CREATING ART, ANIMATIONS, AND MUSIC through Coding

Glen Bull, Jo Watts, and N. Rich Nguyen
Acknowledgements

Creating Art, Animations, and Music through Coding stems from a collaboration between the Department of Computer Science and the School of Education and Human Development at the University of Virginia. Modules with instructional videos and resources that support topics in the book were originally developed with support from a National Science Foundation grant (DRL-1842342, Glen Bull, P.I.). These resources are available on the Make to Learn web site at:

https://www.maketolearn.org/creating-art-animations-and-music/

These modules were initially developed in conjunction with creation of a tool, SoundScope, designed to facilitate exploration of sound and music. SoundScope combines an audio frequency digital oscilloscope with audio oscillators that can be used to synthesize complex sounds and music. SoundScope was designed by Glen Bull, a professor of education and implemented by Rich Nguyen, an assistant professor of computer science. This served as the foundation of a broader collaborative effort.

A subsequent programming environment, TuneScope, was developed that integrates SoundScope with the educational programming language, Snap!. TuneScope extends the music creation capabilities of Snap! through addition of sampled musical instruments such as pianos, guitars, and drums. Eric Stein, a computer science student working under the direction of Rich Nguyen, developed TuneScope. Another computer science student, Yuxin Wu, provided leadership for alignment of each topic with the Advanced Placement Computer Science (APCS) standards.

The art chapters were developed by Glen Bull, Jo Watts, and Alexis Kellam. Jo Watts, manager of the Make to Learn Laboratory, coordinated the overall project. Alexis Kellam, a student in the School of Education and Human Development, served as art editor and provided illustrations for the book. She also provided leadership for development of instructional videos in support of the project.

The music chapters were developed by Glen Bull, Jo Watts, and Joe Garofalo. Joe Garofalo is an associate professor of mathematics education and co-director of the Make to Learn Laboratory. Throughout his career, Joe has integrated music in his mathematics courses. Of greater relevance to this project, Joe also accompanied his daughters, Leah and Ruby, to music camp for many summers, exploring ways in which music can serve as a doorway to other interests.

Michael Littman, a professor of engineering at Princeton University, has served as an advisor and consultant to the project from its inception, providing expertise on the physics of sound and music. Elaine Wolfe, a retired programmer and mathematics teacher, also served as a consultant, providing advice and lending expertise on ways in which digital objects on the screen can be translated into three-dimensional art in the physical world. Gina Bull, retired computer science engineer (and wife of Glen Bull), served as technical editor for the book.

We also would like to acknowledge and thank the sculptor, Bathsheba Grossman, and the illustrator, Peter Reynolds, for their contributions to the effort. Hollyn Slykhuis, a music educator, piloted several of the music activities and provided advice on ways to introduce music in an engaging way. We owe a debt of gratitude to Summer Hayes for her cover of the Robert Johnson rendition of the song, Crossroads.

Most of all, we would like to acknowledge an enormous debt to Brian Harvey and Jens Möenig for creation of the computing environment, Snap!, that made our work possible. We appreciate their support and encouragement throughout this effort and thank them for the many questions that they answered. Brian and Jens created a world and a community that has facilitated and enabled creative work by many others, including this working group.
# Table of Contents

Preface .................................................................................................................. 6  
Introduction to the Snap! Coding System .............................................................. 7  

## Section A. Art

1. Creating Artistic Patterns through Computer Graphics .................................. 19  
2. Exploring Color ............................................................................................... 31  
3. Creating Colorful Patterns ............................................................................ 45  
4. Simulating an Impressionist Painting .............................................................. 55  
5. Designing a Paint Program ............................................................................ 62  
6. Mobile Art ....................................................................................................... 73  

## Section B. Music

7. Computer-Generated Sound and Music ........................................................... 85  
8. Musical Notes and Scales ............................................................................... 97  
9. Chord Player and the Blues .......................................................................... 106  
11. A Tune Recorder / Player ............................................................................. 126  
12. Building a Lyric Display Machine .................................................................. 142  

## Section C. Hypermedia Tools

13. Acoustic Tools ............................................................................................... 156  
14. Color Tools .................................................................................................... 172
Preface

This book was developed to accompany the course, *Creating Art, Animations, and Music through Coding*. The goal of the course is to provide an introduction to creativity at the intersection of technology and the arts.

Each of the six arts chapters provides an introduction to a different artist – the sculptors Alexander Calder and Bathsheba Grossman, the nineteenth century post-impressionist artist, Georges Seurat, the twentieth century artists Mark Rothko and Jason Pollock, and the contemporary illustrator, Peter Reynolds. Each of these artists works in a different style and in different mediums. These styles provide opportunities to emulate these artistic approaches through the medium of digital technologies.

Each of the music chapters also features a different musician, beginning with Robert Johnson, the blues guitarist and singer whose landmark recordings in 1936 and 1937 influenced musicians and groups as diverse as Eric Clapton, the Beatles, and the Rolling Stones. The twelve-bar blues is a tradition that can be traced back to the nineteenth century. This form of the blues enables a musician to play hundreds of different songs with only three chords. In a similar manner, a few basic concepts enable creators to create hundreds of different types of programs with only a few foundational commands.

The course is a collaboration between the School of Education and Human Development and the Department of Computer Science. Although it is approved as an elective in the Bachelor of Arts in Computer Science program at the University of Virginia, the primary goal is not to teach computer science. Rather, the objective is to facilitate creativity and exploration of the arts through coding.

The educational computing language *Snap!* is used as the foundation for the course. This computing language was developed at the University of California, Berkeley, with support from the National Science Foundation and is used in undergraduate courses for non-computer science majors. It was jointly developed by Brian Harvey and Jens Möenig with the goal of making advanced computational concepts accessible to nonprogrammers. Brian Harvey notes, “Languages in the Logo family, including Scratch and Snap!, take the position that our mission is to bring programming to the masses.”

*Snap!* is tightly integrated with a course, the *Beauty and Joy of Computing*, that has been taught as an Advanced Placement (AP) Computer Science course by more than a thousand high school computer science instructors. According to Dan Garcia, who currently teaches undergraduate *Snap!* courses at the University of California, Berkeley, the educational computing language currently has a half-million users who have developed more than four million projects with it.

One of the most important elements of *Snap!* is community drawn together by its philosophy. This active community of users can be found on the *Snap! Forum* at:

https://forum.snap.berkeley.edu/

The active community engaged in active discussion ranges from students in their teens to professionals in their seventies. Almost any question or request for assistance receives a helpful answer within a few hours. It is not unusual to receive a response directly from the creators of *Snap!*, Brian Harvey and Jens Möenig. If you are engaged by the arts, and particularly creation of art and music using digital technologies, we invite you to join this community of users with similar interests. The book *Creating Art, Animations, and Music through Coding* is intended to provide an introduction to that world.
An Introduction to the Snap! Coding System

This initial module provides an orientation to the Snap! programming language. Snap! was developed at the University of California, Berkeley, and is used in computer science (CS) for non-CS majors at that university. Because of its widespread use, a user community has developed around this programming language. The capabilities of this language make it well-suited to explorations in art and music.

Topic 0.1 Securing a Snap! Account

A free Snap! account can be obtained from the University of California, Berkeley web site:

https://Snap.berkeley.edu/Snap

Secure a Snap! account before continuing. The Snap! reference manual is available here:


Section II of the reference manual provides information about securing a Snap! account and saving projects. Review this section before continuing to the explorations that follow below.

Snap! help forums maintained at the University of California, Berkeley, are available here:

https://forum.Snap.berkeley.edu/

Assistance with questions that are not addressed in the reference manual can be obtained through the Snap! forum.

Topic 0.2 The Snap! Workspace

In a typical Snap! session, blocks of code are dragged from the Code Block Palette on the left-hand side of the screen to a Script Area in the middle of the screen. Blocks of code are snapped together to create scripts. Clicking a group of code blocks causes the script to run, performing an action. Often these actions involve movement of sprites on a Stage at the right-hand side of the screen.

A sprite corral beneath the stage indicates which sprite is currently selected. Each sprite has its own separate script space.
Topic 0.3 Snap! Menus

There are several menus that can be accessed in the top left-hand corner of the Snap! screen. The Snap! icon in the top left-hand corner can be used to access the Snap! reference manual.

The File menu to the right of the Snap! icon can be used to save projects and open projects that have been previously saved. This menu can also be used to access costumes for sprites, sounds, and libraries of additional Snap! code blocks.

The Login menu is to the right of the File menu.

A user must be logged into Snap! in order to save projects. Therefore it is a good idea to log in to Snap! at the beginning of every session.

A Settings menu is to the right of the Login menu. This menu provides a number of options for customizing Snap! For example, the Zoom Blocks option can be used to increase the size of the code blocks so that they can be more easily viewed in presentations.

Exploration 0.3

If you have not already, log into Snap! Explore some of the options available under the different menu settings.
**Topic 0.4 The Code Block Palette**

Scripts are created in Snap! by snapping blocks of code together (such as the Move 10 Steps code block in the illustration). The types of code blocks available are displayed in a Code Block Palette at the top left-hand side of the screen. For example, the Motion code blocks are currently highlighted in the palette below. Other categories of code blocks include Looks, Sound, Pen, Control, Sensing, Operators, and Variables. Each category is a different color (e.g., Motion code blocks are blue). Click on the different categories (Motion, Looks, etc.) to access the code blocks associated with that category.

The Motion code blocks direct the movement of sprites (actors that can move about the stage on the right-hand side of the screen.) The Looks code blocks control the appearance of sprites. The Sound code blocks are used to play sounds. The Pen code blocks control the color and thickness of the turtle’s pen. The Control code blocks provide control structures such as the Repeat command. The Sensing code blocks are used to sense the status of Snap! objects and monitor external inputs such as the keyboard and the microphone. The Operators code blocks provide mathematical and logical functions. The Variables palette is used to create and modify variables.

**Exploration 0.4 The Code Block Palette**

Click on each of the categories in the Code Block Palette to get a sense of the types of commands that are found under each category.

**Topic 0.5 A Digital Clock**

Sound is produced by the back-and-forth motion of a vibrating object. The rate at which the back-and-forth motion occurs is an important characteristic of sound. Two closely related terms used to describe this characteristic are frequency and period.

*Frequency.* The rate at which a back-and-forth motion occurs over a given amount of time. The number of back-and-forth events that occur within a given time period is known as frequency.

*Period.* The duration of time for a single back-and-forth event is known as its period.

These terms are used to characterize sound, but are also used to define other non-auditory events. The motion of the rotating flywheel of an engine can be described in terms of revolutions per minute. For example, the flywheel might be described as rotating at a rate of 60 revolutions per minute (rpm). In this instance, the frequency of motion could be described as 60 rpm. If the frequency is 60 rpm, this means that one complete rotation would occur in one second. Therefore, the period in which a single
rotation occurs could be described as 1 second. In this example, frequency and period are related. If the frequency increases, the period in which one rotation occurs will decrease.

The concepts of frequency and period can be applied to the back-and-forth vibration of an object that produces sound in a similar manner. The time scale for vibration that produces sound is typically events per second rather than events per minute, however.

A digital clock is at the heart of every digital computer. The digital clock is used to control the timing and rate at which computer instructions are executed. It can also be used to create a digital timer that can be used to time events.

**Topic 0.6 Rotating the Turtle**

The examples that follow make use of the digital clock to illustrate ways in which code blocks from several different code block palettes are used. In this example a timer will be constructed that will keep track of seconds as the turtle rotates. The user will be able to stop the timer (and the turtle) by pressing the space bar.

The code blocks in the *Motion Palette* are blue. Drag the **Turn Right** code block into the script area and enter 90 degrees as the input to this code block. Then click the code block. Each time the code block is clicked, the screen turtle should turn right 90 degrees.

**Exploration 0.6 Rotating the Turtle**

Explore some of the other code blocks in the *Motion Palette*. What does the **Move** code block do?
Topic 0.7 Controlling the Timing of the Turtle’s Rotation

Go the Control Palette and drag a **Wait** block onto the script area. Snap it into place beneath the **Turn** block.

Then drag a **Forever** block from the Control Palette into the script area.

Drag the combined **Turn** and **Wait** blocks into the open space within the C-shaped **Forever** block.

The combined blocks should look like this.
Click the combined group of code blocks. (A group of code blocks that have been snapped together in this way is referred to as a script.) When the script is executed, an outline or halo will surround the script to indicate that the program is running. The screen turtle should turn right by 90 degrees one time per second. (**Forever** means that the turtle will continue to turn until the user stops the script.)

Click the red *Stop* button in the top right-hand corner of the screen to stop the script (or click the highlighted script).

**Exploration 0.7 Controlling the Timing of the Turtle’s Rotation**

Explore some of the other uses of variables. How could a timer be created that counts down the seconds?

**Topic 0.8 Tracking Elapsed Seconds**

Variables provide a way to keep track of minutes and seconds as the timer counts. To create a *Seconds* variable to track seconds, select the *Variables Palette* and click the *Make a Variable* button.

Enter the variable name *Seconds* in the dialog box that appears. (Leave the default option of “for all sprites” selected.) This variable will be used to track elapsed seconds.
A variable watcher *Seconds* will appear on the stage. All of the variables that have been created are listed at the top of the *Variables Palette*. When the checkbox beside each variable is selected, the *Variable Watcher* will appear on the stage. The variable watcher can be used to monitor the status of the variable.

To count seconds, the seconds variable will need to increase the count each time a second elapses. Drag the code block **Change __ by 1** into the script area. Select the variable *Seconds* from its drop-down menu.

Drag the **Change Seconds by 1** block into the **Forever** block. Click the group of blocks to execute the script. The variable *Seconds* will increase by one each time the turtle turns.
The revised script now functions as a timer. The code block “Set Seconds to ___” can be placed before the *Forever* block to ensure that the timer always begins with a starting value of 0. (This step is called *initialization* because it establishes the initial value of the variable.)

![Code block showing initialization](image1.png)

**Exploration 0.8 Tracking Elapsed Seconds**
Create a timer that tracks elapsed seconds.

**Topic 0.9 Stopping the Timer**
A second script can be added to stop the timer when the space bar is pressed. Begin by dragging the *When* code block from the *Control Palette* into the script area.

![When block](image2.png)

Select the *Sensing Palette* and drag the *Key Space Pressed* block into the hexagonal slot in the *When* block.

![Key Space Pressed block](image3.png)

Then add the *Stop All* block from the *Control Palette.*
Finally, add the **Green Flag** block to the other script. The green flag block will enable the user to start the program by clicking the green flag in the upper right-hand corner of the screen. Pressing the space bar on the keyboard will stop the timer.

![Diagram of Scratch blocks]

**Exploration 0.9 Stopping the Timer**

Explore some of the other code blocks in the *Sensing Palette*. What does the **Mouse Down** code block do? How could this code block be incorporated into the timer?

**Topic 0.10 Counting Revolutions of a Rotating Turtle**

To count the number of revolutions of the turtle that occur within a given time period, add a *Revolutions* counter. Use the **Make a Variable** button in the *Variables Palette* to create a variable named *Revolutions*.

![Diagram of Scratch variables]

Begin by setting *Seconds* and *Revolutions* to an initial state of zero. Point the turtle straight up. This process of establishing the conditions for an initial state is known as *initialization*.
The **Direction** code block reports the direction in which the turtle is pointing. In the illustration below, the turtle is pointed to the right, at an angle of 90 degrees.

The **Direction** block can be combined with an **If** code block to increase the **Revolutions** counter by one each time turtle returns to a heading of 0 degrees after completing a full rotation. (The **Equals** code block used in this example is found under the green *Operators* palette.)

The **If** code block is placed within the **Forever** code block. Each time the turtle completes a rotation, the **Revolutions** counter is increased by one.
The completed procedure initializes *Seconds* and *Revolutions* when the green flag is clicked. It also starts the loop that rotates the turtle and counts revolutions and seconds until the space bar is pressed.

In this illustration, the turtle completed 15 revolutions in 60 seconds. The frequency of rotation, therefore, can be described as *15 revolutions per minute*. Dividing 15 revolutions into 60 seconds yields 4 seconds. The *period* of one rotation, therefore, is 4 seconds.

**Exploration 0.10 Counting Revolutions of a Rotating Turtle.**

Change the *Wait* time to 0.5 seconds. How many revolutions occur in one minute when this change in timing is made? How does this affect the period of time that it takes each revolution to occur?

**Topic 0.11 Saving the Script**

This is a good point to save the script. If the *Save* option (under the *File* menu) is used when the script is first saved, it will be given the default name of “Untitled.” Since that name is not descriptive, use the *Save As* selection the first time that the script is saved.

This will produce a dialog box that allows the project to be given a name. In this instance, the project has been named, “Timer.” The project can be saved on a local computer. The advantage of saving the project in your Snap! account in the cloud is that it can be later retrieved from any computer with an Internet connection. (You must be logged into Snap! in order to save projects.)
After the project is saved, the project name should replace the default name of “Untitled”. After the project is saved and given a name, the *Save* option under the *File* menu can also be used to save future changes and revisions that may be made to the script.
A. Art
1. Creating Artistic Patterns through Computer Graphics

Glen Bull, Jo Watts, and Alexis Kellam

Many artists now use computer graphics to create artistic patterns. Bathsheba Grossman is representative of a new generation of artists who use computers to extend the type of art that they create. The design below was created by Bathsheba Grossman (https://bathsheba.com/crystal/calabiyau/) using a computer-generated pattern.

These artists use computer code to create art. Some of the techniques that they use are available to anyone with an interest in exploring art in this way.

Topic 1.1 Algorithmic Thinking – Finding a Pattern

The first robotic turtles were developed at M.I.T. These robots consisted of Plexiglas hemispheres that rolled about the floor. They were called “turtles” because the Plexiglas hemispheres resembled the shell of a turtle. A pen in the belly of the turtle could be lowered to draw patterns on paper placed on the floor.

Robotic turtles are still used to draw patterns on paper today. However, a turtle on the screen of the computer can be used to create a design more quickly. Screen turtles typically consist of a triangle on the screen of the computer. The Pen Up and Pen Down code blocks are used to raise and lower the pen in the belly of the screen turtle.

The explorations that follow begin with simple patterns. These simple patterns are then used to create more complex patterns.

A square, for example, is one of the most basic patterns that can be created. A square is created by drawing a line and then turning 90 degrees. For example, if the turtle begins in the position shown below (pointing left):

- move 100 steps
- turn 90 degrees

...
then executing the code blocks Move 100 Steps followed by Turn 90 Degrees causes the turtle to draw a line and then turn right.

Repeating the Move and Turn code blocks four times causes the turtle to draw a square.

However, repeating the pattern four times achieves the same result more efficiently. Computers are efficient at repeating a pattern many times. However, the human coder must recognize the pattern in order to take advantage of the computer’s capabilities in this way.

Exploration 1.1 Algorithmic Thinking – Finding a Pattern

Identify the pattern used to create other shapes such as triangles (three-sided figures) and polygons (five sided figures). Assemble and run code blocks that draw shapes such as triangles and polygons.
Essential Knowledge

The example above illustrates the process of repeating a series of steps to draw a square.

Algorithms

The term algorithm refers to a set of instructions that accomplish a task. A computer algorithm refers to a set of instructions that can be implemented by a computer.

CS Principle 3.8 Iteration

A computer often needs to repeat the same series of steps several times. The term iteration refers to a repeated sequence of steps. In the example in the preceding section, a series of steps is repeated to create an artistic pattern.

CS Principle 3.9 Developing Algorithms

In the preceding example, an algorithm for drawing a square is developed. Development of this algorithm depends on recognition that the Move code block and the Turn code block can be combined and repeated to form a square.

Application of Essential Knowledge

An algorithm can be written in multiple ways and still accomplish the same task. A square can be drawn in eight steps; the same pattern can be drawn more efficiently by using a Repeat code block in combination with one Move code block and one Turn code block.

Topic 1.2 Abstraction – Making a Custom Code Block

The limits of human memory place limits on creation of computer code. A human can only retain a limited number of code blocks in memory at one time. Experienced coders circumvent these limits through abstraction. Once code has been developed and tested, it can be abstracted through creation of a custom code block.

Clicking the plus sign near the top of the command palette invokes the option to make a custom block.
A dialog box appears when the *Make a Block* option is invoked. The dialog box includes a space to enter the name of a new custom block. (Note: in most instances the default selection of “for all sprites” is used.)

![Make a block dialog box](image)

When the entered name is confirmed by clicking “OK,” a block editor appears. The working code previously developed and tested can then be dragged into the block editor and snapped into place beneath the header block at the top of the block editor.

![Block Editor](image)
Once this choice is confirmed by clicking “OK,” the newly defined custom block appears at the bottom of the other commands in the command palette.

The new custom block can then be dragged into the script area and used in the same way as any other code block.

Creating a custom block reduces the burden on the coder’s memory. Instead of remembering the correct order of an entire sequence of code blocks, the coder now only needs to remember the name of a single custom block. For this reason, it is important to create a meaningful name for the custom block so that its purpose is clear.

The introduction to a famous computer science textbook, the *Structure and Interpretation of Computer Programs*, asserts in the preface that “Programs must be written for people to read, and only incidentally for machines to execute” (Abelson & Sussman). Elegance in programming is a balance between efficiency (writing code in as few lines as possible) and readability (creating custom code blocks with meaningful names). Readability ensures that code can readily be maintained and re-used in other programs.

**Essential Knowledge**

A procedure is a group of program instructions that have been given a name. In the example above, the instructions for drawing a square are assigned the name “Square”.

23
CS Principle 3.11 Calling Procedures

In Snap!, a custom code block is created by assigning a name to a group of instructions. Custom code blocks can be used in Snap! in the same way that built-in code blocks (known as primitives) are used.

CS Principle 3.12 Developing Procedures

Assigning a name to a group of instructions to create a procedure is one form of abstraction. Once a code sequence is tested and encapsulated in the form of a procedure, the programmer no longer needs to remember the specific details of the procedure, enabling the programmer to focus on other program details.

Application of Essential Knowledge

In the example above, a custom code block named Square replaced a longer sequence of instructions. Replacing an entire sequence of instructions with a named procedure yields several benefits. First, it makes the top level of the program more concise and easier to read. Second, if meaningful names are used, it makes the program easier to follow. Finally, it reduces the cognitive load on the programmer by replacing a longer sequence of code with a procedure name, enabling the programmer to devote their attention to other details of the program.

Exploration 1.2 Abstraction – Making a Custom Code Block

Create custom code blocks for other shapes such as Triangle and Pentagon.

Topic 1.3. Generalization – Creating Inputs for Custom Code Blocks

One of the goals of effective programming is creation of re-usable code. Inputs to custom code blocks make these code blocks more generalizable.

The previously created **Square** code block made a square that was 100 turtle steps on each side. In other circumstances, a square that was 50 steps on a side or 200 steps on side might be needed. One approach to addressing this need consists of creating a new custom block – such as **Small Square** and **Big Square** – for each new application.

However, a more effective approach would be to incorporate an input into the procedure that made it possible to use a single code block to draw a square of any size.

Right-clicking the **Square** code block reveals a drop-down menu that offers the option of editing the code block.
Selection of the *Edit* option accesses the *Block Editor*. If the plus sign (“+”) beside the name *Square* is clicked, another dialog box appears that offers the option of entering an input name.

Once an input name is selected (*Size* in this instance), the choice can be confirmed by clicking “OK.”

Once this choice is confirmed, the input name (*Size*) will appear to the right of the block name (*Square*).
This orange oval representing the input Size can then be dragged into the input slot of the Move code block.

Once this revision of the custom code block is confirmed, an input slot will appear to the right of the name of the code block.

The input slot makes it possible to use the same code block to create squares of three different sizes, rather than creating three different code blocks for the same purpose.

Use of inputs to create more generalizable procedures greatly increases the efficiency of coding.

Essential Knowledge

**CS Principle 3.12 Calling a Procedure**

Procedures are implemented in the form of custom code blocks in Snap!. In Snap!, a code block can be executed (i.e., called) by clicking it. Procedures in the form of code blocks can have inputs. For example, an input to the Square procedure can be used to specify the size of the square. In computer science, the term parameter can be used to describe an input such as size that can be varied in this manner.

**CS Principle 3.13 Developing Procedures**

Use of an input such as size allows the procedure to be generalized. This enables the procedure to be re-used to draw squares with a wide range of different sizes.

Application of Essential Knowledge

In the example above, the numbers “50”, “100”, and “200” are used as inputs to the procedure. The size parameter allows the procedure to be generalized, enabling Square to be used to draw squares of different sizes.

**Exploration 1.3 Generalization – Creating Inputs for Custom Code Blocks**

Create inputs for other shapes such as the Triangle and Pentagon code blocks.
Topic 1.4 Building Complex Code Block by Block

Effective coding begins with construction of basic building blocks that are used to create more complex procedures.

For example, if a series of squares is drawn, turning a bit after each square is drawn, the following pattern results.

In this manner, the Square code block becomes a building block for construction of another custom block, Spin Square.

Surprisingly complex patterns can be generated through construction of a library of custom code blocks in this manner.

Essential Knowledge

CS Principle 3.12 Developing Procedures

An effective coding strategy breaks a larger problem into a series of smaller sub-problems. Procedures are created to solve each of the smaller sub-problems. The larger problem is then solved by combining the procedures that were developed to solve each of the sub-problems.

Procedural abstraction allows a solution to a large problem to be based on the solutions of smaller sub-problems. This is accomplished by creating procedures to solve each of the sub-problems.

Application of Essential Knowledge

In the example above, in order to create the Spin Square procedure, the larger problem was separated into two smaller problems: (1) drawing the square and (2) turning the square. The Spin Square code block is based on the procedure Square. Development of the Square procedure makes the code easier to understand when it is incorporated into a more complex program in the future.

Exploration 1.4 – Building Complex Code Block by Block

Incorporate the Triangle and Pentagon code blocks into other code blocks to create complex patterns.
Topic 1.5 Using Loops to Vary a Parameter

The **Repeat** block represents one type of loop in which the same action is repeated over and over again. A **For** block is another type of loop. One advantage of a **For** loop is that the variable \( i \) changes each time that the loop is executed.

In the example below, the variable \( i \) has a value of “1” the first time the loop is executed. The value “1” is multiplied by 10 in the input to the **Square** code block. Consequently, a square with a size of 10 steps on a side is drawn the first time the loop is executed.

The second time that the loop is executed, a square is drawn that is 20 steps on a side. Since \( i = 2 \) during the second execution of the loop, the input to the **Square** block becomes \( 2 \times 10 \) (i.e., 20).

The last time that the loop is executed, a square that is 100 steps on a side is drawn. In this manner, a series of nested blocks is drawn using just two code blocks, the **For** loop combined with the **Square** code block.

The ability to vary the input of a code block each time that the loop is executed can therefore yield some powerful effects in an efficient way. Almost every program of any complexity will make use of one or more loops in this manner.

**Essential Knowledge**

**CS Principle 3.1 Variables and Assignments**

A variable is an abstraction inside a program that can hold a value. Each variable has associated data storage that represents one value at a time.

**CS Principle 3.3 Mathematical Expressions**

An expression can consist of a value, a variable, an operator, or a procedure call that returns a value. Expressions are evaluated to produce a single value.

Arithmetic operators are part of most programming languages and include addition, subtraction, multiplication, division, and modulus operators.

**Application of Essential Knowledge**

In the example above, we evaluate a mathematical expression that consists of the variable \( i \) and a multiplication operator to draw the squares with different sizes.

**Exploration 1.5 – Using Loops to Vary a Parameter**

Explore creation of nested shapes such as triangles and pentagons through use of loops.
Topic 1.6 Fabricating Art with Patterns

Tools such as inkjet printers, laser cutters, and other fabrication tools can be used to translate digital designs into ornaments, jewelry, and sculptures in the same manner as digital artists.

Print your design with an ink-jet printer. If you have access to a laser cutter or 3D printer, use the pattern to create a three-dimensional ornament.

Graphics drawn with the turtle can be exported as Scalable Vector Graphics (SVG) files for higher resolution output. To access the Scalable Vector Graphic feature, turn on the Log Pen Vectors option in Settings.

After a Snap! program has been used to draw a design, place the mouse cursor on any part of the design and right-click to access the menu with the “SVG Export” option.
The SVG file can then be imported into other graphics programs such as Silhouette Studio. (Silhouette Studio Business Edition includes an option to import SVG files.)

Foil Quill is a third party option available for the Silhouette die cutter that can be used to emboss foil patterns onto materials such as card stock.

The result is an embossed foil pattern obtained by using an SVG vector pattern generated by Snap!.

**Exploration 1.6 – Fabricating Art with Patterns**

Design a card with your own unique artistic design.
2. Exploring Color

*Glen Bull, Jo Watts, and Alexis Kellam*

An understanding of the Snap! color framework explored in this chapter is required for subsequent art modules. These modules include one that simulates an impressionist painting, another module in which a paint program is developed, and a module involving creation of an animated coloring book. Each of these modules makes use of color.

**Creating Art through Coding**

Mark Rothko was an American abstract painter who explored color usage through his artwork. His signature colorfield paintings depict irregular and rectangular areas, known as multi-forms, that encouraged color contemplation and appreciation. The majority of these multi-forms were horizontally painted blocks. Occasionally the orientation of the rectangular multi-forms varied.

Using the pen up and pen down line drawing techniques from Chapter 1 in conjunction with the color palette and size features of the turtle discussed below, Rothko-inspired creations can be made in Snap!
**Topic 2.1 Setting the Turtle's Size and Color**

The previous module introduced the ability of the sprite turtle to draw by raising and lowering a pen in its belly. The color of the line drawn by the turtle can be changed by using the **Set Pen Color** code block.

When the pen color is changed, the body of the turtle also changes to the same color.

The **Set Size** code block is used to increase the size of the turtle. This makes the color easier to see.

**Exploration 2.1 Exploring the Snap/Color Space**

Use the **Set Pen Color** code block to explore the range of different colors available in the Snap/Color space.

**Topic 2.2 Functions: Finding the Turtle’s Pen Color**

The values of the pen colors available in Snap/ range from 0 to 255. The current pen color of the turtle can be determined by using the **Pen Hue** reporter block. For example, the shade of green in the illustration below corresponds to *Pen Hue* number “31”.
In Snap! code blocks that perform an action such as the Move block have the shape of a rectangle.

![move 100 steps]

In contrast, code blocks that report a value are known as reporter blocks. These code blocks have an oval shape. The oval Pen Hue block reports the value of the turtle’s current pen color.

In other computing languages, code that reports a value is known as a function. The term “function” is also used in mathematics. Mathematical functions and functions in computer science are similar but not exactly the same. Using the term reporter block to describe a code block that reports a value eliminates any ambiguity or confusion about the role of this type of block.

The Pen Hue reporter block reports the value of the pen color to more than a dozen decimal places.

![Pen Hue]

This is more precision than will be required in subsequent modules that make use of color. Another type of reporter block, the Round reporter block, can round the pen color value to the nearest whole number.

The Pen Hue block reports that the blue pen color of the turtle is “67.291666666666”. This number is supplied as an input to the Round reporter block. The Round reporter block, in turn, reports that the nearest whole number is “67”.

![Round]

It is common to nest one reporter block within another to perform a series of operations in this fashion.

**Essential Knowledge**

**CS Principle 3.3 Mathematical Expressions**

Most programming languages use mathematical expressions like addition, subtraction, multiplication, and division. Mathematical operators enable the computer to perform mathematical calculations.

**Application of Essential Knowledge**

In Snap! mathematical operators are found in the green Operators palette. These code blocks perform mathematical operations such as addition, subtraction, multiplication and addition.

![2 + 3]

The Round reporter block rounds numbers to the nearest whole number.

![Round 5.32]
Rounding the value of the pen color to the nearest whole number makes it easier to remember the numbers associated with each color in Snap!

**Exploration 2.2 Finding the Turtle's Pen Color**

Change the turtle’s pen color by using **Set Pen Color** to change its color. Then use the **Pen Hue** block to identify the value of the pen color.

Explore the range of color values that are perceived as the same color. How much does a color value have to change before the color is perceived as different? For example, does changing the color value from 37 to 40 make a perceptible difference in the perceived color?

**Topic 2.3 Variables and Loops: Varying the Turtle’s Pen Color**

There are several color variables in the HSL color model: hue (color), saturation (shade), and brightness (light level). The values for each of these variables range from 0 to 255.

A Snap! variable can be created and combined with a loop to vary the turtle’s pen color. The **Make a Variable** option is found under the orange Variables section of the Snap! command palette.

In this illustration, a variable named *Color* has been created and made available to all sprites. (The default option of “for all sprites” will be selected in most of the variables created in subsequent modules.)
Once a variable has been created, the name of the variable will appear directly beneath the *Make a Variable* button.

When the checkbox beside the variable name is selected, the value of the variable appears in a *watcher* on the stage. The *watcher* is a display that shows the value of the variable.

Right-clicking the watcher displays several options, including an option to create a slider that can be used to adjust the value of the variable.

The default value of the slider is 0 to 100. However, options are available (accessed by right-clicking the watcher) to adjust the minimum and maximum values of the slider.

If the oval reporter block containing the name of the variable is dragged from the command palette into the script area, clicking on the reporter block will also display the current value of the variable.
If the **Color** variable block is dragged into the input of the **Set Pen Hue** code block, the **Set Pen Hue** block can be used to set the pen color to the current value of the variable **Color**.

Placing the **Set Pen Hue to Color** block inside a **Forever** loop allows the color of the sprite to be updated as the **Color** slider is moved back and forth. This provides a convenient way to quickly identify the number associated with a given color.

---

**Essential Knowledge**

**CS Principle 3.1 Variables and Assignments**

A variable is a place-holder that can be given any value. Using meaningful variable names makes the program easier to read and clearly indicates what values the variables represent.

**Application of Essential Knowledge**

In our example, a variable named **Color** is created. The value of this variable is controlled by a slider.

**Exploration 2.3 Variables and Loops – Varying the Turtle’s Color**

Use a slider to change the turtle’s color. Then identify the minimum and maximum values associated with the following colors: red, orange, yellow, green, blue, and purple.

<table>
<thead>
<tr>
<th>Color</th>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Red</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Orange</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Yellow</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Green</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Blue</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Purple</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Is the range of values associated with each color band similar or different?
Topic 2.4 Other Color Characteristics: Varying the Turtle’s Shade

The shade of a color is affected by the saturation parameter in the HSL color model.

A variable named Shade can be combined with the **Set Pen Saturation** code block to adjust the shade of a color. For example, if the **Pen Hue** is set to 164 and the **Pen Saturation** is set to 255, the color is perceived as dark blue. However, if the **Pen Hue** is set to 164 and the **Pen Saturation** is set to 27, the color is perceived as light blue.

**Essential Knowledge**

**CS Principle 2.4 Using Programs with Data**

By using variables, programs can process the information they are given to produce new data.

**Application of Essential Knowledge**

When the values of the **Color** and **Shade** variables change, the program can interpret the changing input and alter the color and shade of the pen sprite accordingly.

**Exploration 2.4 Other Color Characteristics – Varying the Turtle’s Shade**

Use a slider to change the turtle’s shade. Then identify the values associated with the following light and dark shades for the following colors: red, green, and blue.

What value is associated with a shade that is completely white (i.e., that has no perceptible color)?

Topic 2.5 Other Color Characteristics: Varying the Turtle’s Light Level

The light level (luminosity) of a color is affected by the brightness parameter in the HSL color model.

A variable named Light Level can be combined with the **Set Pen Brightness** code block to adjust the light level of a color. For example, if the **Pen Hue** is set to 164 and the **Pen Brightness** is set to 255, the color is perceived as blue. However, if the **Pen Hue** is set to 164 and the **Pen Brightness** is set to 50, the color is perceived as dark blue.
**Essential Knowledge**

**CS Principle 2.4 Using Programs with Data**

By using variables in mathematical equations, programs process the information they are given to produce new data.

**Application of Essential Knowledge**

Adding a Light Level variable creates an additional input for the program. The program can use that input to produce a new result.

**Exploration 2.5 Other Color Characteristics – Varying the Turtle’s Light Level**

Use a slider to change the turtle’s light level. Then identify the values associated with the following normal and dark shades for the following colors: red, green, and blue.

What value is associated with a light level that is completely dark (i.e., that has no perceptible color)?

**Topic 2.6 Art Blocks**

The sample illustration in the style of Rothco is composed of bars and blocks of varying colors.

A block can be created in Snap! by using the **Square** procedure developed in the previous section (Creating Artistic Patterns) as a starting point. The primary difference is that while the **Square** code block has a single input, **Size**, the custom **Block** code block has two inputs, **Width** and **Height**.
A user might easily forget whether the *width* or *height* input is entered first. To assist users, the *Title Text* option can be used to add descriptive titles after each input. If the plus sign ("+") after the input *Width* (shown as an orange oval in the illustration below) is clicked, an additional dialog box will appear. Enter the name *Width* and select *Title Text* to create a descriptive title after the input.

In the example below, the custom code block **Block** has been used to draw a rectangular block that is 100 steps wide and 20 steps high.

Rothco art is primarily composed of solid blocks and bars. The basic **Block** procedure can be used as a starting point for creating a solid bar filled with color. A **Solid Bar** code block can be constructed in two steps:

1. First develop a **Go to Block Center** custom code block. This code block places the turtle in the center of a block.

   The **Go to Block Center** code block changes the horizontal position (X) of the turtle by the width of the block divided by two. The vertical position (Y) is changed by the height divided by minus 2.
This change in the vertical and horizontal position of the turtle places the tip of the turtle in the center of the rectangular block.

Once the turtle is in the center of the block, the Fill code block can be used to fill the block with color, creating a solid bar. In order for the code below to function, the tip of the turtle must be surrounded by space, and the pen must be down.

2. Once the Go to Block Center code block has been developed, it can be combined with the Block and Fill code blocks to create a Solid Bar code block.

The Solid Bar code block can be used to create a solid bar filled with color.

Exploration 2.6 Art Blocks

It is good practice to leave the turtle at the same position at the end of the procedure as it was at the beginning of the procedure. Leaving an object in the same position as before is known as state transparency. The custom Solid Bar code block described in this section does not leave the turtle in its original position. Modify the Solid Bar procedure so that the turtle is left in its starting position.

Explore creation of several solid bars of varying sizes and colors. After creating several bars, identify other ways in which the Solid Bar code block might be refined based on your exploration.
2.7 Creating Art in the Style of Rothco

Once a solid bar has been created to your desired dimensions and color, the bar can be turned into a *Costume*. In order to do this, right click on the bar in the stage area.

Choose the option **Pen Trails**. This turns all pens trails into a new costume for the currently selected sprite. The bar can now be found under the **Costumes** tab for the sprite.

Converting the bar into a *Costume* allows it to be manipulated the bar with code. It is helpful for to give each bar costume a name that makes it easily identifiable based on its visual features. This can be done by right clicking the newly created bar costume and selecting the **Rename** option.

Rothko art typically consists of several bar variations. In order to emulate that feature, several bar *Costumes* can be created using different colors, sizes, and orientations. Rothko uses color intentionally in his artwork in order to present an aesthetically pleasing piece. The color of the bar that is being created can be selected using the **Set Pen Color** code block.
Clearing the stage before creating a costume ensures that no other unwanted graphics become part of the costume.

To increase or decrease the size of the turtle after creating a costume, use the **Change Size** block (found under the *Looks* palette).

To change the orientation of the turtle and its associated costume, use the **Point in Direction** block.

The **Stamp** code block can be then used to stamp an image of the block onto the stage. You can move the turtle around the stage in order to position stamped blocks in desired locations. Positioning the bar costumes on the stage in an aesthetically pleasing artistic pattern art in the style of Rothco to be created.

In order to switch between costumes to select bars of different sizes and colors, use the **Switch to Costume** code block (found under the *Looks* palette).

A dropdown menu will list each bar costume (hence the importance of assigning a meaningful name to each costume).

To save an image of the art created, right click the stage area and select “pic…” A copy of the artwork will then be saved locally on the computer’s desktop.
Exploration 2.7 Creating Art in the Style of Rothco

Create an artistic pattern in the style of Rothco by manually dragging the turtle to different locations and stamping bars of different sizes and colors on the stage.

Then create the same design by using a script to move the turtle to different locations on the stage. How can regularities in the pattern of bars be used to create more efficient scripts by incorporating a Repeat code block to take advantage of the regularities?
3. Creating Colorful Patterns

Glen Bull, Jo Watts, and Alexis Kellam

The Snap! color framework explored in the previous chapter can be used to create colorful patterns filled with polka dots and stripes. This exploration, in turn, will lead to simulation of impressionist paintings in the next chapter.

Snap! sprites move about an area known as the Stage. The ability to move the sprites across the stage will play an important role in creation of artistic patterns.

Creating Art through Coding

Jackson Pollock was an American painter in the abstract expressionist movement. Pollock was known for his paint pouring technique that left his canvas covered with paint in a uniquely nontraditional way.

Creating these whimsical and colorful projects in Snap! can be reminiscent of the vibrant paint work done by Pollock. Varying the size of the dots or the thickness of the lines in combination with the randomized positioning of the pen and changes in the pen hue can allow you, the artist, to create art in the style of Pollock.

The illustrations above were created through use of these techniques. The methods used are described in the sections that follow.
Topic 3.1 The Stage

The **Stage** is the space in which the turtle can move about. The default stage is 360 turtle steps high and 480 turtle steps wide.

The **Go to X _ Y _** block can be used to send the turtle to any part of the stage. The X coordinate adjusts the horizontal position of the turtle and the Y coordinate adjusts the vertical position of the turtle.

![Go to X Y block](image)

**Exploration 3.1 The Stage**

Explore the dimensions of the stage by using the **Go To** code block to move the turtle about the stage. What are the X and Y values associated with the center of the stage and each of its four corners (top left, top right, bottom left, and bottom right)?

Topic 3.2 Moving to a Random Position

The **Pick Random** reporter block can be used to pick a random number.

![Pick Random block](image)

By picking a number between the left side of the stage (-240) and the right side of the stage (+240), the **Go To** block can be combined with the **Pick Random** block to send the turtle to a random position on the horizontal axis. The process can be duplicated for the vertical (Y) axis, to send the turtle to a random position on the horizontal and vertical axes.

![Go to X Y random block](image)

This capability will be used to place polka dots at random locations across the stage.

Effective coding encapsulates code that will be reused in a custom code block once the code has been tested and verified to work properly. This reduces the burden on human memory, and makes it possible to create more complex programs than otherwise would be possible.
It is important to assign a meaningful name to a custom code block when it is created. A meaningful name enables the programmer to use the custom code block without the necessity of remembering all of the details of how the code inside the custom block works. In this instance, because the custom code block will send the turtle to a random position on the stage, the custom code block will be named **Go to Random Position**.

**Essential Knowledge**

**CS Principle 3.15 Random Values**

The code block **Pick Random** generates and returns a random number between two values (i.e., within a range that falls between a low number and a high number). Each result within this range is equally likely to occur. Generation of a random number in a program means each execution may produce a different result.

**Application of Essential Knowledge**

In this section, **Pick Random** is used to send a turtle to a given random position on the stage.

**Exploration 3.2 Moving to a Random Position**

Create a **Go to Random Position** custom code block, and use it to move the turtle about the stage. Verify that the turtle randomly moves to every part of the stage given a sufficient number of jumps.

**Topic 3.3 Drawing Dots in Random Locations**

The **Move** block moves the turtle about the stage. For example, **Move 10** moves the turtle forward ten turtle steps. However, when the **Move 0** block is executed, the turtle stays in the same place and does not move forward. If the **Pen** is down, the turtle will draw a dot.

![Move 0 block](image)

The **Set Pen Size** block controls the size of the dot.

![Set Pen Size block](image)

**Move 10** moves the turtle forward ten turtle steps. However, when the **Move 0** block (i.e., “move 0 steps”) is executed, the turtle stays in the same place and does not move forward. If the **Pen** is down, the turtle draws a dot in that spot (as though a blob of ink squirted out).
The **Set Pen Size** block controls the size of the dot.

![Set Pen Size Block](image)

The script that makes a dot can be refined by lifting the pen after the dot is drawn. This will ensure that the turtle only draws dots as it moves about the screen and does not draw lines that connect the dots when the turtle is moved.

![Pen Down, Move, Pen Up Blocks](image)

A custom **Dot** code block can be created once this code is tested and verified. The **Dot** block, in turn, can be combined with the **Go to Random Position** block to place random dots across the screen.

![Repeat Block](image)

An **Repeat** block can be used to control the number of dots that are placed on the stage.

**Essential Knowledge**

**CS Principle 3.8 Iteration**

A computer often needs to repeat the same series of steps several times. The term iteration refers to a repeated sequence of steps. In the illustration above a series of steps is repeated to create an artistic pattern.

**CS Principle 3.11 Calling Procedures**

A procedure is a named group of programming instructions that may have parameters and return values. Procedures are referred to by different names, such as *method* or *function*, depending on the programming language.

A procedure call interrupts the sequential execution of statements, causing the program to execute the statements within the procedure before continuing. Once the last statement in the procedure has executed, flow of control is returned to the point immediately following where the procedure was called.

**CS Principle 3.12 Developing Procedures**

A procedural abstraction may extract shared features to generalize functionality instead of duplicating code. This allows for program code reuse, which helps manage complexity.
Application of Essential Knowledge

In this section, the concept of iteration is applied to draw multiple dots.

Exploration 3.3 Drawing Dots in Random Locations

Create a Dot custom code block. Combine the Dot custom block with the Go to Random Position block to create random dots of varying sizes across the stage.

Estimate how many times the turtle has to make a dot in order to completely color in the entire stage. Then verify this estimate through repeated moves.

Topic 3.4. Dots of Different Colors

Random colors can be generated by assigning a random number as an input to the Set Pen Hue block.

This code can be used as the basis for a custom Pick Random Color code block.

If the Pick Random Color block is combined with the Random Position and Dot code blocks, this sequence of code blocks can be used to place random dots of different colors across the screen.

Essential Knowledge

CS Principle 3.12 Developing Procedures

Creation of custom code blocks (i.e., procedures) improves the readability of a program. Rather than displaying all of the lines of code contained within a custom code block, the reader on sees the central concept or abstraction of the procedure’s function. This also facilitates re-use of code, which helps manage complexity.

Application of Essential Knowledge

In this application, the custom code block Pick Random Color is used to generate random colors.

Exploration 3.4 Dots of Different Colors

Create a Random Color custom code block. Combine the Random Color custom block with the Go to Random Position and Dot blocks to create random dots of different colors across the stage.
**Topic 3.5 Polka Dot Backgrounds**

A polka dot background can be created by using the **Repeat** block to generate the desired number of dots across the stage.

![Repeat block example](image)

The **Hide** code block can be used to hide the turtle. Hiding the turtle speeds up the drawing process.

![Hide block example](image)

The **Warp** code block causes a polka dot background to appear almost instantaneously. The **Warp** block suspends other background operations until execution of the script is completed. This causes operations to take place more quickly.

![Warp block example](image)

The decision about whether to include the Warp block is an aesthetic one in this instance. In some cases, it may be enjoyable to watch the dots appear one by one. In other cases, it may be satisfying to see the polka dot background appear instantly.

**Exploration 3.5 Polka Dot Backgrounds**

Create a **Polka Dot** custom code block that generates a polka dot background on the stage.
**Topic 3.6 Confetti Backgrounds**

A confetti background can be created by replacing the dots with randomly generated lines. A Line code block can be created by moving forward a specified number of steps (with the pen down) and then moving back the same number of steps.

Replacing the **Dot** code block with a **Line** code block produces a striped background.

Turning the turtle a random amount each time produces a confetti background.

The number of variants that can be created is endless. The choices for different variations is dictated by the esthetic sensibility of the coder.

**Essential Knowledge**

The following CS Principles are also applicable to this section:

- **CS Principle 3.15 Random Values**
- **CS Principle 3.11 Calling Procedures**
- **CS Principle 3.12 Developing Procedures**
**Application of Essential Knowledge**

The application is similar to the previous section, except that instead of **Dot** function, the **Line** function is applied.

**Exploration 3.7 Confetti Backgrounds**

Create a **Confetti** custom code block that generates a confetti background on the stage. Then explore other variants and patterns.

**Topic 3.8 Creating a Customized Color Palette**

There are times when it may be useful to create a customized color chart that includes a specific array of colors. For example, the chart below shows a sequence of colors and associated pen numbers associated with each color.

<table>
<thead>
<tr>
<th>Color</th>
<th>Pen #</th>
</tr>
</thead>
<tbody>
<tr>
<td>Red</td>
<td>4</td>
</tr>
<tr>
<td>Orange</td>
<td>9</td>
</tr>
<tr>
<td>Gold</td>
<td>11</td>
</tr>
<tr>
<td>Yellow</td>
<td>17</td>
</tr>
<tr>
<td>Lime</td>
<td>20</td>
</tr>
<tr>
<td>Green</td>
<td>37</td>
</tr>
<tr>
<td>Cyan</td>
<td>48</td>
</tr>
<tr>
<td>Aqua</td>
<td>54</td>
</tr>
<tr>
<td>Blue</td>
<td>61</td>
</tr>
<tr>
<td>Purple</td>
<td>73</td>
</tr>
<tr>
<td>Violet</td>
<td>78</td>
</tr>
<tr>
<td>Magenta</td>
<td>87</td>
</tr>
<tr>
<td>Gray</td>
<td>66</td>
</tr>
</tbody>
</table>

The pen numbers associated with the colors selected can be placed in a list.

```
list 4 9 11 17 20 37 48 54 61 73 78 87 66
dataColorChart
```

The list of pen numbers can then be assigned to a variable.

```
set ColorChart to list 4 9 11 17 20 37 48 54 61 73 78 87 66
```
The **Item** code block can then be used to select a random pen number from the color chart.

The method can then be used to set the **Pen Hue** to a pen number randomly selected from a specific palette of colors.

**Exploration 3.8 Color Chart**

Create a customized color chart and combine it with the **Polka Dot** and **Confetti** programs to create art work in the style of Jason Pollock.
Chapter 4. Simulating an Impressionist Painting

Glen Bull, Jo Watts, and Alexis Kellam

The image on the left is a photograph of a window box with flowers. The image on the right is an impressionist painting of the flowerbox created through a technique known as pointillism. This method creates a painting by placing tiny dots on the canvas to create an impression of the original painting.

In the previous module, dots were placed across the stage to create a polka dot background. The same method can be used to create an impressionist painting.

**Topic 4.1 Using a Sprite as a Canvas**

A photograph is imported into Snap!, where it becomes the costume of a sprite that serves as the canvas for the painting. This is the first module in which more than one sprite is used. One sprite will be used as the canvas for the painting. A second sprite will be used as a paint brush to create the painting on the canvas.

Begin by adding a new turtle sprite that will become the canvas.

Once the second sprite is created, change its name to *Canvas.*
Once this is accomplished, two sprites will exist – the original sprite named “Sprite” and a second sprite named “Canvas.”

Select the Costumes tab of the Canvas sprite and import a photograph by dragging it into the space below the Costumes tab.

The photograph will become the new costume worn by the Canvas sprite. Drag the Canvas sprite to the center of the stage.
Once the Canvas sprite is centered on the stage, uncheck the “draggable” option so that the Canvas sprite cannot be moved from this position by accident.

**Exploration 4.1 Using a Sprite as a Canvas**

Select an existing photograph (or take a new photograph) and import it to use as the costume for the Canvas sprite. This photograph will be used as the basis for an impressionist painting.

**Topic 4.2 Using a Sprite as a Paint Brush**

The original sprite (by default named “Sprite”) will be used as a paint brush. Rename the original sprite, using “Paint Brush” as its new name. When this step is completed, two sprites will be visible in the sprite corral beneath the stage: one named “Paint Brush” and a second named “Canvas.”

The Canvas sprite may cover up the Paint Brush sprite on the stage. If that is the case, the Paint Brush sprite will not be visible. If the Paint Brush sprite is not visible, use the Go to Front Layer code block to bring the Paint Brush sprite to the front layer so that it is visible. (Note: The Paint Brush sprite must be selected in the Sprite Corral when this command is executed.)
When the *Paint Brush* sprite is in the front layer, the **Hue at Myself** reporter block can be used to identify the color of the canvas beneath the *Paint Brush* sprite.

The **Hue at Myself** reporter block can be combined with the **Set Pen Hue** block to change the color of the *Paint Brush* sprite to the color beneath the *Paint Brush* sprite.

This method can be extended to include *saturation* and *brightness* as well as the hue.

This will ensure that the *Paint Brush* sprite reflects the exact shade and color of the canvas beneath it. Once this code has been verified and tested, it can be encapsulated in the form of a custom code block, **Reflect Color**.

The **Reflect Color** custom code block causes the *Paint Brush* sprite to reflect the color of the canvas beneath it.
Essential Knowledge

CS Principle 3.13 Developing Procedures

Assigning a name to a group of instructions to create a procedure is one form of abstraction. Once a code sequence is tested and encapsulated in the form of a procedure, the programmer no longer needs to remember the specific details of the procedure, enabling the programmer to focus on other program details.

Application of Essential Knowledge

In the example above, a custom code block named Reflect Color replaced a longer sequence of instructions. Replacing an entire sequence of instructions with a named procedure yields several benefits. First, it makes the top level of the program more concise and easier to read. Second, if meaningful names are used, it makes the program easier to understand.

Exploration 4.2 Using a Sprite as a Paint Brush

Create a Reflect Color custom block that causes the Paint Brush sprite to reflect the color of the underlying Canvas sprite. Drag the Paint Brush sprite around the Canvas and observe the changes in its color when the Reflect Color code block is executed.

Topic 4.3 Painting the Canvas

After the Paint Brush and Canvas sprites have been created, the Reflect Color block can be combined with the Random Position custom block to move the paint brush to a random position on the stage, reflecting the color beneath the paint brush at the new position. The Paint Brush sprite is then used to place a dot at that position.

The Go to Front Layer block is involved prior to initiation of the Repeat block to ensure that the Paint Brush sprite is in the top layer (in front of the Canvas sprite) when the process is begun.

Once the code is created and tested, it can be used to create a custom Paint Canvas block. The Paint Canvas block can be repeated as many times as desired to create the desired impression of the underlying photograph.

Application of Essential Knowledge

In this section, the main Paint Canvas procedure calls the Go to Random Position, Reflect Color and Dot code blocks to paint the canvas.
**Exploration 4.3 Painting the Canvas**

Create a **Paint Canvas** custom block that causes the *Paint Brush* sprite to create a pointillist painting. Explore the effect of invoking the **Paint Canvas** block with different dot sizes.

**Topic 4.4 Further Explorations in Simulation of Impressionist Paintings**

The **Paint Canvas** block created with the **Dot** block is just a starting point for exploration of impressionist effects. Just as the **Dot** block was replaced with the **Line** block in the previous module to create a confetti background, the **Dot** block can be replaced with the **Line** block in **Paint Canvas** to create a different effect.

Other possibilities include substitution of a starburst in place of a line or a dot. These substitutions can be used to create effects such as the impression of looking at the photograph through frosted glass. Explore the range of effects that can be achieved through different combinations and extensions.
Art and images created in other applications can be imported into Snap! The capabilities of Snap! can also be employed to create a basic paint program in Snap! itself.

This offers a good way to explore the graphics capabilities of Snap! A paint program of this kind can also be customized to incorporate special features such as Instagram-style filters.

**Topic 5.1 Creating a Drawing App**

The first step in creating a paint program is to create a basic drawing app. Note: The sprite’s *Draggable* option must be disabled in order for the method demonstrated to work. Uncheck the sprite’s *draggable* option before continuing.

Then place the **Go to Mouse-Pointer** block in a **Forever** loop. If the sprite’s draggable option is disabled, this code will cause the turtle to follow the mouse pointer as the cursor moves around the stage.
If the pen is lowered before this script is executed, the turtle will draw a line as it follows the cursor around the stage.

```
pen down
forever
  go to mouse-pointer
```

This drawing capability will become the heart of the paint program.

**Exploration 5.1 Creating a Drawing App**

Create a custom **Draw code** block that enables the mouse to be used as a drawing tool.

**Topic 2. Checking the State of the Mouse Button - Introducing the If Statement**

An **If** statement can be used to check whether the mouse button is down. This capability is used to lower the turtle’s pen when the mouse button is down and raise the pen when the mouse button is up.

```
if mouse down?
  pen down
else
  pen up
```

This test can be inserted into the **Forever** loop to ensure that the turtle only draws a line when the mouse button is down.

```
forever
  if mouse down?
    pen down
  else
    pen up
  go to mouse-pointer
```
Essential Knowledge

**CS Principle 3.5 Boolean Expressions**

A *Boolean value* is either true or false.

**CS Principle 3.6 Conditionals**

Selection determines which parts of an algorithm are executed based on a condition being true or false.

Conditional statements affect the sequential flow of control by executing different statements based on the value of expressions in which the code in the *first block of statements* is executed if the expression evaluates to true; otherwise, the code in the *second block of statements* is executed.

Application of Essential Knowledge

In the example above, the *Mouse Down?* block returns a result of *True* when the mouse is down; otherwise, it returns a result of *False*. A conditional statement is also used to control the state of the turtle’s pen (*Down* or *Up*).

**Exploration 5.2 Checking the State of the Mouse Button - Introducing the If Statement**

Refine the *Draw* block so that the mouse only draws when the mouse button is down.

**Topic 5.3 Launching an Application**

The green flag and red stop sign in the upper right-hand corner of the Snap! window provide convenient ways of stopping and starting a program.

When an application is shared with another user, they do not necessarily know the name of the code block (or code blocks) that must be clicked to launch the program. This issue can be avoided by combining the code blocks that launch the program with the *When Green Flag Clicked* control block. Any code blocks attached to green flag control blocks will be launched when the green flag is clicked. Clicking the red stop sign (to the right of the green flag) stops all code.
One other advantage of launching code blocks with the green flag is that several scripts (i.e., separate sets of code blocks) can be simultaneously launched.

**Essential Knowledge**

The following CS principles are employed in the preceding section:

- CS Principle 3.12 Calling Procedures
- CS Principle 3.13 Developing Procedures

**Application of Essential Knowledge**

The custom *Draw* code block is developed and called in the preceding section.

**Topic 5.4 Introducing Lists and Loops: Creating Color Swatches**

A means of selecting color is needed to create a full-fledged paint program. There are several ways in which this might be done. One approach involves creation of a series of color swatches that can be used to adjust the color of the line being drawn.

A series of swatches representing the basic colors can be drawn by creating a list of the *Pen Hues* corresponding to those colors. The list of *Pen Hues* representing these colors is shown below.

```
list  4  9 17 37 48 60 73 ▼▼
```

Some basic setup steps are needed before the color swatches are drawn. The setup steps include clearing the screen, lowering the pen, adjusting the pen size, and setting the *Pen Saturation* and *Pen Brightness* to their maximum values (255).

The Item block can be used to access the list of *Pen Hues*. For example, *Item 1* in the list of *Pen Hues* is “4”:

```
item  1 ▼ of list  4  9 17 37 48 60 73 ▼▼
```

Similarly, *Item 3* in the list of *Pen Hues* is “17”.
When the list of Pen Hues is placed in a loop, the first time that the loop is executed, the value of the variable $i$ will be “1”. Since Item 1 in the list is Pen Hue “4”, the Pen Hue will be set to 4. Then a line 30 steps long is drawn.

The second time that the loop is executed, the value of the variable $i$ will be “2”. Since Item 2 in the list is Pen Hue “9”, the Pen Hue will be set to 9. The loop can be used to access each item in the list of Pen Hues in this manner.

By adjusting the Pen Hue during each iteration of the loop, the color is changed each time the turtle draws a line. This results in a series of lines that look like this:

The basic method of combining a loop with a list of values will be used many times in subsequent units. For example, the same method can be used to play a list of musical notes. This method provides a way of organizing a set of values that represent a concept (such as a palette of colors or a musical scale).

**Essential Knowledge**

**CS Principle 3.2** Data Abstraction

Data abstraction can be created using lists. Developing a data abstraction to implement in a program can result in a program that is easier to develop and maintain.

- An element is an individual value in a list that is assigned a unique index.
- An index is a common method for referencing the elements in a list or string using natural numbers.
- The use of lists allows multiple related items to be treated as a single value. Lists are referred to by different names, such as array, depending on the programming language.

**CS Principle 3.8** Iteration

In computer science, a for-loop is a control flow statement for specifying iteration, which allows code to be executed repeatedly. (This sentence is from Wiki, not in AP CSP guidebook)

**CS Principle 3.10** Lists
In computer science, a list is an abstract data type that represents a countable number of ordered values, where the same value may occur more than once.

Iteration statements can be used to iterate a list.

**Application of Essential Knowledge**

In the preceding section, iteration is combined with a list to draw different colors.

**Exploration 5.4 Introducing Lists and Loops: Creating Color Swatches**

Use the procedures developed in the unit *Exploring Color* to identify a representative range of color values. Create a color palette by placing the Pen Hues associated with these colors in a list. Then develop a custom **Color Swatch** block that creates a color swatch for this palette.

**Topic 5.5 Designing a Color Selection Method**

The method used to create the **Reflect Color** code block in the previous unit can be extended to create a **Select Color** code block for the paint program.

When the **Select Color** block is combined with the **When I am Clicked** code block, the turtle becomes the same color as the color beneath the turtle at the moment that the mouse is clicked. For example, if the cursor is placed on the green color swatch and the mouse button is clicked, the Pen Color is set to “Green”.

**Note:** This method only works when the **Draw** program is running, because the **Draw** code block causes the sprite to follow the mouse cursor. (Consequently, when the mouse button is clicked, the sprite is at the same location as the mouse cursor, and therefore reflects the color under the cursor.)

**Essential Knowledge**

**CS Principle 3.6 Conditionals**

Selection determines which parts of an algorithm are executed based on a condition being *True* or *False*.

Conditional statements affect the sequential flow of control by executing different statements based on the value of an expression.
Explain how the Fill function works and its application.

**Essential Knowledge**

- **CS Principle 3.6** Conditionals

**Application of Essential Knowledge**

In the preceding example, when the "F" on the computer keyboard is pressed, the block is triggered and executes the Fill function.

**Exploration 5.6 Filling the Interior of a Shape with Color**

Use the paint program to draw a shape. Then use the Fill code block to fill the interior of the shape with color.
Topic 5.7 Creating Permanent Paint Swatches

The previously drawn paint swatches provide a convenient way of selecting colors in the paint program. One limitation of the paint swatches is that use of the Clear code block to clear the canvas also erases the paint swatches. This limitation can be addressed by converting the paint swatches into a sprite’s costume.

By adding a second sprite, one sprite can serve as paint brush and the second sprite can provide the color swatches for color selection. Create a second sprite by clicking the new turtle icon below the stage.

Then name the second sprite “Paint Swatches” to distinguish it from the first sprite. Since each sprite has its own script area, the first sprite (i.e., the Paint Brush sprite) will have all the paint scripts. The second sprite will not have any scripts – its costume will simply provide a convenient means of selecting colors.

Then place the cursor on the previously drawn paint swatches on the stage and right-click. A pen trails option will appear. The Pen Trails options converts drawings on the stage into a costume for the currently selected sprite (in this case, the previously named “Paint Brush” sprite).
This action will create a new costume for the sprite consisting of the paint swatches.

![Paint Swatches](image1)

The **Clear** code block can then be used to erase the original drawing. Once this is done, the **Color Swatches** sprite can be positioned at any convenient location on the stage, where it will be used by the **Paint Brush** sprite for color selection.

![Clear](image2)

Each sprite has its own layer. The **Paint Brush** sprite needs to be in front of the **Color Swatches** sprite in order to select colors. This can be ensured by using the **Go to Front Layer** code block prior to the **Draw** code block (in the **Paint Brush** script area), as shown in the illustration below.

![Go to Front Layer](image3)

**Exploration 5.7 Creating Permanent Color Swatches**

Create several different costumes for the **Color Swatches** sprite with different color palettes. Use the different color palettes for different drawings.

**Topic 5.8 Extended Exploration**

- Develop a method to adjust the **Pen Size** using keys on the computer keyboard.
- Develop a method to clear the stage using a key on the computer keyboard.
- Create your own art using the paint application that you developed.

*Extended Exploration Video: [https://youtu.be/Ut2IUydS8NA](https://youtu.be/Ut2IUydS8NA)*
6. Mobile Art

Glen Bull, Jo Watts, and Alexis Kellam

Alexander Calder is a sculptor known for his innovative mobiles – kinetic sculptures so well balanced that the slightest air currents cause them to move and sway. One of the best-known Calder mobiles is in the atrium of the East Wing of the National Gallery of Art in Washington, D.C. This type of mobile art, such as the illustration shown below, can be emulated with a computer.

Illustration by Alexis Kellam

Digital prototypes developed on the computer can also be converted into art in the physical world. The animated art in this module builds upon and combines techniques from previous units.

Topic 6.1 Importing a Drawing

The paint application created in the previous unit provides one method of creating art and drawings using customized tools. Art drawn by hand on paper can also be photographed with a smart phone or camera and imported as the costume of a sprite. To import art or drawings, create a new sprite and name it Canvas.

Then select the Costumes tab of the Canvas sprite and drag the image onto the Costumes tab.
It is not possible to draw directly onto a sprite costume. However, the **Stamp** code block can be used to stamp a copy of the costume onto the stage. Since the stamped copy of the costume is a drawing, it can be filled with paint colors.

To access the stamped drawing, drag the **Canvas** sprite away from the stamped copy beneath the sprite. (If you lose track of which image is the drawing and which is the sprite, it is helpful to remember that a sprite can be moved while a drawing is a static image whose position cannot be moved.)

The drawing on the stage can now be filled with color using the paint program developed in the previous unit. (Note: the **Hide** block can be used to hide the sprite used to stamp the image.)

**Exploration 6.1 Importing Hand-drawn Costumes**
Create a hand-drawn image and import it into Snap! Then enhance the drawing using the paint application developed in the previous unit.
Topic 6.2 Converting a Drawing to a Costume

Any drawing on the Snap! stage can be converted into a sprite’s costume. Once the drawing has been filled with color, create a new sprite. The new sprite in the illustration below has been named “Fish”.

Then right-click the drawing and select the *Pen Trails* option to convert the drawing to a costume. (Note: the mouse pointer must be on a painted area in the drawing to access this option; if the mouse pointer is on an empty space that has not been filled, this option will not be available.)

The drawing on the stage, with color and enhancements added by the paint program, is now the costume of the *Fish* sprite.

Once the drawing has been converted to a costume, the **Clear** block can be used to clear the screen if the old stamped drawing is no longer needed.

**Exploration 6.2 Converting a Drawing to a Costume**

Use the *Pen Trails* option to convert a drawing into a sprite costume.
Topic 6.3 Animating a Drawing

Motion blocks move a sprite, thereby animating the drawing. For example, placing the **Move 10 Steps** block in a **Forever** loop will cause the *Fish* sprite to continue moving until it moves off the right edge of the screen.

The **Go to X: ___ Y: ___** block can be used to return the sprite to the center of the screen.

**Essential Knowledge**

**CS Principle 3.3 Iteration**

The **Forever** block is an *infinite loop*. The original address of the campus of the Apple computer company was at “1 Infinite Loop” in Cupertino, California – an example of computer humor.

**Application of Essential Knowledge**

In this example, the **Forever** block executes the **Move** instruction repeatedly until the **Stop** icon is clicked.

**Exploration 6.3 Animating a Drawing**

Combine a **Move** block with a **Forever** block to animate a sprite. What effect does changing the number of steps used as an input to the **Move** block have upon the motion of the sprite?

**Topic 6.4 Wrapping the Animation around the Stage**

To create an animated diorama with continual motion, animated sprites are wrapped around the stage so that when they exit the right side of the stage, they reappear on the left side of the stage. The X-coordinate of the right side of the stage is 240. By checking to see when the sprite’s horizontal (X) position is greater than 240, the sprite can be reset to -240 (i.e., the left side of the stage) when this occurs.
This test can be combined with the **Forever** and **Move** blocks to create a continuously moving diorama that causes the fish to reappear on the left side of the stage after it exits the right side.

**Essential Knowledge**

**CS Principle 3.3** Mathematical Expressions

An expression can consist of a value, a variable, an operator, or a procedure call that returns a value. Expressions are evaluated to produce a single value.

**CS Principle 3.5** Boolean Expressions

A *Boolean value* is either true or false. In the example above, the **X Position** of the turtle is examined to see if it is greater than 240; the expression re and returns a value of *True* or *False*.

**CS Principle 3.6** Conditionals

Selection determines which parts of an algorithm are executed based on a condition being *True* or *False*. Conditional statements affect the sequential flow of control by executing different statements based on the value of an expression.

In an **If ... Else** code block the *first block of statements* is executed if the expression condition evaluates to true; otherwise the code in the *second block of statements* is executed.

**Exploration 6.3** Wrapping the Animation around the Stage

Refine the code for the previously animated drawing so that the animation wraps around the stage.
Topic 6.5 Animating Separate Elements of an Object

There are times when it may be desirable to group several sprites together. For example, in the illustration below, there are three separate sprites: the body of the helicopter, the main rotor, and the tail rotor. These elements can be grouped together so that all three components move together as one object, while still providing the flexibility for the rotors (which have their own individual script areas) to turn independently of the main body.

To accomplish this, first position the two objects in the desired relationship. In the example below, the main rotor has been positioned in the desired position at the top of the helicopter body.

Then drag the icon of the subcomponent – the rotor in this instance – from the sprite corral at the bottom of the stage onto the helicopter body. Once the icon is over the main component, it will be highlighted by a tan line. Release the mouse button when the tan line appears to group the objects.

Once this is done, the icons in the sprite corral will indicate that the two parts have now been grouped together.
The following script can now be placed in the script area of the main rotor. This script will turn the rotor. The **Wait** is one-tenth of a second (0.1 seconds).

![Script Example]

Note: if the blades of the rotor appear to be off-center as they turn, the **Paint Editor** can be used to set the rotation center of the image so that the blades rotate around the center of the object.

![Paint Editor]

Repeat the grouping process for the tail rotor. Then copy the main rotor script into the script for the tail rotor. After this step is completed, clicking the green flag will cause both the main rotor and the tail rotor to rotate.
Once the rotors are turning properly, the following script placed in the script area for the body of the helicopter will cause the helicopter assembly to move across the screen while the rotors turn. When the helicopter reaches the right side of the stage, it will wrap around and reappear on the left side of the stage.

This method can be used to independently animate subcomponents within a larger grouped assembly.

**Essential Knowledge**
- **CS Principle 3.3** Mathematical Expressions
- **CS Principle 3.5** Boolean Expressions
- **CS Principle 3.6** Conditionals

**Exploration 6.5 Animating Separate Elements of an Object**
Create an animated diorama in which several sprites move independently.

**Topic 6.6 Combining Elements to Simulate a Mobile Object**
The method used to combine the elements of the helicopter in the preceding section can be extended to combine the elements of a simulated mobile.
This illustrative mobile can be separated into five separate components: a red ball, a yellow triangle, a green crescent, and two cross arms (one large and one small). Each of these elements will become a separate sprite that will have its own individual script. The individual components will then be grouped into larger subassemblies in the same manner as the helicopter.

If a Turn code block is placed in the script area of the Small Crossarm sprite, it can be used to tilt the cross arm back and forth.

The following loop causes the crossarm to oscillate back and forth one time. The sine function describes the behavior of naturally oscillating objects. It will be explored in greater depth in future modules involving sound and music (for example, to describe the motion of a vibrating piano string).

Addition of a Forever loop will cause the crossarm to oscillate back and forth indefinitely.
Introduction of a random wait between oscillations simulates the effect that might result from random air currents shifting as they flow past the crossarm.

Finally, addition of a green launch block will cause the crossarm script to execute when the green flag is clicked, along with scripts with green launch blocks in the script areas of the other sprites.

A similar rocking motion is applied to the yellow triangle. Since the yellow triangle is hanging beneath the small crossarm, its oscillation is constrained in a different way.
In this manner two scripts – one variant that controls the motion of the crossarms and a second variant that controls the motion of the red ball, yellow triangle, and the green crescent – are used to create oscillations in each of the individual elements.

Once scripts have been developed for the script areas of each of the five sprites that cause each one to move individually, the yellow triangle and the green crescent can be grouped with the small crossarm. The grouped small crossarm assembly and the red ball can then be grouped with the large crossarm.

Once the groupings have been completed, each of the elements will oscillate in a manner that is controlled by the individual script for that sprite, while moving as a whole within the larger assembly.

**Exploration 6.6 Combining Elements to Simulate a Mobile Object**

Create a simulation of a mobile with separate scripts that animate each element of the mobile. Then construction a physical model of the mobile. Do the elements of the mobile move in a way that resembles the simulation? In what ways does the movement of the physical model and the digital simulation differ?
B. Music
TuneScope is an extension of Snap! that facilitates creation of music through coding. TuneScope can be accessed at: www.tunescope.org

Note: On a Macintosh, TuneScope must be accessed through either the Chrome or Firefox browsers. (TuneScope will not work properly in the Safari browser.)

Electronic Music

Robert Moog developed the first commercially available music synthesizer. The Moog synthesizer was used to produce the album Switched-On Bach. This work became the second classical music album to sell more than a million copies.

The first music synthesizers were constructed using analog electronics rather than digital computers. However, it soon became possible to recreate these same effects using personal computers. Music synthesizers – regardless of whether they consist of dedicated electronic equipment or digital computers – recreate music using audio oscillators and filters.

This chapter provides an introduction to generation of sound and music using TuneScope. TuneScope code blocks provide the capability to generate tones through code. The addition of an audio frequency oscilloscope at the top of the Snap! screen is one of the most noticeable enhancements. The TuneScope display can be used to inspect and analyze acoustic waveforms.
TuneScope provides the capability for synthesizing tones and musical instruments. Music can be generated by a computer in two different ways:

1. through synthesized sound, and
2. through sampled sound.

These topics are discussed in the sections that follow.

**Topic 7.1 Synthesized Sounds – Generating a Tone**

Computers can generate sounds digitally. Pure tones are particularly useful because any regular (periodic) sound such as a musical note can be synthesized by combining a series of pure tones. To generate a tone and display the tone on the TuneScope oscillographic display, first select Snap! as an option from the drop-down *Mode* menu in the upper right-hand corner of TuneScope.

Then select *Oscillators* as an option from the *Audio Input Source* menu.

Once these display settings have been selected, use the **Tone Frequency _ Amplitude** block (found under the *Sound* command palette) to generate a tone and display its waveform on the TuneScope display.

The acoustic waveform will appear in the display once these blocks are executed.
Change the **Tone Number** input from green to red (by clicking the input) to turn off the tone.

![Tone Number](image)

**Essential Knowledge**

**CS Principle 3.14 Libraries**

A software library contains procedures often grouped together within categories that may be used in creating new programs.

Existing code segments can come from internal or external sources, such as libraries or previously written code.

The use of libraries simplifies the task of creating complex programs.

**Application of Essential Knowledge**

TuneScope includes a library of code blocks developed to support creation of music. The **Tone Frequency** _ Amplitude _ block is an example of a code block in this library.

**Exploration 7.1 Synthesized Sounds – Generating a Tone**

Explore different frequency and amplitude settings. What effect does changing the frequency have on the appearance of the waveform in the display tone? What effect does this have on the perceived auditory quality? What effect does changing the amplitude (in the range from 0 to 1.0) have on the tone? (Be sure that the sound on your computer speakers is enabled.)

**Topic 7.2 Synthesized Sounds – Frequency**

The term *frequency* refers to how frequently an event occurs. A tone displayed on the TuneScope display reveals that there is a pattern that is repeated over and over again. Each repetition of the pattern is referred to as one *cycle*.

A tone for which the pattern repeats 100 times in one second is described as having a frequency of 100 *cycles per second*. In honor of the pioneering scientist, Heinrich Hertz, the term *Hertz* (abbreviated Hz) is also used as a term to describe the number of times per second that an event occurs. In this instance, one cycle of the waveform of the tone is repeated 100 times in one second. Frequency is correlated with the perceived *pitch* of a sound. As the frequency increases, the perceived pitch increases. Frequency is a physical phenomenon. Pitch is the perceptual correlate of frequency.

The default TuneScope display shows about 100 milliseconds (i.e., one-tenth of a second) of the waveform. Ten cycles of a tone with a frequency of 100 cycles per second will appear in 100 milliseconds. Counting the peaks of the cycles (as shown in the illustration below) is a good way to confirm that ten cycles of a 100 cycle per second tone will be displayed in 100 milliseconds.
Since there are 1,000 milliseconds in one second, 100 cycles in one second is equal to 1 / 100 second (0.01 seconds). The duration of one cycle is known as the period. Therefore a 100 cycle per second tone has a period of 1 / 100 second. Not all sounds have a discernable pattern that repeats. Sounds that have a repeating period are known as periodic sounds. Sounds that do not have a repeating period are known as aperiodic sounds.

**Exploration 7.2 Synthesized Sounds – Frequency**

Explore tones of different frequencies. When the frequency of a tone is doubled, what is the effect on the perceived sound? What is the lowest frequency tone that you can hear? What is the highest frequency sound that you can hear?

**Topic 7.3 Synthesized Sounds – Intensity**

The height of the waveform is known as its amplitude. In the illustration below, the amplitude of the highest positive peak is labeled + 0.5 and the amplitude of the lowest negative peak is labeled – 0.5.

![Amplitude Graph](image)

The maximum amplitude of a waveform that can be displayed without distortion in TuneScope is 1.0 for the highest positive peak and – 1.0 for the lowest negative peak. If the waveform exceeds those values, it will be distorted.

When the tone is sent to the computer speaker, the high peak of the waveform corresponds to the farthest forward excursion of the speaker cone. The low peak of the waveform on the display corresponds to the farthest backward excursion of the speaker cone. If there are 100 cycles of the acoustic waveform in one second, the speaker cone moves forward and backward 100 times in that second. Thus, the amplitude of the waveform corresponds to *how far* the speaker cone moves while the frequency of the waveform corresponds to *how many times* the speaker cone moves back and forth in one second. When the speaker cone moves a greater distance, it displaces more air, resulting in a more intense sound. The intensity of a sound is measured in decibels.

Intensity is correlated with the perceived loudness of a sound. As the intensity increases, the perceived loudness increases. Intensity is a physical phenomenon. Loudness is the perceptual correlate of intensity.

**Exploration 7.3 Synthesized Sounds – Intensity**

Explore synthesized tones with different amplitudes. When the amplitude of a tone is doubled, what is the effect on the perceived loudness? What is the lowest amplitude tone that you can hear? When the amplitude of a tone exceeds 1.0 on the TuneScope display what is the effect on the perceived quality of the sound generated?
Topic 4. Synthesized Sounds – Complex Tones

Most naturally occurring periodic sounds are not pure tones. Instead, naturally occurring periodic sounds are typically composed of several pure tones. In the illustration below, two pure tones have been generated in TuneScope: a 100 Hz tone and a 200 Hz tone.

![Tone Numbers and Frequencies](image1)

The TuneScope display that results from combining a 100 Hz tone and a 200 Hz tone looks like the illustration below.

![Tone Waveform](image2)

A sound that is composed of two or more pure tones is known as a complex tone. A complex waveform has a quality that is perceived as richer than a pure tone. Most musical instruments generate sounds that are complex waveforms.

Music synthesizers recreate these musical tones by combining pure tones of different frequencies and amplitudes. A series of tones can be created in TuneScope by snapping blocks together in the same manner as the two tones in the previous example.

A **For** loop can be used to generate a series of tones to create a complex waveform. A computer loop executes an action (in this case, turning on a tone) over and over. In this example, the loop is repeated three times. During the first iteration of the loop, the value of the variable “i” is “1”; during the second iteration, the value of the variable “i” is “2”; during the third iteration, the value of the variable is “3”.

![For Loop](image3)

The effect is the same as creating the following three sets of commands manually. In this case, Tone 1 has a frequency of 100 Hz, Tone 2 has a frequency of 200 Hz, and Tone 3 has a frequency of 300 Hz.
A loop makes it possible to experiment with complex waveforms in a fraction of the time that it would have taken to create these waveforms manually by snapping commands together one at a time.

A loop also can be used to quickly turn off a series of tones.

In previous sections, the acoustic characteristics of frequency and intensity have been explored. The perceptual correlate of complexity in a waveform is timbre. The perceived quality of a sound – known as its timbre – is affected by the complexity of the waveform.

**Essential Knowledge**

- CS Principle 3.8 Iteration
- CS Principle 3.11 Calling Procedures

**Application of Essential Knowledge**

In the preceding example, the Tone Frequency _ Amplitude _ block is called three times using a For block to iteratively create three separate tones with different frequencies.

**Exploration 7.4 Synthesized Sounds – Complex Tones**

Explore the effect of combining several tones to create a complex waveform. How does addition of several pure tones affect the perceived quality of the sound generated?
Topic 7.5 Custom Code Blocks

Once you are satisfied with the effect of a sequence of code blocks that you have created, you can create a custom code block to execute that sequence. Click the plus sign “+” in the corner of the Command Palette to make a custom block.

Then assign a name to the new block (Tone Off in this example) in the dialog box that appears.
Click OK to advance to the next dialog box. A *Code Block Editor* will appear. Drag the blocks to turn the tones off into the *Block Editor*.

Then click *OK* to complete the process of creating a custom code block. The newly created code block will appear at the bottom of the code block area in the command palette.

![Block Editor](image)

After dragging the new custom code block into the script area, you can execute the sequence of code blocks by clicking the custom block – in this instance, the newly defined **Tone Off** code block.

**Essential Knowledge**

**CS Principle 3.13 Developing Procedures**

Assigning a name to a group of instructions to create a procedure is one form of abstraction. Once a code sequence is tested and encapsulated in the form of a procedure, the programmer no longer needs to remember the specific details of the procedure, enabling the programmer to focus on other program details.

**Application of Essential Knowledge**

In the example above, a custom code block named **Tone Off** replaced a longer sequence of instructions. Replacing an entire sequence of instructions with a named procedure yields several benefits. First, it makes the top level of the program more concise and easier to read. Second, if meaningful names are used, it makes the program easier to comprehend.

**Exploration 7.5 Custom Code Blocks**

Create a custom code block that generates a complex waveform.
**Topic 7.6 Sampled Sounds – Duration**

Duration is one other parameter of a sound that is important in music. The duration of a tone can be set using a **Wait** code block to establish the length of time that the tone plays before it is turned off.

This capability can be used to create a series of pulses by rapidly turning a tone on and off.

An input to a custom code block can be created by clicking the plus ("+"”) sign to the right of the name of the code block in the code block editor. A dialog box will then appear that enables an input to the code block to be created. In this illustration, the input **Times** will be used to specify the number of times that the pulse will occur.
The orange oval representing the *Times* input is then dragged into the input slot for the *Repeat* block.

![Diagram of Block Editor](image)

The input to the custom code block will then determine the number of times that the tone will pulse on and off.

This method can be extended for complex tones, and can even be used to combine several complex tones. This method will be extended to create notes for musical instruments and musical chords in the next module.

**Essential Knowledge**

**CS Principle 3.13 Developing Procedures**

Assigning a name to a group of instructions to create a procedure is one form of abstraction. Once a code sequence is tested and encapsulated in the form of a procedure, the programmer no longer needs to remember the specific details of the procedure, enabling the programmer to focus on other program details.

Using parameters allows procedures to be generalized, enabling the procedures to be resused with a range of input values or arguments.

**Application of Essential Knowledge**

In the example above, a custom code block named *Pulse* replaced a longer sequence of instructions. The *Pulse* block has one input named *Times*. The input *Times* controls the number of times that the pulse will occur.

**Exploration 7.6 Sampled Sounds - Duration**

Create a custom code block with an input that pulses a complex waveform.

**Topic 7.7 Sampled Sounds – Recording Sounds**

Synthesized sounds, described in preceding sections, are one method of generating sounds with a computer. The other method of producing sounds with a computer consists of capturing and playing
back natural occurring sounds. The computer’s microphone can be used to capture a sample of a naturally occurring sound.

The sound recorder in TuneScope is accessed through the Sounds tab to the right of the Scripts tab. (Note: on a Macintosh, you will be asked to give permission to access the computer’s microphone the first time the sound recording capability is accessed.)

Once a sound has been recorded, it can be assigned any name that is appropriate. In this instance, a ringing sound was created by tapping a fork on a water glass.

**Exploration 7.7 Sampled Sounds – Recording Sounds**
Record several sounds. Assign a meaningful name to each sound sample that has been captured.

**Topic 7.8 Sampled Sounds – Playing Recorded Sounds**
Once a sound has been recorded and assigned a name, it can be accessed through a drop-down menu in the Play Sound code block.

Once a sound has been selected through the drop-down menu, it can then be played by clicking the Play Sound block.
**Exploration 7.8 Sampled Sounds – Playing Recorded Sounds**

Use the **Play Sound** code block to play back recorded sounds. What happens if you play several sounds one after the other by snapping several Play blocks together? What is the difference between the **Play Sound** code block and the **Play Sound until Done** code block?

**Topic 7.9 Designing a Sound Pad on the Computer Keyboard**

Professional musicians use specialized hardware that can be used to create musical loops (i.e., a repeating musical pattern) and sound effects. A sound pad is one type of music controller that is popular with musicians and audio engineers. Rows of keys control sound effects and sound loops. Pressing a key triggers a specific sound or musical loop.

Any sound or code block created in TuneScope can be assigned to a key on the computer keyboard. The **When Key __ Pressed** code block is used to assign a sound or code block to a key on the computer keyboard.

Once a sound or code block has been assigned to a key on the computer keyboard, pressing the key triggers the sound effect or pre-recorded sequence.

**Essential Knowledge**

**CS Principle 3.6 Conditionals**

Selection determines which parts of algorithm are executed based on whether a condition is true or false.

**Application of Essential Knowledge**

In example above, the **When Key Pressed** code block establishes a condition that executes an instruction when the condition is met. In this example, the condition consists of pressing a specific key on the computer keyboard.

**Exploration 7.9 Designing a Sound Pad on the Computer Keyboard**

Assign a series of a half-dozen to a dozen synthesized and sampled sounds to keys on the computer keyboard. Then use a smart phone to record a video performance in which a dramatic reading of a story or poem is accompanied by sound effects.
8. Musical Notes and Scales

Glen Bull, Jo Watts, and Joe Garofalo

A music note is associated with a rate of vibration or frequency. For example, when a piano key is struck, a lever causes a padded hammer to strike a piano string. When struck, the piano string vibrates a specific number of times per second.

A standard piano keyboard has 88 keys. Notes on the left end of the keyboard play notes that correspond to the lower frequencies, while notes on the higher end of the keyboard play notes that correspond to the higher frequencies. The frequency of the lowest note on a standard piano keyboard is 27.5 Hz. The highest note on a piano keyboard is slightly more than 4,000 Hz.

The piano key for the note “C” that is in the middle of the keyboard is known as “Middle C.” Middle C has a frequency of 262 Hz. The frequency of the next C on the keyboard (above middle C) is 524 Hz. In other words, it is double the frequency of middle C.

**Topic 8.1 Musical Scales**

As noted above, each music note is associated with a frequency. The term musical scale refers to the relationships among the frequencies of different notes. Many different musical scales have been developed throughout history.

The modern western musical scale consists of twelve notes. A span of seven white notes and five black notes on a piano keyboard is known as an octave. The octave that begins with middle C is the fourth octave (counting from the left) on the keyboard.

Therefore, middle C can be written as “C4”. In this example, the letter refers to the musical note and the number refers to the octave.
The *Table of Notes and Frequencies* below displays the frequencies for the twelve notes that begin with the note “A” above middle C.

The relationship of each note to the frequency of the note before is always the same. For example, the ratio of C (523 Hz) to the B (494 Hz) is calculated by dividing 523 by 494. The ratio obtained is 1.059. The ratio of each of the other notes to the note before is also the same ratio: “1.059”. This is an important characteristic of the western musical scale.

<table>
<thead>
<tr>
<th>Note</th>
<th>Hz</th>
<th>Ratio of Current Note to the Previous Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>440</td>
<td></td>
</tr>
<tr>
<td>B♭</td>
<td>466</td>
<td>[466 \div 440 = 1.059]</td>
</tr>
<tr>
<td>B</td>
<td>494</td>
<td>[494 \div 466 = 1.059]</td>
</tr>
<tr>
<td>C</td>
<td>523</td>
<td>[523 \div 494 = 1.059]</td>
</tr>
<tr>
<td>C♯</td>
<td>554</td>
<td>[554 \div 523 = 1.059]</td>
</tr>
<tr>
<td>D</td>
<td>587</td>
<td>[587 \div 554 = 1.059]</td>
</tr>
<tr>
<td>D♯</td>
<td>622</td>
<td>[622 \div 587 = 1.059]</td>
</tr>
<tr>
<td>E</td>
<td>659</td>
<td>[659 \div 622 = 1.059]</td>
</tr>
<tr>
<td>F</td>
<td>698</td>
<td>[698 \div 659 = 1.059]</td>
</tr>
<tr>
<td>F♯</td>
<td>740</td>
<td>[740 \div 698 = 1.059]</td>
</tr>
<tr>
<td>G</td>
<td>784</td>
<td>[784 \div 740 = 1.059]</td>
</tr>
<tr>
<td>A♭</td>
<td>831</td>
<td>[831 \div 784 = 1.059]</td>
</tr>
<tr>
<td>A</td>
<td>880</td>
<td>[880 \div 831 = 1.059]</td>
</tr>
</tbody>
</table>

The ratio obtained is the twelfth root of the number 2, which can be represented mathematically by the following symbols:

\[\sqrt[12]{2}\]

This can be calculated in the following way:

\[2 ^ {\frac{1}{12}} = 1.0594630943592953\]

This ratio can be used to calculate each of the notes following “A”. For example, if “A” has a frequency of 440, the note “B♭” (which follows “A”) can be calculated in the following way:

\[440 \times 2 ^ {\frac{1}{12}} = 466.1637615180899\]

The result of multiplying 440 by the twelfth root of 2 yields 466 Hz, which is the correct answer.
This method therefore can be used to play the note “B♭” using 440 as a base frequency:

![Diagram showing tone number, frequency, amplitude, and notes]

An expansion of this method can be used to play all twelve notes of the musical scale using a loop.

![Diagram showing loop with note creation and multiplication]

The technique can be useful as a starting point for exploring musical scales.

**Essential Knowledge**

- **CS Principle 3.1** Variables and Assignments
- **CS Principle 3.8** Mathematical Expressions
- **CS Principle 3.8** Iteration

**Application of Essential Knowledge**

In the example above, a variable named `Note` was created. The initial value of the variable was set to 440. During each iteration of the loop, the variable `Note` was multiplied by the twelfth root of 2. Since the modern Western musical scale uses this mathematical expression to arrange notes on the musical scale, the repeated iterations through the loop play the twelve notes of the scale.

**Exploration 8.1 Musical Scales**

Design a loop that plays a musical scale. How important is precision? Explore the effect of multiplying by “1.059” or “1.06” rather than calculating the result using ten or more decimal places. If only two decimal places of precision are used, what is the frequency of the final note in the scale? Is this enough to make an audible difference?

Explore other scales. For example, multiply by a number other than the twelfth root of two or start with a base frequency other than 440. What are other possibilities for creation of musical scales?
Musical notes that form whole number ratios sound more pleasing. For example, the mathematical ratio formed by the relationship between the notes A and E in the musical scale is approximately a 3:2 ratio.

The frequency of E (659 Hz) subtracted from the frequency of A (440 Hz) yields 219 Hz. The frequency 659 Hz divided by 219 Hz is approximately 3. The frequency 440 Hz divided by 219 Hz yields approximately 2. Consequently, the ratio of E to A is a 3:2 ratio.

Therefore, multiplying 440 (the frequency for A) by the ratio “3 divided by 2” yields 660. This result approximates the frequency for E (which is 659).

Pairs of notes with intervals that form whole number ratios are known as perfect intervals. The interval between notes that form a 3:2 ratio (like the interval between A and E) are known as perfect fifths. Similarly, two notes whose interval form a ratio of 4:3 form a perfect fourth.

The intervals between each successive pair of notes in the modern western scale is always the same (i.e., the twelfth root of 2). One advantage of equal intervals between notes is that two notes such as A and E can be shifted along the scale – for example, to the notes B♭ and F – and the ratio between the notes will remain the same. Shifting notes along the scale in this manner is known as transposing the notes.

**Exploration 8.2 Relationships among Notes**

Explore relationships among pairs of notes whose frequencies form small whole-number ratios such as 3:2 and 4:3. What pairings sound pleasing? Are there pairings that are not harmonious?

Shift pairs of notes along the musical scale. How does shifting pairs of notes in this manner affect the quality of the perceived sound?
Topic 8.3 Complex Tones

The Tone code block generates pure tones. The waveform for a pure tone is a sine wave. A pure tone does not have as much auditory depth as the natural tone generated by a musical instrument such as a piano or guitar.

A richer sound with more depth can be produced by combining several pure tones to create a complex tone. For example, if a 100 Hz tone is combined with a 200 Hz tone, the following complex waveform is generated.

![Complex Tone Waveform](image)

The lowest frequency in a complex waveform is known as the fundamental frequency. Tones above the fundamental are known as overtones. In a natural sound, overtones are typically multiples of the fundamental frequency. A loop can be used to generate a series of overtones.

![Loop for Generating Overtones](image)

The loop shown above generates 100 Hz, 200 Hz, 300 Hz, 400 Hz, and 500 Hz tones. The resulting waveform looks like the illustration below. In this illustration, each of the overtones has the same amplitude as the fundamental frequency. Overtones produced by a musical instrument typically decrease in amplitude.

![Overtones Waveform](image)

In this example, the fundamental frequency (100 Hz) generates the highest peaks. The successive peaks that follow the highest peaks are produced by the overtones. The period of time from one of the high peaks to the next high peak is known as a cycle. A waveform in which each cycle is similar to the one before (as is the case in this example) is known as a periodic waveform (because each period repeats the one before).
The number of cycles that are completed in a second are known as cycles per second. In this example, one cycle is completed in one-hundredth of a second (i.e., ten milliseconds). Consequently, 100 hundred cycles would be completed in one full second. Therefore, the frequency of the sound is said to be 100 cycles per second.

Essential Knowledge

- CS Principle 3.1 Variables and Assignments
- CS Principle 3.8 Mathematical Expressions
- CS Principle 3.8 Iteration

Application of Essential Knowledge

In the example above, a For loop to increase the value of the variable “i” from 1 to 5 during each successive iteration of the loop. The value of the variable, in turn, is multiplied by 100 to create a series of notes in which the frequency of each note is increased during each iteration of the loop.

Exploration 8.3 Complex Tones

Explore creation of complex waveforms are lower in amplitude than the fundamental frequency. Identify factors that seem to contribute to generation of more pleasant and richer-sounding tones.

Topic 8.4 Sampled Instruments

Music synthesizers such as the Moog synthesizer (described in the previous unit, Computer-generated Sound and Music) by combining artificially generated tones to create complex waveforms and by applying other effects to sounds to simulate musical instruments. This approach to using a computer to generate music is known as synthesized sound.

When a sound sample is recorded in TuneScope, vibrations in air move the diaphragm of the computer’s microphone back and forth. This back-and-forth movement is converted into an electrical signal by the diaphragm. In a sense, then, each sound sample captured by the computer represents a snapshot of the distance that the diaphragm of the microphone has traveled from its midpoint at a given moment in time. (When the diaphragm is moving forward, it generates a positive voltage. When it is moving backwards, it generates a negative voltage.)

Each sound sample captured by the computer therefore represents the analog voltage generated by the microphone at that moment in time. This voltage is translated into a digital number. A drop-down menu for the reporter block shown below can be used to identify the number of times per second that sample voltage values were acquired. This characteristic is known as the sample rate.
In this particular instance, a plucked guitar string was sampled at a rate of 48,000 times per second.

The note recorded in this instance was an “A”. If the sound is played back at the same frequency that it was recorded, the note “A” is reproduced.

If the note is played back at a higher rate than it was sampled, a higher frequency is produced. If the reproduction rate is multiplied by two to the twelfth, this will increase the play back rate to more than 50,000 samples per second.

Increasing the playback rate by this amount is sufficient to raise a sound that is 440 Hz when recorded to 466 Hz when played back. In other words, a note that is recorded as “A” (440 Hz) will be played back at 466 Hz (B♭).

By recording a sound at one sampling rate and playing the sound back at a different frequency, a single recorded note can be used to play back any note in the scale.

**Essential Knowledge**

- **CS Principle 3.8** Mathematical Expressions
- **CS Principle 3.12** Calling Procedures
- **CS Principle 3.16** Simulation

Simulations are abstractions of more complex objects or phenomena for a specific purpose. A simulation is a representation that uses varying sets of values to reflect the changing state of a phenomenon. Simulations often mimic real-world events with the purpose of drawing inference, allowing investigation of a phenomenon without the constraints of the real world.

**Application of Essential Knowledge**

TuneScope uses mathematical models (i.e., simulations) to generate tones. In the example above, custom code blocks from the TuneScope library such as `Play Sound _ at _ Hz` were combined with variables to play notes.

**Exploration 8.4 Sampled Instruments**

If a musical instrument is available, record the first note in the scale (“A”). Then play back the recorded sound at different rates to produce all of the other notes in the scale.

Record other periodic natural sounds. For example, tap on a water glass to create a ringing sound. Then adjust the playback frequency to produce all of the notes in the scale.
**Topic 8.5 Musical Notes**

TuneScope includes a number of sampled instruments that can be used to play musical notes. These musical instruments are accessed through the `Set Instrument` code block. This code block enables a musical instrument to be selected from a dropdown menu.

The TuneScope `Set Instrument` code block takes advantage of the capability described in the preceding topic – the ability to record an instrument at one sampling rate and play the sound back at different rates to adjust the reproduced frequency – to play notes that emulate the voices of different musical instruments using the `Play` code block.

In this manner, TuneScope can be used to synthesize musical notes through use of oscillators to create complex tones that approximate natural instruments, or can also be used to reproduce notes electronically by sampling actual instruments.

Previously, computer tones have been expressed in terms of frequency (Hz). Duration of tones have been expressed in seconds (Wait 0.5 seconds). However, musicians use names of notes such as “C, D, E”, etc. to express frequencies. Durations are expressed in terms of quarter notes, whole notes, etc.

A dropdown menu in the TuneScope `Play Note` code block enables musical notes to be selected. (Note designations can also be typed directly into this input slot.)

Once a note has been selected, a second dropdown menu can be used to select the duration of the note. The note duration determines the length of time that the note is played.
Exploration 8.5 Musical Notes

Explore the voices of different sampled musical instruments that can be accessed through the Set Instrument code block. Use the Play code block to play back notes using different instruments.

Topic 8.6 Combining Notes

Several Play Note and Wait code blocks can be combined to play a series of notes. For example, the opening notes of the blues song *Crossroads* are G5, A#5, and A#5

The first note (G5) lasts for one-sixteenth of a note, the second note (A#5) lasts for one-eighth of a note, and the third note (A#5) lasts for one-quarter of a note. The Play Note and Wait code block waits until one note is completed before beginning the next note. The music notation “#” is pronounced “sharp”; for example: “A Sharp” (in this example). (The sharp keys are the black keys on a piano keyboard.)

Exploration 8.6 Combining Notes

Create your own three-note sequence in TuneScope using the Play code block. You can either use notes from a favorite song or create your combination of notes.

Topic 8.7 Tempo

Note durations such as Quarter Note, Whole Note, etc. are relative. A whole note is always longer than a quarter note. (In fact, a whole note is four times as long as a quarter note.) However, the absolute duration of a note is also affected by the tempo. The term tempo refers to how fast or how slow a tune is played. Tempo is expressed in terms of beats per minute (bpm). The tempo is set in TuneScope with the Set Tempo code block.

Musical scores provide a time signature that indicate the number of notes in each measure. For example, a time signature of 4/4 would indicate that there are four quarter notes in one measure, while a time signature of 6/8 would indicate that there are six eighth notes in one measure. For the sake of simplicity, most of the examples in modules that follow will have a time signature of 4/4. In a 4/4 time signature, a tempo of 60 beats per minute would indicate that there are 60 quarter notes in one minute.

Exploration 8.7 Tempo

Explore the effect of tempo on the three-note sequence created in the previous topic.
9. Chord Player and the Blues

Glen Bull, Jo Watts, and Joe Garofalo

The blues is a musical genre that originated with Black musicians in the nineteenth century. The blues influenced many of the popular music forms that followed, including jazz, rhythm and blues, rock and roll, pop, and rock music.

The most common blues format consisted of a line sung over the first four bars, repetition of the line over the next four bars, followed by a concluding line over the last four bars. The song Crossroads, recorded by Robert Johnson in 1936, is a good example of this form.

I went to the crossroad. Fell down on my knees.
I went to the crossroad. Fell down on my knees.
As I asked the Lord above, "Have mercy, now, save poor Bob if you please."

Ooh, standin' at the crossroad. Tried to flag a ride.
Ooh-ee, I tried to flag a ride.
Didn't nobody seem to know me, babe. Everybody pass me by.

Four bars of musical chords were played as each line of the song was sung. When Crossroads is played in the key of C, the following chords accompany each line.

<table>
<thead>
<tr>
<th>C Chord</th>
<th>C Chord</th>
<th>C Chord</th>
<th>C Chord</th>
</tr>
</thead>
<tbody>
<tr>
<td>F Chord</td>
<td>F Chord</td>
<td>C Chord</td>
<td>C Chord</td>
</tr>
<tr>
<td>G Chord</td>
<td>F Chord</td>
<td>C Chord</td>
<td>C Chord</td>
</tr>
</tbody>
</table>

This format, known as the twelve-bar blues, is the most common form of the blues. It consists of four bars per line over three lines, for a total of twelve bars. This format forms the basis of thousands of blues songs. There are countless variations on this format. Because thousands of songs can be played with only three chords, it is also a form that is accessible to musical novices.

Topic 9.1 Play Note and Wait

The **Play Note and Wait** code block was introduced in the previous module (*Notes and Musical Scales*). This code block waits until one note ends before beginning the next.

<table>
<thead>
<tr>
<th>Play</th>
<th>G5</th>
<th>For</th>
<th>Sixteenth</th>
<th>Note Length and Wait</th>
</tr>
</thead>
<tbody>
<tr>
<td>Play</td>
<td>A#5</td>
<td>For</td>
<td>Eighth</td>
<td>Note Length and Wait</td>
</tr>
</tbody>
</table>

In the illustration above, the note G5 is played for one-sixteenth of a beat, followed by A#5 for an eighth of a beat. On a piano, the piano key for the note G in the fifth octave would be struck. The musician would then wait for one-sixteenth of a beat before striking the next piano key.
In contrast, a chord consists of several notes played together at the same time. The **Play Note** code block does not wait until one note ends before beginning the next note.

On a piano, the musician would strike all four piano keys corresponding to the notes A₄, C₄, F₄, and A₅ at the same time.

Some combinations of notes sound more pleasing when played together. For examples, notes whose frequencies form small integer ratios such as 3:2 and 4:3 sound better than some of the other possible combinations of notes when played together. These combinations of notes that sound pleasing when played together are often combined to form chords.

**Exploration 9.1 Play Notes**

Use the **Play Note and Wait** code block to create combinations of notes that sound pleasing. Identify two or three combinations of notes that could be used to form chords.

**Topic 9.2 Play Chord Block**

Rather than using four separate **Play Note** code blocks to play the four notes of the chord, a single **Play Note** code block can be combined with a loop to play the notes in a list.

A more general form of this script can be created through development of a custom code block, **Play Chord**.
The list of notes that form the chord are provided as an input to the Play Chord code block.

While each of the notes in the chord should be played together, the Play Chord block should wait until the chord is completed before continuing to the next chord. This can be accomplished by adding a Rest block at the end of the procedure.

Essential Knowledge

CS Principle 3.2 Data Abstraction

Data abstraction can be created using lists. Developing a data abstraction to implement in program can result in a program that is easier to develop and maintain.

An element is an individual value in a list that is assigned a unique index.

An index is a common method for referencing the elements in a list or string using natural numbers.

The use of lists allows multiple related items to be treated as a single value. Lists are referred to by different names, such as array, depending on the programming language.

CS Principle 3.10 Lists

The Length of List code block reports the number of elements in a list.

CS Principle 3.8 Iteration

CS Principle 3.13 Developing Procedures

Application of Essential Knowledge

In the example above, a list of notes is used to create a musical chord. A For loop is used to develop a custom code block named Play Chord. The custom code block has two inputs: Chord and Note Length.

Exploration 9.2 Play Chord Block

Put one of the combinations of notes developed in Topic 9.1 together in a list. Use the chord list created in this way as an input to the custom Play Chord code block.
**Topic 9.3 Naming Chords**

Because the ukulele only has four strings, it is easier to master than its larger cousin, the guitar. For the same reason, ukulele chords are used in the examples that follow.

The notes A₄, C₄, F₄, and A₅ form an *F* chord on a ukulele. A variable provides a convenient way of associating a name with the notes of a chord. The *Make a Variable* option is found under the *Variables Palette*.

Once the variable *F Chord* has been created, the *Set* code block can be used to assign the value of the variable to the list of notes for that chord.

When the oval block with the variable name is dragged into the script area, clicking the variable name block causes it to display the list of notes associated with the variable.
The variable block can now be used as an input to the Play Chord block. Using the name of the chord improves the readability of the code and makes it more understandable.

**Essential Knowledge**

**CS Principle 3.1 Variables and Assignments**

A variable is an abstraction inside a program that can hold a value. Each variable has associated data storage that represents one value at a time.

Using meaningful variable names helps with the readability of program code and understanding of what values are represented by the variables.

**Application of Essential Knowledge**

In the example above, variables such as *F Chord* and *C Chord*. These variables are used as inputs to the **Play Chord** block.

**Exploration 9.3 Naming Chords**

Create variables for the *C Chord* and *G Chord*. Then use these variables with the **Play Chord** block.

**Topic 9.4 Playing Lists of Chords**

The **Play Chord** block can be combined with a loop to play a list of chords. For example, a list of the chords in the song *Crossroads* can be created by first making a variable named *Crossroads Chords*. Then the **Set** code block is used to create an empty list for this variable. Once this is done, a list will be displayed with an empty input slot for **Item 1** of the list.
The name of the first chord ("C Chord") is typed into the empty slot. Then a slot for Item 2 of the list is added by clicking the plus sign ("+") in the lower left-hand corner of the list editor. This creates a second item that can be used to enter the name of the second chord.

Crossroads begins with four C Chords followed by two F Chords, as shown in the illustration below.

The Var code block reports the value of a variable when the text of a variable name is entered as an input to the Var block. The contents of the variable C Chord can be examined by using this reporter block.

Item 1 of the list of Crossroads Chords is the text “C Chord”.

111
Consequently, when Item 1 of Crossroads Chords is supplied as an input to the Var block, this reporter block returns the list of notes for the C Chord. This method involves two steps:

1. The Item code block retrieves the text “C Chord” from the list of Crossroads Chords.
2. The Var code block then uses the text “C Chord” as an index that is used to retrieve the four notes associated with this chord.

In this manner the items in the Crossroads Chords list can be used to retrieve the individual notes associated with each chord in the list.

The code described above can then be supplied as an input to the Play Chord. The result is that notes associated with the retrieved chord are then played.

This method can be combined with a For loop to play all of the chords in the list of Crossroads Chords.

A more generalized version of this script can be used to create a custom code block, Play Chord List. The Length of List code block enables this custom block to play tunes of any length.
Any list of chords can then be supplied as an input to the **Play Chord List** code block.

![Play Chord List](Crossroads Chords)

**Essential Knowledge**

**CS Principle 3.2** Data Abstraction

Data abstraction can be created using lists. Developing a data abstraction to implement in a program can result in a program that is easier to develop and maintain.

An element is an individual value in a list that is assigned a unique index.

An index is a common method for referencing the elements in a list or string using natural numbers.

The use of lists allows multiple related items to be treated as a single value. Lists are referred to by different names, such as array, depending on the programming language.

**CS Principle 3.10** Lists

The code block **Length of List** reports the number of elements currently in the list.

**CS Principle 3.8** Iteration

**CS Principle 3.13** Developing Procedures

**Application of Essential Knowledge**

In the example above, a list of chord names is assigned to a variable named **Crossroads Chords**. The custom code block **Play Chord List** then is used to play the list of chords.

**Exploration 9.4 Playing Lists of Chords**

Enter the chords for first twelve chords of **Crossroads** in the **Crossroads Chords** list. Then use this variable as input to the **Play Chord List** block to play the song.

**Topic 9.5 Musical Arpeggio**

An arpeggio consists of chords in which there is a delay between notes. The guitarist in *Unchained Melody* plays a slow arpeggio — starting with an F chord and inserting a delay between notes as he walks the guitar pick down strings.

**Exploration 9.5 Musical Arpeggio**

Explore insertion of a **Wait** code block between notes in a chord to create an arpeggio. What delay times sound best?
**Topic 9.6 Chord Combinations**

Changing the order of the notes in a chord can significantly alter the way a chord sounds. Often, notes are rearranged so that the lowest note in the chord matches the base line or so that the highest note matches the melody. For example, in *All Along the Watchtower* performed by Jimi Hendrix, the F chord is played F3, A3, C4, F4. How does this chord sound different from the F chord you created?

A G chord can be created with any combination of the three notes: G, B, and D. A G Chord on the ukulele (which has four strings) could also be created with two G notes (separated by an octave), one B note, and one D note. Try other combinations. The computer can be used to create combinations of notes that would difficult to play on an actual instrument because the finger positions would be too awkward or impossible.

**Exploration 9.6 Musical Chords**

Shift the notes in the F chord to a different octave. When the notes are shifted by an octave, does it still sound like an F chord?

What happens if you shift some of the notes in the chord by one or more octaves, and leave other notes in the chord in the original octave? Find the combination that sounds best to you.
10. Building a TuneScope Drum Machine

Glen Bull, Jo Watts, and Joe Garofalo

In this module, construction of a simple rhythm using a drum machine is explored. A drum machine is sometimes known as a *drum sequencer*. Musicians and DJs use drum machines to build drum tracks. One type of interface for a drum machine resembles the illustration below.

There are two rows of 16 squares. This particular drum machine is set to *sixteenth notes*, so there are 16 possible beats to be played. The top of the grid indicates where the *quarter notes* land within the measure.

![Drum Machine Interface](image)

To play the kick drum *on each quarter note* the squares at the first, second, third, and fourth quarter note location are turned on. This is shown in the example below. In this example, the kick drum plays on the first, second, third, and fourth quarter note of a measure.

![Drum Machine Example](image)

The following example illustrates a simple blues rhythm in which the kick drum is played on the *first* and *third* quarter notes, while the snare drum is played on the *second* and *fourth* quarter notes.

![Drum Machine Blues Rhythm](image)

This is one of the most basic, yet foundational rhythms that nearly all other blues and rock rhythms are constructed from. In the topics that follow, a similar interface for a drum machine will be constructed in TuneScope. TuneScope has several types of drums, including a snare drum and a bass drum.

![Drum Machine Options](image)
To simplify construction, an interface with only eight rather than sixteen slots is created in the example that follows. However, once the foundational principles are understood, a drum machine of any type or design can be constructed. In the interface shown below, each of the eight slots represents an eighth note.

![Drum Machine Interface](image)

**Topic 10.1 Creating a Play Drum Code Block**

If the *Set Instrument* code block is used to set the instrument to *Bass Drum*, the code block **Play C3 for Eighth Note and Wait** creates a synthesized drum sound.

```
Set Instrument To bass drum
Play C3 For Eighth Note Length and Wait
```

These two code blocks can be combined into a custom code block, **Play Bass Drum**.

```
+Play+Bass+Drum+
Set Instrument To bass drum
Play C3 For Eighth Note Length and Wait
```

In the previous example, the *Bass Drum* is played on the first and fifth notes, with empty eighth note slots in between. The following code combines the **Play Bass Drum** code block with **Rest** code blocks to create a sequence that approximates the illustrated drum machine rhythm.

```
Play Bass Drum
Rest for Eighth Length of Time
Rest for Eighth Length of Time
Rest for Eighth Length of Time
Play Bass Drum
Rest for Eighth Length of Time
Rest for Eighth Length of Time
Rest for Eighth Length of Time
```

The entire eight block sequence shown above could be placed in a Forever block to cause this rhythm to repeat indefinitely.

**Essential Knowledge**

- **CS Principle 3.12** Calling Procedures
- **CS Principle 3.13** Developing Procedures
**Exploration 10.1 Creating a Play Drum Code Block**

Create a custom **Play Bass Drum** code block in the manner illustrated above. Then combine the code block with **Rest** blocks to create a bass drum rhythm.

**Topic 10.2 Representing a Drum Pattern with a List**

The drum machine interface described in the introduction to this module represents drum patterns with a series of check boxes. In TuneScope, rather than extending the unwieldy sequence of code blocks shown in the previous topic, the rhythm could be represented in the form of a list in which an “X” represents a drum hit and an empty box represents a rest.

The sequence could be assigned to a **Bass Drum Sequence** variable in this manner:

This pattern enables a **Bass Drum Sequencer** code block to play the bass drum when it encounters an “X” in the list and otherwise rest for one-eighth note. This loop plays the drum sequence in a more efficient manner than the in-line code described in the preceding section.

This loop provides the basic mechanism for building a drum machine in *TuneScope*.

**Essential Knowledge**

- **CS Principle 3.1** Variables
- **CS Principle 3.2** Data Abstraction
- **CS Principle 3.6** Conditionals
- **CS Principle 3.8** Iteration
- **CS Principle 3.10** Lists
- **CS Principle 3.13** Developing Procedures
Application of Essential Knowledge

The drum loop example combines a number of concepts introduced in previous modules, including variables, procedures, iteration, and lists. The most important concept employed is data abstraction. The drum pattern is represented in a format—in this instance consisting of check boxes in a list—that is more efficient than employing in-line code to accomplish the task.

Exploration 10.2 Representing a Drum Pattern in a List

Use a list to create a drum sequence. Then create a drum sequencer script to play the drum sequence.

Topic 10.3 Building a User Interface

The code developed in the preceding sections provides the engine that will become the heart of the drum machine. Construction of a user interface is the next step in the process. The interface will consist of a series of checkboxes. Clicking a checkbox one time places an “X” in the box. Clicking the checkbox a second time removes the “X” from the box. Each box with an “X” represents a drum hit in the sequence.

A checkbox can be constructed by creating a sprite with two costumes, one labeled Off (unchecked) and the other costume labeled On (checked).
A sprite’s costumes are also numbered, in the order displayed. Therefore, the unchecked costume is 
*Costume # 1* and the checked costume is *Costume # 2*. When the checkbox sprite is clicked, if the 
checkbox costume is *Off* (i.e., *Costume # 1*), the sprite switches to the costume labeled *On*. (That is, if 
the checkbox was previously unchecked, it switches to the checked costume and vice versa.) This 
type of user interface is known as a *toggle*, because the costume toggles from one state to another 
each time the sprite is clicked.

If the new costume is *Costume # 2* (as shown in the illustration below), an “X” is placed in the first 
item in the first item of the *Bass Drum Sequence* list. If the new costume is *Costume # 1*, the “X” is 
removed from the first item of the *Bass Drum Sequence* list, leaving an empty box.

The sprite’s costume provides a visual representation of the status, and also provides a way for the 
user to change the status by checking or unchecking the box. Most importantly, toggling the sprite’s 
costume between a checked box and an empty box also changes the state of the corresponding item in 
the *Bass Drum Sequence* list. The *Bass Drum Sequence* loop then processes the list to play the drum 
sequence.
To provide additional user feedback, a click can be added to the end of the script. The **Play Sound** **Click** code block provides audible feedback to the user each time the checkbox is clicked.

Once the first sprite checkbox has been developed and tested, the sprite and its script can be duplicated to create a total of eight checkboxes. Each duplicate sprite will need its script revised to reflect its order in the sequence. For example, in the script for the second sprite *Item 1* would need to be replaced with *Item 2*.

The duplicated sprites with updated scripts can then be arranged in a row to provide the user interface for creating a drum sequence.
Essential Knowledge

- **CS Principle 3.6** Conditionals
- **CS Principle 3.10** Lists

**Application of Essential Knowledge**

A user interface provides a way for a user to change the state of an application without directly writing code. Clicking a series of checkboxes provides a faster and more efficient way of constructing a drum sequence than re-writing the code each time a different drum pattern is created. The checkbox interface is known as a *Graphical User Interface* (GUI). The graphical user interface was invented at the Xerox Palo Alto Research Center (Xerox PARC). Steve Jobs popularized the graphical user interface through introduction of the Apple Macintosh, the first widely adopted computer with a GUI.

**Exploration 10.3 Building a User Interface**

Create a checkbox user interface for a drum sequencer. Then use the interface to explore creation of different drum patterns.

**Topic 10.4 A Drum Sequencer Prototyping Kit**

The code and resources developed in the preceding sections can be combined to create an updated **Bass Drum Sequencer** code block. This code block examines each item in the *Bass Drum Sequence* list and executes the **Play Bass Drum** code block if the current item in the list contains an “X”. By repeating this loop forever, the bass drum sequence repeats over and over again.

```
for i = 1 to length of Bass Drum Sequence
    if item i of Bass Drum Sequence = X
        Play Bass Drum
    else
        Rest for Eighth Length of Time
```

A parallel **Snare Drum Sequencer** code block and interface can be constructed by duplicating the methods used to construct the bass drum sequencer.
Once the drum sequencer code blocks have been constructed for both the snare drum and the bass drum, a main **Drum Sequencer** code block is needed to launch both the bass drum sequencer and the snare drum sequencer.

If the **Bass Drum Sequencer** and the **Snare Drum Sequencer** code blocks were snapped together without using the Launch code block, the **Bass Drum Sequencer** code block would play forever (since it contains a **Forever** loop). Therefore the **Snare Drum Sequencer** code block would never be executed. However, the **Launch** code block can launch both code blocks in parallel.

**Essential Knowledge**

- **CS Principle 3.1** Variables and Assignments
- **CS Principle 3.2** Data Abstraction
- **CS Principle 3.6** Conditionals
- **CS Principle 3.8** Iteration
- **CS Principle 3.10** Lists
- **CS Principle 3.13** Developing Procedures

**CS Principle 4.3** Parallel and Distributed Computing

Parallel computing is a computational model in which a program is broken into multiple smaller sequential computing operations that are executed simultaneously.

**Application of Essential Knowledge**

Executing the snare drum sequence and the bass drum sequence in parallel simplifies the programming task and distributes the work. The order in which code blocks are executed is an important dimension of coding logic. The first computers could only execute program steps one step at a time. The ability to execute two different code blocks in parallel creates many new possibilities and capabilities for program development.

**Exploration 10.4** Drum Sequencer Prototyping Kit

Create a custom **Drum Sequencer** code block that plays two different types of drum sequences (such as a snare drum sequence and a bass drum sequence). Then use the drum sequencer to create a drum track to accompany a song.
Topic 10.5 Visualizing the Drum Pattern

Once the basic drum sequencer has been implemented and tested, there are a number of enhancements and refinements that could be added. Highlighting the active cell when a drum beat is played enables users to follow the rhythm visually.

The active cell in the table can be highlighted by adding a third costume to each sprite.

If the sequencer encounters an “X” in a cell in the table, it switches to the highlighted costume before initiating the **Play Drum** instruction. After the drum is played, the costume is then switched back to the previous costume. This causes the active cell in the table to be highlighted as a drum hit is occurring.

**Essential Knowledge**

- **CS Principle 3.13** Developing Procedures
- **CS Principle 3.12** Calling Procedures

**Application of Essential Knowledge**

Displaying a red dot in the checkbox at the same time the drum hit occurs provides another form of feedback that makes it easier to follow the drum pattern.

**Exploration 10.5 Visualizing the Drum Pattern**

Modify the drum sequencer to provide visual feedback as each drum sequence plays.
Topic 10.6 Controlling the Tempo

The beats per minute can be adjusted using the Set Tempo code block.

```plaintext
set tempo to 40 bpm
set tempo to 80 bpm
```

If a variable named *Beats per Minute* is created, a slider can be created to control the tempo. Right-clicking a variable watcher displayed on the stage accesses a menu that can be used to convert the variable watcher to a slider.

Inserting the code block `Set Tempo to Beats Per Minute BPM` into the drum sequencer loop enables the slider to control the tempo.

```
set tempo to Beats per Minute bpm
```

Essential Knowledge

**CS Principle 3.1 Variables and Assignments**

Application of Essential Knowledge

*Tempo* is not a standard user-defined variable. It is a system variable that affects the entire TuneScope environment. Changing *Tempo* also affects any other programs that also use this system variable.

**Exploration 10.6 Controlling the Tempo**

Add a control to the user interface that enables the user to control the tempo with a slider.
11. A Tune Recorder / Player

Glen Bull, Jo Watts, and Joe Garofalo

Previous modules illustrated how musical notes and chords can be assigned to keys on the computer keyboard. Sequences of notes were created by snapping **Play** code blocks together.

![Play Block Example](image)

In this module, rather than snapping blocks of code together to create musical sequences, a *TuneScope* recorder will be created to record sequences of notes entered by pressing keys on the computer keyboard.

Electronic keyboards can be purchased that can be used to enter notes that are recorded on the computer.

![Electronic Keyboard](image)

Since a computer keyboard is available on every computer at no additional cost, the computer keyboard will be used as a substitute for an electronic piano keyboard in this module.

**Topic 11.1 Playing a List of Notes**

When the computer keys are pressed, musical notes assigned to computer keys will be recorded in a list. A *TuneScope Player* will be used to play the tunes recorded in this manner.

The same type of computer loop used to play chord sequences in previous modules can also be used to play a series of notes. To use a loop to play a sequence of notes, first place the notes in a list in the order in which they will be played. (The **List** code block is found in the **Variables** section of the **Command Palette**.)

![List Block Example](image)

In order to play the notes, a method of accessing individual notes in the list is required. The **Item** code block (also found in the **Variables** section of the **Command Palette**) can be used to access the notes. For example, **Item 1** can access the first item in the list (the note A4), **Item 2** can access the second item in the list, and so on.

![Item Block Example](image)
Therefore, the code block **Play Item 1 of List**:

```
Play item 1 of list A4 B4 E4 for Quarter Note Length
```

Is effectively the same as the code block **Play A4**:

```
Play A4 for Quarter Note Length
```

In the next section, lists will be used to play an entire sequence of notes with a single **Play** code block.

**Essential Knowledge**

**CS Principle 3.10 Lists**

**Application of Essential Knowledge**

In the *Drum Machine* module, lists were used to represent drum sequences. In this module, the application is similar. However, rather than using an “X” to represent a drum hit, each item in the list represents a musical note. In the same way that a drum sequencer code block was constructed to play drum patterns, a tune player can be constructed to play lists of notes.

**Exploration 11.1 Playing a List of Notes**

Create a list of three to six notes. Combine the **Play** and the **Item** code blocks with the list to play one of the notes in the list.

**Topic 11.2 Using a Loop to Play a List of Notes**

If the list is placed in a loop, a single **Play** code block (shown in purple in the illustration below) can be used to play an entire list of notes. (Note: The **Play** code block is found in the *Sound* section of the **Code Block Palette**.)

The **For** loop has an *index* represented by the letter “i”. The value of the index is automatically increased by 1 during each iteration of the **For** loop.

```
for i = 1 to 3

Play item i of list A4 B4 E4 for Quarter Note Length and Wait
```

The index “i” (depicted in orange in the illustration) has a value of “1” during the first iteration of the loop. This value is increased to “2” during the second iteration of the loop and to “3” during the third (and final) iteration of the loop.
The use of a list in combination with a loop makes it possible to use a single Play code block in place of multiple Play code blocks snapped together in this manner.

The use of a list to play a sequence of notes is more efficient than using a separate Play code block to play each note.

**Essential Knowledge**

**CS Principle 3.10 Lists**

**Application of Essential Knowledge**

In previous modules, a loop has been combined with a list to play sequences of chords and sequences of drum patterns. In this example, the same concept is employed to play lists of notes.

**Exploration 11.2 Using a Loop to Play a List of Notes**

Use a For loop to play the list of notes that you previously created.

**Topic 11.3 Storing a List of Notes in a Variable**

A short list of notes can be placed directly into the input slot of the Play code block. For a longer list of notes, it can be convenient to assign a name to a list of notes. This can be done with a variable. Create a variable named Note List and assign the list of notes to it.

(Note: Recall that in previous modules, a checkbox to the left of the variable name under the Command Palette was used to display a table with the items in the list that has been assigned to the variable.)

Once the list of notes has been assigned to the variable Note List, the oval block displaying the variable name can be dragged into the input slot of the Item code block, replacing the actual list. Creation of the variable Note List will make it easier to add additional items to the list of notes in the future.
The variable *Note List* can be used in place of the actual list of notes. When the list is dozens of notes or even hundreds of notes long, use of a variable to represent the list of notes makes the code more readable.

Use of meaningful variable names is crucial to effective programming. A meaningful variable name makes the code more readable.

**Essential Knowledge**

- CS Principle 3.1 Variables and Assignments
- CS Principle 3.10 Lists

**Application of Essential Knowledge**

Replacement of the list of notes with the variable name *Note List* makes the code easier to read and understand. The code block would become increasingly unwieldy as the length of the note list increased (to encompass an entire song, for example). Use of the variable name *Note List* addresses this potential issue.

**Exploration 11.3 Storing a List of Notes in a Variable**

Create a variable and assign the previously created list of notes to the variable. Then use the variable in combination with a loop to play the list of notes.

**Topic 11.4 Playing Lists of Varying Lengths**

The number of notes in *Note List* changes as new notes are added to create and play a tune. When the number of notes in the list changes, the For loop must also be revised. For example, if a fourth note is added to the list, the code block `For I = 1 to 3` must be replaced with `For I = 1 to 4`.

Use of the *Length of List* block in place of a specific number such as “3” or “4” ensures that the loop will continue to work properly even as the number of notes changes.
Essential Knowledge

CS Principle 3.10 Lists

Application of Essential Knowledge

The Length of List code block was previously used in a similar manner to play lists of chords of varying length.

Exploration 11.4 Playing Lists of Varying Lengths

Continue to add additional notes to Note List to develop a musical tune. Use the updated loop with the Length of List block to play the revised list of notes.

Topic 11.5 Playing Lists with Notes of Varying Durations

All of the notes in previous examples have been the same duration – a quarter note. However, a second variable, Note Lengths, can be created and used to specify the duration of each note. (This variable could have also been named Note Duration because that also would have been a meaningful name.)

At this point, two lists have been created – a list of notes and a parallel list of note durations.

The Item code block is used to access the note length in the Play code block in a manner similar to the way in which the Item block is used to access the notes in the list of notes.

Exploration 11.5 Playing Lists with Notes of Varying Durations

Create a parallel list of note lengths to be used in combination with the previously created list of notes. Then play the revised list of notes.
Topic 11.6 Creating a Tune Player

The previously developed code can now be used as the basis for a custom code block, a **Tune Player**.

The **Tune Player** code block plays the notes in *Note List* using the note lengths stored in the variable *Note Lengths*.

Creation of a custom code block, **Tune Player**, means that it is no longer necessary for the programmer to keep track of the specific details of the loop that is used to play the notes.

**Essential Knowledge**

CS Principle 3.13 Developing Procedures

**Exploration 11.6 Creating a Tune Player**

Use the custom **Tune Player** code block to play the previously created list of notes.

Topic 11.7 Adding Notes to the Note List

The **Add Thing to List** code block (found in the Variables section of the Command Palette) can be used to add notes and note lengths to the note lists.
Once these code blocks have been tested and verified to work as anticipated, they can be used to construct an **Add Note** custom code block.

![Add Note Code Block]

This custom block adds a note with a specified length to the **Note List** and **Note Lengths** lists respectively.

![Note Lists]

**Essential Knowledge**
- CS Principle 3.10 Lists
- CS Principle 3.13 Developing Procedures

**Application of Essential Knowledge**

The custom **Add Note** code block extends the flexibility of the tune layer by providing a convenient way to add notes (and corresponding durations) to the **Note List** and the **Note Lengths** lists respectively.

**Exploration 11.7 Creating a Tune Player**

Use the custom **Tune Player** code block to play the previously created list of notes.
Topic 11.8 Recording Notes Entered via the Computer Keyboard

The **When Key Pressed** block assigns an action to a key on the computer keyboard. In this instance, the action adds a note to the **Note List**. A piano keyboard template (shown in the illustration below) can be used to identify keys on the computer keyboard that correspond to musical notes. Note: Controlling data access in this way reduces the possibility of a typing error.

In this piano keyboard template, the number key “1” corresponds to the musical note “C” on the piano keyboard, the number key “2” corresponds to the musical note “C#” on the piano keyboard, the number key “3” corresponds to the musical note “D” on the computer keyboard, and so forth.

The assignment of musical notes to keys on the computer keyboard makes it possible to add new notes to the list by pressing keys on the computer keyboard. There are other possible layouts that could be created depending on the individual needs of each user. For example, the musical note “C” could be assigned to the letter “C” on the computer keyboard. The ability to customize code blocks gives users complete flexibility to customize code blocks in ways that reflect individual preferences.

**Exploration 11.8 Recording Notes Entered via the Computer Keyboard**

Use card stock to create a **Piano Key** template and use it as an overlay for your computer’s keyboard. Then create code in the manner described above that records musical notes entered on the computer keyboard and enters them into **Note List**.

Topic 11.9 Varying Octaves

An effective computational thinking strategy is to solve the simplest instance of a more complex problem as a “proof of concept.” In this instance, all of the notes are assigned to the fourth octave. Now that a basic method for recording notes has been developed as a proof of concept, the note recorder can be refined by adding the capability to vary the octave of recorded notes.

The octave of notes can be varied by creating a variable named **Octave**. The variable can initially be set to a default value of “4”.

```
set Octave to 4
```
The following code blocks enable the octave to be increased or decreased by pressing the up and down arrows on the computer keyboard.

```
when up arrow • key pressed
change Octave • by 1
```

```
when down arrow • key pressed
change Octave • by -1
```

With this refinement of the code, pressing a key selects the musical note. For example, pressing the “1” key on the computer keyboard, supplies the note “A” as an input to the **Add Note** procedure. The **Add Note** code block then uses the **Join** code block (found in the **Operators** section of the **Command Palette**) to combine the note name with the octave. Therefore, if the value of the variable **Octave** is “4”, pressing the “1” key on the computer keyboard will add the note “C4” to the note list.

A piano keyboard has seven octaves. A computer is not constrained by the physical layout of a piano keyboard. Therefore, the number of octaves can be set to the range desired by the user. The refinement of the code shown below resets the variable **Octave** to “1” when its value exceeds “7”, and resets the variable to “7” when its value is less than “1”. This constrains the numerical value of the variable **Octave** to the numbers one through seven.

```
when up arrow • key pressed
change Octave • by 1
if Octave > 7
set Octave • to 1
```

```
when down arrow • key pressed
change Octave • by -1
if Octave < 1
set Octave • to 7
```

**Essential Knowledge**

- CS Principle 3.1 Variables and Assignments
- CS Principle 3.3 Mathematical Expressions
- CS Principle 3.6 Conditionals
Application of Essential Knowledge

In the preceding example, the new code accesses a variable that is external to the code block. As programs become more complex, it can be necessary to document these kinds of details as the program grows.

Exploration 11.9 Shifting Octaves

Recreate your previously created list of notes, but shift all of the notes up or down an octave. Use the custom Tune Player code block to play the new list of notes and listen for the differences between the old and the new lists.

Topic 11.10 Varying the Duration of Notes

A method similar to that used to vary the octave of notes can also be used to vary the duration of notes. Until now, all of the notes have been assigned a fixed duration of one quarter note.

That limitation can be addressed by creating a new variable, Note Duration. The Note Duration variable can be adjusted using the Set Variable code block in the same manner as previous examples.

Creating a variable for the note duration enables this parameter to be adjusted so that the notes are not always the same length.

Once the Note Duration variable has been created and incorporated into the key press code as an input to note length, computer keys can be used to control this variable in a manner similar to the way in which the up and down arrow keys were used to control the value of the Octave variable.

A list of the note lengths is accessed to select the duration of notes. A variable named Duration Index will be used to determine which item in this list is selected.

The Duration Index is initially set to “1” (pointing to the first item in the list, Whole note).
The left and right arrow keys will be used to adjust the duration selected. When the right arrow key is pressed, the *Duration Index* is used to select the corresponding item in the list of note durations.

\[
\text{when \ right \ arrow \ key \ pressed}
\]

\[
\text{set \ Note \ Duration \ to \ item \ Duration \ Index \ of \ list \ Whole, \ Half, \ Quarter, \ Eighth, \ Sixteenth}\\
\text{change \ Duration \ Index \ by \ 1}
\]

The *Duration Index* is advanced each time the arrow key is pressed. Therefore, the next time the arrow key is pressed, the *Note Duration* will be set to the next item in the list (a *Half* note, in this instance).

Eventually the end of the list will be reached. When this occurs, the *Duration Index* is reset to “1” so that it points to the beginning of the list again.

\[
\text{when \ right \ arrow \ key \ pressed}
\]

\[
\text{set \ Note \ Duration \ to \ item \ Duration \ Index \ of \ list \ Whole, \ Half, \ Quarter, \ Eighth, \ Sixteenth}\\
\text{change \ Duration \ Index \ by \ 1}\\
\text{if \ Duration \ Index \ > \ 5}\\
\text{set \ Duration \ Index \ to \ 1}
\]

This enables the duration of each note to be adjusted by pressing the right arrow key. A parallel left arrow procedure enables the left arrow to decrease the *Duration Index*. Consequently, the *Note Duration* can be quickly adjusted by moving back and forth through the list of note durations using the arrow keys.
The Case of the Equal ("=") Key

The `When __ Key Pressed` code block includes the minus ("-”) and plus ("+") keys in its drop-down menu but not the equal ("=") key.

![Keyboard Image]

There are several possible work-arounds. Since the equal ("=") key is not available in the drop-down menu of the `When __ Key Pressed` code block, the plus ("+") key (directly above the equal key) could be substituted instead.

![Code Block Image 1]

One drawback of this work-around is that the Shift key must be held down to access the plus ("+") key. Some users may not recognize or remember that the shift must be held down to access the plus ("+") key. It is desirable to design a user interface that does not require users to make these kinds of adjustments.

While the code is slightly more complex, the following method can be used to access the equal ("=") key directly. The `Key Pressed?` code block (found under the Sensing section of the Code Block Palette) can be placed in a `When <   >` code block. The orange Identity code block (shown as an oval within the blue `Key Pressed?` Block) is then used to insert an equal ("=") sign into the `Key __ Pressed` code block. (Note: The orange Identity code block can be obtained from the Big Num library of the Snap! code library extensions.)

![Code Block Image 2]

The resulting code will insert the note “B” repeatedly as long as the equal ("=") key is held down. To avoid instances of multiple notes being unintentionally added to the `Note List`, a `Wait` code block can be added at the end. If a `Wait` code block is included in the code, only a single note will be added to the note list as long as the user does not hold down the key for more than one second. (A longer wait can be added if necessary.)
Essential Knowledge

CS Principle 3.1 Variables and Assignments

Application of Essential Knowledge

The example in the preceding section continues to extend the way in which variables are used and applied. In this instance, two additional variables – Note Duration and Duration Index – are used to enable the user to vary the durations of notes recorded.

Exploration 11.10 Changing Note Lengths

Use the blocks in this chapter to create a Note List and a Note Lengths list. Use at least three different note lengths. Use the custom Tune Player code block to play the new list of notes and listen for the differences between the note lengths.

Topic 11.11 Incorporating a Rest

Music often includes times during which no notes are played. The musical term for this is a rest. The Rest code block incorporates a delay of a specified duration when no notes are played.

The Add Rest code block is similar to the Add Note code block except that it adds a rest instead of a note to the note list. A corresponding duration (Half, Quarter, Eighth, etc.) is added to the Note Lengths list.

The Add Rest code block could be assigned to any key on the computer keyboard. In this instance, it has been assigned to the “r” (for rest) key.
Pressing the “r” key on the computer keyboard will then insert the word “Rest” into the note list.

After the capability for inserting a rest into the Note List is implemented, the Tune Player needs to be told how to process a rest when it encounters one.

This is accomplished through incorporation of an if ... Else code block. This code block says to Rest for a specified amount of time if the word “Rest” is encountered in the Note List.

Otherwise, the specified note in the Note List is played in the same fashion as before.

**Essential Knowledge**

- CS Principle 3.1 Variables and Assignments
- CS Principle 3.6 Conditionals
- CS Principle 3.10 Lists
- CS Principle 3.13 Developing Procedures

**Exploration 11.11 Adding Rests**

Update the Tune Player so that rests can be incorporated into the lists of notes.
**Topic 11.12 Customizing the Player**

*TuneScope* enables music tools to be customized. The methods used to create the *Tune Recorder / Player* can also be used to customize it. For example, the space bar could be used to start the *Tune Player*.

A code block to reset the note list might also be useful.

This code block sets both the *Note List* and the *Note Lengths* list to empty lists.

To correct errors, the ability to delete a note from the *Note List* and the *Note Lengths* list without completely resetting them would also be useful. This provides the capability to edit the list of notes by removing notes and trying different ones. In this example, when the “d” (for “Delete”) key is pressed on the computer keyboard, the last note in the list is deleted from *Note List*. The last note length in the *Note Lengths* list is also deleted.

When a note is added to the note list, it could be helpful if the *Tune Recorder* also played that note. Adding a **Play** code block at the end of the **Add Note** procedure provides instant feedback when a note is added.
Other enhancements can be incorporated to customize the code blocks and interface to match your individual preferences.

**Essential Knowledge**

- CS Principle 3.10 Lists
- CS Principle 3.13 Developing Procedure

**Exploration 11.11** Customize the Player

Create any of the blocks in this chapter and add them to your Tune Player. Try to think of other features that might make the Tune Player easier to use and how you might create them.

**Further Explorations**

1. Use Tune Recorder to record a short series of musical notes (i.e., a half dozen to a dozen notes). You can either (1) compose your own sequence of notes through experimentation and exploration or (2) copy an existing series of notes from an existing tune. If you choose the second option, be sure to credit the source.

2. Write the lyrics for a two-line jingle to accompany the recorded musical sequence.

3. Use the Tune Player to play the recorded sequence of notes.

4. Sing or chant the lyrics of the jingle that you created while playing the sequence of notes.

5. Create a video of your performance.

**Extensions**

- What are the challenges of using the Tune Player / Recorder to create music?
- Are there additional features that you think would be useful?
- Do you feel that you know enough to create new features using TuneScope code blocks?
A karaoke machine plays the music that accompanies a song with vocal accompaniment removed. This enables a singer to accompany the music. In some cases, a karaoke machine may also display the lyrics accompanying the song. An extension of the TuneScope Player/Recorder created in the previous module can be used to build a machine that displays lyrics. In order to reduce the complexity of code required to build the application, the initial version of the TuneScope application only plays a sequence of notes. Once the simplified lyric display machine is constructed, the principles learned will be used to create a more complete version that also can play chords.

**Topic 12.1 Transposing Lists**

Many karaoke machines include an option to display lyrics for singers who are not familiar with the words. The data structure for playing notes combined with words might look like the table below. This table combines the *Note* and *Note Length* lists created for the TuneScope Recorder Player and adds a third list that contains the lyrics for the song.

The opening guitar notes of the version of Crossroads played by the blues musician Robert Johnson look like this.
Any two lists of this kind can be combined into a single table similar to the one shown in the table view of *Crossroads* above by placing them into another list. This creates a list of lists. However, the list created in this way displays the notes in horizontal rows by default rather than in columns.

The **Transpose** code block (found in the *APL Primitives* library) can be used to transpose the rows and columns of the table formed by combining the two lists. This restores the two lists to their original column orientation.

Each of these lists could be described as a track or channel. A tape recorder also has analog channels consisting of iron oxide particles on an acetate tape. The channels were mixed and combined using electronic mixing boards, leading to use of the term *track* to describe channels with different instruments recorded on them.

The tracks used to construct this application consist of lists of characters representing notes and words. The use of the word track in context of a computing application should not be confused with use of the term to describe an acoustic channel in the music recording industry.

The two combined tracks (i.e., the *Notes* and *Note Lengths* lists) are assigned to a variable named *Music Tracks*. Later these tracks will be expanded to include chords as well as sequences of notes. Assignment to a variable will make it possible to manipulate and play back the combined tracks.
Essential Knowledge

- CS Principle 3.1 Variables and Assignments
- CS Principle 3.3 Mathematical Expressions
- CS Principle 3.10 Lists

Application of Essential Knowledge

Combining lists into a table reduces the possibility of human error. When separate Note List and Note Length lists are kept, there is a possibility of adding a note to Note List without adding a corresponding note length to the Note Length list. Once this occurs, the two lists will be out of synch from that point on. Combining the two separate lists reduces this type of error, and makes it easier to see which note length is associated with a corresponding note.

Exploration 12.1 Transposing a List

Use the List command to combine a list of notes and a separate list of note lengths into a single table. Then use the Transpose block to transpose the lists from rows to columns.

Topic 12.2 Modifying the Tune Player

Before the combined tracks can be played, the Tune Player will need to be modified. The first item in the combined lists now actually consists of a list of two items: the note (G5) and the note length (Sixteeth). In other words, the table consists of a list of lists.

Since there are two lists, two Item blocks will be needed to retrieve the information. The two items in each sublist – the note and note length – can be accessed in the following manner.

In other words, the first item of the two-item list is the note, and the second item of the two item list is the note length.

The updated Tune Player uses Item 1 of Item 1 of the variable named Music Tracks to access the note and Item 2 of Item 1 of the variable named Music Tracks to access the note length. Once a third track containing lyrics is added to the table, Item 3 of Item 1 of the variable named Music Tracks will be used to access the words of the lyrics.
Combining the *Note* list and the *Note Length* list will not only facilitate later addition of a lyric track, it also makes it easier to save tunes that have been composed with the *Tune Recorder* since they can be saved as a single file rather than as two separate files.

**Essential Knowledge**

- CS Principle 3.1 Variables and Assignments
- CS Principle 3.2 Data Abstraction
- CS Principle 3.6 Conditionals
- CS Principle 3.8 Iteration
- CS Principle 3.10 Lists
- CS Principle 3.13 Developing Procedures

**Application of Essential Knowledge**

The ability to access data in a table is one is a foundational concept in computer science. Almost all programs of any complexity store data in this manner. The term *array* is used in computer science to describe this type of data table. A single list is a *one-dimensional array*. A table with rows and columns is a *two-dimensional array*.

**Exploration 12.2 Modifying the Tune Player**

Modify your *Tune Player* in the manner described above and use it to play a tune that has its notes and note durations stored in a table.
Topic 12.3 Exporting and Importing Data

Right-clicking the table containing the Music Tracks data provides access to a dialog box that offers the option of exporting the data.

If this option is selected, the music data can be saved in a file on the local computer. The file is saved in a “Comma Separated Values” (csv) format. This format separates the data with a comma between each item – hence the name, comma separated values.

Most spreadsheet programs, including Microsoft Excel and Google Spreadsheets, can read data saved in a “.csv” file format. When the music table is opened in a spreadsheet, it looks like this:
The lyrics that accompany the music notes can then be typed into a third column of the spreadsheet.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>G5</td>
<td>Sixteenth</td>
<td>I</td>
</tr>
<tr>
<td>2</td>
<td>A#5</td>
<td>Eighth</td>
<td>went</td>
</tr>
<tr>
<td>3</td>
<td>A#5</td>
<td>Quarter</td>
<td>to the</td>
</tr>
<tr>
<td>4</td>
<td>A5</td>
<td>Eighth</td>
<td>cross</td>
</tr>
<tr>
<td>5</td>
<td>G5</td>
<td>Eighth</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>E5</td>
<td>Eighth</td>
<td>road,</td>
</tr>
</tbody>
</table>

Once the updated version of the file with lyrics has been saved, it can be imported back into *TuneScope*. Before importing the updated table, first delete the variable *Music Tracks* from *TuneScope* to avoid creating multiple copies of the table.

The re-imported *Music Tracks* table will now look like this, with a third column containing lyrics.

**Exploration 12.3 Adding Lyrics**

Export your table of notes and note lengths. Then add lyrics using the format described above. Save the updated file and import it back into *TuneScope*. 

147
Topic 12.4 Displaying Lyrics

Addition of a third column to the table containing lyrics makes it possible to update the Lyric Display machine to display lyrics as the music is played. The TuneScope Text Block code block (found under the Looks section of the Command Palette) can be used to display text. The Text Block code block consists of an ID, a title, and the body of the text. In the illustration below, the ID is “1”, the title is “Sample Text,” and the body of the text is “This is some text.”.

The song *You Ain’t Nothing But a Hound Dog* is an inflection point in rock-and-roll music. The song was composed by Lieber and Stoller and first sung by the Black artists Big Mama Thornton. Elvis Presley later sung a revised version of the song. *Hound Dog* was Elvis Pressley’s first song that sold more than a million copies. The table below displays the lyrics for the song, *Hound Dog*. The lyrics are in the third column of the table.
As demonstrated previously, the code block **Item 1** displays the first row of the table.

**Item 3** of **Item 1** displays the third column of the first row, which contains the lyrics.

If the reporter block above is combined with a text block, the first line of the lyrics is displayed.

To display all of the lyrics in the table, a variable named *Phrase* can be used to add each successive line of lyrics to the previous lyrics.

This loop shown above displays all of the items in the third column in the table, including (1) the lyrics, (2) semicolons used as place holders in cells that do not contain lyrics, and (3) the phrase “Clear Text” which is used to indicate that the text should be cleared from the display before beginning the next verse.
Essential Knowledge

- CS Principle 3.1 Variables and Assignments
- CS Principle 3.2 Data Abstraction
- CS Principle 3.8 Iteration
- CS Principle 3.10 Lists
- CS Principle 3.4 Strings

In computer science, a string is a type of data consisting of a string of letters rather than numbers or some other type of data. The act of joining two strings together is known as string concatenation.

Application of Essential Knowledge

The format used for encoding strings of letters in digital computers can be traced back to the nineteenth century. Samuel Morse invented Morse Code to provide a means of transmitting letters over the newly-invented telegraph. Later the telegraph key was replaced with a computer keyboard that could transmit a letter with the press of a single key. An updated version of the Morse Code was called the American Standard Code for Information Interchange (ASCII). Today letters are stored in digital computers in the same format. Instead of recording dots and dashes on a paper tape, the information is stored as 1's and 0's in the digital memory of the computer.

Exploration 12.4 Displaying the Lyrics

Use the Tune Scope Text Block code block to display the lyrics that accompany your song as the music plays.

Topic 12.5 Formatting the Lyric Display

The initial prototype of the lyric display program provides a proof-of-concept that illustrates one approach to displaying lyrics while music plays. In its initial form, all of the formatting characters such as semicolons used as placeholders are also displayed. The semicolons can be skipped by only proceeding when the item in the third cell of the row is not a semicolon.
The updated script incorporating the *If* statement shown above would then look like this.

The updated script now prints all of the lyrics but omits the semicolons used as place holders in cells that do not contain lyrics. However, the phrase “Clear Text” is still mixed in among the lyrics.

The following *If* statement resets the contents of the variable *Phrase* when the words “Clear Text” are encountered. This has the effect of clearing the text displayed in the *Text Block* when *Clear Text* is encountered.
The updated script incorporating the *If* statement shown above would then look like this.

The completed script shown above that displays the lyrics does the following:

1. It clears the text when *Clear Text* is encountered.
2. It does nothing when a semicolon is encountered, skipping ahead to the next line of the program.
3. It displays each line of lyrics in the *Text Block*.

The completed script appears to be complex when viewed as a whole. However, if the script is broken down into smaller parts in the manner described above, each of these three steps can be developed and tested before it is combined with the other parts of the script. An iterative approach to program development is an important element of computational thinking and problem solving.

The script that displays the lyrics makes it possible to control the specific portion of the lyrics that are displayed at any one time in the text box while the music tracks are playing.
Essential Knowledge

- CS Principle 3.5 Boolean Expressions
- CS Principle 3.6 Conditionals
- CS Principle 3.7 Nested Conditionals

Application of Essential Knowledge

One of the most important problem-solving methods in program development consists of the principle of breaking large complex problems into a series of smaller, more easily solved problems. In this instance, the basic strategy for displaying lyrics was tested and verified before the refinement of implementing display formatting was addressed.

Exploration 12.5 Formatting the Lyric Display

Develop code to format the lyrics in a manner that you find esthetically satisfying. There are a number of possible formatting choices. All of the lyrics for each verse could be displayed at once. Alternatively, each word could be individually displayed as each measure of the music is played. The formatting method used will depend on your judgment about the best format.

Topic 12.6 Displaying Lyrics while Music Plays

The completed Karaoke Machine procedure consists of two main loops: (1) the loop that displays the text of the lyrics and (2) the loop that plays the notes (adapted from the the previous Tune Player procedure). By developing and testing each component individually, a relatively complex procedure can be created.
The lyric display machine developed in this manner is an extension of the *TuneScope Player* / *Recorder*. It plays a sequence of notes, just as the *TuneScope Player* did, but also displays the accompanying text of the lyrics. A commercial karaoke machine typically plays musical chords as well as individual notes. The *Karaoke Player* procedure can be adapted to play chords as well as notes.

The **Play** code block can be replaced with the **Play Chord** code block to make this modification. This assumes that the list in the first column of the table consists of chords rather than individual notes.

Once the basic data structure is established, replacement of notes with chords can be accomplished with a direct substitution. A further refinement could extend the application to simultaneously play both notes and chords.

**Essential Knowledge**

- CS Principle 3.1 Variables and Assignments
- CS Principle 3.2 Data Abstraction
- CS Principle 3.5 Boolean Expressions
- CS Principle 3.6 Conditionals
- CS Principle 3.7 Nested Conditionals
- CS Principle 3.8 Iteration
- CS Principle 3.10 Lists
- CS Principle 3.13 Developing Procedures

**Exploration 12.6 Displaying Lyrics while Music Plays**

Combine the lyrics and the music for your song so that they both play at the same time. Run the program while you sing along as the lyrics are displayed. Think about ways you might revise and enhance the program.
C. Hypermedia Tools
13. Acoustic Tools

Glen Bull, Joe Watts, and Joe Garofalo

In previous modules, waveforms have been displayed in the TuneScope display in the top half of the TuneScope screen. The process of graphing the waveform has been automated to facilitate the initial exploration of waveforms.

The foundation for exploring the underpinnings of digitized sound has now been established. The process of recording a sound was described in the previous module.

This process converts the back-and-forth movements of a computer’s microphone into an electrical signal. The amplitude of the varying electrical signal is repeatedly measured and converted into a series of digital numbers. This process is known as analog-to-digital (A/D) conversion. The process indirectly measures minute movements of the microphone, and converts that movement into a series of digital numbers.

Even a short segment of digitized sound may consist of thousands of sound samples. If the digitization rate is 48,000 samples per second (for example), a single second of speech will result in a list of 48,000 numbers. These numbers represent the voltage levels generated by the back-and-forth movement of the diaphragm of the microphone used to capture the digitized sound.

In article “The Magic Number Seven, Plus or Minus Two” the psychologist George Miller reports that the average individual can only hold about seven numbers in short term memory at one time.
(This is one of the most cited scientific papers of all time.) The limits may vary by one or two, but under no circumstance is it possible for a person to process or understand a list of 48,000 numbers.

This list of numbers representing a second of digitized speech, however, is more comprehensible if it is presented as a graphical representation rather than in number form. The coordinate system for the default Snap! stage spans the range from -240 to +240 units on the horizontal (X) axis, and from -180 to +180 units on the vertical (Y) axis.

These coordinates provide the framework for graphing any type of data in Snap!, including a segment of sound.

**Topic 13.1 Graphing Sound Samples**

Prior to graphing the waveform, the turtle is positioned on the left side of the screen. The screen is cleared, and the turtle’s pen is lowered.

![Graphing Sound Samples Diagram]

Depending on the computer, the digitized sound samples may consist of numbers such as 0.27 and 0.42. To be clearly seen, the positive peaks of the waveform should ideally fall between 20 and 100 on the vertical axis of the stage. The sound samples can be multiplied by an appropriate number to increase the scale if necessary. In the example below, the samples of the sound utterance “One Two Three” have been multiplied by 100 and assigned to a variable named `Sound`.

![Graphing Sound Samples Code]
Once these setup and scaling activities are completed, a loop can be used to plot the sound samples. The utterance is slightly more than 2 seconds long. At a sample rate of 48,000 samples per second, more than 96,000 samples were collected.

Dividing 48 into 48,000 yields the result of 1000. Consequently, if every 48\textsuperscript{th} point is plotted, each point will represent a one millisecond time increment (since there are 1000 milliseconds in a second).

The default width of the stage is 480 steps. If each step in the graph represents a one millisecond time increment, it is still only possible to graph approximately a half-second of the utterance (0.48 seconds).

This time scale can be adjusted by calibrating the number of sound samples that are graphed. If every 96\textsuperscript{th} sound sample is graphed, each point will represent two milliseconds, and it becomes possible to graph nearly a full second of sound on the stage. If every 192\textsuperscript{nd} sound sample is graphed, each point will represent four milliseconds, and it becomes possible to graph nearly two seconds of sound on the stage.

Since the recording of the words “one, two, three” is approximately two seconds long, graphing every 192\textsuperscript{nd} point results in a time scale that makes it possible to display the entire utterance on the stage. This can be accomplished with a For loop that graphs every 192\textsuperscript{nd} sound sample in the utterance. The default increment for a For loop is 1; in this instance, the increment has been increased to every 192\textsuperscript{nd} point. After each point is plotted, the turtle moves over one step. By this means, the amplitude of each sound sample is graphed.

The graph shown in the illustration below results. The words “one”, “two”, and “three” can be clearly seen. The up and down variations in amplitude correspond to minute variations in the back-and-forth movements of the diaphragm of the microphone that captured the sound.
The graph can be displayed much faster if the procedure is placed in a **Warp** block. The **Warp** code block suspends all background operations until the plotting loop is completed.

The script to graph the sound has two components: (1) a graph setup procedure that positions the turtle and clears the screen, and (2) a loop that moves through the table of sound samples and graphs these values on the stage.

These two components can be used to build the code blocks **Graph Setup: Position the Turtle** and **Graph the Sound**. These two blocks, in turn, can be used to create the custom code block, **Plot Sound Samples**, that plots the values of the sound samples to create a graph of the sound wave.
The **Plot Sound Samples** code block accepts the samples of a recorded sound as an input and plots the results.

Displaying the sound samples graphically makes it possible to gain a better understanding of the characteristics of the waveform.

**Essential Knowledge**
- CS Principle 3.1 Variables and Assignments
- CS Principle 3.3 Mathematical Expressions
- CS Principle 3.8 Iteration
- CS Principle 3.10 Lists
- CS Principle 3.13 Developing Procedures

**Application of Essential Knowledge**

The list of thousands of sound samples is impossible for a human to interpret. When the same sound samples for the utterance “one two three” is plotted, each of the individual words is readily discerned. If the same list of numbers is sent to a speaker cone, the back-and-forth movements of the speaker cone are heard as a spoken phrase. Thus, the same list of numbers serves as the basis for both a graph and a reproduced utterance.

**Exploration 13.1 Graphing Sound Samples**

Use the **Plot Sound Samples** code block to graph different types of recorded sounds: speech, music, noise, etc. How does the display of the different types of sounds differ? Change the time scale of the graph displayed on the stage.

**Topic 13.2 Time Scale**

A sense of the time scale is often useful. For example, if 20 cycles of a waveform are displayed, determination of the frequency of the waveform requires calculation of the duration of the sound segment, which makes knowledge of the time scale a prerequisite for obtaining this information. Other characteristics of sound such as onset of the sound (the rate at which the amplitude of the waveform increases at the beginning of an utterance) require a known time scale.

In the previous section, every 192\textsuperscript{nd} sound sample was graphed. Since 48,000 samples per second were acquired when the sound was digitized, each sound sample plotted represents a 4-millisecond time increment. Calculation of the timeline involves two steps:
1. First, the absolute position of the turtle on the X-axis (i.e., the horizontal axis) must be determined. The \textbf{X-Position} code block can be used to obtain this value. Since the coordinates of the default stage range from -240 on the left to +240 on the right, the position of the turtle ranging from 1 to 480 can be determined by adding 240 to the turtle’s current location. The result of this calculation can be assigned to a variable named \textit{Time (seconds)}.

\begin{verbatim}
forever
set Time (seconds) to x position + 240
\end{verbatim}

2. Since each sound sample represents a time increment of 4 milliseconds (when every 192\textsuperscript{nd} sound sample is graphed, given a sampling rate of 48,000 samples per second), this result must be multiplied by four to obtain the location in milliseconds. For simplicity, this result is rounded to the nearest millisecond.

\begin{verbatim}
forever
set Time (seconds) to round x position + 240 \times 4
\end{verbatim}

To obtain time in seconds, this result can be divided by 1000 (since there are 1000 milliseconds in a second). In subsequent calculations below, results will be displayed in milliseconds.

When this loop is executed, positioning the turtle (recast in the role of a time cursor) slightly to the right of the center of the stage yields a value of 1000 milliseconds (i.e., one second). Since the duration of the utterance “One, Two, Three” is about two seconds in length, this result confirms that the calculation of the time scale is accurate.

\begin{verbatim}
set Time (milliseconds) to 1000
\end{verbatim}

This code can be encapsulated in a custom code block that uses the turtle as a cursor to indicate the time in milliseconds based on its position on the horizontal axis.
The ability to use the turtle as a marker that indicates the time scale is useful in a number of different analyses of the acoustic waveform.

**Essential Knowledge**

- CS Principle 3.1 Variables and Assignments
- CS Principle 3.3 Mathematical Expressions
- CS Principle 3.13 Developing Procedures

**Application of Essential Knowledge**

The graph of an acoustic waveform represents the amplitude of the sound sample (i.e., the value of the number representing each sample) over time. Therefore, a knowledge of the time base (i.e., the time increment represented by each sample) is crucial to interpretation of the graph.

**Exploration 13.2 Time Scale**

Use the turtle as a time marker to determine the start of the graph of the waveform of the word “One.” Then determine the time in milliseconds at which the word ends. How long is the word “One”? How does this duration compare with the length of the words “Two” and “Three”?

**Topic 13.3 Sound Segments**

There are times when it may be useful to isolate a portion of an utterance or a specific sound segment for analysis. The **Numbers from __ to __** code block can be used to isolate a portion of a list. For example, if only the workdays in a list of the seven days of the week are desired:

```
list Sunday Monday Tuesday Wednesday Thursday Friday Saturday Sunday

Item numbers from 2 to 6 of list: Sunday Monday Tuesday Wednesday Thursday Friday Saturday Sunday
```

The list of workdays can be obtained by requesting Items 2 through 6 (Mon through Fri) in the list:

```
Item numbers from 2 to 6 of list:
```

The same method can be used to obtain a portion of a sound recording. If there are 48,000 sound samples in a second (for sound digitized at that rate), then Items 24,000 to 48,000 will yield the portion of the utterance from 0.5 seconds to 1.0 seconds.

In fact, plotting **Items 24,000 through 48,000** of the utterance “One, Two, Three” yields the word “One” just as we would expect (since this word begins about a half-second into the recording).
This capability can be incorporated into a custom **Sound Segment** code block.

![Sound Segment code block](image)

One benefit of creating a custom **Sound Segment** code block is that this code block can not only be used to graph a sound segment but also to play the sound segment to verify that the portion of the utterance obtained is the desired segment.

![Plot and play sound segments](image)

It may be more convenient to enter the start and end of the sound segment in milliseconds rather than in sound samples. This conversion can be accomplished by multiplying milliseconds by 48 (since 48 times 1000 milliseconds yields 48,000 sound samples). The addition of the label $Ms$ after the input slots to indicate the units of measurement employed improves clarity.

![Sound Segment code block with milliseconds](image)

The **Sound Segment** code block can now be used with the start and end of the desired sound segment indicated in milliseconds.

![Plot and play sound segments with milliseconds](image)

The time cursor developed in the previous section can be used to identify the beginning and end of a desired sound segment. Once these values are obtained, they can be used as inputs to the Sound Segment code block to isolate portions of the utterance.

**Essential Knowledge**

- CS Principle 3.3 Mathematical Expressions
- CS Principle 3.10 Lists
- CS Principle 3.13 Developing Procedures
Application of Essential Knowledge

The ability to isolate a specific segment of the sound is an essential tool for interpretation of the graph. Pointers to the beginning and end points of the segment in the list serve as an index for isolating a portion of the sound.

Exploration 13.3 Sound Segments

Use the Sound Segment code block to identify and isolate consonants and the vowels in the words “Two” and “Three”.

Topic 13.4 Amplification

When a digitized sound is played, the numbers in the list of sound samples control the displacement of the speaker cone. The larger the number, the greater the displacement. Greater displacement of the speaker cone, in turn, results in a sound that is perceived as louder.

Therefore, the loudness of the sound can be increased by increasing the amplitude of the sound samples. The multiplication operator is used to multiply the value of a number.

The multiplication operator can be used to multiply all of the numbers in a list.

For example, each item in the list of sound samples can be doubled by multiplying by two.
If the graph of the waveform results in the following plot,

then multiplying the values in the list of sound samples by two results in a graph that is double the amplitude of the previous one.

If the sound samples that have been increased in amplitude are played, the sound is perceived as louder.

This method can be used to create a custom **Amplify** code block to control the amplification of the sound.

If the amount of amplification is greater than one, the amplitude of the sound is increased.
If the amount of amplification is less than one, the amplitude of the sound is decreased.

Essential Knowledge

- CS Principle 3.3 Mathematical Expressions
- CS Principle 3.10 Lists
- CS Principle 3.13 Developing Procedures

Application of Essential Knowledge

The amplitude of the waveform corresponds to the attribute of the sound perceived as loudness. Increasing or decreasing the amplitude by multiplying or dividing all the numbers in the list provides a way of deterring the relationship between a physical attribute of the sound and the perceived loudness.

Exploration 13.4 Amplification

Use the Amplify code block to increase and decrease the amplitude of a digitized sound. Graph each result and play the resulting sound. How does the amplitude of the graph of the waveform correspond to the perceived loudness of the sound?

Topic 13.5 Adjusting Frequency

Previously, we adjusted the amplitude of a sound (perceived as loudness) by multiplying each number in the list of sound samples. Similarly, we can adjust the frequency (perceived as pitch) of the recording by varying the playback speed.

This can be accomplished by first setting a variable, Playback Speed, to the sampling rate. The default rate at which sound samples in TuneScope are recorded is 48,000 samples per second. Therefore, in most cases the sample rate will be 48,000 samples per second.

When the sound is played back at the same speed at which it was recorded, a normal pitch is heard. In other words, the recorded frequency and the frequency played back are the same.

However, increasing the playback speed increases the frequency played back in comparison with the frequency at which the sound was recorded.
Similarly, decreasing the playback speed decreases the frequency.

This method can be used as the basis for development of a custom code block to adjust the frequency of a sound.

This code block can then be used to adjust the frequency of a recorded sound when it is played back.

Essential Knowledge

- CS Principle 3.1 Variables and Assignments
- CS Principle 3.3 Mathematical Expressions
- CS Principle 3.10 Lists
- CS Principle 3.13 Developing Procedures

Application of Essential Knowledge

The concept known as sampling rate refers to the rate at which sound samples were recorded. The sampling rate determines the time base of the recorded sound. The same time base used to record the sound must be used in playback in order to reproduce the sound with fidelity. If the time base is increased or decreased in playback, the frequency of the sound will change, affecting the perceived pitch of the reproduced sound.

Exploration 13.5 Adjusting Frequency

Use the Change Frequency code block to increase and decrease the frequency of a digitized sound when it is played back. How does the frequency of the sound correspond to its perceived pitch?

Topic 13.6 Reversing a Sound

If the items in a list are displayed from the end of the list to the beginning, the numbers are reversed. In the example below, the numbers “4, 5, and 6” are reversed.
The **Length of Sound** code block can be used to identify the total number of sound samples in a list. In this example, there are a total of 97,920 sound samples in the list.

![Length of Sound Code Block]

By playing the list of sound samples from the **Length of Sound** (97,920) to Item 1 in the list, the sound is played in reverse.

![Play Sound & Reverse Code Blocks]

This method can be used to create a custom code block, **Reverse**, to reverse the sound.

![Reverse Code Block]

The **Reverse** code block reverses a sound, starting with the end of the list of sound samples and continuing to the first item in the list of sound samples.

![Play Sound Reverse Code Block]

**Essential Knowledge**
- CS Principle 3.10 Lists
- CS Principle 3.13 Developing Procedures

**Application of Essential Knowledge**

All of the examples described above involve *list processing* – i.e., processing a list in one manner or another. In this instance, the process involves reversing the order of the numbers in the list, which causes the reproduced sound to be played in reverse order.

**Exploration 13.6 Reversing a Sound**

Use the **Reverse** code block to play a list of digitized sound samples in reverse. Play several vowels in reverse to examine the effect that this has upon perception. Are vowels still recognizable when they are reversed? Then play several types of consonants in reverse. What effect does this have upon perception?
**Topic 13.7 Echo**

If a sound begins to play and then is played a second time with a slight delay, the result is perceived as an echo.

Musicians often use this method to create reverberation. A permanent list that combines one list of sound samples with a second, delayed list of samples can be created by placing a series of empty cells at the beginning of the second list. If the items from -10,000 to the **Length of the List** are displayed, the first 10,000 items in the list will be empty.

When the empty cells in the list of sound samples are played back, the sound will be silent during this portion of the list. This, in effect, delays the playback for that amount of time.

The list with the silence at the beginning can be combined with the original list by using the plus (“+”) operator.

When the combined lists created in this manner are played back, there is a slight delay before the second list of sound samples begins, creating the effect of an echo. Increasing or decreasing the number of empty cells in the second list controls the length of the delay.
This technique can be used to create a custom **Echo** code block to create an echo with a specified delay that controls the timing of when the second sound begins playing.

The custom **Echo** code block can be used to specify the amount of echo that is incorporated into the playback.

In a live performance setting, such as an auditorium, the amount of delay is related to the amount of time that it takes for the sound to travel to the walls and reflective surfaces of the auditorium and return. These reverberations affect the quality of musical performances. Listener’s perceptions of recorded performances are also affected by these factors.

**Essential Knowledge**

- CS Principle 3.3 Mathematical Expressions
- CS Principle 3.10 Lists
- CS Principle 3.13 Developing Procedures

**Application of Essential Knowledge**

Reproduction of a sound a second time, with a slight delay between the first and the second reproduction, simulates the effect of a sound wave bouncing off the walls and ceiling of a room to create the effect of an echo. In an actual room, the reverberation results from the sound wave reflecting from many different surfaces rather than just one.

**Exploration 13.7 Echo**

Use the **Echo** code block to explore the effect on different lists of digitized sound samples. Attempt to create the perception of a large auditorium by controlling the amount of echo employed in the playback.
Topic 13.8 Combining Sound Effects

The **Reverse**, **Amplify**, and **Echo** code blocks can be combined with one another.

For example, a reversed sound with echo added can be played by combining those two code blocks.

**Exploration 13.8** Combining Sound Effects. Explore the effect of combining the **Reverse**, **Amplify**, and **Echo** code blocks in various combinations.

The acoustic tools developed for manipulating and digitizing sound samples can perform operations on the list of sound samples to increase or decrease the loudness, increase or decrease the pitch, play the sound backwards, add an echo, or isolate specific words or sounds within a longer recording.

**Summary**

The sound effects created through custom code blocks such as **Amplify** and **Echo** are illustrative of many other types of operations that can be performed to create other sound effects. For example, a high-pass filter removes some of the lower frequencies. Conversely, a low-pass filter removes some of the higher frequencies. These methods are the basis for many popular effects now used to create and record music. In various combinations, these methods are also the basis for creating and manipulating synthesized speech.
The same transformations applied to sound (described in the preceding module) can also be applied to images. Each individual dot within a picture on a computer screen is known as a picture element, abbreviated as *pixel*.

### Topic 14.1 Sound and Color

The *Pixels of Costume* code block is used to display the values of the pixels. Each row in the illustration below represents one pixel within the image. The columns shown represent the red, the green, and blue components of pixels. A fourth component known as the alpha element controls transparency and opacity.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>52</td>
<td>224</td>
<td>204</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>52</td>
<td>224</td>
<td>204</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>52</td>
<td>224</td>
<td>204</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>52</td>
<td>224</td>
<td>204</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>52</td>
<td>224</td>
<td>204</td>
</tr>
</tbody>
</table>

The value of the pixels can be doubled by multiplying the value of the pixels by 2.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>104</td>
<td>448</td>
<td>408</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>104</td>
<td>448</td>
<td>408</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>104</td>
<td>448</td>
<td>408</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>104</td>
<td>448</td>
<td>408</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>104</td>
<td>448</td>
<td>408</td>
</tr>
</tbody>
</table>

Switching to a costume in which the pixels are doubled results in a new image that has a lighter shade than the previous image.

Multiplying each sample in a list of sound samples by two results in a louder sound. Multiplying each pixel image by two results in a brighter image.

If the *Red*, *Green*, and *Blue* columns are multiplied by zero, the color is removed from the image, leaving a black image.
The Alpha column (i.e., Column D) was multiplied by one, which has the effect of leaving each value unchanged.

<table>
<thead>
<tr>
<th>3306</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>204</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>204</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>204</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>204</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>204</td>
</tr>
</tbody>
</table>

**Essential Knowledge**

- CS Principle 3.3 Mathematical Expressions
- CS Principle 3.10 Lists

**Application of Essential Knowledge**

The principles involved in manipulating lists of sound samples described in the preceding module (Acoustic Tools) are equally applicable to tables of picture elements (pixels). The primary difference is that four columns of information are required to describe a pixel. Mathematical operations on a table (i.e., a two-dimensional array) are termed *matrix operations*. In the same way that mathematical operations on a list of sound samples can be used to alter the quality of the sound, mathematical operations on an array of pixels can alter the attributes of an image.

**Exploration 14.1 Sound and Color**

Create a sprite costume that is a rectangle composed of a single color. Use matrix operations to adjust the values of the pixels. How do these adjustments affect the sprite's appearance?

**Topic 14.2 Controlling the Color of an Image**

Once the color has been removed from the image, it provides a blank canvas that can be used to explore the effects of adding different amounts of color back into the image. For example, if 255 is added to the Red column (i.e., Column A), the image becomes red.

The numeric values of the Red, Green, and Blue columns after this operation are shown in the table below.

<table>
<thead>
<tr>
<th>3306</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>255</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>255</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>255</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>255</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Now repeat the operation by removing the color from the image again. Then add back 255 (the maximum amount) to the second (Green) column.

Was the result what you expected? Repeat the operation for the third (Blue) column.

The minimum and maximum pixel values range from 0 to 255: a range of 256 values. The color values are stored in an eight-bit word in the memory of the computer. Each bit can record two states: either 0 or 1. Since two possibilities for each of eight bits yields a total of 256 possibilities, that is the maximum range of values that can be recorded for each color. For that reason, this color range is referred to as eight-bit color. (Some systems use 16 bits or 24 bits to record color values, which yields a much broader range of color possibilities.)

The range of 256 possibilities could have been encoded as 1 to 256 rather than 0 to 255. (Both ranges have a total of 256 possibilities.) However, from the beginning of the computer age, computer scientists have always begun with 0 when counting.

It is possible to put a value greater than 255 in the input slot. However, values greater than 255 do not have an effect on the color displayed.

The fourth column, known as the alpha channel, controls the opacity of the image. A value of 255 or greater in this column will cause the image to be completely opaque.

**Essential Knowledge**
- **CS Principle 3.3** Mathematical Expressions
- **CS Principle 3.10** Lists

**Application of Essential Knowledge**
Adding zero to a column leaves the values of the numbers in the column unchanged. In a similar manner, multiplying by one also leaves the values unchanged. This method can be useful in leaving the values of one column in a matrix unchanged while adjusting the values of other columns.

**Exploration 14.2**
Use the matrix operations described above to change the color of the sprite’s costume.
**Topic 14.3 Removing Color from an Image**

To remove the color from the image and create an image that is completely opaque, a value of 255 can be added to the fourth column (i.e., the alpha column).

![Switch to costume](image)

This will result in an image that has zero’s in the first three columns and 255 (the maximum value) in the fourth column. The result is an image that is completely opaque.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

A custom code block can be used to remove the color from an image, leaving the image opaque. This transformation can be convenient when the silhouette of an image is needed (for example).

![Remove Color](image)

The illustration below combines the steps of removing color (through multiplication by zero) and then adding back in the desired value for a given column.

![Switch to costume](image)

If variables are created for each of the colors, sliders can be created to control the value of each color. A *watcher* that displays the value of each variable can be displayed on the stage by clicking the checkbox beside each variable name. The watcher can be converted to a slider control (as described in a previous module).
Essential Knowledge

- **CS Principle 3.3** Mathematical Expressions
- **CS Principle 3.10** Lists
- **CS Principle 3.13** Developing Procedures

Application of Essential Knowledge

The matrix operations described in this section combine the operations of addition and multiplication in a single code block. The power of matrix operations of this kind is that the operations are performed more efficiently than performing the same task with a loop in which the operation is performed on each element one number at a time during each iteration of the loop.

**Exploration 14.3** Removing Color from an Image

Write a program that performs the operations described above using a loop. Compare the efficiency of the operation performed with a loop with the efficiency of the equivalent function performed with operators in the manner described in the preceding section. How does the time required to perform the operation with a loop compare with the amount of time required to perform the equivalent operation in which all of the numbers of a column are multiplied in a single operation?

**Topic 14.4 Color Mixer**

Right-clicking the displayed variable yields a drop-down menu that includes options to create a slider and set its maximum and minimum values. In this case, the minimum value set would be “0” and the maximum value would be “255”.

Once the variables *Red*, *Green*, and *Blue* are used to control the values of each color (as shown in the illustration below), the sliders can be used to control the colors displayed in the image. The opacity is left unchanged.
Once the combined blocks are verified to work properly, they can be used to create a **Color Mixer** block.

If the **Color Mixer** block is placed in a **Forever** loop, sliders for Red, Green, and Blue can be used to mix the colors of the image.

Almost all paint and drawing programs provide a means of entering red, green, and blue values. Therefore, once a desired shade or color is identified, these values can be used in any paint program.

The reverse is also true. If a combination of red, green, and blue values is identified in a paint or drawing program, these combinations can be used to dynamically adjust values in a computer program.

**Essential Knowledge**

- **CS Principle 3.1** Variables and Assignments
- **CS Principle 3.3** Mathematical Expressions
- **CS Principle 3.10** Lists
- **CS Principle 3.13** Developing Procedures
Application of Essential Knowledge

Use of sliders to control the color mixer is more efficient than directly revising the code blocks. A well-designed user interface can encourage exploration that would be tedious to implement through direct revision of the code.

Exploration 14.4 Color Mixer

Create a color mixer in the manner described in the preceding section. Then experiment with constraining the range of numbers controlled by the sliders to restrict the mixer to a specific color palette.

Topic 14.5 Color Filters

A color filter can be created by adding or subtracting the values of the pixels in the Red, Green, and Blue channels. For example, in this instance, a value of 50 has been added to each pixel in the Red channel. The bouquet of flowers has a distinctly rosy hue.

Subtracting 50 from the value of the pixels in the Red channel results in an image that de-emphasizes red.

A red filter can be created by using this method to increase or decrease the value of the pixels in the Red channel.

The red filter makes it possible to explore the effect of increasing or decreasing the value of the pixels in the Red channel.
Construction of parallel green and blue filters can be used in creation of an **Adjust Red Green Blue** code block.

The **Adjust Red Green Blue** code block makes it possible to interactively explore the effect of increasing the value of the red pixels in an image while simultaneously decreasing the value of the green pixels (for example). The illustration below has a distinctly red tinge, while the green colors are muted and de-emphasized.

The fourth channel that controls the transparency / opacity of the image is known as the **Alpha** channel. The term *channel* refers to a specific portion of the light spectrum. Consequently, the red, green, and blue components are often referred to as channels, since red, green, and blue are components of the visible light spectrum. The Alpha channel was invented at George Lucas’ animation and imaging company, *Industrial Light and Magic*. It has now become a standard term for image processing. Since the Alpha component is another element of color in computer graphics, by extension it has also become known as a channel even though it is not related to the visible light spectrum in the same manner that the red, green, and blue channels are.
Essential Knowledge

- CS Principle 3.3 Mathematical Expressions
- CS Principle 3.10 Lists
- CS Principle 3.12 Calling Procedures
- CS Principle 3.13 Developing Procedures

Application of Essential Knowledge

The examples in this section illustrate the way in which a programmer can create a series of tools that serve such as the custom Red Filter, Green Filter, and Blue Filter code blocks and use these tools as the basis for creation of more complex applications. This is a more efficient approach to problem solving because it breaks a larger problem down into smaller parts. Creation of tools is also more efficient because in some cases the tools can be reused in other applications.

Exploration 14.5 Color Filter

Create a custom color filter. Use it to modify a photograph that you have taken.

Topic 14.6 Controlling Transparency

Subtracting the value of the pixels in the Alpha channel results in an image that is more transparent (i.e., less opaque).

Increasing the value of the pixels in the Alpha channel results in a gray box surrounding the image, as shown in the illustration below.
This is because the background pixels surrounding the image that were formerly transparent are now opaque. Consequently, a gray box surrounds the bouquet of flowers in the image below.

![Image with background pixels]

This limitation can be addressed by resetting the image to its original state, with zero values in the cells of the background pixels, and then subtracting to achieve the desired level of transparency.

![Code blocks for adjusting transparency]

If these code blocks are placed in a loop so that the value of the pixels in the Alpha channel are continually reset, a slider can be created that can be used to interactively adjust the transparency of the image.

![Slider for adjusting transparency]

A similar approach can be used to create a procedure to adjust the transparency level of the image.
The **Transparency** code block can be used to specify an image and apply a specified level of transparency.

The four channels that control the pixels of an image – *Red*, *Green*, *Blue*, and *Alpha* – are standard components of most graphics and image processing programs. The ability to control these attributes through a computer program provides capabilities that go beyond those of a static image processing program.

**Essential Knowledge**
- **CS Principle 3.3** Mathematical Expressions
- **CS Principle 3.10** Lists
- **CS Principle 3.13** Developing Procedures

**Application of Essential Knowledge**

The concept of the Alpha channel is often not well-understood by novices who first begin using commercial image manipulation programs such as Photoshop. By working directly with the underlying pixels, it becomes easier to understand the capabilities of this channel and the way in which it can be used to adjust an image.

**Exploration 14.6 Controlling Transparency**

Create your own color filter system that can be used to manipulate images in the style of Instagram filters. Create a custom filter and apply it to an image.

**Further Explorations**

Add color filter capabilities to the paint program developed in Module 5 (Designing a Paint Program).