BONUS APPLICATIONS

In this document you'll find bonus applications to go along with Chapters 2, 6, 7, and 9. We walk through detailed explanations of each application, and you can find the scripts for these applications in the *.zip* file you downloaded from *http://nostarch.com/learnscratch/*.

Chapter 2

Jump_NoCode.sb2

In this section, you'll create a game called *Survival Jump*, which highlights different ways to make sprites move. In this game, players have to use the arrow keys to make a sprite jump over various objects coming from the right side of the Stage. See Figure 1 for an overview of the sprites involved. To keep the game simple, we won't penalize players for touching the object. Instead, we'll just play a sound each time we have a collision.

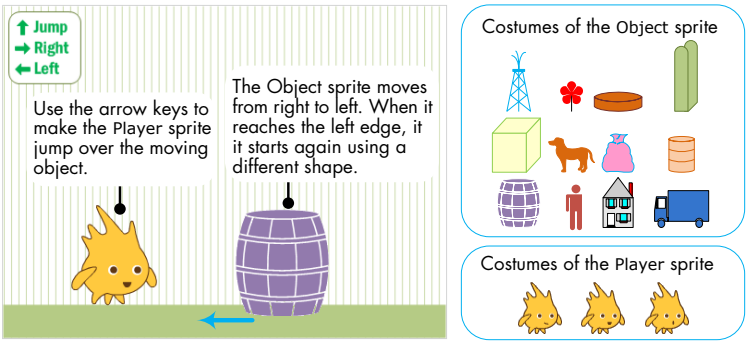


Figure 1: Leap over the moving objects to survive.

Open up *Jump_NoCode.sb2*, which contains all the sprites and the sounds needed for the game. The following discussion will walk you through creating the game, starting with the Player sprite scripts in Figure 2.

Figure 2: The scripts for the Player *sprite*

When the green flag is clicked, we first move the *Player* sprite to an appropriate location at the bottom-left part of the Stage ❶. We then start a **forever** loop ❷ to change the **Player** sprite's costume so the character looks like it's moving. You can find the **next costume** block in the *Looks* palette.

The other three scripts shown in Figure 2 respond to the arrow keys. When the right arrow is pressed, we move the sprite 100 steps to the right by changing its *x*-position ❸. To slow down the motion and make it feel more realistic, we break the motion into 10 hops of 10 steps each. The left arrow key does the same thing but in the opposite direction ❹.

The up arrow ❺ also moves the sprite, but now we want the sprite to jump up quickly and then drop at a slower rate. The total steps in the up and down directions are equal, at 200 steps.

Test what you've written so far by clicking the green flag. You should see the animation of the Player sprite, and you should be able to move the sprite around with the arrow keys. When you're done, press the stop icon and then add the script in Figure 3 to the Object sprite.

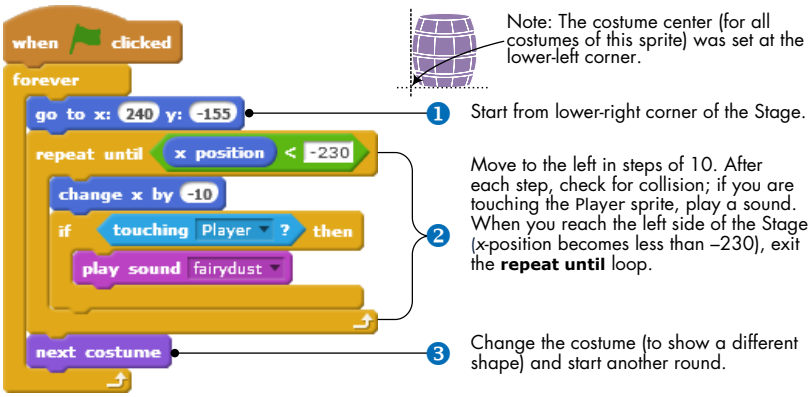


Figure 3: The script for the Object *sprite*

First, we move the Object sprite to the lower-right corner of the Stage ❶ and start moving it to the left ❷. After each move, we play a sound if the object hits the player. The object moves left until it reaches the edge of the Stage. Then, the script changes the object's costume ❸ and moves back to the start of the **forever** loop to repeat the whole sequence.

That was the last script, so click the green flag icon to test the game and start dodging objects. When you learn about variables and decision making in later chapters, you'll be able to make this game more complex. You could, for example, keep score, have the player lose energy with each collision, change the graphics effects of the moving object, clone the moving object—and more. Let your imagination have fun!

Chapter 6

This section explores two more games: a Guess My Number program and a Scratch version of Rock, Paper, Scissors.

Guess My Number

GuessMy Number.sb2

The number-guessing game selects a random integer between 1 and 100 and prompts the player to guess that number. The application then tells the player if the guess was higher or lower than the secret number by displaying "too high" or "too low," respectively. The player has six chances to guess the secret number. If the guess is correct, the player wins the game; otherwise, the player loses. The interface for this application is shown in Figure 4.

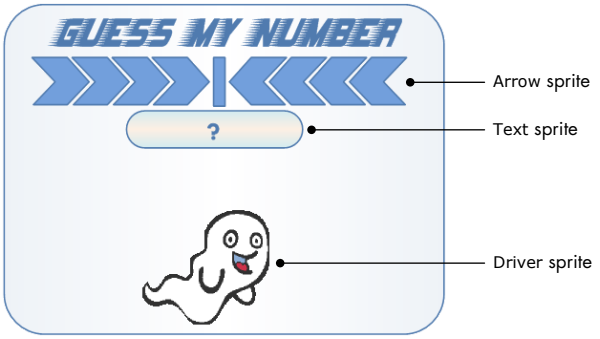


Figure 4: The user interface for the Guess My Number game

The application contains three sprites. The Driver sprite controls the flow of the application. The Arrow sprite provides animated feedback about how to adjust the next guess. The Text sprite shows the status of the game after each guess.

The game starts when the green flag is clicked. In response, the Driver sprite executes the script shown in Figure 5.

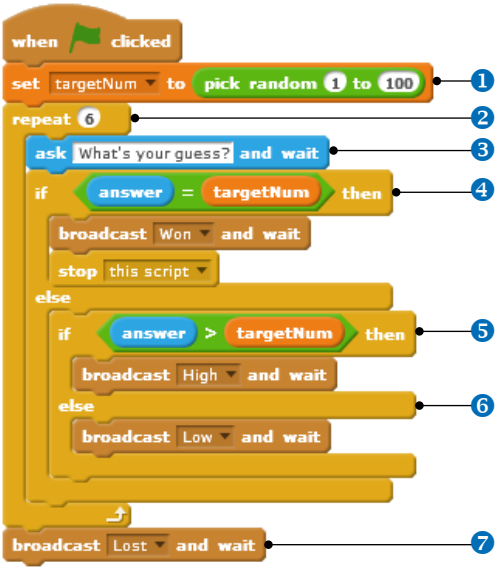


Figure 5: Script for the Driver *sprite*

The script starts by setting targetNum to a random integer between 1 and 100 ❶. It then starts a loop for getting and checking the player's guesses ❷. The loop repeats for (at most) six times. During each iteration of the loop, the player is prompted to enter a guess ❸. The value entered is saved in the built-in variable answer. The script then compares the player's guess with the target number. If the player guessed correctly ❹, the script broadcasts the Won message, which will be acted upon by the other two sprites as you'll see

below, and executes the **stop** block to end the script. If the guess was incorrect, the script checks whether the guess was higher than the target ❺. If yes, it broadcasts the High message to the other two sprites. Otherwise ❻, the player's guess is lower than the target number, and the script broadcasts the Low message. If the player fails to guess the number after six trials, the loop ends, and the script broadcasts the Lost message ❼.

The scripts for the Text sprite are shown in Figure 6, which also shows the five costumes used by this sprite. This sprite switches its costume in response to the different broadcast messages.

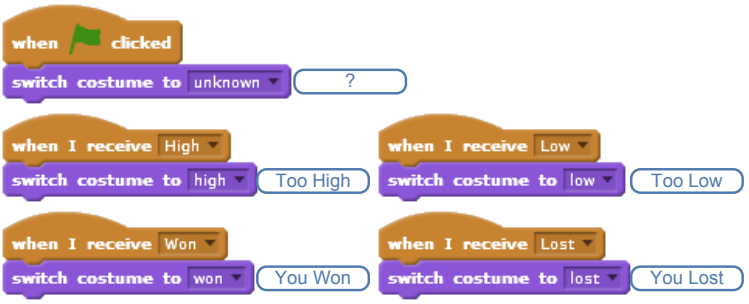


Figure 6: Scripts for the Text *sprite*

The scripts for the Arrow sprite are shown in Figure 7. This sprite has 10 costumes (also shown in the figure), which provide a simple animation effect. The scripts switch between these costumes to provide the desired animation.

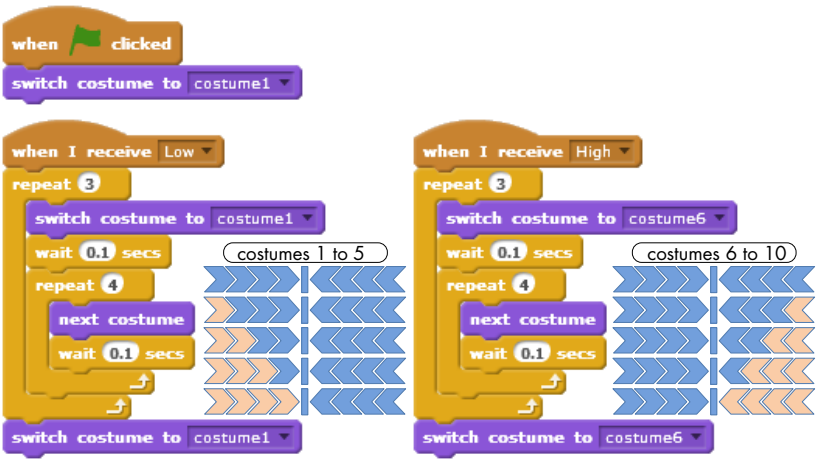


Figure 7: Scripts for the Arrow *sprite*

When the sprite switches between costumes 1 through 5 in fast succession, the arrow appears to move to the right, telling the user to increase the guess. Similarly, when the sprite switches between costumes 6 through 10, the arrow moves to the left, telling the user to lower the guess.

TRY IT OUT 1

GuessMyNumber.sb2

Load *GuessMyNumber.sb2* and play it to see how it works. Why do you think the program limits the number of guesses to six? Is it possible to guess the target number in six or fewer tries? What is the best strategy? (Hint: If your first guess is 50, then feedback such as "too low" will allow you to eliminate half of the possible numbers.) Modify the program to keep track of the player's number of guesses. Then have the Driver sprite display a message that uses this number at the end of the game.

Rock, Paper, Scissors

RockPaper.sb2

In this section, we'll create a Rock, Paper, Scissors game that lets you play against the computer. The interface for this game is shown in Figure 8. The hand symbols on the three buttons represent (from left to right) rock, paper, and scissors. The player selects an action by clicking on the corresponding button. The computer, on the other hand, selects an action randomly. The following rules determine the winner: paper beats (wraps) rock, rock beats (breaks) scissors, and scissors beat (cut) paper.

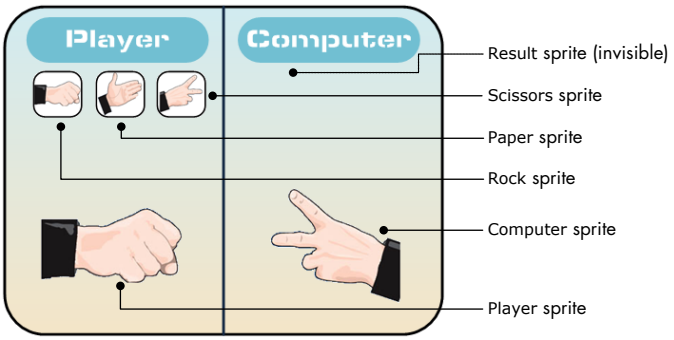


Figure 8: The user interface for the Rock, Paper, Scissors game

The application contains six sprites as shown in Figure 8. The Player sprite is responsible for showing the player's selection, whereas the Computer sprite is responsible for showing the computer's choice. The Rock, Paper, and Scissors sprites represent the three buttons, and the Result sprite (not shown in the figure) shows the result of the game.

The game starts when the player clicks on one of the three buttons to choose an action. The scripts associated with the possible actions are shown in Figure 9. Each button sets the choice1 variable to a value that identifies the player's choice and then broadcasts the Start message to tell the other sprites that the player has made a selection.

The Start message is handled by the Stage, which executes the script in Figure 10.

Script for the Rock sprite | Script for the Paper sprite | Script for the Scissors sprite

Figure 9: The event handlers for the three action buttons

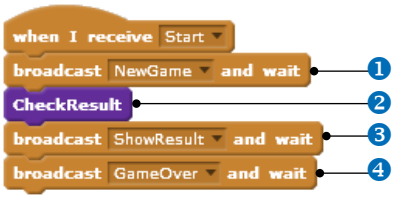


Figure 10: Script for the Stage to respond to the player's choice of action

The script starts by broadcasting the NewGame message to tell the Player and the Computer sprites to display their choice ❶. It then calls its **CheckResult** procedure to find out who won this round of the game ❷. After that, it broadcasts ShowResult to tell the Result sprite to show the result ❸, followed by GameOver ❹ to let the other sprites prepare for another round. Let's examine these scripts one at a time.

The NewGame message is handled by all six sprites. The Rock, Paper, Scissors, and Result sprites simply hide themselves when they receive this message. The Player and the Computer sprites, on the other hand, execute the scripts shown in Figure 11.

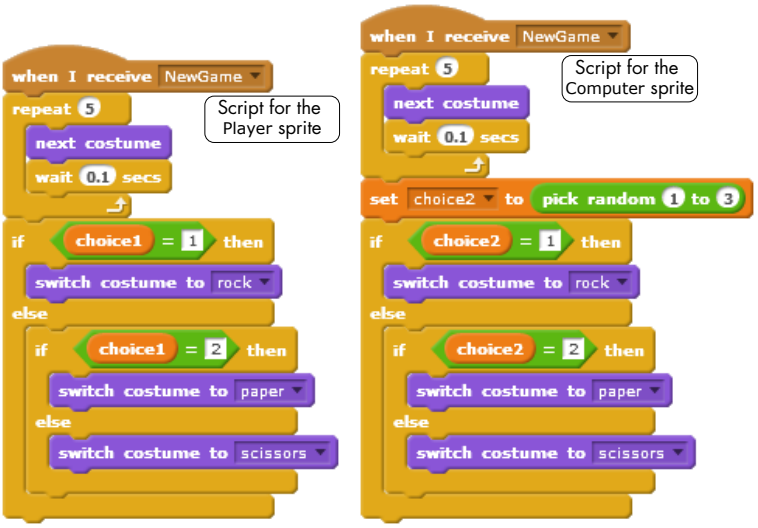


Figure 11: The scripts prompted by the NewGame *message for the* Player *(left) and the* Computer *(right) sprites*

Both sprites switch their costumes five times in a row very quickly (for a fun visual effect), and then they show their selected costume. While the costume of the Player sprite is decided by the user via the choice1 variable (which is set when the user clicks one of the three buttons), the Computer sprite sets the value of the choice2 variable to a random number between 1 and 3 and sets its costume accordingly. Both the Player and the Computer sprites have three costumes each that represent rock, paper, and scissors.

Once the Player and the Computer sprites have shown their selection, the Stage calls its **CheckResult** procedure, shown in Figure 12. Note that the nested **if/else** blocks have been separated to make the code easier to understand. The procedure compares the values of choice1 and choice2 and sets the value of the winner variable accordingly. Remember that a 1 corresponds to rock, a 2 corresponds to paper, and a 3 corresponds to scissors.

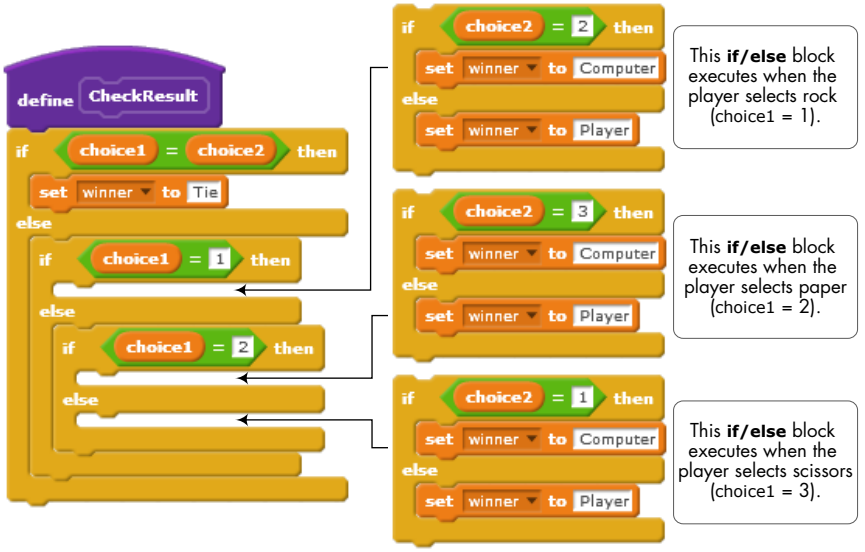


Figure 12: The ***CheckResult*** *procedure of the Stage*

After checking the result and setting the value of the winner variable, the Stage broadcasts the ShowResult message. The message is handled by the Result sprite, which executes the script shown in Figure 13. After checking the value of the winner variable, this code shows one of its three costumes accordingly. The **repeat** block just adds a visual effect to make the result display more clearly.

Finally, when the ShowResult script ends, the Stage sprite broadcasts GameOver to indicate the end of this round. In response to this message, the three button sprites show themselves, and the Result sprite hides itself. The game is ready for the next round.

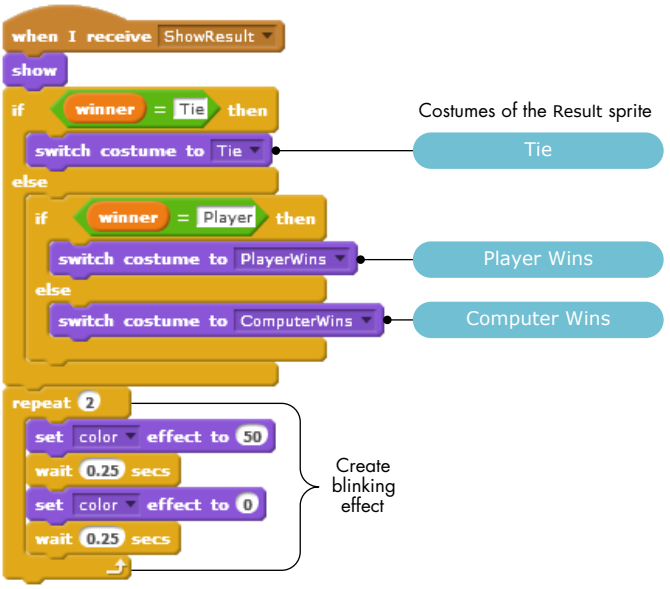


Figure 13: The script prompted by the ShowResult *message for the* Result *sprite*

RockPaper.sb2

TRY IT OUT 2

Load *RockPaper.sb2* and play it several times to see how it works. Make the game keep track of how many times the player has won, lost, and tied. Also, display the score information at the end of each round.

Chapter 7

The first bonus script for Chapter 7 is a game that you can use to test your counting skills. You'll also build two science simulations in this section: one for planetary motion and another to demonstrate the motion of a single gas molecule inside a container.

Match That Amount

MatchThat Amount.sb2

The game shows an amount of money in pennies, and it asks the user to find the least number of coins needed to give an equivalent amount of money. The interface for the game is illustrated in Figure 14.

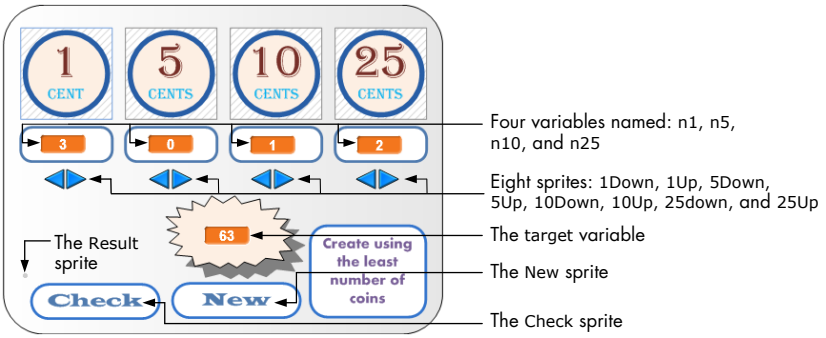


Figure 14: The user interface for the Match That Amount application

The game contains 11 sprites: eight arrow buttons for setting the answer, a New button for starting a new problem, a Check button for checking the answer, and a Result sprite for giving feedback to the user.

The game starts when the user clicks the New button to display a new problem. In response, the New sprite runs the script shown in Figure 15 (left). The script sets the target variable to a random number between 1 and 250 and then broadcasts NewProblem. The only sprite that traps this message is the Result sprite, which simply clears its current message (if any) by executing a **think** command with a blank string, as shown in Figure 15 (right).

Figure 15: Scripts for the New *sprite (left) and the* Result *sprite (right)*

The script expects the user to pick an answer by clicking the left and right arrow buttons below the different coins. The buttons have nearly identical scripts; the only difference is which variable they control. The scripts associated with the 1Up and the 1Down sprites are shown in Figure 16. The scripts for the other buttons work in a similar manner: The 5Up and 5Down sprites update the variable n5, the 10Up and 10Down sprites update the variable n10, and the 25Up and 25Down sprites update the variable n25.

Figure 16: Scripts for the 1Up *and* 1Down *sprites*

After setting an answer, the user clicks the Check button to see whether that answer is correct. In response, the Check sprite broadcasts CheckAnswer, which is received and processed by the Result sprite via the script shown in Figure 17.

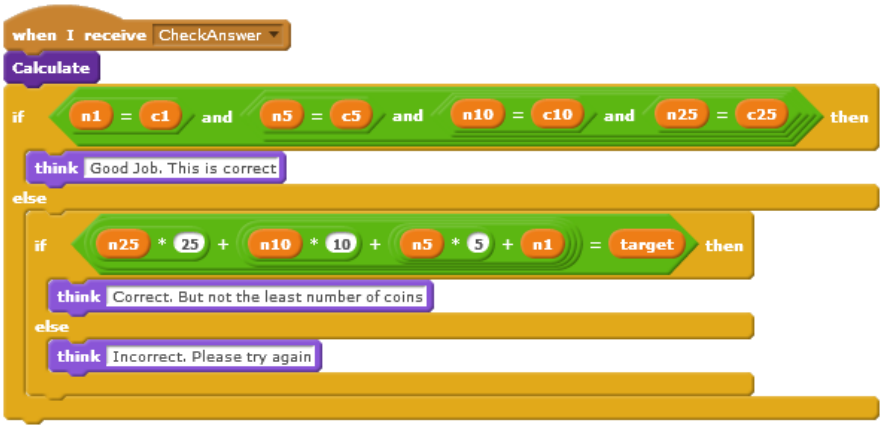


Figure 17: The script prompted by the CheckAnswer *message for the* Result *sprite*

The script first calls **Calculate** to compute the least number of coins needed to match the target amount. The computed values are saved in four variables named c1, c5, c10, and c25, which indicate the correct number of 1¢, 5¢, 10¢, and 25¢ coins, respectively. The script then compares the values specified by the user (saved in n1, n5, n10, and n25) with the correct values. If the values match, then the user's answer is correct. If there is a mismatch but the answers add up to the target amount, then the script tells the user that fewer coins could have been used. Otherwise, the answer is incorrect, and the user is asked to try again. Let's now examine the **Calculate** procedure, shown in Figure 18.

The variables c1, c5, c10, and c25 indicate the correct number of pennies, nickels, dimes, and quarters, respectively. The rem variable is used to keep track of the remaining number of pennies after the value of the other selected coins is subtracted. The procedure starts by initializing the four variables to 0 and setting rem equal to the target amount. The first loop finds the optimal number of quarters by repeatedly subtracting 25 from the target value until there are fewer than 25 pennies. The second loop finds the optimal number of dimes by repeatedly subtracting 10 from the remainder, and the third loop uses the same technique to find the optimal number of nickels.

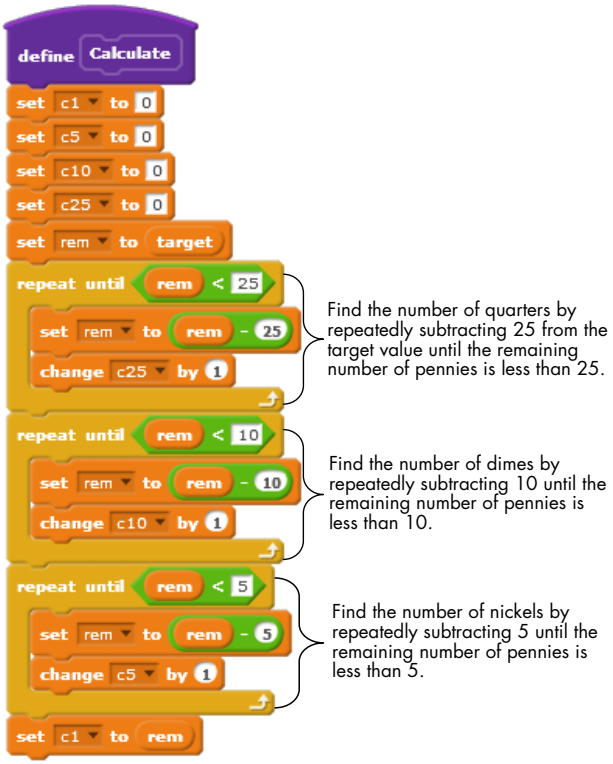


Figure 18: The **Calculate** *procedure*

MatchThat Amount.sb2

TRY IT OUT 3

Load *MatchThatAmount.sb2* and play it several times to understand how it works. Modify the **Calculate** procedure to use the **mod** operator (from the *Operators* palette) instead of the **repeat until** blocks.

Planetary Motion

Orbit.sb2

In this section, we'll explore planetary motion for a simple solar system that contains the Sun and a single planet, which we'll call Earth, as shown in Figure 19.

According to Newton's law of gravity, the gravitational force, F, between the Sun and Earth is given by

$$F = G\frac{Mm}{r^2}$$

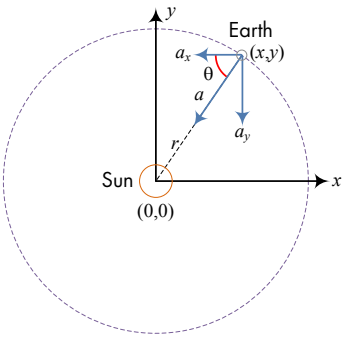


Figure 19: The orbit of Earth around the Sun

where M and m are the masses of the Sun and Earth, r is the distance between them, and G is the universal gravitational constant. Using Newton's second law, the acceleration, a, of Earth toward the Sun is calculated as follows:

$$a = \frac{F}{m} = G\frac{M}{r^2}$$

If the Sun is located at (0,0) and Earth's current position is (x,y), the x- and y-components of the acceleration are

$$a_x = a\cos\theta = -G\frac{Mx}{r^3}$$

and

$$a_y = a\sin\theta = -G\frac{My}{r^3},$$

where the negative sign indicates that the acceleration is directed toward the Sun.

Since acceleration is change in velocity per unit of time, during each time interval, Earth's horizontal velocity v_x changes by a_x, and Earth's vertical velocity v_y changes by a_y. In addition, since velocity is change in position per unit of time, Earth's horizontal position changes by v_x and its vertical position changes by v_y for each time interval. That means for each time interval in our simulation, we need to calculate Earth's distance to the Sun, change v_x (by $-a_x$) and v_y (by $-a_y$) and change Earth's x-position (by v_x) and y-position (by v_y).

The last thing we need to consider is which units correspond to the scale of the problem. Measuring distance in astronomical units (where 1 AU $\approx 1.5 \times 10^{11}$ meters) and time in years (with 1 year $\approx 3.156 \times 10^7$ seconds), we have

$$M \times G \approx 1.99\left(10^{30}\right)\text{kg} \times 6.673\left(10^{-11}\right)\frac{\text{m}^3}{\text{kg sec}^2} \approx 39.2\,\text{AU}^3 \,/\, \text{year}^2$$

The user interface for our simulation is shown in Figure 20. The application contains three sprites named Sun, Earth, and Show.

The Show sprite has two costumes that show a checkbox in its checked and unchecked states. Clicking this button causes the sprite to switch between these two costumes and broadcast either the ShowOrbit or HideOrbit message. When the Earth sprite receives the ShowOrbit message, it puts its pen down to draw the orbit on the Stage, and when it receives HideOrbit, it puts its pen up and clears the Stage. You'll find the corresponding scripts in *Orbit.sb2*.

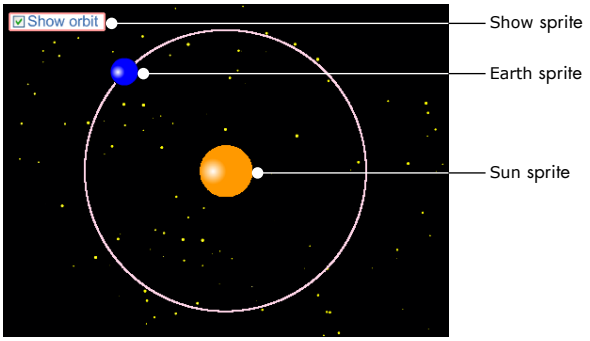


Figure 20: The user interface for the planetary motion simulation

The script for the Earth sprite, which drives the simulation, is shown in Figure 21.

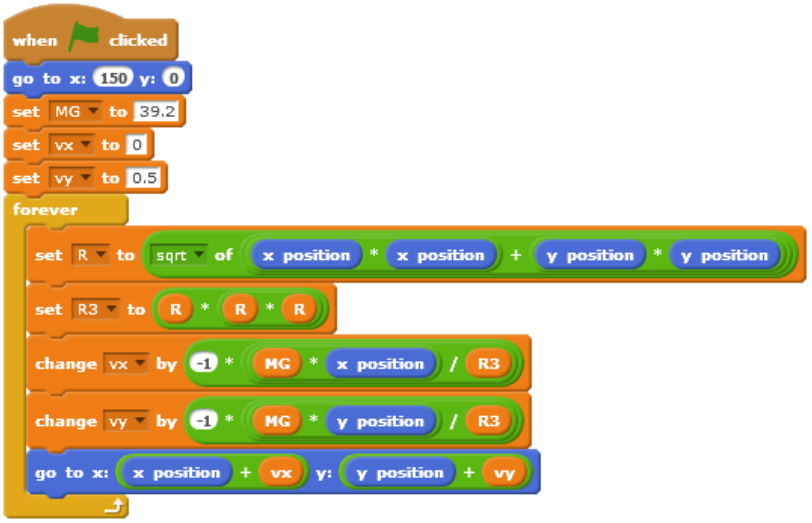


Figure 21: Script for the Earth *sprite*

The Earth is first moved to point (150,0) on the Stage and is given a small initial velocity in the positive *y*-direction. The infinite loop then performs the following steps repeatedly:

1. Compute the current distance (R) to the Sun and find the cube of this distance (saved in the variable R3).
2. Update the horizontal and vertical components of the velocity, as described above.
3. Update the *x*- and *y*-coordinates of the Earth and move it to its new position.

TRY IT OUT 4

Orbit.sb2

Load *Orbit.sb2* and run it to understand how it works. Although the orbit in Figure 20 looks circular, it actually isn't. Check the monitor for the R variable to see how Earth's distance from the Sun changes during the simulation.

Molecules in Motion

MoleculesIn Motion.sb2

According to the kinetic theory of gases, the molecules of any gas are in constant and random motion. If a gas is trapped in a container, its particles constantly collide with each other and with the walls of the container. The speed of motion is directly proportional to the temperature of the gas. The simple simulation presented here focuses on how a single molecule's motion changes when it collides with the walls of its container. The user interface for the application is shown in Figure 22.

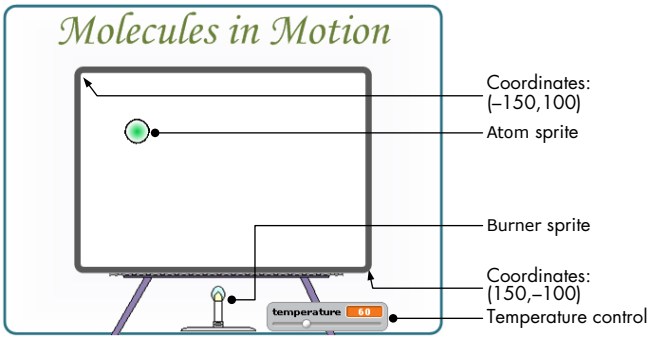


Figure 22: The user interface for the Molecules in Motion simulation

When the sprite collides with a wall of the container, it should bounce off that wall at an angle that is equal to the *angle of incidence* (the angle at which the sprite hit the wall), as demonstrated in Figure 23.

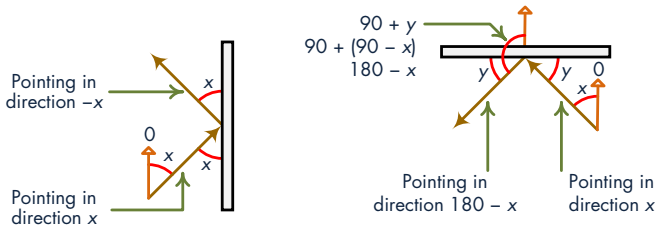


Figure 23: Direction after colliding with the right wall (left) and the top wall (right)

Figure 23 (left) shows that if a sprite pointing in some direction (x) hits the right (or left) wall, it will bounce off the wall in the direction $-x$. Similarly, Figure 23 (right) shows that if a sprite pointing in some direction (x) hits the top (or bottom) wall, it will bounce off in the direction $180 - x$.

The application contains two sprites: the Atom sprite and the Burner sprite. The Burner sprite has four costumes that show flames of different sizes, and its script switches between the costumes to animate the flame. You'll find the burner's scripts in *MoleculesInMotion.sb2*; the script for the Atom sprite is shown in Figure 24.

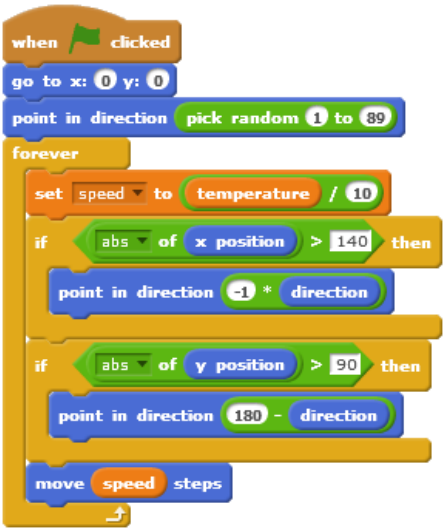


Figure 24: Script for the Atom *sprite*

The sprite moves to the origin of the Stage, selects a random initial direction, and enters a **forever** loop. During each iteration of the loop, it sets the speed variable based on the current value of the temperature variable, which the user controls via a slider. The two **if** blocks check the sprite's current x- and y-coordinates to see whether it has collided with any of the walls of the container. In the case of a collision, the sprite sets its new heading as described above. The sprite then moves some distance (specified by the speed variable) in its new direction.

Chapter 9

Chapter 9 covered lists, and the three bonus applications for this chapter feature them as well. The first application is a two-player game about sorting fractions and decimals. The second is a program that spells whole numbers. The third demonstrates the sieve of Eratosthenes, an algorithm for finding all prime numbers less than 100.

Sort 'Em Out

SortEmOut.sb2

This game involves sorting fractions and decimals. The user interface for the game is shown in Figure 25.

Each player gets 5 random cards from a deck of 31 cards, and a player can press the deal button to receive a new card. Players can then drag a new card over one of their original five cards to replace it, or they can drag it over the recycle bin image to discard it. The first player to arrange cards in ascending order wins the game.

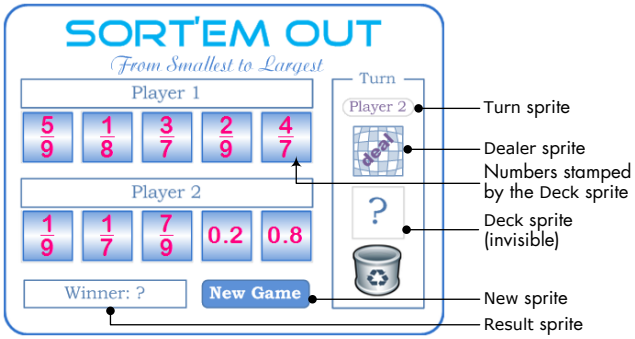


Figure 25: The user interface for the Sort 'Em Out game

The application contains five sprites. The Dealer sprite, represented on the interface by the deal button, manages the flow of the application. The Deck sprite has the costumes of the 31 cards (see Figure 26). It shows the players' cards and contains the logic that determines whether the dealt card was dropped over another card or onto the recycle bin. The 31 cards are added to the costume list of this sprite in order, from smallest to largest, where the first costume corresponds to the 0.1 card, the second costume corresponds to the 1/9 card, and so on. The New sprite represents the New Game button, the Turn sprite shows which player's turn is next, and the Result sprite shows the winner.

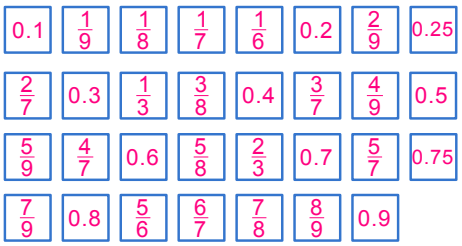


Figure 26: The 31 costumes of the Deck *sprite*

NOTE *This application has many scripts, but this section explains only those related to list management. You can find the missing details in the* SortEmOut.sb2 *file.*

When a player clicks the New Game button, the New sprite broadcasts the NewGame message. In response, the Dealer sprite executes the script shown in Figure 27.

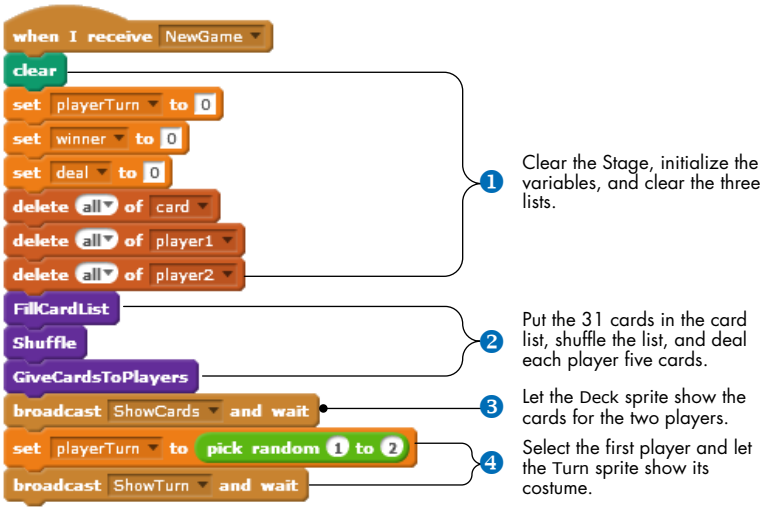


Figure 27: The script activated by the NewGame *message for the* Dealer *sprite*

This script uses the variables and lists described in Table 1 to keep track of all the information for the players and cards in Sort 'Em Out.

Table 1: Variables and Lists for the New Game Script

Name	Type	Description
playerTurn	variable	Indicates whose turn is next. A value of 1 means player 1, a value of 2 means player 2, and a value of 0 means the game hasn't started.
winner	variable	Indicates the winner of the game. Again, a value of 1 means player 1, a value of 2 means player 2, and a value of 0 means not decided yet.
deal	variable	Indicates whether or not to deal a new card. When a player deals a new card, this flag is set to 1 to indicate that no more cards should be dealt until the player does something with the dealt card.
card	list	Contains the available cards for dealing.
player1	list	Contains the first player's five cards.
player2	list	Contains the second player's five cards.

The script starts by clearing the Stage from the stamps of the previous game. It then initializes the three variables (playerTurn, winner, and deal) to 0 and clears the three lists (card, player1, and player2) in preparation for a new game ❶. The script then calls **FillCardList** and **Shuffle** to put the 31 cards in the card list in random order and calls **GiveCardsToPlayers** to deal each player five cards, starting from the top of the card list ❷.

(I'll explain these procedures soon.) After that, the script broadcasts the ShowCards message to tell the Deck sprite to show the two players' cards on the Stage ❸. The Deck sprite reads the contents of the player1 and player2 lists, and it stamps the costumes that correspond to each element at a predefined location on the Stage (see *SortEmOut.sb2* for the details). At last, the script selects the first player randomly and broadcasts the ShowTurn message to the Turn sprite ❹. The Turn sprite responds by displaying a costume that reads either "Player 1" or "Player 2," based on the current value of the playerTurn variable.

Now, let's explore the **FillCardList** and the **Shuffle** procedures, shown in Figure 28.

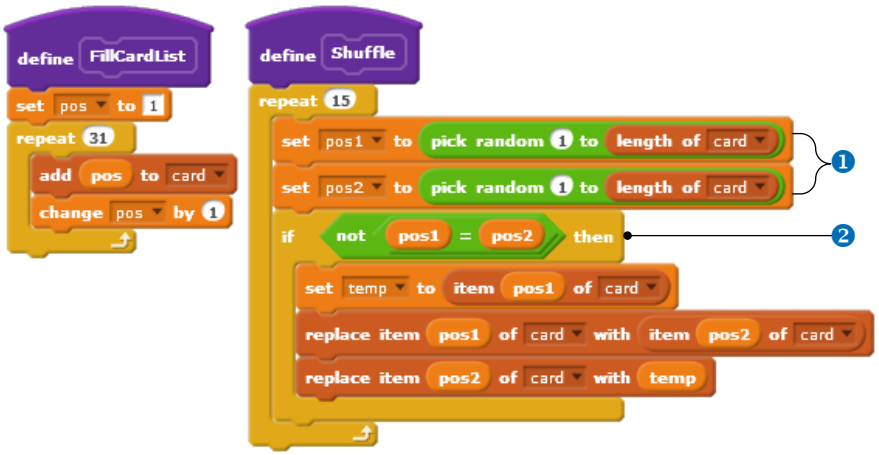


Figure 28: **FillCardList** *(left) and* **Shuffle** *(right) belong to the* Dealer *sprite.*

The **FillCardList** procedure adds the numbers 1 through 31 to the card list. The **Shuffle** procedure rearranges the elements of the card list by swapping their positions randomly. It does this by selecting two random positions in the list ❶ and swapping their contents using a temporary variable named temp ❷. The count of 15 for the **repeat** loop is arbitrary.

The **GiveCardsToPlayers** procedure, which is shown in Figure 29, simulates the process of dealing cards from the top of the deck to the two players in alternate turns.

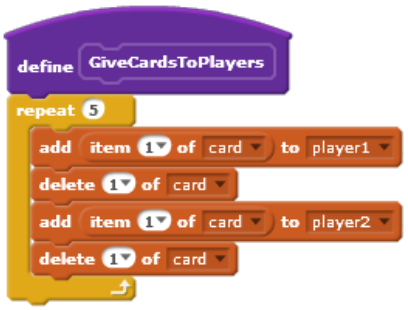


Figure 29: The Dealer *sprite's* **GiveCardsToPlayers** *procedure*

The procedure takes the top card from the deck (the first item of the card list) and gives it to player 1. It then takes the new top card and gives it to player 2. The process repeats until each player has five cards. At the end of this procedure, both the player1 list and the player2 list will have five numbers (representing five cards), and the card list will have the 21 numbers remaining for dealing.

Now the two players are ready to start playing. The player whose turn it is clicks the deal button to see the card at the top of the deck. In response, the Dealer sprite executes the script shown in Figure 30.

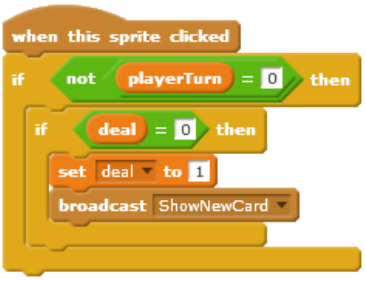


Figure 30: The script that runs when the Deal *button is clicked*

If the game has been initialized (that is, if the playerTurn variable isn't 0) and the deal flag is 0 (which means that the sprite is not currently waiting on a player's action), the script sets the deal flag to 1 (to ignore additional clicks of the deal button) and broadcasts the ShowNewCard message to the Deck sprite. When the Deck sprite receives this message, it wears the costume of the card specified by the first element of the card list and makes itself visible to allow the player to drag it over the Stage. The Deck sprite then tracks the player's drag-and-drop actions. Nothing happens until the player drops the card either on top of one of the five cards or onto the recycle bin.

The Deck sprite communicates the user's action back to the Dealer sprite via the variable dropTarget. If the player drops the card onto the recycle bin, the sprite sets dropTarget to 0, hides itself, and broadcasts the GotUserChoice message. If the player drops the card over one of the five cards, the sprite sets dropTarget to 1, 2, 3, 4, or 5 (based on which of the five cards was chosen), stamps the image of the dragged card over the selected card, and broadcasts the GotUserChoice message. Check *SortEmOut.sb2* for the details of the drag-and-drop procedure.

When the Dealer sprite receives the GotUserChoice message, it executes the script shown in Figure 31.

Figure 31: The GotUserChoice *script of the* Dealer *sprite*

If the dropTarget variable is set to 0, then the player has chosen to discard the dealt card. In this case, the script calls **RecycleCard** to put the card back at the end of the deck and then gives the next player a turn ❶. Otherwise, if the player has chosen to replace a card, the script calls **SwapCards** to replace the player's card with the dealt card ❷. The **SwapCards** procedure also checks whether the player's latest move has produced an ordered set of cards and sets the winner variable accordingly. When **SwapCards** returns, the script checks the value of the winner variable ❸. If winner has a nonzero value (meaning that a winner has been detected), the script broadcasts the ShowWinner message (to the Result sprite) to show the winner of the game and ends the game. Otherwise, it calls **NextPlayer** to give the next player a turn.

Let's now look at the three procedures called from the GotUserChoice script, starting with the **RecycleCard** and the **NextPlayer** procedures, shown in Figure 32.

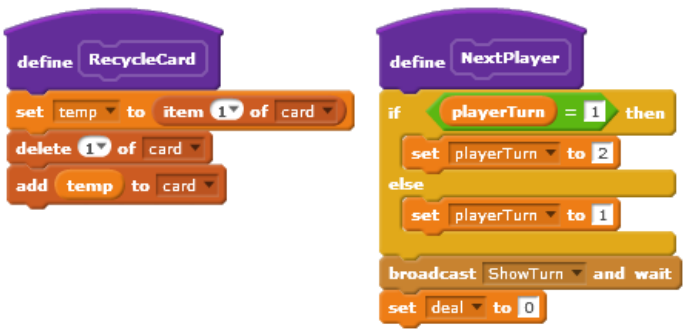


Figure 32: The **RecycleCard** *(left) and the* **NextPlayer** *(right) procedures of the* Dealer *sprite*

The **RecycleCard** procedure moves the first (top) card from the deck to the end of the deck. The **NextPlayer** procedure first switches the value of the playerTurn variable and then broadcasts the ShowTurn message to tell the Turn sprite to show the correct image for the next player. The procedure also resets the deal flag back to 0 to indicate that it is ready to process the next click of the deal button. The third procedure, **SwapCards**, is shown in Figure 33.

Figure 33: The **SwapCards** *procedure of the* Dealer *sprite*

The procedure replaces the element at position dropTarget in the list for the current player with the dealt card (which is the first element in the card list) ❶, and it moves the replaced element to the end of the card list ❷. The procedure then calls **CheckPlayer1List** (or **CheckPlayer2List**) to see whether the player's five cards are in order. The **CheckPlayer1List** procedure is shown in Figure 34. The **CheckPlayer2List** procedure is almost identical, except for its target list (player2) and the value it assigns to the winner variable (2).

The **CheckPlayer1List** procedure compares the first and second elements, the second and third, and so on ❶. If the second element in any of these pairs is less than the first element, then the list is not sorted ❷. Otherwise, the list is in the proper order, and the winner variable is set accordingly ❸.

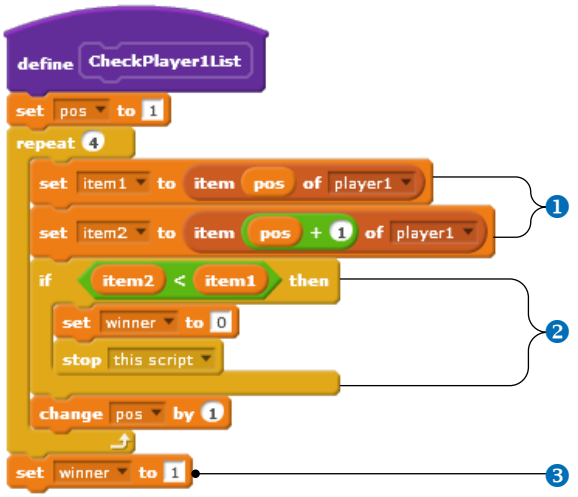


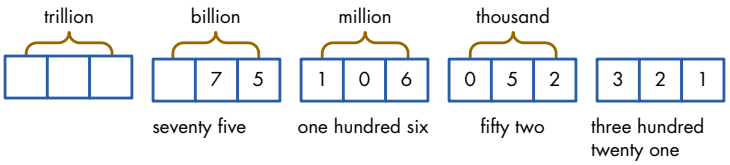
Figure 34: The **CheckPlayer1List** *procedure of the* Dealer *sprite*

Say That Number

SayThatNumber .sb2

This section presents an application that asks the user for a number and says that number in words. If the user inputs 3526, for example, the program will say "three thousand five hundred twenty six".

As demonstrated in the example in Figure 35, the idea is to break the number, from right to left, into groups of three digits. Each of these groups is then spelled out individually, followed by a multiplier word (such as *thousand*, *million*, and so on) if needed.



seventy five billion, one hundred six million, fifty two thousand, three hundred twenty one

Figure 35: Spelling out a number in words

First, we'll need to get a number from the user. Then, our script will need to break that number into groups of three digits and save the resulting groups in a list (called group), as illustrated in Figure 36. Once the digits are in a list, our script can cycle through the elements in reverse order, spell out the digits of each group, and append the appropriate multiplier word to the group.

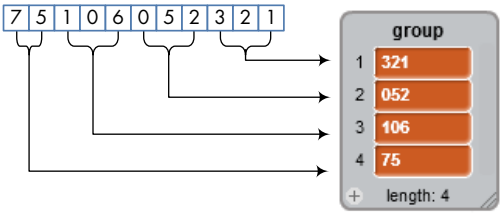


Figure 36: Breaking the user's number into three-digit groups

Our first procedure, called **BreakNumber**, is shown in Figure 37. It works on the number input by the user, which is saved in the built-in variable answer.

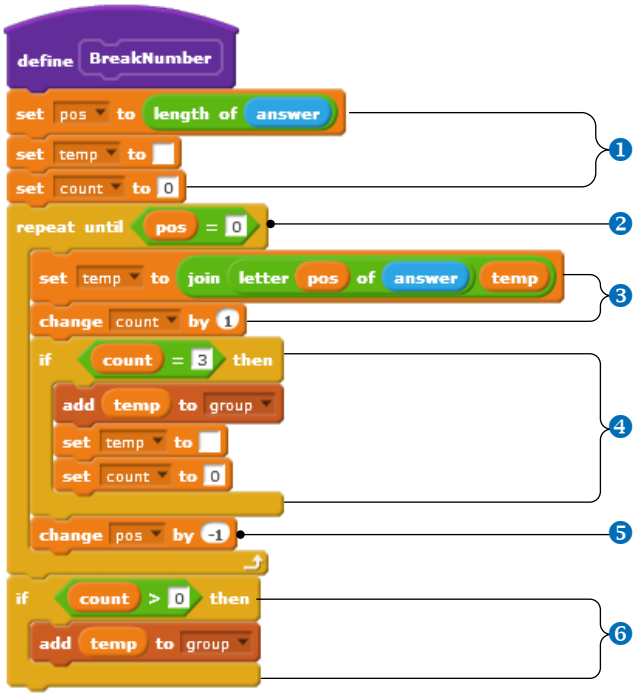


Figure 37: The **BreakNumber** *procedure*

The procedure divides the input number into groups of three digits and saves these groups into the group list. The procedure also uses three variables: pos is an index to the individual digits of the input number, temp temporarily holds the three digits of each group, and count keeps track of the number of digits added to temp.

The procedure first sets pos to index the first digit (from the right) of answer, empties the temp string, and sets count to 0 to indicate that temp is currently empty ❶. It then starts a **repeat until** loop to cycle through all the digits of answer from right to left ❷. Note that pos is changed by

–1 at the end of each iteration ❺. Inside the loop, the procedure appends one digit from answer to the temp string and changes count by 1 ❸. The added digit is the one whose index is given by the pos variable. When count becomes 3 ❹, meaning that we've accumulated a three-digit group, the procedure adds the value stored in the temp variable to the list, empties the temp string, and resets count to 0 in order to prepare for extracting the next three digits of answer. If count is greater than 0 when the loop ends ❻, then our last group has fewer than three digits, and we simply add this group to the list.

Now we need to write a procedure that spells a three-digit number. If the number is anything from 100 to 999, we'll spell out the digit in the hundreds place followed by the word *hundred*. We'll then remove the hundreds digit from the number so we only have to spell a two-digit number. For example, if the input number is 321, this step produces the text "three hundred" and changes the input number to 21 (that is, 321 – 300).

To spell a two-digit number, we have to consider two cases. If the number is between 11 and 19 (inclusive), we can spell the number as one word (*eleven*, *twelve*, and so on) and be done. If the number is greater than 19 or exactly 10, we can spell the tens digit (*ten*, *twenty*, and so on) and then remove it, leaving us with a one-digit number to spell. For the input number 21, this step produces the text "twenty" and changes the input number to 1 (that is, 21 – 20).

Spelling a one-digit number is the simplest step. If the digit is between 1 and 9 (inclusive), we'll spell out a single word (*one*, *two*, and so on) that corresponds to the input digit.

In addition to the group list, the application uses the four lists shown in Figure 38 to store the substitution words needed for its spelling task. The first three lists (digit, ten, and teen) will be consulted to extract the appropriate words for spelling a three-digit number, whereas the multiplier list is used to get the appropriate multiplier for each group. Note that the first entry of the multiplier list contains an empty string because the first three digits (from the right) do not need a multiplier word.

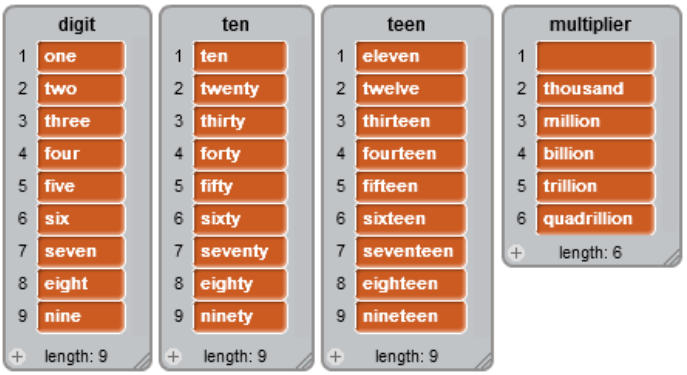


Figure 38: The lists used in the Say That Number application

Our **SpellTrio** procedure, which spells any three-digit number according to the above steps, is shown in Figure 39. The figure also includes a flowchart (right) that highlights the steps that occur when the input number is 321; the active path is shown in red.

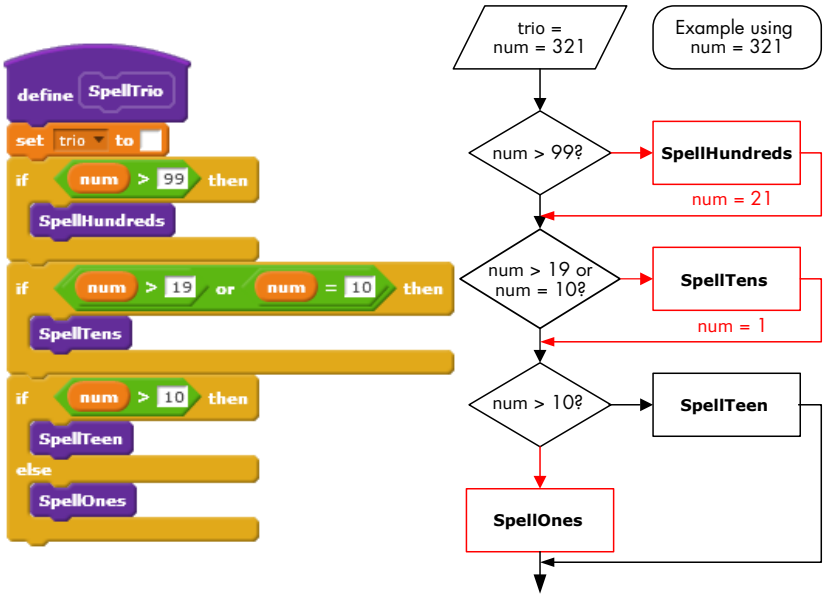


Figure 39: The ***SpellTrio*** *procedure*

The num variable specifies the three-digit number to be spelled. This procedure constructs and saves the spelled number in the variable trio. To make the procedure more readable, we created four blocks of our own that implement the different execution paths. Some of these subprocedures change the value num, as you'll see next.

The four subprocedures called by **SpellTrio** are shown in Figure 40. The procedures use a temporary variable (named d1) to hold the largest digit of the input number and use that digit as an index to one of the three word lists (digit, ten, or teen).

To explain how these scripts work, consider the flowchart of Figure 39. Since num = 321 in this example, **SpellTrio** starts by calling **SpellHundreds**. This procedure performs the following four steps (which correspond to the four statements of the procedure):

1. Set d1 = 3, (the first digit of 321).
2. Set trio = "three" (the third item in the digits list).
3. Add a space followed by "hundred" to trio. So trio becomes "three hundred".
4. Set num = num – (d1 * 100) . Since d1 = 3, num becomes 21 (that is, 321 – 300).

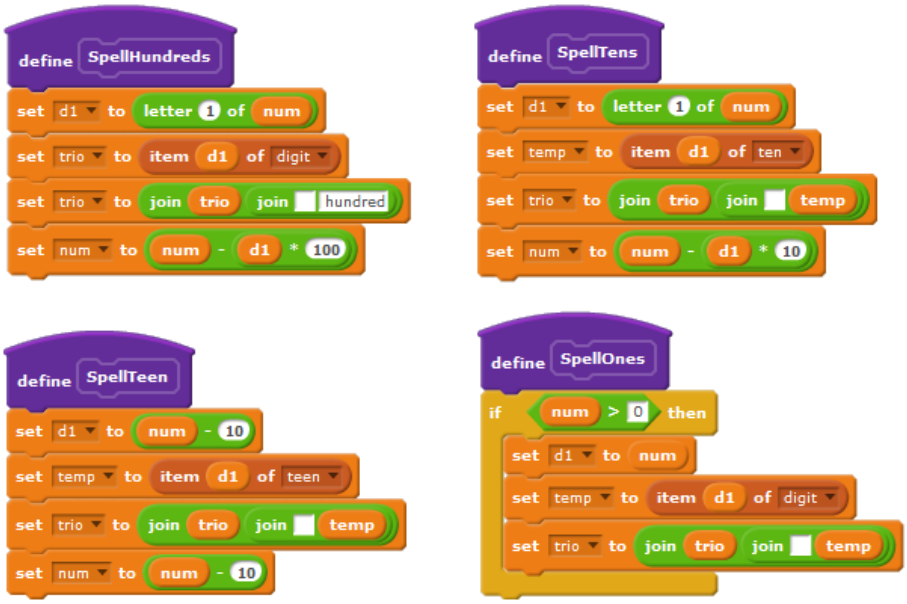


Figure 40: The four subprocedures used by **SpellTrio**

When **SpellHundreds** returns, **SpellTrio** checks num again. Since 21 is greater than 19, **SpellTrio** calls **SpellTens**. This procedure performs the following four steps:

1. Set d1 = 2, (the first digit of 21).
2. Set temp = "twenty" (the second item of the ten list).
3. To trio, add a space followed by the temp string. So, trio becomes "three hundred twenty".
4. Set num = num – (d1 * 10) . Since d1 = 2, num becomes 1 (that is, 21 – 20).

When **SpellTens** returns, **SpellTrio** checks num again. Since 1 is not greater than 10, **SpellTrio** calls **SpellOnes**. Since num = 1, this procedure sets temp = "one", the first digit of the digit list, and appends this string to trio, causing it to become "three hundred twenty one".

Equipped with these procedures, all we have to do now is call the **SpellTrio** procedure repeatedly for each element in the group list and append the appropriate multiplier after spelling each group. Our **SpellNumber** procedure is shown in Figure 41.

The procedure uses two variables: ans holds the words of the spelled-out number, and index is used to cycle through the items of the group list. The procedure first empties the ans string and initializes index to point to the last entry of the group list ❶. It then starts a **repeat** loop to cycle through all the items of the group list ❷. During each iteration of the loop, the procedure sets the num variable to the three-digit number currently being spelled and calls **SpellTrio** to spell out that number ❸.

When **SpellTrio** returns, we append its return value (which is saved in the trio variable) to ans, followed by the appropriate multiplier word from the multiplier list ❹. The index variable is also used to get the right entry from the multiplier list.

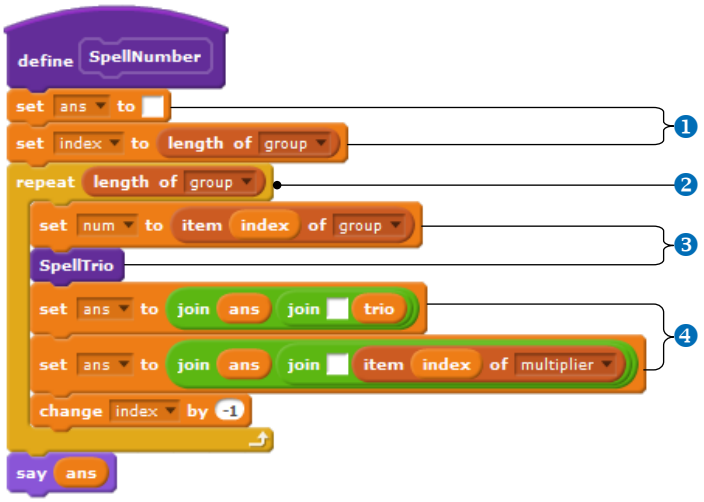


Figure 41: The **SpellNumber** *procedure*

The main procedure that gets the user's input and drives the flow of the application is shown in Figure 42. It starts by emptying the group list of any entries added in a previous run. It then prompts the user to enter a number and waits for input. After getting the user's input, it calls **BreakNumber** to break the input number into groups and save those groups into the group list. It then calls **SpellNumber**, which works on the entries of the group list to spell out the input number.

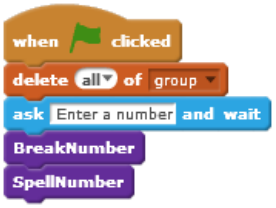


Figure 42: The main script for the Say That Number application

The Sieve of Eratosthenes

Sieve.sb2

Prime numbers have fascinated people since ancient times. The Sieve of Eratosthenes procedure, which is illustrated in Figure 43, provides one way to find all the prime numbers less than a given number. It works as follows:

1. Cross out number 1 because 1 is not a prime number.
2. Circle number 2 because 2 is prime.
3. Cross out all multiples of 2 (because they are not primes).
4. The next non-crossed-out number after 2 is 3. This number is thus a prime, and we circle it.
5. Cross out all multiples of 3 (because they are not primes).

6. The next non-crossed-out number after 3 is 5. This number is thus a prime, and we circle it.
7. Cross out all multiples of 5 (because they are not primes).
8. Continue in this manner. The circled numbers at the end are prime numbers.

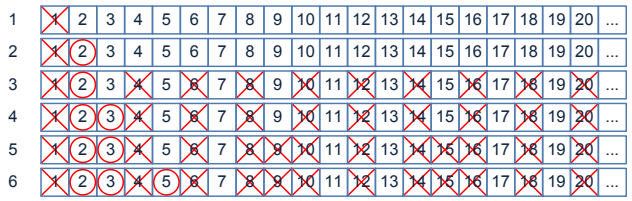


Figure 43: Using the sieve of Eratosthenes to find prime numbers

In this section, we'll develop an application that demonstrates the sieve of Eratosthenes. The interface for this application is shown in Figure 44.

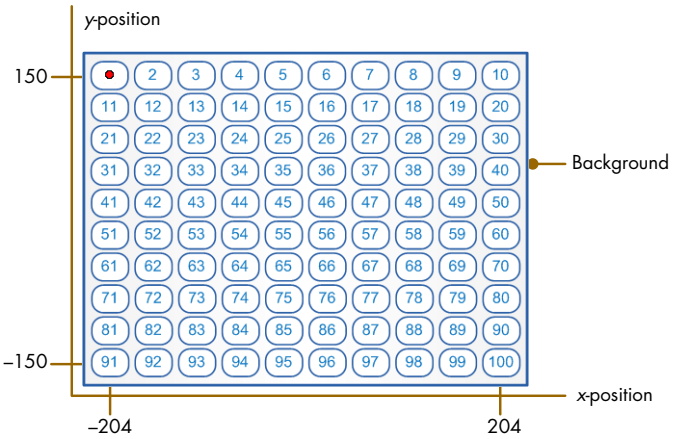


Figure 44: The Sieve of Eratosthenes that we'll use in our script

A sprite will move over these numbers, crossing out the nonprime numbers as it goes along. At the end of the script, the remaining numbers will be the prime numbers below 100.

The application contains one sprite (named Driver), which has two costumes. The first costume (dot) is a small red circle that the sprite wears when moving over the numbers. The second costume (blank) is a white rectangle that the sprite uses to wipe out nonprime numbers by stamping this image over the nonprime numbers.

The application uses a list (numList) with 100 elements to keep track of prime and nonprime numbers. The first element of the list corresponds to 1, the second element to 2, and so on. When the application ends, the value stored in each element of the list will be either a 1 or a 0 to indicate whether the corresponding number is a prime or not, respectively.

Let's explore how our sieve works in Scratch in detail. When the green flag icon is clicked, the Driver sprite runs the script shown in Figure 45.

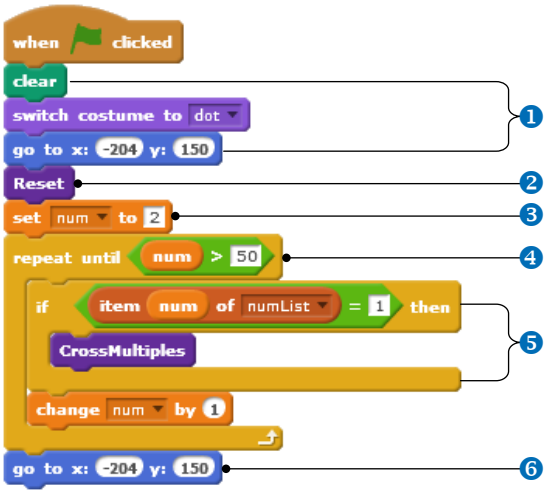


Figure 45: The main script of the Driver *sprite*

The script clears the Stage (in case the sieve was run previously), commands the sprite to wear its red dot costume, and moves the sprite to the upper left cell (over the 1 in the sieve) ❶. The script then calls the **Reset** procedure ❷, which initializes the 100 elements of numList to 1. After that, it sets the variable num to 2 (since this is the first prime number) to prepare for crossing out its multiples ❸, and it starts a **repeat** loop ❹ to initiate the process of crossing out the multiples of each discovered prime number. Inside the loop, every time we find a list element whose value is 1 (meaning that we've found a new prime number), we call **CrossMultiples** to cross out the multiples of that prime ❺. When the loop terminates, the sprite returns to the upper-left corner of the Stage ❻. The **CrossMultiples** procedure is shown in Figure 46.

The **CrossMultiples** procedure uses a variable (named m) to hold the multiples of the input number (num). The **SetPosition** procedure (shown in Figure 47) moves the sprite to the correct location on the Stage as specified by the current value of m ❶. The script starts a loop to examine the numbers 2 × m, 3 × m, 4 × m, and so on, until the examined multiple exceeds 100 ❷. If the list element corresponding to any of these multiples has a value of 1, the sprite switches to its blank costume and stamps over that number on the Stage to wipe it out ❸. It also sets the list value that corresponds to that multiple to 0.

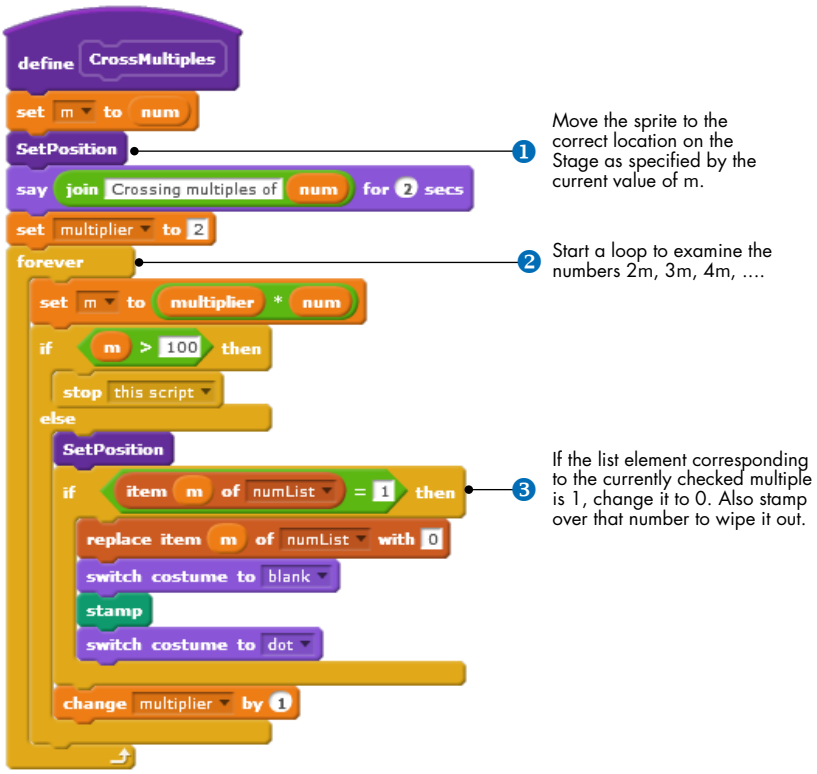


Figure 46: The **CrossMultiples** *procedure*

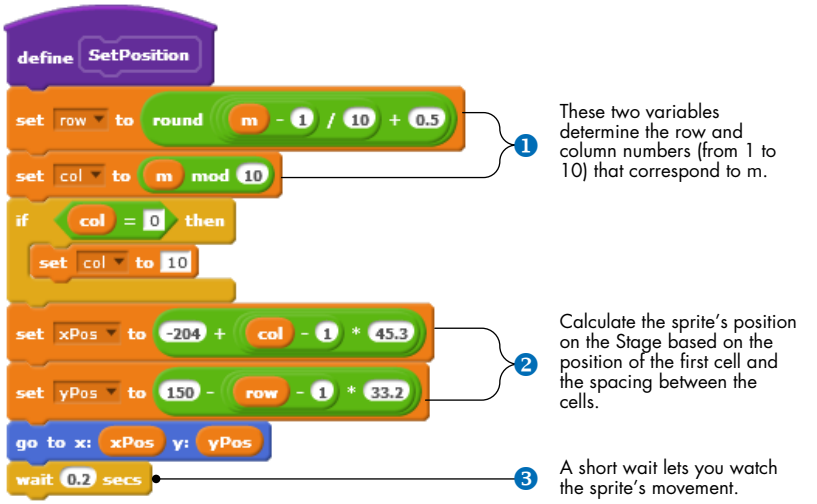


Figure 47: The **SetPosition** *procedure*

The **SetPosition** procedure sets the sprite's location in accordance with the value stored in m. The two variables row and col are used to determine the row and column numbers (from 1 to 10) that correspond to m ❶. Based on the position of the first cell (which contains 1) and the spacing between the cells on the Stage, the procedure calculates the xPos and the yPos of the sprite ❷. I've also included a short **wait** time ❸ so you can watch the sprite's movement on the Stage.

Figure 48 shows the result of running this application. All nonprime numbers have disappeared, and the remaining numbers are prime.

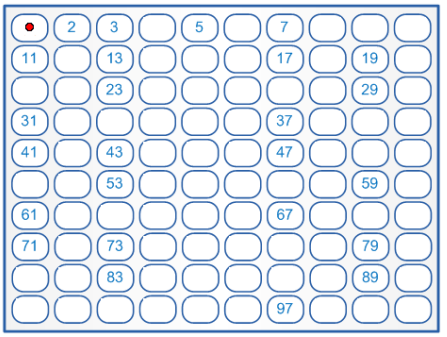


Figure 48: The output of the Sieve of Eratosthenes application