks and HTML tags. Check to see that HTML and CSS options are checked under settings.

Content details	Teaching Suggestions	Time
<p>Code: Text</p> <pre data-bbox="191 478 732 1066"> &lt;!DOCTYPE html&gt; &lt;html&gt; &lt;head&gt;   &lt;title&gt;iTeach- ComputerScience&lt;/title&gt; &lt;/head&gt; &lt;body   bgcolor="#ffffbc6"&gt;   &lt;h1 align="center"&gt; Art Gallery to open soon!&lt;/h1&gt;   &lt;p&gt;     In a new exhibition at   ...   ... comes from watching and being watched.   &lt;/p&gt; &lt;/body&gt; &lt;/html&gt; </pre>	<p>Step 1: Demonstrate to students on how to create a basic page by dragging the blocks for the framework.</p> <p>Step 2: Ask the students to add color and fonts.</p> <p>Step 3: Encourage students to switch to “text-mode” to type in HTML text.</p> <p>Step 4: Point out the basic structure and problems that can happen if students miss small details in the HTML code. Note that they can avoid these problems by dragging and dropping using block-mode.</p> <p>Step 5: Click on “Share” and copy – paste the URL into a Web browser.</p> <p>Point out the layout and color and comment on the site’s visual appeal.</p> <p>Here is the <a href="#">code</a> for the program.</p> <p>Output</p> <div data-bbox="755 1430 1230 1621" style="background-color: yellow; padding: 5px;"> <p style="text-align: center;"><b>Art Gallery to open soon!</b></p> <p><small>In a new exhibition at birforms gallery, Rafael Lozano-Hemmer asks viewers to consider the socio-politics of contemporary technology through playful participation. The premiere of a new interactive sculpture in Interior (2015) is perhaps the best indicator of the artist’s desire to create what he calls “playful landscap technologies of oppression.” The work is a brightly illuminated inside-out disco ball made of 1,600 one-1/4 mounted on acrylic into which a visitor inserts his or her head. Because the mirrors face inward, participants disco ball are confronted with a wall of fractured images of their own reflection. Meanwhile, other visitors can see the person’s head inside the ball. As a result, External Interior cleverly plays on the tension that c watching and being watched.</small></p> </div>	<p>Demonstration: 30 minutes</p>


### 11.1.5 Lesson Plan II

This lesson focuses on the creation of a page using the CSS. It demonstrates the use of CSS and how to add it to the program from the previous lesson plan. Check to see that the HTML and CSS options are checked under settings.

Content details	Teaching Suggestions	Time
<p>Code – HTML</p> <pre data-bbox="191 327 743 890"> body {   background: cyan;   font-family: sans-serif;   text-align: center;} button {   background: cornflowerblue;   color: white;   border: none;   border-radius: 6px;   font-size: 18px;   margin: 8px; } #rs { background: gray;} img { border-radius: 6px;} ul { list-style-type: none; padding: 0;} ul li { display: none;} ul li.shown { display: block;} </pre> <p>Code – CSS</p> <pre data-bbox="191 940 743 1339"> body {   background: cyan;   font-family: sans-serif;   text-align: center; } button {   background: cornflowerblue;   color: white;   border: none;   border-radius: 6px;   font-size: 18px;   margin: 8px; } </pre>	<p>Step 1: Open the program and click on run.</p> <p>Step 2: Point out the HTML part of the code and show how the structure of an HTML file can be dropped into the program in block-mode.</p> <p>Step 3: Show the CSS code and tinker with values to demonstrate change in appearance.</p> <p>Step 4: Compare with the first program and show students how CSS helps</p> <p>Here is a sample <a href="#">program</a>.</p> 	<p>Demonstration: 15 minutes</p> <p>Student Practice: 30 minutes</p>

### 11.1.6 Lesson Plan III

Students will now write a JavaScript program that is embedded in the webpage they have been modifying. The Art Gallery page. Note, the program is a very simple one because it takes what has been taught in Lesson Plan II and adds the JavaScript information to it.

Content details	Teaching Suggestions	Time
<p>Code: JavaScript</p> <pre> speed(100); for(var i=0;i&lt;6;i++) {   pen(random(color), 2);   for (var j = 0; j &lt; 50; ++j)   {     rt(30, j);   } } </pre> <p>Code: HTML</p> <pre> &lt;!DOCTYPE html&gt; &lt;html&gt;   &lt;head&gt;     &lt;title&gt;iTeach- ComputerScience&lt;/title&gt;   &lt;/head&gt;   &lt;body     bgcolor="#ffffbc6"&gt;     &lt;h1 align="center"&gt; Art Gallery to open soon!&lt;/h1&gt;     &lt;p&gt;       In a new exhibition at ...     ... and being watched.     &lt;/p&gt;   &lt;/body&gt; &lt;/html&gt; </pre> <p>Code: CSS</p> <pre> body {   font-family: sans-serif;   text-align: center; } button {   background: cornflowerblue;   color: white;   border: none;   border-radius: 6px;   font-size: 18px;   margin: 8px; } </pre>	<p>Step 1: Pull up <a href="#">program</a>: FirstPageArtGalleryDesigns.</p> <p>Ensure that the Pencil Code environment is in JavaScript mode with HTML / CSS selected.</p> <p>Step 2: Run the program. (It looks similar to the program the students modified in the previous lesson plan.)</p> <p>Step 3: Point to the JavaScript program. This is one of the programs they previously designed.</p> <p>Step 4: Explain that this activity combines many techniques they have already learned.</p> <p>Step 5: Click on the Share and copy / paste the URL into a new Tab in the browser to view the webpage.</p> <p>Teaching Tip: Use a right mouse button click on View Source to show how the JavaScript code is embedded into the HTML page.</p> <p>Output:</p> <div data-bbox="711 1283 1214 1541" style="background-color: yellow; padding: 5px;"> <p style="text-align: center; margin: 0;"><b>Art Gallery to open soon!</b></p> <p style="font-size: 8px; margin: 0;">In a new exhibition at bilforms gallery, Rafael Lozano-Hemmer asks viewers to consider the socio-political implications of contemporary technology through playful participation. The premiere of a new interactive sculpture titled External Interior (2015) is perhaps the best indicator of the artist's desire to create walls that "playful landscapes out of technologies of oppression." The work is a brightly illuminated inside-out ball made of 1,600 one-way mirrors mounted on acrylic into which a visitor inserts his or her head. Because the mirrors face inward, participants dominating the disco ball are confronted with a wall of fractured images of their own reflection. Meanwhile, other visitors in the gallery can see the person's head inside the ball. The result, External Interior cleverly plays on the tension that comes from watching and being watched.</p>  </div>	<p>Demonstration: 20 minutes</p>
<p>Encourage students to create their own JavaScript code and view the program execution embedded in a webpage. Student Practice: 25 minutes</p>		

## 11.1.7 Lesson Plan IV

This lesson focuses on the creation of a slide show using the JavaScript program. It demonstrates the use of arrays (the simplest data structure in programming). This culminating lesson enables students to put together all of the techniques presented in all the lesson plans in this chapter to create an interesting webpage.

Content details	Teaching Suggestions	Time
<p>Code:HTML</p> <pre data-bbox="201 695 591 1062">&lt;!DOCTYPE html&gt; &lt;html&gt;   &lt;body&gt;     &lt;h1&gt;My Slide Show&lt;/h1&gt;     &lt;button id="bb"&gt;Back&lt;/button&gt;     &lt;button id="ff"&gt;Forward&lt;/button&gt;     &lt;div&gt;        &lt;img id="im" src="/img/cake"&gt;     &lt;/div&gt;     &lt;button id="rs"&gt;Restart&lt;/button&gt;   &lt;/body&gt; &lt;/html&gt;</pre> <p>Code: CSS</p> <pre data-bbox="201 1142 542 1829">body {   font-family: sans-serif;   text-align: center; } button {   background: cornflowerblue;   color: white;   border: none;   border-radius: 6px;   font-size: 18px;   margin: 8px; } #rs {   background: gray; } img {   border-radius: 6px; } ul {   list-style-type: none;   padding: 0; } ul li {   display: none; } ul li.shown {   display: block; }</pre>	<p>Step 1: Open the <a href="#">program</a> js-slideshow. Ensure that the Pencil Code environment is in JavaScript mode and it has HTML and CSS options checked.</p> <p>Step 2: Point out that each button has a small functional module under it that gets executed when clicked. (Refer to concepts in Chapter 5 – Functions.)</p> <p>Step 3: Show that the images get selected from the Internet using /img capability. (Chapter 3 – Input / Output).</p> <p>Step 4: Point out the HTML and CSS sections of the program that enable this program to become a part of a webpage.</p> <pre data-bbox="721 1234 1230 1451">function goforward() {   pointer += 1;   if ( pointer &gt;= food.length ) {     pointer =0;   }   document.getElementById ('im').src = (food[pointer]) }</pre> <pre data-bbox="721 1604 1062 1820">var food = [   "/img/orange juice",   "/img/cupcake",   "/img/potato chips", ]</pre>	<p>Demonstration: 30 minutes</p>

Content details	Teaching Suggestions	Time
<p>Code: JavaScript</p> <pre> document.getElementById('bb').addEventListener('click', goback);  document.getElementById('rs').addEventListener('click', restart);  document.getElementById('ff').addEventListener('click',goforward);  var food = [   "/img/orange juice",  "/img/cupcake",   "/img/potato chips", ] var pointer =0; function goforward() {   pointer += 1;   if (pointer &gt;= food.length) {     pointer =0;   } }  document.getElementById('im').src = (food[pointer]); }  function goback() {   pointer -= 1;   if (pointer &lt; food.length) {     pointer =food.length-1;   } }  document.getElementById('im').src = (food[pointer]); }  function restart(){ document.getElementById('im').src = food[pointer]; pointer = 0; } </pre>	<div data-bbox="721 275 1230 953" style="text-align: center;"> <h2 style="margin: 0;">My Slide Show</h2> <div style="display: flex; justify-content: center; gap: 20px; margin: 5px 0;"> <span style="background-color: #4a86e8; color: white; padding: 5px 10px; border-radius: 5px;">Back</span> <span style="background-color: #4a86e8; color: white; padding: 5px 10px; border-radius: 5px;">Forward</span> </div>  </div>	
<p>Encourage students to change the images. Increase the size of the array and add more elements. Encourage students to tinker with the CSS and HTML values to improve the visual appeal of the program.  Student Practice:120 minutes</p>		



# Chapter 12: Traversing Data Using JQuery

## 12.0.1 Objectives

This unit introduces the basics of the jQuery library. Many webpages today use jQuery to create interactive features, and familiarity with jQuery will allow students to understand a large number of programming resources on the internet. Students will learn the concept of a jQuery selection, and use jQuery methods and events to create a simple interactive program.

## 12.0.2 Topic Outline

- 12.0 Chapter Introduction
  - 12.0.1 Objectives
  - 12.0.2 Topic Outlines
  - 12.0.3 Key Terms
  - 12.0.4 Key Concepts
- 12.1 Lesson Plans
  - 12.1.1 Suggested Timeline
  - 12.1.2 CSTA Standards
  - 12.1.3 Lesson Plan I using the Timer () program.
  - 12.1.4 Lesson Plan II on traversing an array.

## 12.0.3 Key Terms

JQuery	JQuery Object
CSS Selector	JQuery Method
Turtle library	set
Arguments	method

## 12.0.4 Key Concepts

### Introduction to jQuery

JQuery is the most popular library used in web pages because it is a convenient way to examine and alter visual elements. Here is an example using jQuery.

```
$('#div').hide();
```

Each use of jQuery has three steps:

1. A CSS Selector is used to find a set of elements on the page.
2. A jQuery Object is created representing the set of elements.
3. A jQuery Method is called to do some operation on the set of elements.

In this example, 'div' is the CSS selector, \$('#div') is the jQuery object, and .hide() is the jQuery method. This line of jQuery code means: "Find all the <div> elements on the page and then hide them."

Querying databases involves a three-step process of finding, gathering, and manipulating. JQuery applies this database technique to the interface elements of an HTML page, treating a single page as a database.

## Creating jQuery Objects with \$

The function that creates jQuery objects is the most important function in the jQuery library. Because it is used so often, the jQuery library provided a short and unusual name for this function: \$. Although it may look strange, \$ is just a regular function name that happens to use a symbol.

A jQuery object holds a set of elements: the set may contain zero, one, or multiple elements on the page. Here are some example uses of \$:

jQuery constructor	Creates a jQuery object containing this set.
<code>\$('p')</code>	All the <p> elements in the document.
<code>\$('.special')</code>	All the elements with class="special" in the document.
<code>\$('#buy')</code>	The element with id="buy" in the document.
<code>\$('&lt;img src="/img/cat"&gt;')</code>	A new <img> element with the given src, not yet inserted into the document.
<code>\$('&lt;p&gt;Hello&lt;/p&gt;')</code>	A new <p> element with the given text content, also not yet inserted into the document.

The CSS selector language used to find elements with \$ is the same language used in CSS, so anything you learn about CSS selectors can also be used in jQuery. If no elements match a selector, the function returns the empty set.

The \$ function can also create elements using HTML syntax (such as in the last two examples above). In this case, it returns a set containing one newly-created element, not yet placed in the visible document.

A jQuery object has a .length attribute that gives the size of the set. For example you can use the expression `$('p').length` to count the number of <p> elements in the document.

## Using jQuery Objects

There are a number of methods that can be used to operate on any jQuery object. Some examples:

<code>\$('p').fadeOut();</code>	Smoothly fades the elements, then hides them.
<code>\$('p').css({ background: red });</code>	Alters the CSS styling of all selected elements.
<code>\$('p').html('Read &lt;b&gt;this&lt;/b&gt;');</code>	Replaces the HTML content of all selected elements.
<code>var t = \$('p').text();</code>	Reads text content of the first selected <p> element.
<code>\$('input').val(10);</code>	Set the value within all the selected <input> boxes.
<code>var v = \$('input').val();</code>	Reads the value of the first selected <input> element.

<code>\$('#img').attr({src: '/img/cat'});</code>	Changes every <code>&lt;img&gt;</code> <code>src</code> attribute to <code>"/img/cat"</code> .
<code>\$('#img src="/img/dog"&gt;').appendTo('body');</code>	Creates a dog image and adds it to the <code>&lt;body&gt;</code> .
<code>\$('#warn').remove();</code>	Removes the element with <code>id="warn"</code> .
<code>\$('#img').bk(100);</code>	Use the turtle <code>"bk"</code> function to move all <code>&lt;img&gt;</code> s.

Students who have used Pencil Code will find jQuery familiar because every Pencil Code turtle is a jQuery object. The Pencil Code turtle library is an extension to jQuery that adds a number of turtle methods such as `"pen"`, `"fd"`, `"bk"`, `"rt"`, and `"moveto"` to the set of jQuery methods. Programmers can use these methods to move any visual element on the screen.

The main turtle can be accessed using the jQuery call `$('#turtle')`, so the CoffeeScript program `"fd 100"` from the very first section of this book is the same as JavaScript and jQuery program `$('#turtle').fd(100)`.

### Experimenting with jQuery

It is helpful for students to experiment with individual jQuery methods. Using the `"gear"` menu, they can create a Pencil Code project that includes the following HTML.

```
<html>
  <body>
    <h1>My favorite things</h1>
    <p>Pizza: </p>
    <p>Watermelon: </p>
  </body>
</html>
```

There are enough elements in this document to try each of the jQuery examples above. Students can enter the jQuery code directly into the `"Test panel"` on the right pane of Pencil Code, or they can enter the code to run in a JavaScript or CoffeeScript program on the left.

There are two things to notice with jQuery:

1. Changes are usually made as soon as you run the code, although some changes can be animated over time.
2. Although the changes you make affect the visible document, they do not change the HTML of the program itself.

The HTML in a program is the `"starting state"` of the HTML page. Once a program adds, removes, or alters elements, it can end up looking different from the HTML page the programmer originally wrote - but if the program is run, it will start with the original HTML.

### Using jQuery to Provide Dynamic Output

jQuery is useful for creating user interfaces with a screen of dynamic output that changes over time. For example, with jQuery, you can keep a timer and update a number every second. Here is a JavaScript program that does this.

```

$('<h1>Countdown</h1>').appendTo('body');
$('h1').css({textAlign: 'center'});
var count = 10;
forever(1, function() {
  $('h1').html(count);
  count -=1;
  if (count < 0) {
    $('h1').html('blast off!');
    stop();
  }
});

```

This program uses “forever” to set up a function that is called once per second until stop() is called. Here is an explanation of each of the jQuery calls in the program.

<code> \$('&lt;h1&gt;Countdown&lt;/h1&gt;').appendTo('body'); </code>	Creates an <h1> element and adds it to the <body>.
<code> \$('h1').css({textAlign: 'center'}); </code>	Sets the CSS of the <h1> so its “text-align” is “center”.
<code> \$('h1').html(count); </code>	Changes the HTML contents of the <h1> with a variable.
<code> \$('h1').html('blast off!'); </code>	Changes the HTML contents of the <h1> to “blast off!”d.

jQuery allows a program to provide real-time information on the screen by updating the contents of any visual element.

### Using jQuery Events to Collect User Input

In previous sections of this manual, input was collected by handling clicks in on-screen buttons. JQuery makes it simple to collect input events on any set of elements using the “.on” method. Here is an example.

```

$('h1').on('click', function(e) {
  log('You clicked on an h1');
})

```

The first argument of the “on” method is the event name and the second argument is the event handler function. These event handlers are the same as those used since Chapter 3. The main difference here is that it is easy to connect the same event handler to a whole set of elements at once. It is also easy to handle events other than the “click” event. Here is a partial list of events that can be handled this way.

<code> \$('h1').on('click', function(e)...) </code>	e.pageX and e.pageY represent the page coordinates of the click.
<code> \$('h1').on('dblclick', function(e)...) </code>	e.pageX and e.pageY are coordinates of a double-click.
<code> \$('h1').on('mousemove', function(e)...) </code>	e.pageX and e.pageY are coordinates of mouse motion.

<code>\$('#h1').on('keydown', function(e)...) </code>	e.which is the numeric code of a key being pressed.
<code>\$('#h1').on('keyup', function(e)...) </code>	e.which is the numeric code of a key being released.

Many other events can be captured; their names and descriptions can be found on the Web.

### Combining Input and Output with jQuery

Students can create useful interactive interfaces by combining input and output with jQuery. For example, the program below combines an on('click') event handler with .rt and .attr so that the image is spun and switched whenever it is clicked.

```
var trees = [
  '/img/elm-tree',
  '/img/maple-tree',
  '/img/pine-tree',
  '/img/cypress-tree',
  '/img/oak-tree'
];
$('#').appendTo('body');
$('#img').on('click', function() {
  $('#img').rt(360);
  $('#img').attr('src', random(trees));
});
```

### 12.1.1 Suggested Timeline: 1 55-minute class period

Instructional Day	Topic
2 Days	Lesson Plan I
2 Days	Lesson Plan II

### 12.1.2 Standards

CSTA Standards	CSTA Strand	CSTA Learning Objectives Covered
Level 3 B (Grades 9 – 12)	Collaboration (CL)	Use project collaboration tools, version control systems, and Integrated Development Environments (IDEs) while working on a collaborative software project.
Level 3 A (Grades 9 – 12)	Computing Practice Programming (CPP)	Use advanced tools to create digital artifacts (e.g., Web design, animation, video, multimedia)
Level 3 A (Grades 9 – 12)	CPP	Create and organize Web pages through the use of a variety of Web programming design tools.

### 12.1.3 Lesson Plan I

This lesson plan focuses on designing programs using JQuery commands using the timer program.

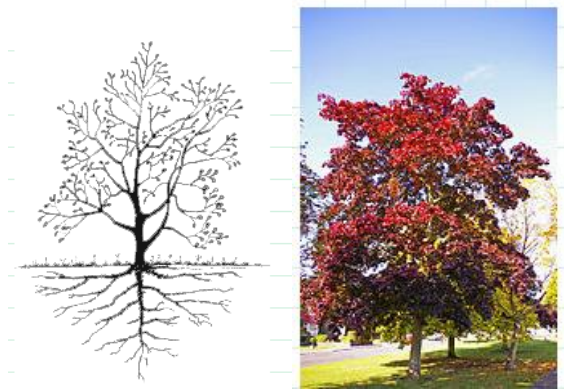
Content details	Teaching Suggestions	Time
<p>Code:</p> <pre data-bbox="201 373 717 779">\$('&lt;h1&gt;Countdown&lt;/h1&gt;').appendTo('body'); \$('h1').css({textAlign: 'center'}); var count = 10; forever(1, function() {   \$('h1').html(count);   count -=1;   if (count &lt; 0) {     \$('h1').html('blast off!');     stop();   } });</pre> <p>Output</p> <p><b>Countdown</b></p> <hr/> <p><b>8</b></p>	<p>Step 1. Demonstrate the timer <a href="#">program</a>.</p> <p>Step 2. Point out the key concepts where the various jQuery commands are explained.</p> <p>Step 3. Explain to the students that the \$ symbol calls a function.</p> <p>Step 3. Show the commands and their explanation and alt-tab between output and the actual program.</p> <p>Step 5. Encourage students to tinker with the code and modify it.</p>	<p>Demonstration: 15 minutes</p> <p>Student Practice: 30 minutes</p>

## 12.1.4 Lesson Plan II

This lesson plan demonstrates the power of JQuery for traversing data stored in arrays. It addresses traversal and showing data storage in 1D Arrays.

Content details	Teaching Suggestions	Time
<p>Code:</p> <pre data-bbox="201 407 743 810">var trees = [   '/img/elm-tree',   '/img/maple-tree',   '/img/pine-tree',   '/img/cypress-tree',   '/img/oak-tree' ]; \$('&lt;img src="/img/tree"&gt;').appendTo('body'); \$('img').on('click', function() {   \$('img').rt(360);   \$('img').attr('src', random(trees)); });</pre>	<p>Step 1. Demonstrate the magicTree <a href="#">program</a>.</p> <p>Step 2. Show students how the click function responds to the mouse click.</p> <p>Step 3. Point out the array named trees.</p> <p>Step 4. Point to the appendTo jQuery command.</p> <p>Step 5. Encourage students to modify the contents of the array and notice the various kinds of images that can be displayed.</p> <p>Step 6. Encourage students of try other jQuery commands in the code and see the results.</p>	<p>Demonstration: 15 minutes</p> <p>Student Practice: 30 minutes</p>

### Output



# APPENDIX A

## Pencil Code – Recommended Coding Standards

### Variable Naming Conventions:

- Variable and method names like `countPegs`, `x`, or `total` are lowercase, with occasional upperCase characters in the middle. Some call this the interCap method for naming variables. For example, a proper variable name is `numStudents` rather than `NumStudents` OR `num_students` OR `numstudents`.
- Constant names are UPPERCASE, with an occasional UNDER\_SCORE. For example, `BANK_FEE` is a good constant name rather than `BANKFEE` or `bankFee`.
- There are spaces after keywords like “if” and surrounding binary operators like “=” or “+”.
- There should be NO space after a function name like `connectionCost` or like `sqrt`. (In CoffeeScript, no space is even allowed between the function name and the parentheses in a function call!)
- You should use constants where appropriate.
- Every function must have a comment.
- Functions must be short: a rule of thumb is to limit them to 30 lines of code.

The following rules specify when to use upper- and lowercase letters in identifier names.

- Prefer lowercase for variable and function names (maybe with an occasional upperCase in the middle to help separate words); for example, `firstPlayer`.
- When using all-uppercase for constants, use underscores to separate words for example, `CLOCK_RADIUS` rather than `CLOCKRADIUS`.

Names must be reasonably long and descriptive. Your program is considered to be self-documenting if its variable names are descriptive.

### Braces:

- Braces should never be omitted where they are allowed. Although JavaScript allows single-line if, while, and for statements without braces, the braces should always be used with these statements. For example, in the example above, the braces after the “if” are not strictly required by the language, but they should be included. Missing braces here is one of the leading sources of bugs in professional code! Many professionals have learned over time to always include the braces.
- In CoffeeScript, of course, braces are not used because indents are used to reflect the nesting structure.



Indents:

- In JavaScript, every block of code surrounded by a `{ }` should be indented evenly by two spaces. Even though intents are not required in JavaScript, they should be used just as you are required to use them in CoffeeScript.
- Pencil Code by default uses 2-space indents.

### Sample Code to illustrate the programming standards stated above:

```
// How far apart are the points which must be connected
// to the origin? Five sets them five points apart.
INCREMENT_AMOUNT = 5;

// connectionCost totals up the cost for connecting each
// point within an (x, y) rectangle to the origin, assuming
// the points are in a grid determined by INCRMENT_AMOUNT
// and the cost per unit distance is given by costPerMile.
// Only points farther than minDist are included.
function connectionCost(x, y, minDist, costPerMile) {
  var total = 0;
  for (var i = 0; i < x; i += INCREMENT_AMOUNT) {
    for (var j = 0; j < y; j += INCREMENT_AMOUNT) {
      var dist = sqrt(x * x + y * y);
      if (dist > minDist) {
        total += costPerMile * dist;
      }
    }
  }
  return total;
}
```

## APPENDIX – B

### Links to the list of programs used in the manual

	Chapter	Links to the programs
2	Lines and Points	Lesson Plan II: <a href="http://teachersguide.pencilcode.net/edit/chapter2/dotRow">http://teachersguide.pencilcode.net/edit/chapter2/dotRow</a> <a href="http://teachersguide.pencilcode.net/edit/chapter2/smiley">http://teachersguide.pencilcode.net/edit/chapter2/smiley</a> Lesson Plan III: <a href="http://gym.pencilcode.net/draw/#/draw/first.html">http://gym.pencilcode.net/draw/#/draw/first.html</a> <a href="http://guide.pencilcode.net/edit/explainer/turns">http://guide.pencilcode.net/edit/explainer/turns</a> <a href="http://guide.pencilcode.net/home/explainer/curves">http://guide.pencilcode.net/home/explainer/curves</a> <a href="http://activity.pencilcode.net/home/worksheet/flower.html">http://activity.pencilcode.net/home/worksheet/flower.html</a> Lesson Plan IV: <a href="http://teachersguide.pencilcode.net/edit/chapter2/collage">http://teachersguide.pencilcode.net/edit/chapter2/collage</a>
3	Input/Output	Lesson Plan I: Drag and drop the blocks as shown in the lesson. <a href="http://teachersguide.pencilcode.net/edit/chapter3/questionBot">http://teachersguide.pencilcode.net/edit/chapter3/questionBot</a> Lesson Plan II: <a href="http://teachersguide.pencilcode.net/edit/Chapter3/Button">http://teachersguide.pencilcode.net/edit/Chapter3/Button</a> <a href="http://teachersguide.pencilcode.net/edit/Chapter3/Keydown">http://teachersguide.pencilcode.net/edit/Chapter3/Keydown</a> <a href="http://teachersguide.pencilcode.net/edit/Chapter3/MouseClick">http://teachersguide.pencilcode.net/edit/Chapter3/MouseClick</a> Lesson Plan III: <a href="http://teachersguide.pencilcode.net/edit/Chapter3/imgBot">http://teachersguide.pencilcode.net/edit/Chapter3/imgBot</a> Lesson Plan IV: <a href="http://teachersguide.pencilcode.net/edit/Chapter3/shapeBot">http://teachersguide.pencilcode.net/edit/Chapter3/shapeBot</a> Additional Ex.: <a href="http://gym.pencilcode.net">http://gym.pencilcode.net</a>
4	Loops	Lesson Plan I & II: <a href="http://guide.pencilcode.net/edit/loops/">http://guide.pencilcode.net/edit/loops/</a> Lesson Plan III: <a href="http://teachersguide.pencilcode.net/edit/chapter4/rainbowCircles">http://teachersguide.pencilcode.net/edit/chapter4/rainbowCircles</a> Lesson Plan IV: <a href="http://teachersguide.pencilcode.net/edit/chapter4/QuestionBot">http://teachersguide.pencilcode.net/edit/chapter4/QuestionBot</a> Lesson Plan V: <a href="http://teachersguide.pencilcode.net/edit/chapter4/interactionWhile">http://teachersguide.pencilcode.net/edit/chapter4/interactionWhile</a> Lesson Plan VI: <a href="http://teachersguide.pencilcode.net/edit/chapter4/forever">http://teachersguide.pencilcode.net/edit/chapter4/forever</a>
5	Functions	Lesson Plan I: <a href="http://teachersguide.pencilcode.net/edit/functions/remotcontrol">http://teachersguide.pencilcode.net/edit/functions/remotcontrol</a> Lesson Plan II: <a href="http://teachersguide.pencilcode.net/edit/functions/ShapeBot">http://teachersguide.pencilcode.net/edit/functions/ShapeBot</a> Lesson Plan III: <a href="http://teachersguide.pencilcode.net/edit/chapter5/starsInTheSky_I">http://teachersguide.pencilcode.net/edit/chapter5/starsInTheSky_I</a> <a href="http://teachersguide.pencilcode.net/edit/Chapter5/StarsInTheSkyII">http://teachersguide.pencilcode.net/edit/Chapter5/StarsInTheSkyII</a> <a href="http://teachersguide.pencilcode.net/edit/Chapter5/StarsInTheSkyIII">http://teachersguide.pencilcode.net/edit/Chapter5/StarsInTheSkyIII</a> Lesson Plan IV: <a href="http://teachersguide.pencilcode.net/edit/chapter5/starsInTheSky_I">http://teachersguide.pencilcode.net/edit/chapter5/starsInTheSky_I</a> <a href="http://teachersguide.pencilcode.net/edit/functions/moon">http://teachersguide.pencilcode.net/edit/functions/moon</a>
6	Selection Statements	Lesson Plan I: <a href="http://teachersguide.pencilcode.net/edit/chapter6/WakeUp">http://teachersguide.pencilcode.net/edit/chapter6/WakeUp</a> <a href="http://teachersguide.pencilcode.net/edit/chapter6/pattern">http://teachersguide.pencilcode.net/edit/chapter6/pattern</a> <a href="http://teachersguide.pencilcode.net/edit/chapter6/DiceRoll">http://teachersguide.pencilcode.net/edit/chapter6/DiceRoll</a> Lesson Plan II:

	Chapter	Links to the programs
		<a href="http://teachersguide.pencilcode.net/edit/chapter6/questionbot">http://teachersguide.pencilcode.net/edit/chapter6/questionbot</a> Lesson Plan III: <a href="http://teachersguide.pencilcode.net/edit/chapter6/genieComplete">http://teachersguide.pencilcode.net/edit/chapter6/genieComplete</a> Lesson Plan IV: <a href="http://teachersguide.pencilcode.net/edit/chapter6/hiLo">http://teachersguide.pencilcode.net/edit/chapter6/hiLo</a> Lesson Plan V: <a href="http://activity.pencilcode.net/home/worksheet/race.html">http://activity.pencilcode.net/home/worksheet/race.html</a>
7	Learning a Second Language: JavaScript	Lesson Plan I: <a href="http://teachersguide.pencilcode.net/edit/chapter7/spiral">http://teachersguide.pencilcode.net/edit/chapter7/spiral</a> <a href="http://teachersguide.pencilcode.net/edit/chapter7/RandomSpirals">http://teachersguide.pencilcode.net/edit/chapter7/RandomSpirals</a> Lesson Plan II: <a href="http://teachersguide.pencilcode.net/edit/chapter7/brokenScene">http://teachersguide.pencilcode.net/edit/chapter7/brokenScene</a>
8	Introducing Lists and One-Dimensional Arrays	Lesson Plan I: <a href="http://teachersguide.pencilcode.net/edit/chapter4/rainbowCircles">http://teachersguide.pencilcode.net/edit/chapter4/rainbowCircles</a> Lesson Plan II: <a href="http://teachersguide.pencilcode.net/edit/chapter7/dotOfArrays">http://teachersguide.pencilcode.net/edit/chapter7/dotOfArrays</a> Lesson Plan III: <a href="http://teachersguide.pencilcode.net/edit/chapter8/push_pop">http://teachersguide.pencilcode.net/edit/chapter8/push_pop</a> Lesson Plan IV: <a href="http://teachersguide.pencilcode.net/edit/chapter8/bar">http://teachersguide.pencilcode.net/edit/chapter8/bar</a> Lesson Plan V: <a href="http://teachersguide.pencilcode.net/edit/chapter8/pie">http://teachersguide.pencilcode.net/edit/chapter8/pie</a> Lesson Plan VI: <a href="http://teachersguide.pencilcode.net/edit/chapter8/searchtxt">http://teachersguide.pencilcode.net/edit/chapter8/searchtxt</a>
9	Nested Loops	Lesson Plan I: <a href="http://teachersguide.pencilcode.net/edit/2DArray/box">http://teachersguide.pencilcode.net/edit/2DArray/box</a> <a href="http://teachersguide.pencilcode.net/edit/chapter9/nest">http://teachersguide.pencilcode.net/edit/chapter9/nest</a> Lesson Plan II: <a href="http://teachersguide.pencilcode.net/edit/chapter9/colorfulMatrix">http://teachersguide.pencilcode.net/edit/chapter9/colorfulMatrix</a> <a href="http://teachersguide.pencilcode.net/edit/chapter9/triangle">http://teachersguide.pencilcode.net/edit/chapter9/triangle</a> <a href="http://teachersguide.pencilcode.net/edit/chapter9/XMasTree">http://teachersguide.pencilcode.net/edit/chapter9/XMasTree</a>
10	Recursion	Lesson Plan I: <a href="http://teachersguide.pencilcode.net/edit/chapter10/recursiveSquares">http://teachersguide.pencilcode.net/edit/chapter10/recursiveSquares</a> Lesson Plan II: <a href="http://teachersguide.pencilcode.net/edit/chapter10/spiral">http://teachersguide.pencilcode.net/edit/chapter10/spiral</a>
11	Building a Website: Using HTML, CSS in the Pencil Code Environment	Lesson Plan I: <a href="http://teachersguide.pencilcode.net/edit/chapter11/FirstPageArtGallery">http://teachersguide.pencilcode.net/edit/chapter11/FirstPageArtGallery</a> Lesson Plan II: <a href="http://teachersguide.pencilcode.net/edit/chapter11/OnePage">http://teachersguide.pencilcode.net/edit/chapter11/OnePage</a> Lesson Plan III: <a href="http://teachersguide.pencilcode.net/edit/chapter11/FirstPageArtGalleryDesigns">http://teachersguide.pencilcode.net/edit/chapter11/FirstPageArtGalleryDesigns</a> Lesson Plan IV: <a href="http://teachersguide.pencilcode.net/edit/chapter11/slideshow">http://teachersguide.pencilcode.net/edit/chapter11/slideshow</a>
12	Traversing Data Using JQuery	Lesson Plan I: <a href="http://teachersguide.pencilcode.net/edit/chapter12/timer">http://teachersguide.pencilcode.net/edit/chapter12/timer</a> Lesson Plan II: <a href="http://teachersguide.pencilcode.net/edit/chapter12/magicTree">http://teachersguide.pencilcode.net/edit/chapter12/magicTree</a>



## Appendix C - Pacing Guide

### 1 semester course

Almost 12 Weeks:

Time	Pencil Code Topic	Programming Concept	Examples / Comments etc.
3 days	Basic Blocks: Move, Art, Text, Images	Sequencing	Chapter 2,3
2 days	Blocks revisited: Sound, Text, Controls	Input / Output	Chapter 3 Sample programs (2) Spiral Program: Beginnings of a QuestionBot.
5 days	Operators, Controls	Iterations	Battery of programs Spiral Program: QuestionBot
2 days	Transitioning to text-mode (CoffeeScript)	Iterations	Practice tracing loops Battery of programs
3 days	Operators	Functions	Battery of programs Spiral Program: QuestionBot Creation of pieces of scenes. Chapter 5 <i>(One large program)</i>
3 days	CoffeeScript programming	Iterative Development process	Design a scene. <i>(Use the Middlebury College <a href="#">course</a> on Pencil Code for ideas.)</i> Chapter 5
5 days	Control (CoffeeScript)	Conditionals	Battery of Programs (Chapter 6) Spiral Program: Questionbot Complete.

<b>Time</b>	<b>Pencil Code Topic</b>	<b>Programming Concept</b>	<b>Examples / Comments etc.</b>
5 days	HTML – CSS	Building a website	Build a simple page using HTML blocks in Pencil Code. Chapter 11
3 days	JavaScript programming	Iterative Development process	Design a complex pattern. Or Design a Scene lab Chapter 7, 5
5 days	Block – text mode (JavaScript)	Arrays & Nested Loops	Battery of simple programs – Chapter 8 Hi Lo – Chapter 6 (Code in JavaScript)
5 days	HTML- CSS – Block language based program	Embedding a program in a HTML file	Build a website (multiple pages) with embedded JavaScript program. Chapter 11
3 days	Text mode (JavaScript)	A light introduction to Recursion	Chapter 10 A battery of 3 programs.
5 days	Block – text mode	Use of JQuery	Chapter 12
5 days	Large Project	Putting it all together. (No use of HTML)	Complex Patterns, ASCII Art
5 days	Open-ended partner project	Putting it all together (Use of HTML)	Art / Music / Stories