Computer Science Concepts in Scratch

(Scratch 1.4)

Version 1.0

Michal Armoni and Moti Ben-Ari

Contents

Introduction

This book will familiarize you with the Scratch visual programming environment. Scratch users have many different goals: some will build games for entertainment, while others will construct simulations of the natural world to use in teaching and learning. The Scratch system can be used by people with different skills: designing a game or a film, creating images, or programming. The authors of this book focus on a specific goal: using Scratch to encourage learning of *computer science*. Of course, the skills and knowledge that you learn from this book will enable you to use Scratch for any purpose you wish, but we emphasize understanding important concepts and ideas of computer science, and not, for example, those needed to construct a film or a game.

Why computer science?

Computers have entered into all phases of our life: most films that we watch have computer-generated special effects. Modern “gadgets” contain computers: music players, cell phones, cameras. Most of our money exists not as coins and bills but as data stored in the banks’ computers. Doctors send us to be tested by machines that are controlled by computers and, in many cases, even the results of the tests are images that are generated by computers. Every time we travel—by airplane, train or even by car—we entrust our safety to computers.

People who build computerized systems, especially those who develop the *software* or *programs* for the computers, are responsible for translating our wishes and requirements into a form that can be understood by computers, which are—after all—persistent and speedy but basically rather stupid. Developing software is a fascinating and challenging occupation because of the need to deal with two such different types of creatures: people and computers. We hope that familiarity with software development in Scratch will bring you closer to the field of computer science and perhaps even encourage you to consider studying computer science in the future.

Why Scratch?

Almost everyone knows how to *use* a computer. We use them for surfing the internet (email, downloading music and videos, chatting, shopping), for writing documents and for playing games. Aren't you curious to learn how these amazing applications are *built*? Well, they are built as *computer programs*, which are written in *programming languages* that a computer can understand and run. Unfortunately, programs like internet browsers and word processors are very complex—they can have millions of instructions in a programming language—and the languages themselves were designed for professional programmers and are difficult to learn. Here is where Scratch comes to our rescue. Scratch is easy to learn and you can build programs for a computer (or, as we say, *to program the computer*) immediately when you start to work with Scratch. Furthermore, Scratch supports the use of graphics, animation and sound without requiring that you understand the technical details.

Don't let the colorful presentation of Scratch fool you! It is not a computer game. Scratch is a real software development environment, and experience with Scratch will provide you a glimpse of what it is like to program a computer professionally. It is lots of fun working with Scratch, and the programs you develop can be exciting games or interesting simulations, but during the process of creating these programs you will face the same challenges faced by a professional programmer: What is the computer capable of doing? How can our wishes and needs be translated into instructions that the computer can understand and run? How are errors diagnosed and corrected? We are certain that studying Scratch in depth will be a fascinating experience.

The structure of the book

You don’t learn how to program a computer by reading a book but by writing programs. A book can only guide you and provide explanations. Therefore, this book is structured as a collection of ***examples*** and ***tasks***. Each chapter teaches a new concept (or perhaps a group of related concepts), but the concept is always introduced in order to solve a specific problem: counting oranges, animating dancing images, building a computer game, and so on. Each chapter starts with a simple task, but as soon as we solve a task, we add new aspects to the task, where the new aspect asks for an extension to the existing task or changes a requirement to make it more complex. Each task in the sequence of tasks will require that you use some new concept of computer science and either a new construct of Scratch or a different way of using one that you already know.

We suggest that you work on each task according to the following guidelines:

- *Read* the task.
- *Think* how you would solve the task and *write* your solution in English.
- Consider if you already know enough constructs in Scratch in order to translate your solution in a Scratch project and try to do so.
- If not, look for new Scratch constructs. Try using the new construct.

For each task, the book provides a *very detailed explanation* of how to solve it. In addition, you can download full solutions as Scratch projects that you can examine and run. Please try to solve tasks by yourself without looking at the explanations, but if you are having difficulty, read the explanations and then look at the solutions. Even if you solved the task without difficulty, we suggest that you read the explanations because they emphasize concepts of computer science that played a role in the solution.

A number of the tasks are designated as ***optional*** and some sections have ***Additional material*** in the title. Some of these concern advanced concepts that you may wish to leave for later once you have more experience with Scratch. Most of them, however, concern topics that are not central to learning important concepts of computer science, such as drawing images or changing the background image.

To facilitate using the book as a reference, framed text summarizes the material in each section. A frame with a bold border:

> **New concept:**

is used for *computer science concepts* that are not limited to the Scratch environment, while a frame with a normal border:

> **New construct in Scratch:**

is used for *programming constructs* in Scratch.

A word to the learner

The more effort you make to solve the tasks by yourself before reading the explanations, the more successful your learning will be. But, if you encounter a difficulty, *don't hesitate to read the explanations and to examine the solutions*. As you work through the tasks in the book, you will come to understand better the spirit of Scratch and you will find that it becomes easier to learn new concepts and constructs.

Please don't give up whenever you cannot solve a problem. Solving tasks and translating the solutions into a programming language are difficult and demand that you invest a lot of effort. Everyone who learns to program encounters such difficulties and it takes a while to become proficient. Look upon each task as a challenge that you have face and overcome. Rest assured that the more experience you gain, the more your confidence in your ability will increase.

A word to the teacher

There is no point in asking us: "Am I allowed to . . . ?" because the answer is always "yes." While we believe that we have chosen an effective order in which to present the concepts, as well as the appropriate tasks with which to illustrate them, feel free to change the order of presentation, skip chapters, propose other tasks, or change the learning process in any way you please. Furthermore, it is clear that each student will progress at a different rate. There is no point in slowing down students who wish to forge ahead in order to investigate new topics or to build other projects.

The only thing that we would like to insist upon is that you not compromise on the goals of the book. We ask that you ensure that students really are learning the basic concepts of computer science like control structures, data handling, communications, and so on. Students can find it very seductive to spend all their time drawing images, changing graphics effects and composing music. While design is important, Scratch has no advantages over other software for design. We prefer to look upon graphics design as a way of motivating students and simplifying the programming process, but this should not come at the expense of achieving our goal: bringing computer science closer to the students.

Chapter 1

First Steps

Scratch is a visual programming environment that we will use to become acquainted with concepts of computer science. Scratch does this by enabling you to create computerized *animations*. Initially, the Scratch environment will be unfamiliar and we will invest effort in learning the details of how to use it. Very quickly you will find that you have mastered the environment and we will progress from creating very simple animations to more complex ones. Even the complex animations will be built step-by-step to help you learn the new concepts and constructs. Scratch is not used in professional software development, but the concepts that will learn as you create projects in Scratch will appear again and again when you proceed to study more advanced computer science.

Example 1

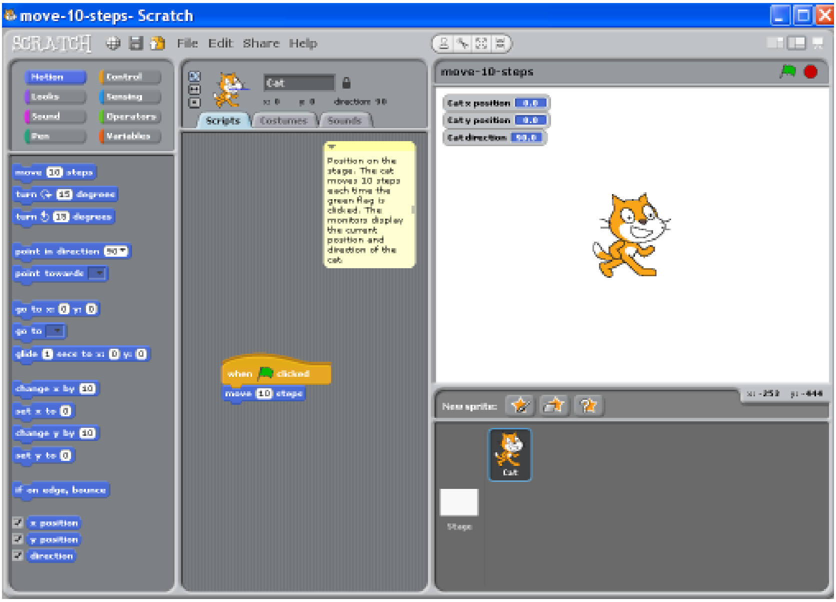
Our first animation in Scratch

Our first encounter with Scratch will be with a very simple animation that we have prepared for you. Open the Scratch environment by double-clicking on the orange-and-white cat icon on the desktop. The Scratch window will open. At the top of the window you will see four words: File, Edit, Share, Help. Click on File to open a menu of commands and then click on the line Open.... You will see a list of folders that appears beneath the folder Projects. Click on the folder ch01 that is associated with this first chapter, and then click the button OK. Within the list, click on project move-10-steps that contains the first example and click again on OK.

Program file name: move-10-steps

A description of the Scratch window

The Scratch window is displayed below. On the right side you will see a large white square. This is the ***stage*** upon which the figures of the animations will move. The figures are called ***sprites***. In this example there is one sprite, the Scratch cat. In the top left corner of the stage, we have arranged to display the position and direction of the cat sprite who is standing at the center of the stage at a point numbered $(0, 0)$ and facing to the right in the direction 90° (90 degrees).



Position on the stage. The cat moves 10 steps each time the green flag is clicked. The monitors display the current position and direction of the cat.

```
when clicked
move 10 steps
```

Cat x position 0.0
Cat y position 0.0
Cat direction 90.0

The stage measures 480 units wide by 360 units high. The positions of the center and the four corners of the stage are shown in Figure 1.1.

Below the right corner of the stage, the current position of the mouse pointer is displayed, for example, $x : 100 \;\; y : -50$. Move the mouse pointer around without looking at the display and try to guess where it is; then look at the corner display and see if you were approximately correct. Drag the cat with your mouse and see how his position changes. Before continuing, return the cat to the center of the stage, as near as you can to $(0, 0)$.

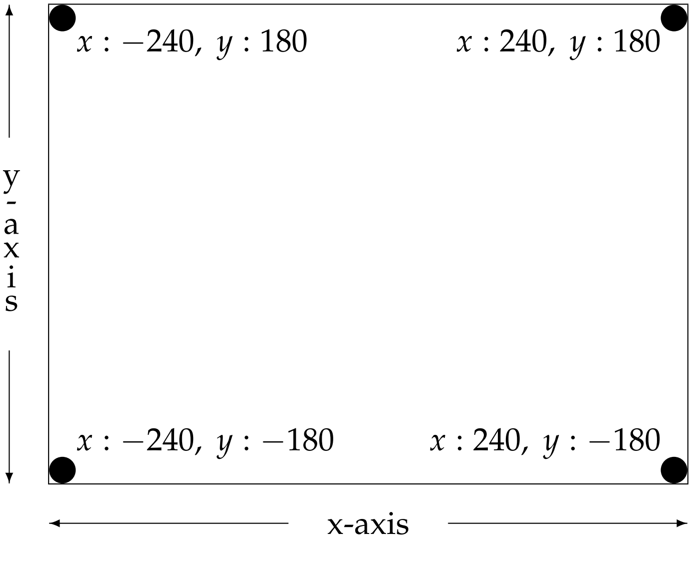


Figure 1.1: x- and y-positions on the state

Running an animation

Click on the green flag that appears above the right corner of the stage.

? What happens?

The cat has changed its position and moved a little bit to the right. You can see this in the display of the position (the top left corner of the stage): on the x-axis the cat has now moved to position 10, while before its position was zero. The position on the y-axis has not changed nor has the direction.

Click again on the green flag.

? What happens now?

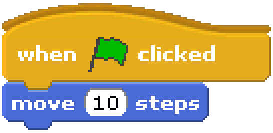
Again, the cat has changed its position and moved 10 steps to the right. Clicking repeatedly on the green flag will cause the same action to take place again and again: a movement of the cat 10 steps to the right on the x-axis. This will continue until the cat reaches the edge of the stage and (almost) disappears. If you continue to click the green flag, nothing will happen.

Scripts

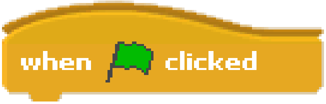
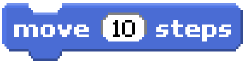
Look at the long gray panel that appears to the left of the stage in the center of the Scratch window. This panel contains the ***script*** that is run in order to move the cat.

Every sprite in Scratch has a set of scripts that describe its behavior. Sprites do not act as they wish, but only according to clear instructions in the scripts that we write for them.

As you might expect, the script for the cat in this project is very simple:



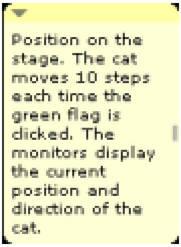
```
when 🏁 clicked
move 10 steps
```

It consists of two blocks containing instructions. The first  when 🏁 clicked means that the following instructions will be run when you click the green flag with the mouse. The second  move 10 steps means that the sprite (the cat) must move 10 units on the stage in the direction it is facing.

New concept: Instruction, running a program
An ***instruction*** causes a sprite to perform an action. Instructions must be clear and unambiguous.
When a program is ***run*** the actions specified in the instructions are carried out.

The instruction move to there is ambiguous because we don't know where "there" is. Such an instruction could not be part of a program. Instead, the instructions have to be very clear like `move 10 steps in direction 90`.

The yellow box that appears in the script area contains a ***comment***, which is a description of the script in English. Comments are very important because they help the reader understand the Scratch program. While you understand what a script does when you write it, a week or two later, you might not remember it so well. If someone else tries to work with your scripts, she will be glad to see the scripts explained in comments.



Position on the stage. The cat moves 10 steps each time the green flag is clicked. The monitors display the current position and direction of the cat.

New construct in Scratch: block, script, comment

A ***block*** is a colored graphical element that is used to construct programs in Scratch. Each block is labeled with an instruction that causes the computer to perform an action when the block is run.
A ***script*** is a structured collection of blocks that specifies the behavior of a sprite.
A ***program*** in Scratch consists of the definition of all the sprites (how they look) and all the scripts of each sprite (how they behave).
A ***comment*** is a textual description of what a script is supposed to do.

Comments are *not* part of the program! That is, Scratch runs the instructions in the blocks of the scripts and ignores the comments. You can write a comment that says "the cat moves left," but if you use a block that instructs the cat to move right, it will move right. Comments are very helpful when trying to understand a program, but you must always remember that a comment may not correctly describe what the instructions are doing.

Example 2

A sequence of motion instructions

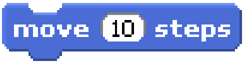
Let us make the cat's behavior more complex by having it make more than one movement when the script is run.

Task 1

> Construct an animation that causes the cat to move 100 steps and then turn 90 degrees counterclockwise each time it is run (by clicking the green flag).

Program file name: move-steps-turn-left

Changing values in instructions

First, let us increase the length of the movement that the cat makes from 10 steps to 100 steps. The instruction `move...steps` is a ***general instruction*** that does not tell us how many steps to move. In order to actually use the instruction in a script, we must create a ***specific instruction*** that specifies the number of steps to move. The script for the previous example used the block , which contains a window with the value 10. We want to change this to 100 so that the specific instruction will cause the cat to move 100 steps. Click on the window and it will change to a dark blue color. You can now type in the number 100, which will replace the number 10. Click the green flag above the stage and you will see that the cat moves 100 steps. Use the mouse to move the cat to another place on the stage; click again on the green flag and see what happens.

? What happens if we enter a negative number -100 into the window in the block?

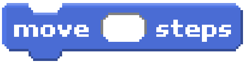
Negative numbers cause the movement to be made in the opposite direction from positive numbers. If the sprite is pointing right, a move by a positive number causes it to go to the right, while a move by a negative number causes it to go to the left. If the sprite is pointing left, the movement is reversed: positive to the left and negative to the right. To check that this is true, enter different values in the window of the move block, click the green flag, and look at the values in the display in the upper left corner of the stage. In the next section we will learn how to change the direction of a sprite and you should check again that "positive" and "negative" movements are relative to the direction of the sprite.

Before continuing, change the value in the move block back to positive 100.

New concept: general and specific instructions

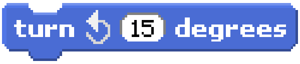
A ***general instruction*** becomes a ***specific instruction*** when specific values are added to the instruction. These values turn an ambiguous instruction that cannot be run into an unambiguous instruction that can be run.

New construct in Scratch: move

The instruction  causes the sprite to move from its current position by the number of steps given within the window. One step is one unit on the stage. If the value is positive, the movement is in the direction the sprite is facing; otherwise, it is in the opposite direction.

Changing the direction of a sprite

We now add an additional aspect to the behavior of the cat: changing its direction. Look at the long gray panel on the *left* of the Scratch window. You will see a large collection of blocks that we can use when building scripts. We will change the direction of the cat using the blue block



, which is labeled with the instruction turn and an arrow that is curved in the counterclockwise direction. (This is the third block from the top). To insert this block into the script, drag it and drop it underneath the *move* block.

> *Drag-and-drop* means do the following actions: (1) move your mouse until the mouse pointer points to the block; (2) press the left mouse button and *hold it*; (3) without releasing the button, move your mouse in the direction of the script in the middle panel; this is called *dragging* because you will see the block ”dragged” along with the mouse pointer; (4) when you reach the point where you want to insert the block, release the left mouse button and the icon for the block will ”drop” into place. A bulge in the bottom of a block will fit into a notch in the top of the block below it.

Scratch makes sure that you can only drop a block in places where it is permitted. A white line will appear at these places as you drag the block. In this script, the white line will appear when you position the mouse above the *move* block, as well as when you position it below the block, because the *turn* block is permitted in both places. Move the mouse until the white line appears below the *move* block and drop the *turn* block there.

The turn instruction is also a general instruction that has to have a value entered into its window in order to make it a specific instruction that can be run. Change the number within the window of the turn block to 90. This instruction tells the cat to turn in a counterclockwise direction by 90°.

Directions on the stage

Directions on the Scratch stage are measured around a circle like on a clock, so we can talk about *clockwise* movement and *counterclockwise* movement. Unlike a clock which divides its directions into 12 units (hours) and 60 units (minutes or seconds), directions in Scratch are divided into 360 units called *degrees* and indicated by a small circle °. Figure 1.2 shows the degrees associated with eight arrows that start at the center of the stage and point horizontally, vertically and diagonally.

The 360 degrees of the circle go from -180 degrees to $+180$ degrees. The direction 90 degrees (written 90°) points right, 0° points up, -90° points left and 180° (or -180°) points down. Given a direction D°, the opposite direction is $D^\circ - 180^\circ$ or $D^\circ + 180^\circ$, so, for example, the opposite of 45° is $45^\circ - 190^\circ = -135^\circ$, and the opposite of 0° can be written as 180° or -180°.

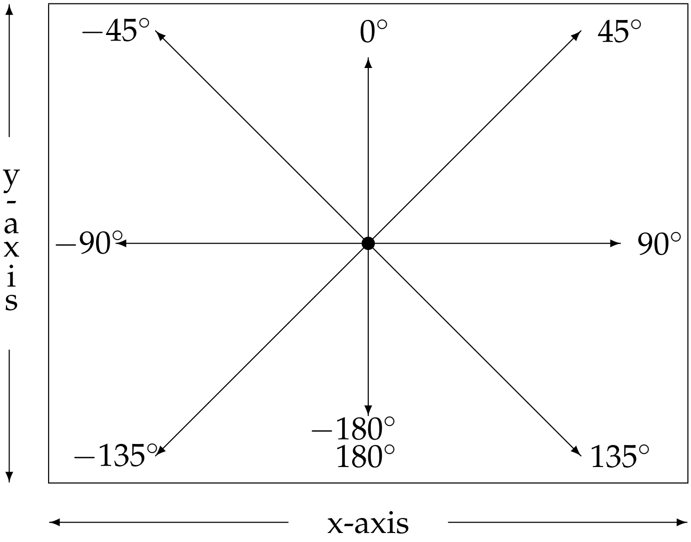


Figure 1.2: Directions on the stage

New construct in Scratch: turn

The instruction turn ↺ ( ) degrees causes the sprite to ***turn in a counterclockwise direction*** by the number of degrees that is specified in the window. Similarly, the instruction turn ↻ ( ) degrees causes the sprite to ***turn in a clockwise direction*** by the number of degrees in the window.

Sequential run

In the script for Task 1, every click on the green flag causes a *sequence* of two instructions to be run one after the other in the order they appear in the script: first the *move* instruction and then the *turn* instruction. Click on the green flag several times and see how the cat moves on the stage. Look at the display of the cat's position and direction in the upper left corner of the stage and explain how the instructions cause these values to change, and in what order they are changed.

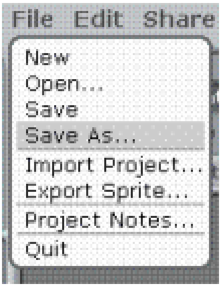
New concept: sequence of instructions and sequential run

A ***sequence of instructions*** is a set of instructions placed one after another.

In a ***sequential run,*** the instructions in the set are run in the same order that they appear in the sequence.

Before we finish working on this script, let us update the comment in the yellow box that is in the script area. Click within the box and change the text so that it explains what the new script does. It is important that you always write and update these comments so that it will be easy to understand what a script does.

Saving a project



Let us save the new project without erasing or damaging the existing one. Click the word File at the top of the window and then click the line Save As in the menu. A window will open showing the current folder; type in the name of the new project. Make sure not to use the same name as the old project, because that will replace the old one with the new one. Now click on OK to save the file; an animated frame in the Scratch window indicates that the save was carried out successfully.

Example 3
Starting an animation from a fixed place

We now extend the script for the cat to include a longer sequence of instructions.

Task 2

> Construct an animation that causes the cat to move 100 steps and then turn 90 degrees counterclockwise each time it is run (by clicking the green flag). For each run, the cat will start at the same place: the middle of the stage $(0, 0)$ facing right.

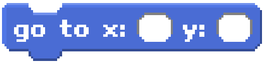
Program file name: initialize-and-move

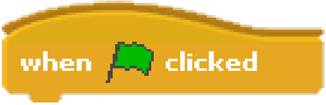
Initialization

The two scripts that we have written so far have caused the cat to move. After running each of these scripts, the cat is at a new position, so that running the script again causes the cat to move *starting from* its new position. In other words, the effect of running a script depends on the *initial position* of the cat.

Let us now develop a script that results in the *same* behavior each time it is run, regardless of the cat's initial position. To achieve this, we write a script whose first instructions move the cat to a fixed position. Look at the list of blocks in the left panel.

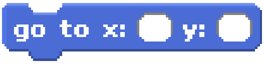
? Which instructions can be used to do this?

We can use the instruction .

Drag and drop the block with this instruction, placing it at the beginning of the script immediately after the block . Remember that a white line will appear between the two blocks to indicate that you can drop the block there. Scratch will automatically move the existing blocks to make room for the new one. The middle of the stage is at position $(0, 0)$, so if the windows within this block do not have zeros, click them and enter zeros.

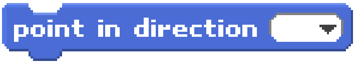
Assigning initial values in a script is called *initialization*. Here, we are ensuring that the initial position of the cat is always in the center of the stage.

New construct in Scratch: go to

The instruction  go to x: y: causes the sprite to change its position and ***go to a new position*** that is specified by the x (horizontal) and y (vertical) values in the windows.

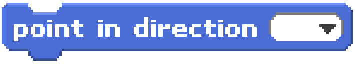
We would also like to start the cat's movement when it is pointing in a fixed direction. Following the initialization of the cat's position, we will place an instruction that will initialize the cat's direction. Look again at the list of blocks.

? What instruction do you think we can use to set the direction?

The instruction  point in direction (the fourth from the top) enables us to point the cat in the direction given by the number in the window. Drag-and-drop this block so that it appears after the block with the *go to x: y:* instruction. Check that the white area has the value 90, meaning that the cat sprite faces to the right. If not, click on the little arrow in the white area and choose *(90)-right*.

Save this script under a new name using the instruction Save As.

New construct in Scratch: point in direction

The instruction  causes the sprite to ***change the direction*** in which it faces to the direction specified in the window. By clicking on the small arrow in the window you can select one of the four main directions (up, right, down, left); alternatively, you can type in any direction (0–360) that you please.

Click on the green flag. The cat initially returns to the center of the stage facing right (if it was not there already); then it moves 100 steps to the right and turns counterclockwise until it faces upwards. Click the green flag again and again, and you will see that the cat returns to the same position every time, because it starts its movement from the same place, the center of the stage facing right. The four instructions of the script are always run in the same sequential order that they appear after the top block for the green flag.

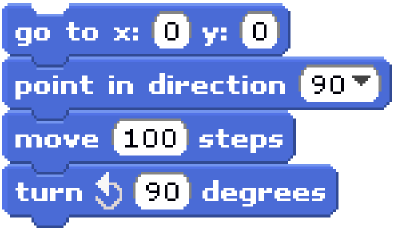
You can use the mouse to drag-and-drop the cat and place it anywhere on this stage. Move the cat to an arbitrary position on the stage and then click the green arrow. Explain what happens.

New concept: initialization
Initialization refers to instructions that set values at the beginning of a program. Most sequences of instructions will have an initialization part consisting of several instructions that ensure that the sequence always starts running from the same state.

Absolute and relative motion

The first two instructions in the script for this task are very different from the second two instructions.

? Can you explain the difference?



```
go to x: 0 y: 0
point in direction 90▾
move 100 steps
turn ↺ 90 degrees
```

The instructions *go to x: y:* and *point in direction* are called ***absolute instructions***, because the motion that results is the same no matter where the cat is and in what direction it is pointing. The result depends only on the values that are given to the instructions (in the windows). For example, *go to x: 0 y: 0* will always return the cat to the center of the stage and *point in direction 180* will always cause the cat to face the bottom of the stage.

The instructions *move* and *turn* are called ***relative instructions***, because the motion that results is relative to the current position and direction of the cat. That is, although the number of steps that the cat takes and the number of degrees it turns are specified in the windows of the instructions, the final position and direction of the cat depend on its current position and direction. For example, if the cat is pointing to the right, the instruction `move 100 steps` will cause it to move 100 steps to the right, while if the cat is pointing up towards the top of the stage, the instruction `move 100 steps` will cause it to move 100 steps upwards. Similarly, the cat is pointing upwards, the instruction `turn counterclockwise 90 degrees` will cause it to face the left side of the stage, while a second run of this instruction will cause it to point downwards.

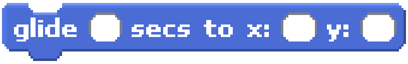
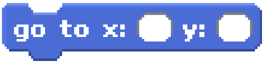
New concept: absolute and relative instructions
An ***absolute instruction*** is one where the result of running the instruction does not depend on the current state, but only upon the values specified in the instruction.
The result of running a ***relative instruction*** depends both upon the values specified in the instruction and the current state.

Example 4
Continuous motion

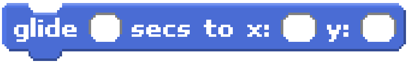
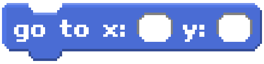
Task 3

> Construct an animation that causes the cat to move around the stage, starting in the lower left corner and returning there.

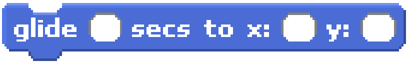
Program file name: move-around-stage

The scripts that we wrote for the previous tasks cause the cat to move but you can't really see the motion of the cat because it happened so fast. There is another instruction  glide ( ) secs to x: ( ) y: ( ), which is similar to  go to x: ( ) y: ( ), except that it causes the motion to be gradual so that it can be seen. The `glide` block (the eighth block from the top) has three windows that must be filled with values to make it into a specific instruction: the first value is the duration of the gliding motion, that is, the number of seconds that the movement will take. The second two windows are for the x and y positions *to which the sprite will move*. You will have to experiment with the value for the duration to see which one gives the best visual display.

? Is the glide instruction an absolute instruction or a relative instruction?

Let us compare  glide ( ) secs to x: ( ) y: ( ) with  go to x: ( ) y: ( ). The `goto` instruction is clearly an absolute instruction because its only effect is to move the sprite immediately to a final point, regardless of where it currently is. The `glide` instruction achieves the same final state but the time of gliding means that we are interested in the appearance of the sprite as it moves to the final point. This depends on the current state: if the sprite is close to the final point it will move slowly, while if it is far away it will move fast. We see that the `glide` instruction is a relative instruction because its action depends on the current state of the sprite.

New construct in Scratch: glide

The instruction  causes the sprite to ***move to the position*** specified in the windows labeled x and y. The ***time*** it takes to reach this position from its current position is specified in the first window.

Writing a description of the script for this task

Let us use the instructions that we have learned so far to write the script that will cause the cat to travel completely around the stage. The cat will start in the lower left corner and move in a counterclockwise manner around the stage.

Before writing the actual script in Scratch, take a piece of paper and draw the movement of the cat on the stage. This will help you understand the sequence of instructions that must be run for the cat to successfully complete its journey. Next, make a list of the separate motions that the cat must do. The list of movements will be as follows:

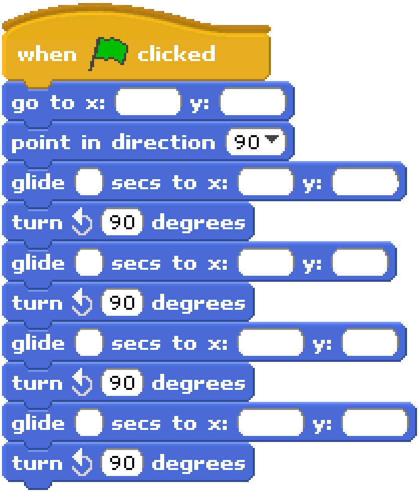
1. *move the cat to the lower left corner of the stage*
2. *point the cat to the right*
3. *the cat moves to the lower right corner of the stage*
4. *the cat turns to face upwards*
5. *the cat moves to the top right corner*
6. *the cat turns to face left*
7. *the cat moves to the top left corner*
8. *the cat turns to face down*
9. *the cat moves to the bottom left corner*

New concept: a description of the behavior of a sprite
One of the first steps in developing a programming project is to write out in words a ***description*** of the actions to be performed by the program. A description is written as a ***sequence of steps***. The description must be clear and unambiguous, although it need not be fully detailed.

You will be tempted to skip over the step of writing a description of a program; it is much more fun to start programming in Scratch immediately. However, experience has shown that as programs get longer and more complex it becomes extremely important to think about the design of the program first and to write down the design. Don't give into the temptation to start programming right away!

Constructing a script for this task

This description can be easily translated into a Scratch script. Here it is (but without the specific values in the *go to* and *glide* blocks):



```
when ⚑ clicked
go to x: ( ) y: ( )
point in direction (90 ▾)
glide ( ) secs to x: ( ) y: ( )
turn ↺ (90) degrees
glide ( ) secs to x: ( ) y: ( )
turn ↺ (90) degrees
glide ( ) secs to x: ( ) y: ( )
turn ↺ (90) degrees
glide ( ) secs to x: ( ) y: ( )
turn ↺ (90) degrees
```

We have not yet learned how to place the block with the green flag in the script, so let us start with an existing script and remove all its blocks except for the top one. Bring your mouse pointer until it is pointing to the first block just below the one with the green flag. Press and hold the left mouse button; you will see that the block you clicked on and *all the ones below it* can now be dragged. If you drag them into the left panel with the list of blocks and drop them, you will see that they disappear. Now you can create a new script with a sequence of nine blocks corresponding to the list we made above.

The position of a sprite refers to its center

The *position* of a sprite usually refers to its *center* when it is displayed on the stage. Since we do not want the sprite to disappear, make sure not to move it too close to the edge of the stage. To find how far a sprite can be moved without disappearing, place the mouse cursor on its center, click and hold down the left mouse button. Move the sprite to an edge of the stage. Just before it begins to disappear, make a note of the x or y position of the mouse position as displayed below the right corner of the stage. You can always change the values of move instructions to correct the position.

Running the script

Test the script by running it (clicking on the green flag) and observing if the motion of the cat is what it is supposed to be. If not, look very carefully at each of the instructions in the script and compare it with the list of actions that you wrote down. It is very easy to make a mistake like writing 9 for 90 or 100 for 10.

Add a comment to explain what the script does. To do so, point the mouse to an empty area on the gray background of the script area (the middle panel) and click the *right* mouse button. You will see a small menu whose third entry is add comment; click on that entry with the left mouse button. A yellow window will open and you can write your comments there.

When you have finished, be sure to save the project under a new name.

Exercise 1

> Create a script that causes the cat to start in the upper right corner and travel around the stage in a clockwise manner.
>
> **Guidance:** Draw the movement on a piece of paper and make a list of the separate movements that are required so that the cat can complete its journey successfully.

Program file name: move-around-stage-clockwise

Make it a habit to plan your animations before you write the scripts in Scratch; you will find that this will simplify your work, especially when the animations become more complicated.

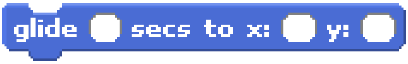
Summary

In this chapter, we met Scratch for the first time and we learned basic concepts of computer science, as well as constructs for writing programs in Scratch.

Concepts

Sprites: Scratch in an environment for creating animations of ***sprites***. Sprites move on a ***stage***. The positions on the stage are described by x- and y-positions from -240 to 240 along the x-axis and from -180 to 180 on the y-axis. The center of the stage is at the point $(0, 0)$. At any time when a Scratch program is run, each sprite has a ***position*** as well as ***direction*** in which it faces.

Sprites do not move by themselves; they only move according to ***instructions***, which are collected into sequences called ***scripts***. The instructions are run ***sequentially***, one after another, in the order that they appear from top to bottom in the script. Scripts can begin running when the ***green flag*** is clicked.

General and specific instructions: Many instructions are ***general instructions***, for which we need to supply values in order to obtain a ***specific instruction*** that can be run. For example, the  glide ○ secs to x: ○ y: ○ instruction needs three values to turn it from a general instruction to a specific instruction: the number of seconds of the glide, and the x- and y-values of the position that the sprite will glide to.

Absolute and relative instructions: There are two types of instructions: ***absolute instructions*** whose effect is entirely determined by the values that are supplied in the instruction itself. For example, the absolute instruction go to x: 100 y: 100 causes the sprite to move to position $(100, 100)$, regardless of the sprite's current position motion. The result of running a ***relative instruction*** depends both on the values in the instruction and on the current state of the sprite. For example, the instruction move 10 steps causes the sprite to move 10 steps in the direction it is *currently facing*, starting at its *current position*.

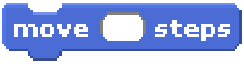
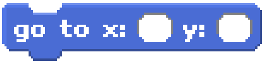
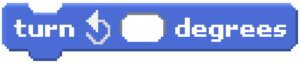
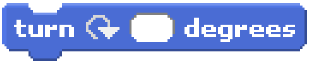
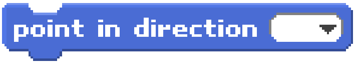
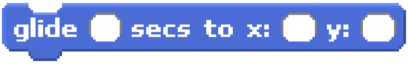
Initialization: We usually want the behavior of a sprite to be the same each time it is run. This requires that we give initial values to the state of the sprite such as its position and direction. The instructions placed at the beginning of a script so that it will always start running in the same state are called the ***initialization*** of the script.

Written description: Before beginning to construct scripts from Scratch blocks, a ***description*** of the behavior of the sprites should be written down in English. This gives an overview of the behavior without going into the details such as the actual positions and directions. Descriptions are written as a set of ***steps*** that need to be run to achieve the required behavior of the sprites.

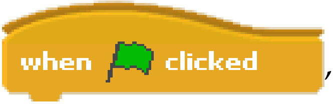
Comments: Comments explain parts of programs in written English. They help the reader of the program understand the program but have *no* effect on the running of the programming.

Scratch instructions

Motion instructions: Most of the blocks we have used so far are used for motion instructions from the blue Motion palette:

-  move steps: the sprite moves a number of steps in its current direction;
-  go to x: y:: the sprite moves to a specific position;
-  turn degrees : the sprite turns a number of degrees counterclockwise;
-  turn degrees: the sprite turns a number of degrees clockwise;
-  point in direction: the sprite turns to face in a specific direction;
-  glide secs to x: y:: the sprite moves to a specific position in a specific amount of time.

These are all general instructions and you have to turn them into specific instructions by supplying values such as positions and directions.

Control instructions: When the first block of a script is , it causes the script to begin running when the green flag is clicked.

Scratch techniques

Blocks: Scripts are constructed using ***blocks*** containing instructions. The construction is accomplished by ***dragging and dropping*** the blocks: dragging the blocks using the mouse and dropping them at an appropriate place.

Scripts from blocks: Blocks fit together to form scripts. A bulge in the bottom of a block fits in to a notch in the block below it.

Comments: Comments can be attached to scripts by right-clicking in the stage area and selecting `add comment`. The text of the comment is written in the light yellow box that appears.

Chapter 2

Multiple Sprites

The animations that we developed in the previous chapter had a single sprite. In this chapter, we will develop animations that have multiple sprites.

Example 1
An animation with two sprites

Task 1

> Construct an animation with two sprites: the first is our familiar cat and the second will be a dog. The cat and the dog will appear on the stage and move concurrently (that is, simultaneously, at the same time). The cat will cross the stage from left to right, while the dog will cross the stage from bottom to top.

Program file name: cat-meets-dog

We will create the animation in stages, starting with the cat and its script, and then will we construct the dog and its script.

Creating a new script

In the last chapter, we started with an existing sprite and its script, and made modifications to the script. Here, we will build the entire animation ourselves. When Scratch first begins to run, it shows the cat sprite (called `Sprite1`), but the middle panel has no scripts in it. Since the cat is one of the sprites that we require, we do not need to change the image. We do need to construct the script that will cause the cat to move as required.

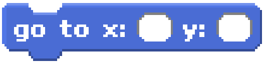
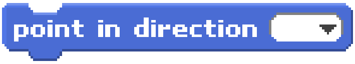
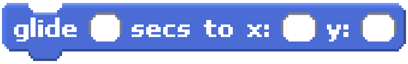
> At any time when you are working with Scratch, you can start a new project by clicking on the menu File and then selecting New.

? What are the elements of the motion of the cat?

First, we have to position the cat at its initial position at the *left edge* of the stage facing to the right; then, we have to tell the cat to move to its final position. A written description is:

1. *move the cat to its initial position*
2. *move the cat to its final position*

? What Scratch instructions can implement these two steps?

Step 1 requires two absolute instructions:  `go to x: ( ) y: ( )` will move the cat to its initial position and  `point in direction ( ▾)` will cause it to point it in its initial direction. The second step uses the relative instruction  `glide ( ) secs to x: ( ) y: ( )` to move the cat to its final position. Drag and drop blocks with these instructions to the script area in the middle of the Scratch window. We need to enter values in order to change these general instructions into specific instructions.

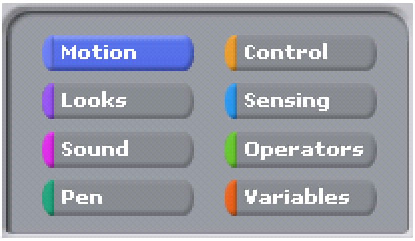
? What values need to be stored in the windows?

Look again at the diagram of the stage in the previous chapter. The x-value of the left edge of the stage is -240, but we will give the cat an initial x-position of -200 so that we can see the entire cat. (The position of a sprite is the *center* of the image used to display it.) The y-value of the middle of the stage is 0 and we can use this value because there is enough room above and below the middle to display the entire cat. The initial direction is 90° so that the cat points to the right. We leave it to you to enter appropriate values in the windows of the glide instruction.

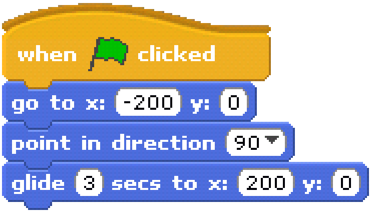
Control instructions

We are almost done, but not quite. Recall that all our scripts had a orange block at the top that enables a script to start running when the green flag is checked. In the palette that appears when Scratch is run, there are no orange blocks, only blue blocks whose instructions relate to motion.

> The left gray panel is called the ***blocks palette***. A palette is a tray used by a painter to hold paints of various colors; the painter dips a brush into one of the paints on the palette and then draws on a canvas. In the same way, you "draw" a script by "dipping" your mouse cursor into a block on the palette!



Look above the set of blue blocks and you will see a small rectangular area with eight buttons in it. There is a blue button labeled Motion and another seven buttons whose left edges are colored. Click on the button with the orange edge labeled Control. The button will now be colored completely orange and in the blocks palette you will see a long list of orange blocks. The first block in the list contains the instruction `when green flag clicked`, which is exactly what we need. Drag this block and drop it at the top of the sequence of motion blocks that you already built in the center panel. The full script for the cat is:



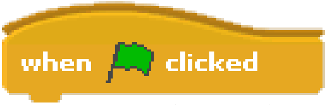
```
when ⚑ clicked
go to x: -200 y: 0
point in direction 90▾
glide 3 secs to x: 200 y: 0
```

The order in which you place the blocks within the script doesn't matter; only the final form of the script is important. So you could have started the script with the orange block and then added the motion blocks.

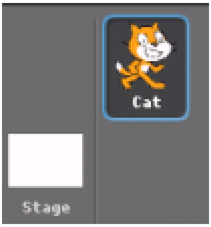
Blocks that can only appear at the start of a script

There is something strange about the orange block we added to the script. The *top* of the block is curved and does not match the edge of any of the other blocks. This is intentional and ensures that the instruction that starts the running of a script can only be the first instruction in a sequence. No no block has a curved *bottom* edge that could match the top of this one.

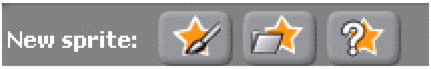
New construct in Scratch: start running a script (green flag)

The instruction  indicates that the the instructions beneath it will be ***run*** when the green flag is clicked. This block can only appear at the beginning of a script.

Adding a new sprite



Let us now add the new sprite. There is a rectangular gray panel at the lower right corner of the Scratch window; this panel is used to add and modify the sprites in the animation. On the left you can see a small white rectangle labeled Stage; this is the white background of the stage and later we will learn how it can be changed. On its right you can see a small icon that represents the cat sprite that already exists in our animation. Just above the panel you will see three buttons:



Click on the middle button which is used to select an existing image for a new sprite. Click on the folder labeled Animals to select it and then click OK. (The folder Animals will probably already be selected when the menu is shown; you can tell by the blue background around the icon for the folder.) Scroll down through the display of animals until you find images of dogs; click on the third image, dog2-c, which shows a walking dog.

> To ***scroll*** through the images, bring your mouse cursor until it points to the vertical button at the right side of the menu; press and hold the left mouse button and drag the vertical button up

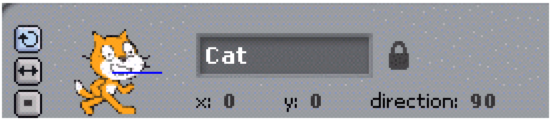
and down until you get to the area that you want; now release the button.

In the sprite area at the lower right of the Scratch window, you will see a small icon with the image of the dog. If you click on this icon, the script panel in the middle of the window will show the script for the dog sprite. Of course it will be empty because we have yet to construct a script for the dog. Practice clicking on the cat and the dog icons and see how the script area changes. There will be a frame around the icon of the sprite that was last selected and whose script appears in the script area.

New construct in Scratch: creating a new sprite

The middle button above the sprite area enables you to ***add a new sprite*** to your program. You can choose any one of the images in the Scratch library or you can choose any image that you wish from one of your folders. Most images formats like GIF, JPG and PNG can be used.

Naming sprites



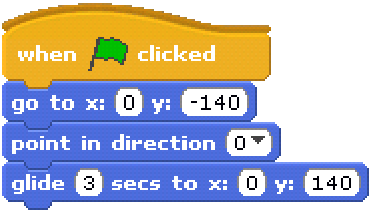
Cat

x: 0 y: 0 direction: 90

Before continuing, let us give names to our sprites instead of the names `Sprite1` and `Sprite2` that Scratch gave for us. Above the sprite panel in the center of the window, you will see a small rectangular area with the icon for the sprite and to its right the name of the sprite. Click here and type in a new name such as `Cat` or `Tom` for the cat, and `Dog` or `Rover` for the dog. Remember that you will have to click on the icons in the sprite area in the lower right corner to change the display from one sprite to another.

Construct the script for the dog

Click on the icon for the dog sprite to select it. Use the script for the cat as a model and change the values of the instructions so that they cause the dog to move from the bottom center of the stage to the top. Here is the script that we constructed:

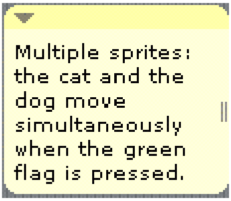


```
when 🟩 clicked
go to x: 0 y: -140
point in direction 0▾
glide 3 secs to x: 0 y: 140
```

When you are done, click on the green flag; what happens?

Both the dog and the cat sprites move at the same time!

You can see how the two sprites move, how they meet at the center of the stage and how the dog passes over the cat.



Multiple sprites: the cat and the dog move simultaneously when the green flag is pressed.

It is important to document every script with an appropriate comment. Move the mouse to an empty (gray) area in a script and right-click; select add comment from the menu that appears. Write your comment in the yellow box that appears.

Save the project that you have just constructed.

New concept: concurrency—sprites running concurrently

When a program is run, the scripts of all sprites are ***run concurrently***. This means that the scripts are run at the same time (one instruction after another) without one script waiting for another to finish, or being dependent on one another.

Exercise 1

Everyone knows that dogs like to chase cats. Construct an animation that has both the cat and the dog start in the lower left corner of the stage. The cat moves to the upper right corner and the dog chases after it.

Guidance: The script for the dog will have three parts: first, the initialization; then, the dog will slowly chase the cat until the dog reaches the center of the stage; finally, the dog will chases the cat faster and catch up with it. You can use glide instructions to cause the dog to chase the cat. Adjust the times of the glides to achieve the desired effect.

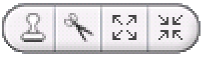
Program file name: dog-chases-cat

Additional material on Scratch: Mouse modes

In this section we describe some advanced ways of manipulate sprites and scripts using the mouse. You can skip this for now and return to it whenever you want to.

You will certainly have noticed that the script for the dog was exactly the same as the script for the cat, except for the values entered into the windows of the blocks. Instead of constructing the script for the dog block by block, it is easier to *duplicate* the script for the cat, move it to the script area of the dog and then change the values in the instructions.

Duplicating a script



Ensure that the cat sprite is selected and that its script appears in the script area. Above the left corner of the stage, you will see a toolbar consisting of four buttons. Click on the left button, whose image looks like a rubber stamp; this changes the mouse to work in ***duplicate mode*** and the mouse cursor turns into a small rubber stamp. Now, bring the mouse cursor to the cat's script, press the left button and hold it; a new copy of the script will appear to the sprite area and you can drag it. Drop it on the icon for the dog. Note that the mouse cursor has returned to its normal shape and the arrow button has been reselected. Click now on the dog icon and you will see that you have a copy of the cat's script in the dog's script area.

Whenever you drag and drop a script, the block that the mouse pointer points to *and all blocks below it* are moved. Therefore, to drag-and-drop an entire script, make sure that the mouse pointer points to the first block, usually the one with the orange block for *`when green flag clicked`*.

The duplicate button can also be used to duplicate a sprite together with all its scripts and costumes (see below). Click on the button so that the rubber stamp icon appears; now click on a sprite in the sprite area below the stage. A new copy of the sprite will appear, although it will have a different name.

New construct in Scratch: duplicating sprites and (portions of) scripts

The left button above the stage can be used to ***duplicate a (portion of) a script***. Click on the button and then press and hold the left mouse button on a block in a script. This will create a copy of the block and all blocks below it. Drag-and-drop the copy blocks to someplace in the same script area or to another sprite.
The button can also be used to ***duplicate a sprite***. After clicking on the duplicate button, click on a sprite or its icon in the sprite area and another copy will appear. You can drag and drop either the sprite or its icon to move them.

Erasing a sprite and changing the size of a sprite

The toolbar has three other buttons. The second button from the left is used to delete a sprite or a script ✂. The two buttons on the right can be used to increase or decrease the size of a sprite on the stage. Experiment with these buttons, but be sure to save the project first so that you can load a fresh copy of the project later.

The buttons duplicate and delete behave

somewhat differently from the increase and decrease buttons. Once you have duplicated or deleted a sprite or a script, the mouse cursor returns to its original form, and you will have to click on the same button to perform the action again. After increasing or decreasing the size of a sprite, the mouse remains in that mode so you can increase or decrease the size repeatedly. To return to the normal mouse mode, click the left mouse button when the cursor points to an empty area of the stage.

New construct in Scratch: changing the size of sprites, deleting sprites and (portions of) scripts

To ***increase or decrease the size of a sprite***, click on ⤢ or ⤡ and then click (repeatedly if necessary) on the sprite.

To ***delete a sprite***, click on ✂ and then on the sprite. To ***delete a (portion of) a script***, click on ✂ and then on a block; the block and all the blocks below it will be deleted.

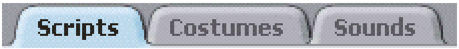
Additional material on Scratch: Changing costumes

You don't have to use the initial images of the sprite that are supplied by the Scratch software. The images, called *costumes*, can be changed, or you can create your own. Modifying the appearance of the sprites is fun, but we suggest that you don't spend too much time doing so.

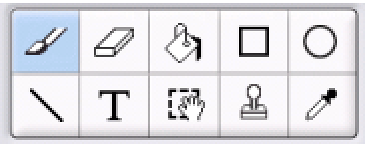
Task 2

Change the costumes of the cat and the dog.

Program file name: costumes



Click on the icon of the cat's sprite in the sprite area in the lower right of the Scratch window. The script panel (the gray panel in the middle of the Scratch window) has three tabs at the top. When the tab labeled Scripts is selected, the scripts for the sprite are displayed. Click now on the tab labeled Costumes. The gray panel displays a list of the costumes of the currently selected sprite. Click on the image of the cat's costume and then click on the button labeled Edit. This opens a window labeled Paint Editor, which is similar to the Paint program in Windows. We will talk you through one modification of the costume for the cat and then let you experiment with other options when you change the costume for the dog.



In the upper left corner of the window there is a double row of buttons. Click on the Fill button, which is the middle one in the upper row; it looks like a can of paint with paint spilling out of it. Now click on one of the colors in the lower area. Move your mouse pointer to the image of the cat displayed in the grid in the right area of the window. Click somewhere within the orange head of the cat; it will now change to the color you have chosen. Click on another color and then click within the black nose of the cat; the color of that area will change also. Don't be afraid to make a mistake, because you can always go back by clicking on the Undo button just above this set of buttons.

New construct in Scratch: editing a costume

Clicking on the tab labeled Costumes in the script area causes the set of costumes for the currently selected sprite to be displayed. Select one costume by clicking on its icon and then click on Edit. The Paint Editor is shown in a new window and can be used to modify the image of this costume.

Summary

Concepts

Concurrency of multiple sprites: Several sprites can take part in a Scratch animation and each sprite has its own scripts. Clicking on the green flag causes the scripts of all sprites to be run concurrently—at the same time.

Scratch techniques

Building scripts from blocks: New scripts are created by dragging and dropping blocks from a palette. The blocks in Scratch are displayed on the ***blocks palette*** at the left of the screen. You can select which list of blocks you want displayed by clicking on the colored buttons above the palette. In addition to the ***Motion instructions*** (colored blue), we used an instruction from the ***Control instructions*** palette (colored orange): the instruction *`when green flag clicked`* that causes the script to start running.

Creating sprites: You can create additional sprites by selecting the image of the sprite after clicking on the the middle button above the sprite area. Each sprite has its own script. Clicking on the icon of the sprite causes its script to be displayed. The name of each sprite can be changed in the area above the script area.

Mouse actions menu: Clicking on one of the buttons of the toolbar just above the stage changes the mode of the mouse. Selecting one of these buttons causes the mouse to: (1) duplicate sprites and scripts, (2) delete sprites and scripts, (3) make sprites larger, (4) make sprites smaller. To return to the normal mouse mode, click outside this menu.

Costumes: The appearance of a sprite in Scratch is determined by its costume. New costumes can be created and existing costumes can be modified by selecting the Costumes tab above the script area. Costumes can be imported from a file, or they can be created and modified using the Paint Editor.

Chapter 3

Short Scripts, Long Runs

The animations we have created share a common characteristic: their scripts were finite sequence of motion instructions. The instructions in the sequence were run one after another when the green flag was clicked. They included a variety of instructions (absolute motion, relative motion), and they were of different lengths, from a sequence of one instruction to a sequence of nine instructions. Nevertheless, the general structure of all the scripts was similar, and the animations, too, were very similar: a finite sequence of movements that occurred one after another, always in the same order. The number of movements was always the same as the number of motion instructions in the script. In this chapter, we will enrich the animations that we create by putting the basic instructions together in new ways.

Example 1
Infinite Run

We want to create an animation that runs indefinitely without stopping. Of course, it is impossible to write an infinite sequence of instructions, so we need to learn how to create an infinite run from a finite set of instructions. We will start with the last animation from the previous chapter, where the cat and the dog sprites cross the stage, one from left to right and the other from bottom to top.

Task 1

> Modify the animation from the previous chapter so that the cat and the dog cross the stage again and again without stopping: the cat will move continuously from left to right and then right to left, and the dog will move continuously from bottom to top and then top to bottom.

Program file name: meet-forever

We break down the task into parts and deal with each one by itself. First, let us analyze the motion of the cat. To start, we place the cat at its initial position and facing in its initial direction. Now we wish to construct a sequence of instructions that cause back-and-forth motion, such that when the cat reaches an edge of the stage it turns around and faces in the other direction. The movement of the cat will be constructed of short movements (say 10 steps) repeated again and again. After each movement, the cat will check if it is touching the edge of the stage, and, if so, it will change direction to face in the opposite direction.

A description of the program is as follows:

```
0. when the green flag is clicked
    1. go to the starting position ($x = -200, y = 0$)
    2. face in the initial direction (90°, right)
    3. repeat again and again
        3.1 move 10 steps
        3.2 if you have reached the edge, turn around
```

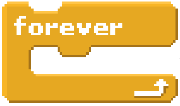
We already know how to translate steps 0, 1, 2 and 3.1 into Scratch instructions. This leaves instructions 3 and 3.2.

For step 3.2 there is a motion instruction if on edge, bounce which appears as the fourth block from the bottom in the blue palette.

New construct in Scratch: bouncing at the edge of the stage

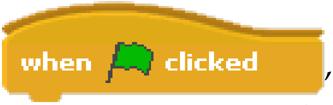
The instruction if on edge, bounce causes the sprite to ***reverse its direction*** if it is touching the edge of the stage. For example, if it is moving in direction 90° (right) and is touching the edge, it will turn to face in direction −90° (left).

Repeated run of instructions

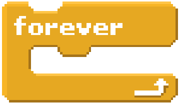
For step 3, we need an instruction that doesn't modify the animation by itself, but rather causes other instructions to be ***run repeatedly***. Instructions that affect the sequence in which other instructions are run are called ***Control instructions***, because they are used to control the running of other instructions. These instructions can be found in the orange palette. The instruction that we want is ***infinite run*** instruction defined by the fourth block from the top: 

.

The structure of this block is different from the previous blocks that we have used. They had small bulges and sockets that enabled them to be joined one on top of another to form a sequence of instructions.

The one exception was the block 

, which can only be the *first* block in a sequence since it has no socket on the top.

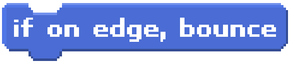
The 

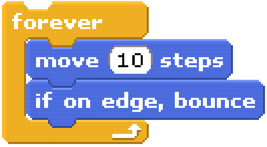
block has a socket on top so that it can appear after another block, but it does not have a bulge on the bottom, so no block can be placed after it.

? Why can't you add a block after the `forever` block.

You can't add another block because the `forever` block runs "forever" so no block that appears after it would ever be run.

The block looks like a "mouth" that is open on its right side. This enables the block to enclose other blocks that are to be run repeatedly. Let us see how to construct step 3 of the animation described above.

Drag the *forever* block from the palette and drop it in the script area. Now change to the Motion palette (the blue one) and drag the block for the *move* instruction so that it is *within* the "mouth" of the *forever* block. As usual, you will see a white line that indicates that you can drop the block there. Now locate the block  `if on edge, bounce` and drag it, dropping it after the *move* block but still within the *forever* block:



```
forever
    move 10 steps
    if on edge, bounce
```

The meaning of this structure is that the two instructions corresponding to the blocks enclosed within the *forever* block will be run again and again, forever.

Complete the script by adding the blocks for instructions that implement steps 0, 1, 2 *above* the *forever* block. Click on the green flag and check that the cat moves indefinitely without stopping, and that when it reaches the edge of the stage it changes direction.

Although the script is constructed from only six instructions, the animation runs indefinitely without stopping. This is because the infinite run instruction causes two other instructions to be run again and again. Any animation, even an infinite one, can be stopped by clicking on the ***red stop sign*** that is next to the green flag.

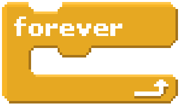
Create a similar script for the dog sprite. It will be the same as the script for the cat, except that the dog will start in a different position, facing in a different direction. Click on the green flag and you will see both the cat and the dog sprites moving at the same time indefinitely.

Save your animation as a new project.

> **New concept: infinite run**
> An ***infinite run*** occurs when some instructions are run again and again without indefinitely.

Infinite runs are common in programs like web browsers that are always willing to accept a new request from the user.

New construct in Scratch: infinite run

The instruction 

causes the instructions included within its "mouth" to be ***run again and again*** without stopping. No blocks can be placed after this block.

Exercise 1

The cat is tired of being chased by the dog and calls his relative, the wildcat, to scare the dog away. Initially, the wildcat is positioned in the center of the stage and faces the dog. The dog jumps around the wildcat (above, to its right, below, to its left) each time barking at the dog. Whenever the dog jumps, the wildcat turns to face it; the wildcat is so scary that the dog continues jumping forever.

Guidance: Write precise descriptions of the behavior of the wildcat and the dog, and then translate them into Scratch scripts. Run your project, document it with comments and save it. An appropriate image for the wildcat sprite is cat4 in the folder Animals. You will want to use the Scratch instruction point towards ▼ that causes the sprite to point towards another sprite (or the current position of the mouse cursor).

Program file name: big-cat

Example 2
infinite run with a condition

In the animation of the previous example, the cat and the dog pass each other without any interaction. Animations are much more interesting if there is some interaction between the sprites that take part.

Task 2

The cat and the dog will start from the center of the stage and move towards its edges. The dog will move upwards from the center, while the cat will move to the right. When the two sprites are touching, they will say something to each other even as they continue their motion. When they move far enough away so that they are no longer touching each other, they will stop their motion and stop speaking.

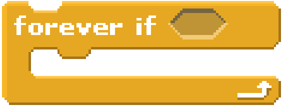
Program file name: say

To make it easy to observe the interaction, we will increase the amount of time that they are near each other by slowing down the motion: each small movement of the sprites will be for 5 steps instead of 10.

In this animation, we are interested in running a sequence of instructions again and again, but some of the instructions (the ones that cause the cat and the dog to speak) will only be run if the sprites are touching each other. A description of the actions of the cat is as follows:

```
0. when the green flag is clicked
    1. go to the starting position ($x = 0$, $y = 0$)
    2. face in the initial direction (90°, right)
    3. repeat again and again, if you are touching the dog
        3.1 move 5 steps
        3.2 say Help
```

To construct this script in Scratch, we have to find instructions for steps 3 and 3.2. Let us start with step 3.

The instruction for step 3—repeated run together with the check—can be found in the orange palette of Control instructions. The appropriate block is 

which is the 10th block from the top.

This instruction is similar to the *forever* instruction except that it contains a ***condition*** that must be true if the enclosed instructions are to be run. (The condition is entered in the window with the angled ends, as explained below.) The condition is checked each time *before* the enclosed instructions are to be run. The instructions are run if the condition is true. If the condition is false, the enclosed instructions are not run, though it is possible that later on the condition will become true and the enclosed instructions will be run the next time the condition is checked (see Example 3, below).

The condition "touching?"

What remains to do is to implement the second part of step 3 which ensures that steps 3.1 and 3.2 are run only if the cat and dog sprites are touching. In Scratch, every sprite has ***sensors*** which can sense various situations. Here, we want the cat sprite to sense if it is touching the dog sprite. This can be expressed as a question:

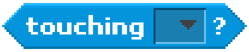
> *Am I (the cat sprite) touching the dog sprite?*

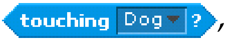
A question which can be answered "yes" or "no" is called a ***condition***.

Often we use a different terminology. A condition is a statement like:

> *I (the cat sprite) am touching the dog sprite*

that can be determined to be "true" or "false."

The blocks for conditions can be found in the light blue palette Sensing. The third block from the top is  touching ▾ ? and it has a window in which we can select the object that this sprite may be touching.

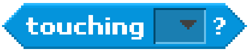
Let us now construct the script. Drag-and-drop the `forever if` block onto the script area. After the word `if` there is a window with angled ends, which indicates that a condition must be entered. Drag and drop the block `touching` into the window. Now, click on the small window within the block `touching` and select the word `Dog` from the list. The block now reads  touching Dog ▾ ?, where the question reminds you that it is asking the question:

> *Am I (the cat sprite) touching the dog sprite?*

If the answer is "yes" ("true"), instructions enclosed within the mouth of the `forever if` block will be run; if the answer is "no" ("false"), the instructions will not be run.

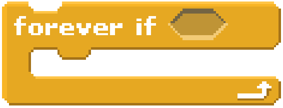
New concept: condition
A ***condition*** is a question whose answer is "yes" or "no," or a statement that is "true" or "false."

New construct in Scratch: the condition "touching ?"

The condition  is true if the sprite that is running this script is *touching* whatever is within the window, such as another sprite.

New concept: infinite run with a condition
An ***infinite run with a condition*** checks the condition before each run of the enclosed instructions. If the answer to the condition is "yes" or "true," the enclosed instructions are run; otherwise ("no" or "false"), they are not run. In either case, the infinite run continues and does not stop.

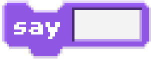
New construct in Scratch: infinite run with a condition

The instruction  causes the condition to be checked again and again. If the condition is true, the instructions included within the "mouth" of this instruction are run; otherwise, they are not.

Talking sprites

Step 3.2 requires the cat to say `Help`. As in comic books, sprites "say" something by having a balloon displayed with the words to be said; the balloon is connected by an arrow to the sprite. The instruction for speaking is , which is the fourth block from the top in the purple palette Looks. Drag and drop it within the "mouth" of the *forever if* block. Type the words to be said (here `Help`) into the first window of the block and the number of seconds that the balloon should appear (here 0.5) into the second window: .

New construct in Scratch: talking

The instruction causes a balloon to appear near the sprite that runs it; the balloon contains the words written in the first window and it appears for the number of seconds written in the second window.
There is another version of this instruction . The balloon will appear indefinitely until the end of the animation or until another *say* instruction is run.

Complete the script for the cat by adding blocks for steps 0, 1, 2, 3.1.

Create a script for the dog sprite that is similar to the one for the cat sprite except that it moves from the center towards the top of the stage. As long as it is touching the cat sprite, it will say `Yum` for one half second. Run the script by clicking on the green flag. Add comments and save the animation in a project with a new name.

Example 3
Multiple scripts

We have created several animations with two sprites, each sprite with its own behavior. The behavior of each sprite is determined by a single script that was written for that sprite. Scratch supports multiple scripts for each sprite. This is convenient if we want to affect the behavior of the sprite in several different ways.

Task 3

> Construct an animation where the cat and the dog move across the stage indefinitely, bouncing off the edges of the stage. Whenever they meet, the cat will say `Help` and the `dog` will say Yum.

Program file name: meet-forever-and-say

The new animation will have *two* scripts for each sprite. The first script that describes the movement will be the same as the script in the first example, while the second script will check the condition that the two sprites touch as in the second example. Both scripts for both sprites will start with the instruction when green flag clicked, so that all four scripts will start running ***concurrently***—at the same time—when the green flag is clicked.

Start by opening the project that contains the first animation. Click the button Save as to save this project under a *new* name. By doing this we will already have the first script for each sprite.

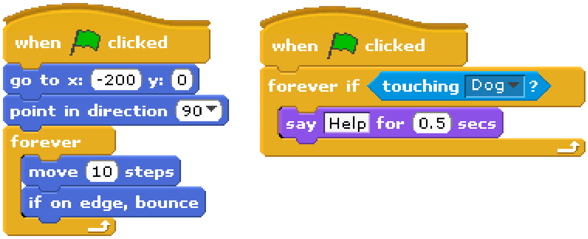
The script for talking

What does the second script for each sprite do? It has to check repeatedly if a sprite is touching the other sprite and if so to say something. For the cat sprite, this will be:

```
0. when the green flag is clicked
    1. repeat again and again, if you are touching the dog
        1.1 say Help
```

This is exactly the same as the script in the second example, except that the motion instructions have been removed. That is because the responsibility for moving the cat is assigned to the first script, while the second script is only responsible for the instructions related to touching another sprite.

The script area already contains the first script, but you can create the second script in any blank part of the script area by dragging and dropping the appropriate blocks. Here is the pair of scripts for the cat sprite:



```
when ⚑ clicked
go to x: -200 y: 0
point in direction 90▾
forever
    move 10 steps
    if on edge, bounce

when ⚑ clicked
forever if touching Dog▾ ?
    say Help for 0.5 secs
```

Similarly, create a second script for the dog sprite that says `Yum` when the sprite touches the cat. Click on the green flag and watch the animation. Since you have already created a new name for this project, click the Save button to save it after you have added comments.

> **New concept: concurrency of several scripts running at the same time**
> A sprite can have more than one script. All the scripts are run *concurrently*—at the same time—whether they are in the same sprite or different sprites.

Exercise 2

Construct an animation with three sprites: a girl, the Sun and a pair of sunglasses. The sunglasses are magical and automatically sense when the Sun is about to shine in the eyes of the girl. The girl sits at a fixed point on the stage. The Sun rises and sets by moving up and down the stage. The magical sunglasses are initially placed to the right of the girl. Whenever the sunglasses touch the Sun, they quickly move to cover the girl's eyes and then return to their initial position.

Guidance: For the sprites use a girl from the People folder, the Sun is in the Fantasy folder and the sunglasses are in the Things folder. Place the girl at the left edge of the stage (x=−150) and place the Sun and the sunglasses about 100 steps (x=−50) to the right of the girl's position.

Program file name: sunglasses

Additional material on Scratch: Changing the background

All the animations we have created so far have had sprites moving on a blank stage. Here we show how to change the background of the stage to make more colorful animations.

Task 4

> Modify the animation for the previous task by changing the background.

Program file name: background

Look in the sprite area in the lower right corner of the screen. To the left of the cat and dog sprites you will see a white rectangle labeled by the word Stage. Click here and you will see that the script area changes to the one for the stage. The script area will be empty because we have not written scripts for the stage. Click on the tab labeled Background. From here, you can add a new background image or modify the current background of the stage. You can even use a picture that you have taken with a digital camera as a background.

Click on the button Import and select an image to be used for this background. For example, select the folder Nature and click OK; you will see a set of small images of the various backgrounds. Click on one of them, for example, forest, and then click OK. This image will replace the white background on the stage, but the scripts will remain the same, as you can see by clicking on the green flag.

Save the animation as a new project. Before saving the project you may want to delete the white background. Click on the image of the background and then click on the small button with an X to the right of the button labeled Copy.

> **New construct in Scratch: changing the background**
>
> You can change the background of the stage. A background can be created by the Paint Editor, or you can import an image from a file or from a digital camera. Changing the background is performed from the area displayed when the Backgrounds tab of the stage is selected.

Summary

Concepts

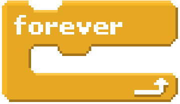
Concurrent run of multiple scripts for one sprite: A sprite can have multiple scripts that are executed *concurrently*—at the same time. Interesting animations can be constructed by having multiple sprites, each with multiple scripts, running concurrently.

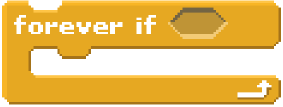
Control instructions: ***Control instructions*** do not affect the animation directly like motion instructions, but rather describe how other instructions will be run.

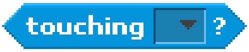
Infinite run: During an ***infinite run***, a sequence of instructions is run again and again, indefinitely.

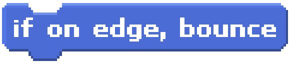
Infinite run with a condition: An infinite run may have a ***condition*** associated with it. A sequence of instructions is run again and again, indefinitely, but only when the condition is true. If the condition is not true, the enclosed instructions will not be run, although they will be run later if the condition ever becomes true. An infinite run with a condition is meaningful only if there are multiple scripts, so that some other script can make the condition true or false.

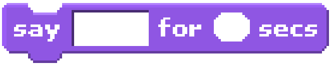
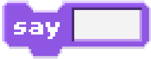
Scratch instructions

Infinite run: The instruction  causes an ***infinite run*** of the sequence of instructions within its "mouth."

Infinite run with a condition: The instruction  causes the sequence instructions within its mouth to be run indefinitely but only on those runs when the condition is true. The condition is written in the small window after the word `if`.

Touching: The condition  from the light blue Sensing palette is used to check if the sprite is ***touching*** another sprite. It can also be used to check if the mouse is touching the sprite or if the sprite is touching the edge of the stage.

Touching the edge: The instruction  is used to change the direction of motion of a sprite when the sprite ***touches an edge*** of the stage.

Speaking: The instruction  from the Looks palette causes a balloon containing text to appear above the sprite for a period of time. The instruction  is similar, but is not limited in time and appears until the animation completes.

Scratch techniques

Changing the background: The background on the stage can be changed by selecting an image from a file. A background can be created using the Paint Editor or a digital camera.

Chapter 4

Communications Between Sprites

In the previous chapter we created an animation that contained interaction between sprites: when the cat sprite touched the dog sprite, there was a reaction—talking—in the behavior of both sprites. In this chapter we will learn more about interaction between sprites, both in the way that they react and in the events that occur. To do this we will learn new instructions in Scratch that will enable us to write richer and more interesting scripts.

Example 1
The opening kickoff

Task 1

> Construct an animation of the opening kickoff in a game of soccer. Our familiar cat will pretend to be the famous player Pele. He waits for the referee to signal that the game is to start and then he kicks the ball.

Program file name: cat-kicks

The project will be developed in two stages. First, we need to select the images for the three sprites and place them in their initial positions facing in their initial directions. Second, we need to cause each sprite to perform its actions: the referee signals, Pele kicks the ball and the ball moves.

Defining the sprites

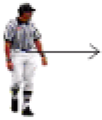
? Which sprites will participate in this animation?

It is clear that there will be one sprite for the player Pele and another one for the referee, but in addition we need a sprite for the ball, because the ball is an object that participates in the animation. The ball will wait at the opening position until it is kicked by Pele. Since it moves when it is kicked, the ball must be a sprite with its own script that contains motion instructions.

Guidance (Selecting the sprites): Any new project in Scratch starts with the cat sprite on the stage, so all you have to do is change its name to `Pele` (or, if you prefer, the name of your favorite player). Next, add a sprite for the referee, as was explained in Chapter 2. In the library of Scratch images, you can find an appropriate image called referee2 in the folder People. Change the name of the sprite to `Referee`. Now add a third sprite for the ball using the image called soccer1 in the folder Things. Choose a name for this sprite, too.

Initializing the sprites

Exercise 1



For each of the three sprites, plan and construct scripts that perform initialization of the sprite's position and direction when the green flag is clicked. Drag the sprites to arbitrary positions on the stage with your mouse, and then click on the green flag to check that they move to their correct initial positions. Write comments and save the project.

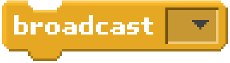
Initiating the action

In a real game of soccer, the referee blows his whistle to start the game and we will do the same here. The referee sprite will notify the other sprites to kickoff:

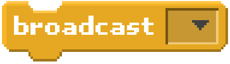
1. *initialize position and direction*
2. *say* `Kick`
3. *notify the other sprites to kickoff*

? Why do we have two actions: *Say* and *Notify*?

The *say* instruction in Scratch causes a balloon to appear on the stage. It is seen by the people who watch the animation on the computer screen, but it does affect the other sprites at all. We need another instruction that will cause the referee sprite to notify the other sprites that participate in the animation. There will be two instructions: one for the referee to *send a message* and another for the other sprites to *receive the message*, after which they will begin their actions.

To implement step 3, *notify*, use the instruction  which is a Control instruction that can be found in the orange palette, the seventh block from the top. This is a normal block that can be included anywhere in the sequence of blocks of the script. For the referee sprite, place it after the *say* instruction. There is a window that can be used to select the message to be sent. Click on the window, select `new ...` and type in the word `Kick` as the name of the message. Add a comment explaining the script for the referee and save the project.

New construct in Scratch: broadcast

The instruction  causes the sprite ***send a message*** to all other sprites. The name of the message is chosen from those available in the window or you can define a new message.

Scratch prefers the word *broadcast* over *send* because the message is sent to all the sprites in the animation just as a TV show is broadcast to all people in an area.

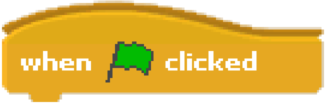
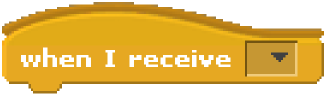
Pele receives the notification to kickoff

Pele is supposed to kick the ball after the referee sends the message `Kick`:

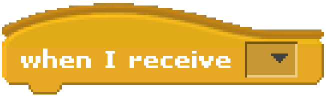
```
1. When I receive the message to kickoff
    2. Kickoff
```

? How does a sprite receive a message?

The script for the sprite has to start running when a certain event happens, namely, receiving a message. The word *when* alerts us that the instruction will be similar to *`when green flag clicked`*: clicking the green flag is also an event and *when* it is clicked a script begins running.

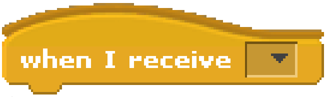
Like  when clicked the top of the block  when I receive (orange Control palette, the ninth block from the top) is curved so that it can only be the first block in the script. *When* the event of receiving a message occurs, the script following the block is run.

The sprite for the Pele will have two scripts: one that starts when the green flag is clicked and initializes the position and direction of Pele, and a second one that starts when the message is received from the referee and causes the Pele to kick the ball.

Drag the block  when I receive and drop it in an empty place in the script area. Click on the window and select the message `Kick`. (Once we have defined a message in one instruction, the message appears in the menus of other instructions related to messages.)

New concept: communications through message passing
Sprites can communicate with each other by ***sending and receiving messages***. When a sprite sends a message, it is broadcast to ***all*** the sprites in the project. When a sprite receives a message, any scripts that wait for this particular message begin to run.
By using messages, we can *coordinate* the runs of scripts in different sprites, ensuring that one script begins to run only after another script has run some of its instructions.

New construct in Scratch: receiving a message

The instruction 

can appear only as the first block in a script. When the message specified in the window is ***received***, the script begins to run.

Pele kicks the ball

? What sequence of instructions is appropriate for the action of kicking a ball?

This action is composed of two separate actions: Pele kicks the ball and then the ball moves as a result of the kick. The first action is performed by the Pele sprite, while the second action must be performed by the ball sprite. The reason is that ***running an instruction affects only the sprite whose script contains the instruction***.[1]

The action for Pele is very simple: he moves until his sprite touches the ball sprite:

```
1. when I receive the message to kickoff
    2. say Let's go
    3. move (until you touch the ball sprite)
```

Step 2 is added so that you can see when Pele has received the notification from the referee and is about to kick the ball. You will have to experiment with the number of steps in the move instruction so that Pele touches the ball sprite.

New construct in Scratch: size and position of a sprite

The position of a sprite is the position of the ***center*** of its image.
A sprite touches another sprite when ***any part*** of the sprite touches ***any part*** of the other sprite.

[1] An exception would be the instruction of broadcasting a message, but even here it is the message which affects other sprites, not the broadcast instruction.

Write the scripts for the Pele sprite. There will be two scripts: one for the initialization that will be performed when the green flag is clicked and one for kicking the ball that will be performed when the `Kick` message is received. It is important that there be two scripts so that the ball isn't kicked until after the referee blows his whistle.

Add a comment for the Pele sprite and save the project. Although we haven't finished the project, it is a good idea to save our work frequently in case the computer crashes or in case we want to start over from an intermediate stage.

The ball is kicked

Now it is the turn of the third sprite, the ball, to play its role in the animation. The ball must start moving when it is kicked by Pele:

```
0. when you are kicked by Pele
    1. move left to the edge of the stage
```

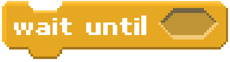
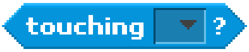
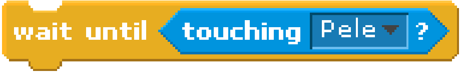
We previously decided that the Pele and ball sprites should start when the referee blows his whistle, that is, when they receive the message to kickoff. However, the ball should not move until it is kicked by Pele. We defined that the ball is kicked when the Pele sprite touches the ball sprite. Therefore, the description of the ball's script should be as follows:

```
0. when the Kick message is received
    1. wait until you touch Pele
    2. move left to the edge of the stage
```

The script starts with the orange Control instruction for receiving the message. For step number 2, a `glide` instruction can be used so that we can see the motion of the ball after it is kicked.

? What about step 1: waiting until you touch Pele?

We need an instruction that will ***wait*** until a *condition is true*. The meaning of the instruction will be: this script will *stop running* and only start again when the condition becomes true. Conditional waits are used to coordinate the running of several sprites: one sprite waits for a condition and another sprite is responsible for ensuring that the condition eventually becomes true.

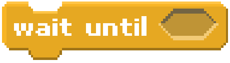
The block wait until is the fourth block from the bottom in the orange Control palette (You will have to scroll the palette to find this block.) A condition must be placed in the window with angled ends. The appropriate condition is touching ? which we used in previous animations. In the window of the condition, we will select the sprite Pele: wait until touching Pele ? .

New concept: conditional wait

A ***conditional wait*** causes a sequence of instructions to stop running until the condition becomes true. When the condition is true, the sequence of instructions resumes running.

A conditional wait is used to coordinate actions.

New construct in Scratch: conditional wait

The instruction  causes the run of the script to ***stop and wait until the condition that appears in the window is true***. When that happens, the script resumes its run at the block following this one.

Construct the scripts for the ball: the initialization when the green flag is clicked and the action of being kicked when the whistle is blown. Experiment with the values in the glide instruction until you get a pleasing animation. Add a comment to the ball sprite and save the project.

Summary: Let us summarize the sequence of actions that you see when the animation is run. When the green flag is clicked, all three sprites simultaneously move to their initial positions and directions. The referee says `Kick`, Pele says `Let's go` and kicks the ball (moves until he touches the ball). The ball is kicked (moves to the other side of the stage) when it is touched by Pele.

Exercise 2

Expand the project so that there are two players kicking two balls at the same time when the referee blows his whistle.

Guidance: For the second player use the dog sprite (named, perhaps, Maradona) and a second ball. The scripts for the Maradona sprite and the second ball sprite will be the same as for Pele and first ball, except for their positions and movements. Place Maradona and his ball below Pele and his ball. Make sure that the second ball only responds to a kick from Maradona and not to a kick from Pele!

Program file name: cat-and-dog-kick

Exercise 3

In the previous exercise, the ball kicked by Maradona reaches the referee. Expand the animation for the referee so that he falls down when he is touched by the ball.

Guidance: The animation of falling can be done by changing the direction of the referee. This will occur when the ball touches him.

Program file name: cat-and-dog-kick-referee-falls

Exercise 4

Construct an animation of two dogs who compete to get a bone. Initially, the dogs line up and wait for the cat to signal the start of the race for the bone. The first dog to arrive at the bone, picks it up and runs away with it. Ensure that one of the dogs runs faster than the other so that he will reach the bone first.

a. Design the animation described above.

Guidance: Decide on initial positions and directions for the four sprites, using the illustration above as a rough guide, and construct the initialization parts of the scripts. Decide what it means "to run away with the bone" and write instructions to implement this action.

Use three messages to control the action: one from the cat to tell the dogs to start, and one from each of the dogs to the bone to tell it to follow that dog. You have to ensure that only one of the messages is actually sent.

Program file name: get-bone1

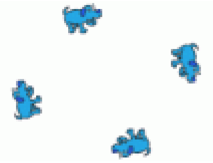
b. Modify the project from the previous exercise to use two messages instead of three: the same message is sent by both dogs to the bone. It is now the responsibility of the bone sprite to ensure that it only follows the dog that touched it first.

Program file name: get-bone2

c. Can you solve the problem with only one message that will be sent from the cat to the two dogs? The scripts of the bone must all start when the green flag is clicked.

Program file name: get-bone3

Exercise 5



Construct an animation where four dogs are running a relay race around the edge of the stage. Each dog will start in one of the corners and move around the stage in a clockwise direction.

a. First write an animation where the four dogs move continuously around the stage when the green flag is clicked.

Program file name: relay-race1

b. Now write an animation where the dogs start moving one by one. Initially, only the first dog moves. When he is halfway to the second dog, the second dog also starts moving. When the second dog is halfway to the third dog, the third dog starts moving, and similarly for the fourth dog.

Guidance: A dog sends a message `start moving` to the next dog:

```
1. when I receive the message start moving from the previous dog
    2. move halfway to the next dog
    3. send a start moving message to the next dog
    4. forever
        4.1 move around the stage
```

Program file name: relay-race2

Summary

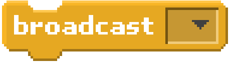
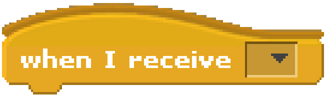
Concepts

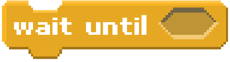
Communication and coordination by messages: Sprites can communicate with each other and coordinate their actions by sending and receiving messages: one sprite

broadcasts a message that is ***received*** by all the other sprites. One or more of the receiving sprites can react to the event of receiving a message by starting to run a script.

Conditional wait: When a ***conditional wait*** is run, the instructions that follow it are not run until the condition is true. When the condition is true, the following instructions can be run.

Scratch instructions

Sending and receiving messages: ***Sending and receiving messages*** are Control instructions found in the orange palette. The instruction  causes a message to be sent to all other sprites. You must name the message to be sent. The instruction  is a control instruction that can appear only as the first instruction in a script. When the message is received, the instructions below it in the script are run.

Conditional wait: The instruction  is a ***conditional wait*** instruction. The condition is written in the small window after the until.

Chapter 5

On the Dance Floor—Repeated Run Again

In Chapter 3 we saw how instructions for repeated run can be used to create very long animations—even infinite ones—from a finite (often very short) sequence of instructions. We learned two instructions for repeated run, both of them for infinite runs. The first, `forever`, causes the enclosed instructions to be run forever, while the second instruction, `forever if`, also leads to an infinite run, but one in which the enclosed instructions are run only if a condition is true. In this chapter, we will study more instructions for repeated run, but these instructions will be for finite runs of the enclosed instructions. We will study the instructions through a sequence of animations of dancing sprites, where each animation displays a more complex dance.

Example 1
A simple dance—repeated run for a fixed number of times

Task 1

> The dancer moves across the stage in one direction, then she moves back in the reverse direction, and finally she stops.

We
want the dancer to move across the stage in one direction and then back in the other direction.

? What instructions can we use?

One possibility is to simply use two *move* instructions, one to move her from left to right and the other from right to left. However, these instructions will simply cause her to jump from the initial position to the final position and this will not look like a dance. Another possibility is to use the glide instruction; we explore this possibility in an exercise.

Exercise 1

Write a script that implements the solution suggested above.

Guidance: Open a project and import a new sprite. You can find an appropriate one in the folder People. There are several images for the character Cassy, several of which show her dancing. Choose the image called cassy-dancing-3, which shows her with two outstretched arms, one upwards and one downwards.

The dancer's initial position is $(-100, 100)$ with initial direction 90° (pointing to the right). Use two `glide` instructions: one to move the dancer 100 steps to the right and another to move her 100 steps back to the left. Use a `turn` instruction to change the direction of the dancer by 180° between the first and second glides.

Program file name: dancer-glides

The animation in Exercise 1 will seem strange: when the instruction `turn` 180 `degrees` is run, the image for the Cassy sprite will *rotate* 180° so that she is standing on her head! For an explanation of this behavior and how to change it, see the Technical note at the end of the chapter, although you can work through this chapter without making the change.

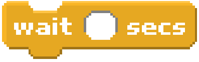
Dancing as repeated short steps

The animation that you created in Exercise 1 is nice but the resulting movement does not really look like dancing. Let us try a different method: instead of moving the dancer 100 steps in one instruction we will cause her to move in small amounts, say 10 steps at a time. Between movements we will have the dancer wait a short amount of time, $\frac{1}{5} = 0.2$ second. We will barely notice the waits, but they will give the impression of a dance step. Here is a description of actions that will move the dancer 100 steps in 10 separate movements of 10 steps each:

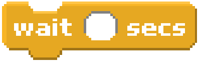
```
1. move 10 steps
2. wait 0.2 seconds
3. move 10 steps
4. wait 0.2 seconds
   . . .
19. move 10 steps
20. wait 0.2 seconds
```

The wait instruction

? Do we know the instructions needed to create this script?

We are already very familiar with the motion instructions, but we have yet to use the type of wait instruction that we need. In Chapter 4, we use a ***conditional wait***, which causes the run to wait until a certain event happens; here, we need a simpler instruction, a ***timed wait*** that causes the run to wait for a fixed amount of time. The instruction we need is , the fourth block from the top of the orange Control palette. This is a general instruction and in order to turn it into a specific instruction we have to enter a value into the window; this value is the number of seconds to wait.

New construct in Scratch: timed wait

The instruction  causes the script ***to stop running for the period of time specified in the window***. The period is given in seconds and can be fractional, for example, 0.2 second. When the period has passed, the run continues with the next instruction.

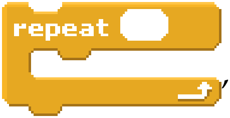
In order to move the dancer 100 steps, we can write a sequence of ten *`move 10 steps`* instructions with a *`wait`*

`0.2 sec` instruction between each two move instructions, altogether twenty instructions. After changing the direction of the dancer, we again need the same sequence of twenty instructions. It would be very boring to construct a script with so many instructions; certainly it would be extremely difficult if we had been asked to write a dance with 100 instructions.

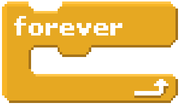
Fixed repeated run

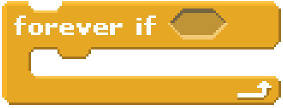
What we need is a way to indicate that an instruction or a sequence of instructions (here, a sequence of the two instructions move and wait) should be run a fixed number of times (here, ten times). A written description of the movement of the dancer in one direction would be:

```
1. run 10 times
    1.1 move 10 steps
    1.2 wait 0.2 seconds
```

The instruction for repeated run is 

, which appears as the sixth block from the top in the palette for Control instructions. This block has a window in which we enter the number of times that we want the enclosed instructions to be run. It has a "mouth" that contains the instructions to be run again and again.

Compare this block with the blocks 

and 

that we used in Chapter 3.

? Does block *repeat...* differ from the blocks for *forver* and *forever if*?

The bottoms of the blocks for the *forever* and *forever if* instructions are straight; they do not have a bulge that would allow other blocks to be placed after them. Clearly, since the infinite run is never going to terminate, it does not make sense to ask what instruction will be run after it, so a block for infinite repeated run can only be the last block in a script. However, a ***fixed repeated run***—where we know how many times the enclosed instructions will be run—need not be the last instruction in a sequence of instruction, so it can appear anywhere within the script. As you can see, the bottom of the repeat instruction has a bulge that can fit into the notch in the top of another block.

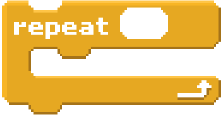
Construct the part of the script that corresponds to the written description above, duplicate it and put the two parts in place of the glide instructions in the script from Exercise 1. This script now contains two fixed repeated runs, or, as they are usually called, ***loops***. This term is used because after the enclosed instructions are run the script continues by turning back to the beginning of the repeat instruction. If you trace the run with your finger, it looks like a piece of rope that loops back upon itself. The number of turns or loops is determined by the number given in the window of the repeat instruction.

Write comments for this animation, save the project under a new name, and click on the green flag to run it.

Program file name: dancer-moves-loop

New concept: fixed repeated run
A ***fixed repeated run*** causes a sequence of instructions to be run again and again. The number of runs is fixed in advance and specified in the instruction.

New construct in Scratch: fixed repeated run

The instruction 

causes the enclosed instructions to be ***run the number of times that is specified in the window***.

In the rest of the chapter, we will expand on this basic example. Each new animation will be based on the same approach that we used in Example 1 (breaking down the dance into small movements with short waits between them). In the optional sections at the end of the chapter, we will show how to modify the basic dance to obtain a more dynamic one.

Exercise 2

> Add another dancer to the animation. He will dance in the *opposite direction* from the first dancer Cassy, starting at point $(100, 100)$, facing in the direction left $-90°$. He will move 100 steps and then reverse direction and move 100 steps.
>
> **Guidance:** Choose a different image for this new sprite, for example, Jay in the People folder.

Program file name: two-dancers-move

Example 2
Until we meet again—conditional repeated runs

In Exercise 2, we added an additional sprite to the animation, a second dancer who danced simultaneously with the first dancer. However, there was no interaction between the two sprites. They pass one over the other and their meeting does not affect their motion at all. Now, let us try to cause some interaction between the two sprites.

Task 2

> The two dancers move towards each other.
> When they meet, they change direction and
> begin moving in the opposite direction.

Program file name: two-dancers-move-until-touch

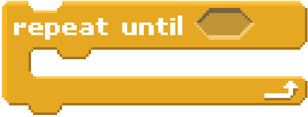
As before, each of the dancers will move 10 steps at a time followed by a short wait, but now the change of direction will not occur after a specific number of runs of this pair of instructions. Instead, when a meeting of the two sprites occurs, it will cause each of the sprites to change direction and to move in a new direction.

Repeated run that depends on the occurrence of an event

Here, too, we need an instruction for repeated run, but we don't want to decide ahead of time on the number of runs of the enclosed instructions as we did in Example 1. Instead, we want to *terminate* the repeated run when an event occurs, and this event will be a meeting between the two dancers. Here is a written description of the movements of the dancer:

```
1. run until you meet the other dancer
    1.1 move 10 steps
    1.2 wait 0.2 seconds
```

The first step specifies a ***conditional repeated run***. The end of the repeated run is not defined by the number of loops of the instruction, but by the occurrence of an event. Before running the enclosed instructions, a check is made to see if the event has occurred. If not, steps 1.1 and 1.2 are run, and the run returns to step 1, where the event is checked again. If, at the beginning of the loop, the event has occurred, then the conditional repeated run (including its enclosed instructions) is considered to have terminated.

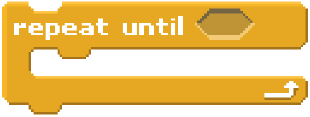
The instruction for conditional repeated run is 

, the third block from the end in the palette of the Control instructions. The block for this instruction has a notch on the top and a bulge on the bottom so that instructions can appear before and after it. As with all repeat instructions, this instruction has "mouth" to enclose the instructions that are to be repeated. The window following repeat until has angled ends, which means that a condition must be placed there. This instruction is thus similar to the conditions instructions *forever if* and *wait until* until from Chapters 3 and 4. The condition we need here is the familiar one *touching...?*.

New concept: conditional repeated run

A ***conditional repeated run*** enables a sequence of instructions to be run again and again. The number of runs is not fixed in advance; instead, the sequence is run until a condition becomes true.

New construct in Scratch: conditional repeated run

The instruction  causes the enclosed instructions to be run ***until the condition specified in the window becomes true***.

For each of the two dancer sprites, create the part of the script consisting of the conditional repeated run instruction and the enclosed *move* and *wait* instructions. Place these blocks at the appropriate positions within the script for the two dancers. Write comments for this animation, save the project under a new name, and click on the green flag to run it.

What happens during the conditional repeated run?

When the run of one of the scripts reaches the instructions for conditional repeated run, a check is made if the two sprites are touching. Since this is not true in the initial state of the animation, the `move` and `wait` instructions enclosed within the `repeat until` instruction will be run. Then, another check is made for the occurrence of the event of touching the other sprite; again, this does not occur, so `move` and `wait` are run again. This continues until the check discovers that one sprite has touched another. When this happens, the enclosed instructions are no longer run; instead, the run of the script continues with the instruction that follows the `repeat until` instruction. In this case, it is the instruction that changes the direction of the sprite.

Exercise 3

> Is it possible that the instructions enclosed by `repeat until` are never run, not even once? Change the script so that the initial positions of both sprites are the same, for example, the point (0,100). Click the green flag to run the animation and explain what happens.

Exercise 4

> Consider the second loop of the script for each dancer which is a fixed repeated run. What will happen if we change these loops to use the conditional repeated run instruction? Make this change to the script and explain what happens.

In Exercise 4, we learned an important lesson about conditional repeated run. We must be certain that the condition will eventually become true; otherwise, the instruction will become an infinite repeated run, and the instructions that follow it will never be run. This possibility did not concern us in Chapter 3 because the repeated runs were intended to be infinite, nor did it concern us in the example at the beginning of this chapter, where the number of repeated runs was specified by a fixed number.

Example 3
Dancing on and on—repeated runs within repeated runs

Task 3

> Modify the animation of the dancers so that the dance continues again and again, forever (or until the red stop button is clicked).

That is, we want to repeat the portion of the dance that we created in Example 2.

? How can we do this?

We have already learned in Chapter 3 how to construct an infinite repeated run. The instruction that we need is . Recall that within this instruction for infinite repeated run, we can include an *arbitrary* sequence of instructions; in particular, we can include the entire sequence of instructions for the dance from Example 2. The sequence of instructions for the dance itself contains two instructions for repeated run, in this case conditional repeated runs.

? Can these instructions be included within an infinite repeated run?

Yes, why not? However, we have to be careful when we put the two together.

Let us look again at the script in Example 2 for the dancing girl. Which part of this script should be included in the infinite repeated run instruction ?

? Should the first instruction when green flag clicked be included?

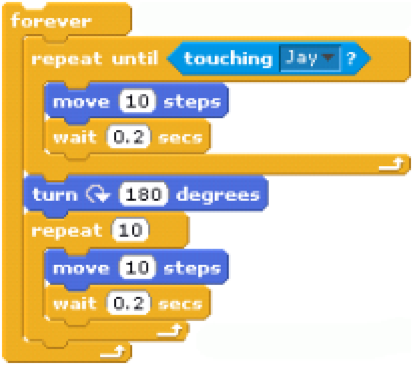
Of course not! Clicking on the green flag should be done once at the beginning of the script, not every time the dancer begins her movements. In fact, the top of the block for this instruction does not have the notch that would allow it to be placed within a sequence of instructions; it can only appear as the first instruction in the sequence.

The next two instructions are used to initialize the dancer sprite—to place her in her initial position facing in her initial direction.

? Should these instructions be run repeatedly?

Again, the answer is no. The word "initialization" means that these instructions prepare the sprite for the dance, but they are not part of the dance itself. There is no need to initialize the sprite more than once.

The dance that we want to run repeatedly is described by the rest of the script that begins with the first conditional loop and continues until the end of the script. Therefore, all we have to do is enclose this part of the script within the block for the infinite run instruction forever. Drag this block from the orange Control palette until its top is between the initial instructions and the first loop; its "mouth" will open and enclose the rest of the script.



```
forever
  repeat until <touching Jay ?>
    move 10 steps
    wait 0.2 secs
  turn ⟳ 180 degrees
  repeat 10
    move 10 steps
    wait 0.2 secs
```

Exercise 5

Make this change in the scripts for the two dancers. Click the green flag and explain what happens.

Program file name: dancers-disappear

Analyzing the problem

The change did not lead to the animation that we expect. In order to understand what happened, let us follow the run of the script step-by-step:

- At the beginning, each of the sprites turns to its initial direction, the boy dancer to the left and the girl dancer to the right.
- They perform their first dance, moving toward each other until they touch.
- As a result of the meeting, they change their directions and continue the dance.
- Now, the *`forever`* instruction causes the first conditional repeated run to begin again; the dancers will continue to move until they meet again. Unfortunately, this will never happen, because the dancers continue to move in the directions they moved in during the second part of the dance. In Scratch, whenever the sprite tries to move off the stage, it remains partially visible at the edge. This is

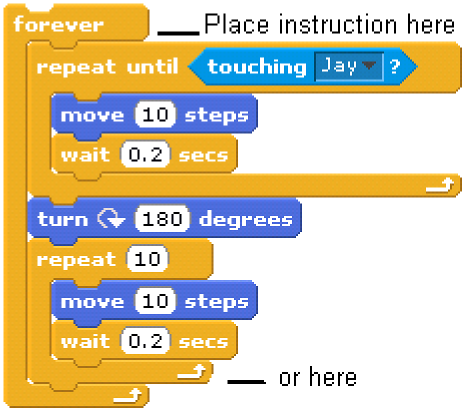
similar to what happened in Exercise 4: since the meeting never occurs, the run is infinite, although we never intended for it to be infinite.

Trying to fix the problem

How can we solve this problem? We have to add additional instructions for changing the directions of the dancers *before* they begin the repeated run of the first part of the dance. Clearly, the correct instruction to use is turn ↻ 180 degrees, the same instruction that we used between the two parts of the dance.

? Where shall we place this instruction?

There are two possibilities. As we have seen, the problem occurs before the dancers return to run the first part of the dance. Therefore we can place the turn instruction either before the first part of the dance or after the second part of the dance, just before the end of the infinite repeated run instruction:



```
forever
  repeat until <touching Jay ?>
    move 10 steps
    wait 0.2 secs
  turn ↻ 180 degrees
  repeat 10
    move 10 steps
    wait 0.2 secs
```

It doesn't appear that there should be any difference between placing the instruction in either of these two places, because the beginning of the first part of the dance is run immediately after the end of the second part of the dance. Place the *turn* instruction before the first part of the dance so that it will be run immediately after the initializations of the dancers. Click the green flag and see what happens.

The problem appears again: the position of the turn instruction is important

The change in direction changed the direction that was set during the initialization. As a result, the dancers move away from each other and the event of their meeting will no longer occur. Therefore, we choose the second possibility, and place the turn instruction that reverses the direction after the second part of the dance. We now know that this instruction will not be run immediately after the initial direction has been set. Add this instruction to the script and check that the animation is what we expect.

Update the comments for this project and save it under a new name.

Program file name: dance-indefinitely

In this example, we have seen that one repeated run instruction can enclose another, but this has to be done carefully and we have to check that the resulting script results in the animation that we want.

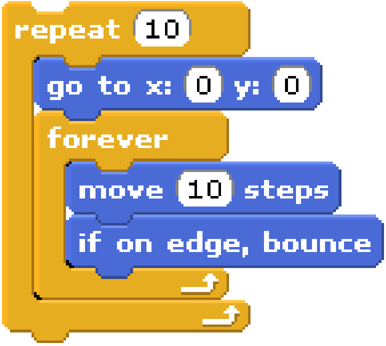
New concept: nesting of repeated run instructions
A repeated run instruction of any kind can enclose arbitrary instructions, including other repeated run instructions. When this occurs, we say that the repeated run instructions are ***nested***.

Possible combinations of infinite repeated run

? Are all combinations of repeated run instructions possible?

? Can an infinite repeated run be placed within another repeated run instruction?

The answer is no. There is no point in placing an infinite repeated run instruction (with or without a condition) within another repeated run instruction, because it will never end; therefore, the instructions that come after it will *never* be run. Even if the infinite repeated run instruction is the *last instruction* within another repeated run instruction, it will not be possible to return to the start of the containing instruction and run the other instructions. The action specified by the script on the right would be exactly the same if the outer `repeat...` were removed.



```
repeat 10
  go to x: 0 y: 0
  forever
    move 10 steps
    if on edge, bounce
```

Example 4
Becoming a choreographer—controlling the dance

In the animations we have created so far, the sequence of movements has been specified in advance. The person watching the animation has no influence on what will happen and can only watch the animation like a movie. Interacting with the computer—as in a computer game—can often be more interesting than just watching a movie.

Task 4

Modify the animation so that you can influence the movements of the dancers. They will still dance back and forth, but the direction of the dance will not be controlled by an event (the two dancers meeting) or by a number (the number of steps taken), but by a command that you will give:

- When the girl dancer moves right, she will continue in that direction until you press the *left arrow key*. When she is moving left, she will continue to do so until you press the *right arrow key*. When a key is pressed her direction changes by 180°.
- The boy dancer will move up and down the stage instead of left and right. When he moves down, he will continue in that direction until you press the *up arrow key*. When he is moving up, he will continue to do so until you press the *down arrow key*. When a key is pressed his direction changes by 180°.

Program file name: user-controls-dance

Here is a description of the first part of the girl's dance:

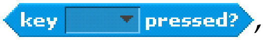
```
1. run until the left arrow key is pressed
    1.1 move 10 steps
    1.2 wait 0.2 seconds
```

and here is the description of the second part:

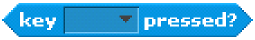
```
1. run until the right arrow key is pressed
    1.1 move 10 steps
    1.2 wait 0.2 seconds
```

As you can see, the instructions within each loop are the same as before, but the repeated run instruction is different. Both of them must be conditional repeated run instructions repeat until, where the condition is no longer that of touching another sprite but of a key being pressed.

Responding to a key press

The condition we need is , the ninth block from the top in the light blue palette Sensing. Click on the small arrow in the window to choose which key will be sensed. For the first condition choose the left arrow and for the second condition choose the right arrow. Make these changes in the script for the girl dancer.

New construct in Scratch: sensing a key press

The condition  is true if the user is currently ***pressing a key*** and it is false if the user is not pressing a key. The key that is sensed must be specified by choosing it from the menu that appears when the arrow is clicked.

The movements of the boy dancer

We chose two *different* keys—the *up arrow key* and the *down arrow key*—for controlling the boy dancer; otherwise, both dancers would respond to the same event. The boy dancer will have his initial direction downwards 180°. In the first part of the dance he will move downwards and then, after turning 180°, he will move back up the stage. The loops for the two parts of the dance are similar to those for the girl dancer:

```
1. run until the up arrow key is pressed
    1.1 move 10 steps
    1.2 wait 0.2 seconds
```

```
1. run until the down arrow key is pressed
    1.1 move 10 steps
    1.2 wait 0.2 seconds
```

Make these changes, click on the green flag and check that the animation works as we wanted to. Add comments to the scripts and save the project under a new name.

Exercise 6

> Construct an animation with three sprites: an elephant, a horse and a cow. When the green flag is clicked they move to their initial positions one above the other near the left edge of the stage and pointing to the right 90°.
>
> a. Following initialization, each animal moves to the right in 60 movements of 5 steps each.
>
> Program file name: fixed-run
>
> b. Modify the animation so that each animal has a key associated with it (left arrow for the horse, right arrow for the cow, up arrow for the elephant). Following initialization, each animal moves to the right until its associated key is pressed.
>
> Program file name: play-run1
>
> c. Construct an animation whose behavior is opposite to the previous animation. Following initialization, each animal moves only *when* its

associated key is pressed. When the key is released, it stops moving. If you press the key again, the movement will start again.

Program file name: play-run2

d. Combine the two previous animations. Following initialization, each sprite starts to move and stops when its key is pressed. If the key is pressed again, the sprite will start to move again, and so on.

Guidance: You may need to add wait instructions so that you will have time to press and release a key.

Program file name: play-run3

Additional material on Scratch: Changing costumes

Realistic animation

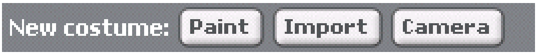
The basic dance animation that we have constructed in this chapter is lacking an essential aspect: although we see movement and this movement reminds us of a dance (because of the short waits between each step), the moving sprite always looks the same with its arms and legs always in the same position. A more believable animation of a dance would have sprites continuously change their appearance. In animated cartoons, this is done by creating several, slightly different, images that are quickly shown one after the other, thus giving the appearance that the character is moving. For example, to show that a character is walking, the animators draw several images of the character with its legs and arms in different positions and then these images are displayed in quick succession.

Task 5

> Modify the animation from Example 3 so that the dance is more believable. The sprites should change their appearance while they are moving.

Program file name: dancers-change-costumes

There can be more than one costume for each sprite. In Chapter 2 we showed how to import a new costume for use by a sprite and how to modify a costume using the Paint Editor. When a script is run, we can include instructions that change costumes.



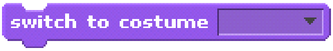
We will show how this is done for the girl dancer. In the middle panel of the Scratch window, click on the middle tab Costumes. The girl dancer has only one costume, which is marked with a blue border indicating that this is the active costume. Above the costume, you will see the words New costume: followed by three buttons, one for creating a new costume by using the Paint Editor, the second to Import a costume that already exists on the computer, and the third for importing a costume from a Camera.

Add a new costume for the dancer by clicking on the button Import and selecting a new costume from the folder People; for example, choose the costume cassy-dancing-1 and click OK. You will now see two costumes for the dancer; the second one has a blue border indicating that it is the active costume. By clicking on the icon for a costume you can make it the active costume; try this and you will see that the blue border changes, as does the image of the sprite on the stage.

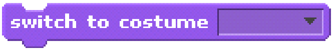
New construct in Scratch: adding a costume to a sprite

Additional costumes can be added to a sprite. Select the Costumes tab above the script area and create the new costume either by: drawing it using the Paint Editor, importing from a file, or transferring an image from a digital camera.

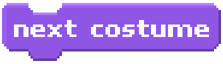
Instructions for changing costumes

We now show how to change between costumes during the dance. Choose the purple palette Looks and you will see that the first two blocks contain instructions that change the costume. The first one  switch to costume changes changes the costume of the sprite so that it is displayed with the costume that you can choose by clicking on the arrow in the window within the block.

New construct in Scratch: changing to a specific costume

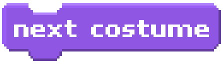
The instruction 

changes the costume that the sprite is displayed with. The costume is specified by choosing from the menu that appears when the arrow in the window is clicked.

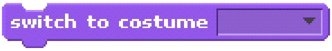
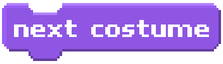
The second instruction 

is simpler; it simply changes the costume of the sprite to the next one in the list in the center panel. For example, if there were four costumes and the current one (with the blue border) is the second one, then running the instruction *next costume* would cause the third costume to become the current one. Running the instruction again would cause the fourth costume to become the current one, and running it yet again would cause the first costume to become the current one, since the first costume is considered to be the one after the last costume.

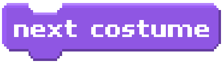
New construct in Scratch: changing to the next costume

The instruction  next costume ***changes the costume*** that the sprite is displayed with to the ***next costume***, where the "next" costume is the one that appears after the current on in the costume panel. If the current costume is the last one, then the "next" costume is the first one.

? Are these two instructions absolute or relative instructions?

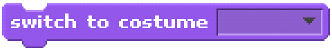
The instruction  switch to costume is an *absolute* instruction because it changes the costume to a specific new one without considering what the current costume is. The instruction  next costume is a *relative* instruction because the new costume depends on the current one.

Creating an animation effect by changing costumes

Let us use the instruction  next costume to cause the dancer to change her appearance again and again during the dance.

? Where shall we place this instruction?

Since the dance is constructed from repeated runs of a short sequence of two instructions (move and wait), we can add the next costume instruction to this sequence. Do this for both parts of the dance.

Since we are changing costumes during the dance, it is important to specify an initial costume for the sprite, just as we needed to specify an initial direction for the sprite when we change direction during the dance. Otherwise, if we run the animation several times, different costumes could appear depending on where the previous run of the animation was stopped (by clicking the red stop button next to the green flag). To initialize the costume we will use the absolute instruction . Click on the arrow and choose one of the costumes of the dancer to be the initial one.

Click on the green flag to run the animation, stop it and start it again to check that it always starts with the same costume.

Exercise 7

> Make a similar change for the boy dancer. The sprite that we have chosen for this dancer (jay) has only one costume in the folder People. Change the sprite by deleting the jay costume and by adding two costumes dan1 and dan5. Make the appropriate changes in this script for the boy dancer, add comments and save the project under a new name.

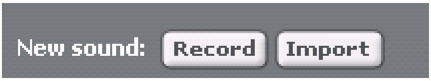
Additional material on Scratch: Adding background sounds

The Scratch environment supports the use of sounds. They can be associated with sprites or with the background. Here we explain the use of sounds in the context of background sounds.

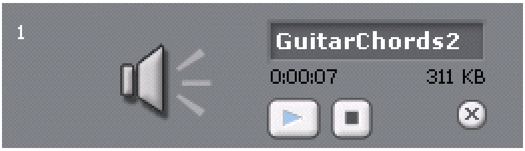
Task 6

> Add music to the animation of the dance in Example 3.

Program file name: dance-with-music



Select the stage image in the sprite panel in the lower right-hand corner of the Scratch window. In the script area click on the tab Sounds tab. The sounds that participate in the animation will appear here and initially there will be none, unlike the costumes for the stage where there is a default white background. To add a new sound you can Import it from the library of sounds that is supplied with the Scratch environment, or you can Record your own sound by using a microphone that is connected to the computer.



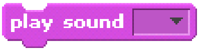
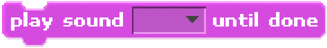
Once the sound has been added to the list, it can be chosen as a background sound. Click on the Import button to import a sound and choose the folder Music Loops. Click on the entry GuitarCourds2 in the list that appears, and the computer will play the sound so that you can be sure that this is the sound you want. Click OK. The sound will appear in the list of sounds that can participate in the animation. Underneath its name is written the length of the sound (here 7 seconds), and there are buttons for playing the sound, stopping the sound, and erasing the sound from the list.

New construct in Scratch: adding sounds

To add a sound click on the Sounds tab in the script area. Sounds can be imported from files in many formats such as (wav and mp3). You can also record a sound from a microphone attached to the computer.

Specifying the background sound

Now that the sound appears in the list, it can be used as a background sound. Since the stage is responsible for the background sounds, we have to create a script for the stage that will cause the sound to be played. Click on the Scripts tab. For the stage, the set of blocks of instructions that can be included in the scripts is different from the set of blocks that you already know about from writing scripts for the sprites. In particular, the blue Motion palette is empty, because the background cannot move! Similarly, the light blue Sensing palette does not have blocks for the condition `touching...?`, because the background is always touching all the sprites, as well as the mouse cursor.

Select the dark pink palette Sounds. The first two instructions cause sounds to be played. The first instruction  play sound causes the sound to begin playing and the run continues immediately with next instruction, while the second instruction  play sound until done causes the sound to be played until it ends, and only then does the run continue with the next instruction. Both instructions have a window with an arrow that you use to choose which sound will be played.

New construct in Scratch: playing sounds

The instructions play sound ▾ and play sound ▾ until done cause the sprite or the background to play the sound whose name appears in the window. A script containing the first instruction continues running as soon as the sound starts to play, while a script containing the second instruction stops running until the sound has finished playing.

Drag the second instruction, play sound `GuitarChord2` until done, and drop it in the script area for the stage. The window will already show the sound `GuitarChord2` because that is the only sound that appears in our list of sounds. Add the instruction when green flag clicked as the first instruction in the script. Click on the green flag and you can hear that the sound is played at the same time that the two dancers move on the stage. However, it is hard to call this a background sound, because after 7 seconds the sound is finished and the animation becomes quiet.

? How can we ensure that the background sound will be played continuously?

We have to cause the sound to be played again and again, as long as the dance continues. Since the dance of both dancers is controlled by an infinite repeated loop, the sound, too, should appear in such a loop. Enclose the block *`play sound GuitarChord2 until done`* within an infinite repeated run instruction. Click the green flag and check that the sound is played as long as the dancers dance. Add comments to the project and save it under a new name.

Changing the sound during the animation

The background sound does not have to remain the same throughout the animation. Let us use two *`play`* instructions for different sounds. If the instructions are enclosed within an infinite repeated run instruction, the two sounds will alternate indefinitely. We will do something else; we will change the sound as the result of an event that occurs during the animation.

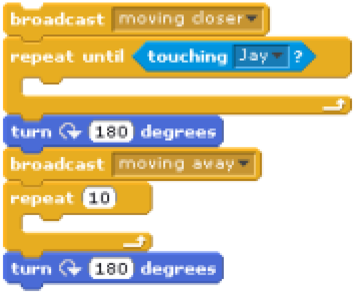
Task 7

> Modify the animation so that the background will play one sound when the two dancers move towards each other and another sound when they move away from each other.

Program file name: dance-change-music

Using communication to indicate that the sound should be changed

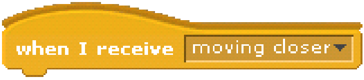
The two dancers must communicate with the stage in order to inform the stage whether they are moving towards each other or away from each other. The stage then uses this information to decide which sound to play. Two messages will be used: one for moving towards each other and the other for away from each other.



```
broadcast moving closer
repeat until touching Jay ?

turn ↻ 180 degrees
broadcast moving away
repeat 10

turn ↻ 180 degrees
```

It is not necessary for both dancers to broadcast these messages since their movements are already coordinated; that is, we have already arranged it so that either both dances move towards each other or they move away from each other at the same time. Let us arbitrarily decide that the girl dancer will be responsible for sending the messages. The dancer will send a message `moving closer` before the first dancing loop and a message `moving away` before the second dancing loop.

Changing the music after receiving a message

Now we have to change the behavior of the stage so that it can receive the messages and change the sound that is played. As explained in the previous chapter, the instruction for receiving a message has to be the first block in any script. Therefore, we will replace the block *when green flag clicked* with the block  when I receive moving closer.

? What about the second message moving away?

You will need a separate script for receiving the message `moving away`. It will be the same as the script for receiving `moving closer` except that it will play a different sound, for example, `GuitarChord1`. Run the animation.

? Does it do what you expected?

Analyzing the problem

The result sounds terrible because the sounds partially overlap. The reason is that the instruction play sound until done appears in two different scripts. The run of each script stops while the sound is being played and waits until it is finished. However, it is possible for the dancer to send the message `moving closer` and before seven seconds are up, the dancers meet and move away, causing the girl dancer to send the message `moving away`. The two sounds will be played at the same time.

To solve this problem, each script should stop the playing of sounds before it begins playing its own sound. This can be done with the instruction stop all sounds, the third block from the top in the dark pink Sounds palette. Place this instruction before each `play` instruction. Check if the animation does what you expect. Add comments to the project and save it under a new name.

New construct in Scratch: stopping all sounds

The instruction stop all sounds causes ***all sounds to stop playing*** immediately. This includes all sounds that are being played in all the scripts both of the background and the sprites.

Exercise 8

So far we have shown how the background can play sounds. In this exercise, you will have several *sprites* (a drum, a guitar and a trumpet) play sounds. The instructions for playing the sounds will appear in the scripts for the sprites.

a. Construct an animation in which each instrument plays its own tune 5 times.

Program file name: fixed-tune

b. Modify the animation so that the instruments play their tunes until an appropriate key is pressed. Choose a key for each of the instruments; for example: left-arrow for the drum, right-arrow for the trumpet and up-arrow for the guitar. You might have to keep the key pressed for a while before the instrument stops playing. Why is that?

Program file name: play-tune1

c. Modify the animation so that the instruments don't play their tunes until an appropriate key is pressed. An instrument plays its tune as long as its key is pressed and stops when the key is released. Pressing the key again will cause the tune to start again.

Program file name: play-tune2

d. Modify the animation from exercise b: The instruments start playing when the green flag is clicked and they stop when an appropriate key is pressed. However, pressing the key again will cause the tune to start again. You may want to add *wait* instructions so that the user has sufficient time to press a key and release it.

Program file name: play-tune3

Summary

Concepts

Finite repeated runs: In ***fixed repeated run***, the number of runs is specified in advance as a certain number, while in ***conditional repeated run***, the loop is run until a specific event happens or until a condition becomes true. When using a conditional repeated run, we have to make sure that the event actually happens so that the run does not become infinite.

Nesting instructions: It is possible to include one type of repeated run within another, except that an *infinite repeated run* cannot be included within any other repeated run.

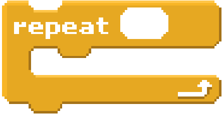
Timed wait: A script can be made to stop running for a specified amount of time.

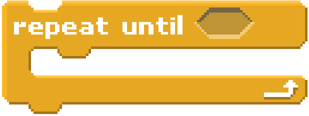
Interaction with the user: The user can interact with a program by using the keyboard. The program can find out which key was pressed and modify its run depending on the key.

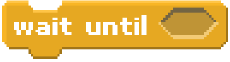
Changing costumes: A sprite can have several costumes and the costume can be changed during the animation.

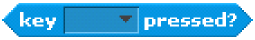
Sounds: Sounds can be played during the running of a project.

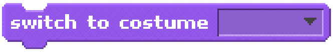
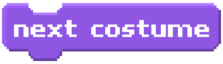
Scratch instructions

Finite repeated runs: ***Fixed repeated run*** is implemented by the instruction . The number of runs is entered in the small window.

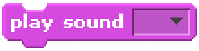
Conditional repeated run is implemented by the instruction . The window must contain a condition.

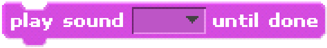
Waiting: The instruction  causes the script to ***stop running for a period of time*** expressed in seconds (including fractions of seconds).

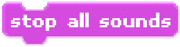
Interaction with the user: The condition  is true when the user has pressed the key specified in the window.

Changing costumes: Add a costume to the animation by selecting the Costume tab and painting the costume or importing it from a file or a digital camera. The absolute instruction  changes the costume to the one specified in the window. The relative instruction  changes to the next costume in the list in the script panel.

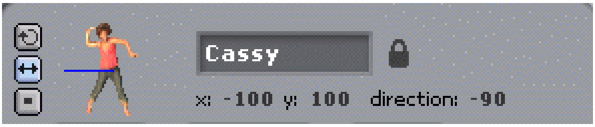
Sounds: Sounds can be played by the background or by a sprite. The Sounds tab is used to add sounds to the project. Instructions for sound appear in the dark pink palette:







Technical note: turning an image



Cassy

x: -100 y: 100 direction: -90

Scratch enables you to limit the way that a sprite responds to *turn* and *point* instructions. Look at the area above the script area. So far you have used this only for changing the name of the sprite, and you may have noticed that it displays the current position and direction of the sprite below the name. To the left of the icon for the sprite, you will see three buttons. By default, the upper button with a round arrow is selected. This allows the sprite the rotate through a full circle 360° like a gymnast doing cartwheels. If you click the middle button with the two-way arrow, then the sprite always stands upright and can only spin 180° like an ice skater or a dancer. Click this button for the dancing sprites in this chapter. The third button causes the sprite to ignore all *turn* and *point* instructions.

Chapter 6

Remembering Things—Variables

In this chapter we will develop a simple game in order to introduce *variables*, which are used to remember values and to change values that were previously stored. The values can represent, for example, the number of points that a player wins in a game.

Example 1
The growing and shrinking dragon—changing the size of a sprite

In the first example the user will not participate; instead, we only

demonstrate how to change the size of a sprite. We will work through the example using a specific sprite. Let us choose the image dragon1-a found in the folder Fantasy.

Task 1

> Construct an animation with one sprite, a dragon. When the green flag is clicked, the size of the dragon will be reduced by one half.

Program file name: change-to-half-size

The behavior of the dragon sprite is as follows:

```
0. when the green flag is clicked
   1. reduce the size of the dragon to one half of its full size
```

Initializing the size

? Does this project need initialization?

When the project is constructed by importing the image for the dragon sprite, the size of the sprite is its original size. The first time that we run the script, the size is reduced; however, the second time the script is run, the size of the sprite is already half of its original size, so we see no change! Therefore, we must initialize the size of the sprite to its full size so that we can see that it is in fact reduced to the half-size:

```
0. when the green flag is clicked
    1. initialize the size of the dragon to its full size
    2. reduce the size of the dragon to one half of its full size
```

In Scratch, every sprite has a size and there are instructions that can change the size. Blocks for instructions that change the size can be found in the purple palette Looks:

- `change size by ( )` is a relative instruction that changes the current size to a new size by adding a value.
- `set size to ( ) %` is an absolute instruction that changes the size to a new size without taking the current size into account.

Step 1 can be expressed in Scratch using the instruction `set size to (100) %`, while step 2 can be expressed using the instruction `set size to (50) %` .

New construct in Scratch: setting and changing the size of a sprite

The instruction `change size by ( )` changes the current size of a sprite to a new size by adding a value.
The instruction `set size to ( ) %` changes the size of a sprite to a new size that is a percentage of its original size when it was imported into the project.

It is worthwhile waiting a bit

Create a script that implements the description given above and run it by clicking on the green flag.

? What happens?

The first time that the script is run, we see the dragon reduced in size, but if we click the green flag again, nothing seems to happen. The reason is that the size of the dragon is reduced *immediately* after it is set to its full size, so we don't have a chance to see it at full size after the initialization. In order to see the change, introduce a short wait between the two steps:

```
0. when the green flag is clicked
    1. initialize the size of the dragon to its full size
    2. wait 2 seconds
    3. reduce the size of the dragon to one half of its full size
```

Update the script to implement this behavior and run it.

? Is the new image half the size of the original one?

Displaying the size of the dragon

Since we never see dragons in real life, it is hard to tell when the sprite is at its original size and when it is at half its original size. It would be nice if the numeric value of the sprite's size were displayed on the stage. In the purple palette Looks, just below the block for `set size to...%`, is a ***reporter block*** □ size. Click on the square box next to the reporter block: ☑ size . You will see that the value of the size of the sprite is displayed in the upper left hand corner of the stage. This display is called a ***monitor***. If you wish to remove the monitor, simply click again on the square to the left of the reporter block.

With the reporter block checked, click on the green flag and note that the monitor correctly reports the size of the sprite: initially 100 and two seconds later 50.

Write comments for your project and save it with an appropriate name.

> The optional section on *mouse modes* in Chapter 2 explained how to directly change the size of a sprite using the grow and shrink buttons (the two right buttons on the toolbar above the stage . Try changing the size of the sprite using these buttons and check that the monitor always displays the current size.

Example 2
The size of the dragon changes to the value of a variable

In the first example, we showed how the size of a sprite can be set and changed within a script. At the end of the example, we noted that you can change the size directly by clicking on the image on the stage, although it is difficult to give the sprite a specific size. Now we would like to enable the user to set the size of the dragon to a specific numerical value.

Task 2

> Modify the animation so that the user can supply a numerical value for the size of the dragon.

Program file name: change-to-smaller-size

We need a way to remember a value and then to use that value in the instruction set size to %. Places to remember information are called ***variables***. You can create a variable to remember the size of the dragon; then, you can ***store*** a value in the variable. Once the variable stores a value, you can ***use*** the value for any purpose, for example to change the size of the dragon using the instruction *`set size to ... %`*.

Suppose that you want to store some money. You can use a *box* and put the money into the box. You can also look into the box and see how much money you have. A variable is like this box: if you have a value such an amount of money you can store it in a variable and you can look into the variable to see how much money you have.

Be careful, though, because there are differences between a real box and a variable. You can add and remove money from a real box, but for a variable there are only two actions that you can do: store an entirely new value into the variable (using an instruction like `set size to...%`) or read the value that the variable contains (using a reporter like `size`). When you read the value of a variable, the contents of the variable are not changed.

New concept: variables
A ***variable*** is a memory cell that can remember a value. A value can be ***stored*** in a variable, which remembers the value until it is changed by a later store. The current value within a variable can be ***read***.

Creating a variable

The instructions relating to variables are displayed in the red-orange Variables palette. Initially, there are two buttons in this palette Make a variable and Make a list.

Unlike the objects in the other palettes, these are ***buttons*** that the user can click on, not blocks that can be used in scripts. They are not colored and they don't have shapes like blocks. Only after clicking one of these buttons will "real" instructions that operate on variables and lists appear. (Lists will be discussed in Chapter 9.)

Click on the button Make a variable. A window pops up and asks you for the name of a variable. Choose a name such as s and write the name in the window. Click OK to create the variable.

> You can also choose if you want this variable to be used by all sprites or by only one sprite. Although our current project has only one sprite, we will later add other sprites, so leave the choice For all sprites unchanged.

If this is the first variable in the project, the button *Delete* variable will appear beneath the button for *Make a variable.* Use this button if you accidentally create a variable that you don't need. For example, if you want to change the name of a variable, you will have to delete it and make a new variable.

Once a variable is created, several new blocks will appear underneath the buttons. This first is a block with rounded ends that is labeled with the name you gave for the variable and to its left a small square . This is a ***reporter block*** that represents the variable (just like the reporter for `size` that you learned about above). Click the square box to cause a monitor for the variable `s` to appear on the stage. If the monitor appears, you can remove it by clicking again on the square box.

> The block for `size` represents a variable that stores the size of the sprite, and is set and changed by the Scratch environment. You can read its value and but you cannot change it directly. It displays the actual size of the sprite. The block for `s` represents a variable that you created; what you do with it is up to you.

New construct in Scratch: creating a variable

Clicking the button causes a new variable to be ***created***. You must give a ***name*** to the variable and you decide if it is to be ***accessible*** to all sprites or only to this one. When the variable has been created, a ***reporter block*** appears in the palette.

When the first variable in a project is created, a number of new instruction blocks appear in the Variables palette; these are discussed below.

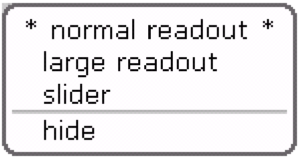
New construct in Scratch: displaying the value of a variable

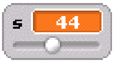
The value of any variable (including internal variables created by Scratch itself like `size`) can be displayed in a ***monitor*** on the stage. In the palette, there is a small square next to each reporter for a variable. Clicking on the square causes a check mark to appear or disappear; when the check mark appears, the monitor is displayed on the stage. The monitor can be dragged and dropped anywhere on the screen.

Storing values into a variable

? How can you change the value of the variable s?

One way of changing the value of a variable is to use a graphical icon to *scroll* through a range of values just like you scroll through a web page that is too large to fit on the screen. The icon used is called a ***slider***.



Right-click on the monitor that appears on the stage and select slider from the menu that pops up. The variable will be displayed with a slider below the name and value of the variable:  . By changing the position of the knob on the slider you can change the value of the variable. Recall that a variable can contain only one value at a time, so every time you move the slider, the new value is remembered and the old value forgotten.

To move the slider, place the mouse cursor on the knob, click the left mouse button and hold it down. As you move the mouse left and right, the knob on the slider moves and with it the value of the variable.

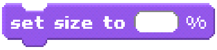
New construct in Scratch: sliders

Right-clicking on the monitor for a variable will cause a menu to appear; select slider. ***The value of the variable can be changed by dragging the knob on the slider left or right***. The current value is continuously displayed in the monitor.

Reading and using the value of a variable

Let us now change the behavior of the dragon so that the final size of the dragon can be controlled by the user. This size will be the value of the variable s as set by the user with the slider:

```
0. when the green flag is clicked
    1. initialize the sprite to its original size
    2. wait 2 seconds
    3. set the size of the sprite to the value of the variable s
```

The instruction  can be used to implement step 3.

? But how do we cause it to use the value of the variable s?

Previously, we entered values into instructions in one of two ways. For some instructions like move ○ steps we typed a number in the small window; for others like point in direction ▾ we clicked on the arrow to select a value from a menu.

In addition, there were cases where *a value was obtained from another block*. For example, in Chapter 4, we used the block wait until ⬡, where the condition was another block touching ▾ ?. The shape of a block determines where it can be used. The block for the condition *touching...?* has angled ends and therefore it is legal to place it in the window of the *wait until* block that also has angled ends.

For the block set size to ○ %, the rounded shape of the window means that any ***value block*** can be used. If you look at the block for the variable s, you will see that it also has rounded ends (in fact, since the variable name is so short, the block looks like a circle). Drag this block and drop into the window of the set size instruction. We obtain the instruction set size to s % which means:

> set the size of the sprite to the *current value* of the variable s.

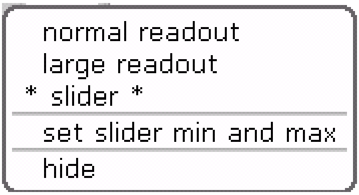
Run the script several times by clicking on the green flag, but before clicking, change the value of s by dragging the slider on the monitor for the variable. Check that the dragon is first displayed at its original size and then 2 seconds later at the size corresponding to the current value of the variable. Write comments for your project and save it with an appropriate name.

> Reading the value of a variable does not change the value that is stored there. It is like using the value of a telephone number written on a piece of paper; the number remains there until it is erased.

Changing the limits of the slider

By changing the value of the variable s, we have managed to make the dragon smaller than 100% of its original size. But everyone knows that dragons are very big!

? How can the user make the dragon bigger than 100% of its original size?



The user can change the limits of the range of the slider. Right click on the slider. The menu has the entry set slider min and max. Select this entry and enter 200 for the Max of the slider and click OK. The slider will now let the user enter values from 0 to 200. Set the slider to 200 and click the green flag. Add a comment to the project and save it with a new name.

Program file name: change-to-new-size

New construct in Scratch: setting the limits of a slider

Right-clicking on the monitor for a variable will cause a menu to appear. Select set slider min and max to enter the values for limits of the slider.

Repeated variation of the size of the dragon

Our project still isn't realistic, because you have to set the size using the slider *before* clicking the green flag to start the animation. In many computer games, you are allowed to change values at any time when the program is running.

Task 3

> Modify the animation so that the size of the dragon can be changed as often as you want by moving the knob on the slider.

Program file name: change-size-repeatedly

Exercise 1

> Plan the modifications needed for Task 3 and change the scripts as needed. Document and save the project.
>
> **Guidance:** Choose an appropriate control instruction.

Example 3
Adding buttons to the game

The project that you developed in the exercise allows the user to change the size of the dragon. Suppose now that you wish to return the size of the dragon to its original value 100%. You could do this by carefully dragging the slider for s so that its value becomes 100. However, it would be easier if there were a single button to click that could return the value of s to its initial value.

Task 4

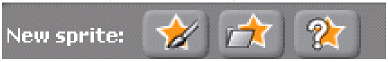
> Add a Reset button to the game. Clicking on this button causes the dragon to be displayed at its original size.

Program file name: change-size-with-reset

Even though the button is not an animated character and will not move on the stage, it still has to be defined as a sprite in order to respond to the event of being clicked on. The sprite behaves according to the following description:

```
0. when Reset clicked
   1. change the value of the variable s to 100
```

Since the script for the dragon sprite sets its size to the value of the variable s, changing this value in the script for the button sprite will cause the dragon's size to return to 100%. It will also set the slider position to 100 so that additional changes start from there.

The project archive accompanying this book contains a file reset-button.gif with the image that can be used for the reset button. To make the button sprite, create a new sprite by clicking on the middle button of the toolbar  and selecting this file. If you are curious to learn how we created the image for the Reset button, you can read the following description; otherwise, you can continue below with the development of the script for the button.

Initially, create the sprite for the `Reset` button by importing the button image from the Things folder. The button is blank so we need to add a label. Click on the Costumes tab in the middle panel and then click on the Edit button to run the Paint Editor. You will see the image of the blank button.

Click on the icon T (the second in the lower row of tool icons in the left middle of the Paint Editor screen). T stands for Text and you can now enter the text that will label the button. You will see two new images on the screen: a vertical blue line and a small black square. The blue line is a cursor for entering the text that is just like the cursor you are familiar with from using a word processor or entering text in your web browser.

Type in the word `Reset`; if you make a mistake, correct it using the arrow keys and the delete and backspace keys. The word that you type may not be exactly where you want it—in the center of the image for the button. The small black square can be used to drag and drop the entire line of text: Bring the mouse cursor to the black square, press the left mouse button and hold it down; now you drag the text to its correct position and release the mouse button to drop it into place. Click on OK when you have finished editing the button image.

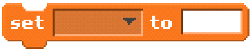
Now that we have a costume for the button sprite, let us construct the script that corresponds to the description above. The first step is *when* `Reset` *clicked*, which is similar to *when green flag clicked* and *when message received*, that is, the instructions are to be run *when* an event occurs.

The instruction (the third block from the top of the orange Control palette) is shaped just like the block when green flag clicked so that it can be used only as the first block in a script. The meaning of the block is that the blocks that follow it in the script are run only when the *sprite* `Reset` is clicked.

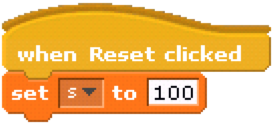
New construct in Scratch: responding to the event of clicking on a sprite

The instruction exists in the palette of every sprite with the name of the sprite appearing between `when` and `clicked`. The instruction can only appear as the first block in a script. ***The script is run when the sprite is clicked.***

Changing the value of a variable

Step 1 above requires that the value of the variable s be *changed by the script* and not by dragging the slider or typing a value on the keyboard. To do this we use the absolute instruction , which is the first block in the red-orange Variables palette that contains an instruction.

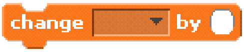
The first window must to be replaced by the name of the variable whose value we want to change. Since the names of all the variables in a project are known to Scratch, they are listed in a menu which can be opened by clicking on the arrow in the window. (Since there is only one variable in the current project, its name will already appear in the window.) Enter in the second window the new value that you want the variable to have by clicking in the window and typing the value. Check that the project works as required. Save the project under a new name after you write comments explaining the changes you made.



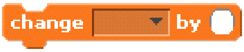
```
when Reset clicked
set s to 100
```

New construct in Scratch: setting the value of a variable

The instruction is an absolute instruction that ***sets*** the value of the variable selected in the first window to the value that appears in the second window.

If you want to change the current value of a variable, you can read the current value, change the value and then *set* the variable to this new value. Since this action is frequently done, Scratch provides a shortcut: an additional instruction that performs this action. The instruction  adds the value in the second window to the current value of the variable selected in the first window. If you want to subtract a value, simply add the negative of the value: *change s by -4* subtracts 4 from s.

New construct in Scratch: changing the value of a variable

The relative instruction  that ***adds*** the value in the second window to the current value of the variable selected in the first window.

When we compared a variable to a box containing money, we said that you have to store an entirely new value in a variable, unlike the real box where you can add or remove money. The *`change...by...`* seems to allow that but it is best to think of this instruction as described above: a shortcut for reading the value of a variable and then setting it to a new value.

Copying a value from one variable to another

Suppose that we have found an ideal size for our dragon, but we want to experiment a bit more with the size before making a final decision.

Task 5

Add two buttons to the animation:

Clicking the button `Save` will remember the current size of the dragon, while clicking the button `Restore` will change the size of the dragon to the value that was stored when `Save` was last clicked.

Program file name: change-size-with-save-and-restore

As with the `Reset` button, clicking on one of the two new buttons is an event so we need to respond to each of these events. Clicking on `Save` causes the current size of the dragon to be *remembered*; this is an indication that we need a new variable, which we will name `save`. Here is a description of the behavior:

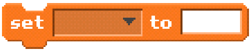
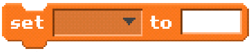
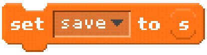
```
0. when Save clicked
    1. copy the current value of the variable s to the variable save
```

In the other direction, clicking on the Restore button will cause the remembered value in `save` to be stored back into the variable `s` so that it can change the size of the dragon:

```
0. when Restore clicked
    1. copy the current value of the variable save to the variable s
```

Create the new variable `save` and make sure that the small square next to the variable is checked so that a monitor for it will appear on the stage.

? How can be copy the value of one variable to another variable?

The instruction  sets (stores) a value into a variable. It can be used not only to store a fixed value like 100 but also to store the current value of another variable. Recall that the reporter for a variable returns its current value; it can be used in the second window of the instruction  to obtain the current value of a variable and set it into another variable. This implements the action of *copying*:  means copy the *current* value of the variable s and make it the value of the variable save. The previous value of save will be lost.

? Should the monitor for save be displayed with a slider?

The answer is no. The value of the variable save will be changed only by the script for the button Save. Since the user does not change it directly, no slider is needed.

Exercise 2

> Modify the program by adding new sprites for the two buttons. You can use the images we have prepared (save-button.gif, restore-button.gif) or you can create them using the blank button image and the Paint Editor. Construct the scripts for the Save and Restore buttons. Run the program and check that it performs as required.

Exercise 3

> In Task 3 in Chapter 5 (dance-indefinitely), the dancers moved towards each other until they touched and then moved away 10 times 10 steps.

a. Modify the animation so that the user can control the number of times the dancers move 10 steps in each direction.

Guidance: Add a variable `steps` with a slider and define reasonable limits on the slider. The dancers will use the value of this variable as the number of times they move 10 steps in each direction.

Program file name: dance-steps1

b. Modify this animation so that the two dancers need not have the same number of repetitions.

Guidance: Define two variables `Jay-steps` and `Cassy-steps`.

Program file name: dance-steps2

Exercise 4

a. Construct an animations for a rocket. The rocket is initially in the lower left corner of the stage; when the green flag is clicked, it moves at a constant speed until it reaches the upper right corner of the stage.

Guidance: Use a *`repeat until`* instruction that encloses a *`move`* instruction. The number of steps in the *move* instruction is the value of a variable Speed. Experiment with the initial position and direction of the rocket and with its speed until you are satisfied with its motion.

Program file name: rocket1

b. Modify the project so that the user can control the speed of the rocket. What happens if the speed of the rocket is negative? Explain.

Guidance: Display a slider for the variable Speed and choose approprite limits for the value of the variable.

Program file name: rocket2

c. Acceleration is the rate at which the speed increases or decreases. A real rocket accelerates (moves faster and faster) because as fuel is used up the rocket weighs less and there is less gravity and air resistance as it climbs above the earth. Modify the animation so that instead of the speed being determined by the user, it is initialized to zero and accelerates (increases) by a constant value each time the *move* instruction is run.

Program file name: rocket3

d. Modify the animation so that the *acceleration* increases.

Guidance: Add a variable `ChangeBy` with an initial value of 1. For each time the ***move*** instruction is run, the value of `Speed` will increase by the current value of `ChangeBy`, while the value of `ChangeBy` will increase by 1.

Program file name: rocket4

e. We would like to measure how much faster the rocket is in the animation of exercise (d) compared with the rocket in the animation of exercise (c). Unfortunately, the animations are too fast to measure, so we would like each rocket repeat its motion a large number of times, perhaps 20 times. That should take long enough so that you can measure the times of the animations on your watch or by using the stopwatch feature of your cell phone.

Guidance: Define another variable whose value will be determine the number of times that the motion of the rocket will be repeated. Measure the total time that the animation takes and divide by this number to obtain the average time that it takes the rocket to move from corner to corner. Experiment with the value of the variable until you get consistent measurements of the average time of the motion of the rocket.

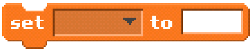
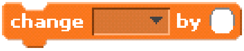
Program file name: rocket3a, rocket4a

Summary

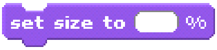
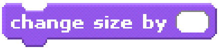
Concepts

Variables: A variable can remember one value. ***Reading*** a value does not cause the value in the variable to change, but ***writing*** a new value replaces the value that was previously stored in the variable.

Scratch instructions

Changing the value of a variable: To change the value of a variable, use the absolute instruction  or the relative instruction . The first instruction changes the value of a variable given in the first window to the value given in the second window. The second instruction changes the current value of the variable specified in the first window by the value given in the second window.

The new value can be either a number or the value of another variable (obtained by placing the reporter representing of the variable in the second window). The `set...to...` instruction with a variable in the second window *copies* the value of the second variable to the first variable.

Changing the size of a sprite: The size of a sprite can be changed in a script using the absolute instruction  or the relative instruction . The purple palette Looks contains a reporter for the size of a sprite. If you check the small square, the size will be displayed in a monitor on the stage. The monitor for size cannot contain a slider, because the user can only modify its value indirectly using these instructions.

Scratch techniques

Changing the size of a sprite: Above the stage are two buttons that turn the usual cursor into a cursor for growing or shrinking the sprite.

Creating a variable: Variables are created in Scratch by clicking on the button Make a variable in the red-orange Variables palette. A name must be given to a variable and you can decide whether the variable can be seen by all sprites or just one sprite (although we have not used the second possibility yet). When the first variable in a project is created, a group of blocks containing instructions for working with variables appears.

Reporters and monitors: When any variable is created, a ***reporter***—a block that represents its value—is created. The block has rounded ends and is a ***value block*** that can be used in other blocks where a value is needed, such as the blocks for setting the size of a sprite. Clicking the small square next to this block causes a ***monitor*** for the variable to be displayed on the stage. The monitor is automatically updated by Scratch to show the current value of the variable.

Sliders: Right-clicking on the monitor brings up a menu where a ***slider*** can be selected. The slider appears below the display of the value of the variable; it enables the user to change the value of a variable by dragging the knob on the slider. Right-clicking on the slider brings up a menu with the entry set slider min and max that can be used to change the range of the values of the slider.

Chapter 7

It Depends—Conditional Run

Control structures are central to the construction of programs because they enable us to control the order in which instructions are run. So far we have learned the following control structures: *repeated run* (finite or infinite, conditional or repeated a fixed number of times); running a script in response to an *event* (clicking on the green flag, receiving a message, pressing a key or clicking on a sprite); *waiting* (for a period of time or an event to occur before continue to run a script). In this chapter we will learn another structure for controlling the running of instructions in a script. It causes the running of a sequence of instructions to be dependent on a condition becoming true.

The project that we construct this chapter is a bit more complicated than the one that we constructed in the previous chapter. It is based on the well-liked game, Pac-Man. In the game, the player uses the keys to control the movement of the Pac-Man character so that it can navigate through a maze without hitting the walls. The goal is to maneuver the Pac-Man so that it "eats" objects placed within the maze. We will not develop the full game; for example, we will not have monsters who try to "eat" the Pac-Man. Feel free to extend the game once you have worked through this chapter.

We will construct the game in stages: first, we will cause the Pac-Man to move; then, we handle the event of its hitting the wall of the maze; next, we enable to user to control its movement; finally, we place bunches of bananas in the maze for the Pac-Man to eat. The major complications in the game will be to have it restart from the beginning when Pac-Man hits a wall.

Example 1
Walking through a maze

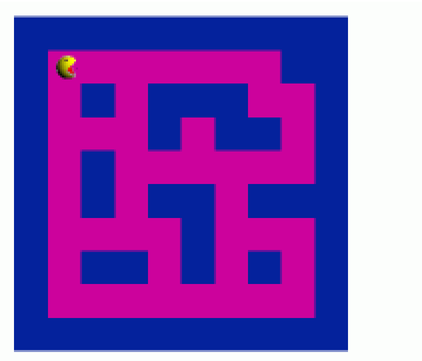
Task 1

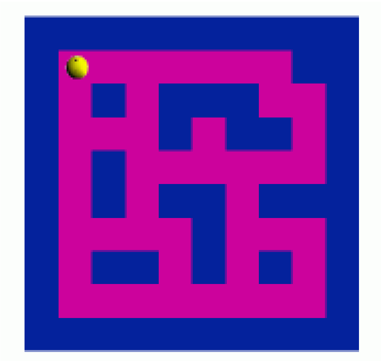
> Construct a project where the Pac-Man character is placed at an initial position within a maze. It opens and closes its mouth repeatedly.

Program file name: pacman-opens-and-closes-mouth

The maze can be either an appropriate background or a sprite; we choose to make it a sprite. Since both the maze and the Pac-Man are not part of the standard Scratch library, we have prepared an outline of a project that already contains these two sprites (file name costumes). Initially, the maze sprite will have no scripts, though we will add some later.

Next, we implement the action of opening and closing Pac-Man's mouth. This effect can be obtained by repeatedly changing between two costumes, one with an open mouth and one with a closed mouth. Changing costumes was explained in one of the optional sections of Chapter 5. If you have not already studied it, go back and do so before continuing with this chapter.





Here is a a description of the behavior of the sprite for this task:

```
0. when the green flag is clicked
    1. initialize the position, direction and costume
    2. repeatedly change the costume
```

Exercise 1

> Fill in the details of the description of Pac-Man's behavior: the three initializations, a repeated run and the instructions for changing the costumes. Implement the detailed description as a Scratch script, run it and check that it works. Add comments and save the project.
>
> **Guidance:** Click on the Costumes tab in the script area and you will see that there are two costumes defined for the Pac-Man sprite, one with its mouth open and one with its mouth closed.
>
> Initialize the sprite so that it is placed at position $(-115, 115)$, facing right with its mouth open. Changing the costume should be done as described in the Optional section of Chapter 5. In order for the animation to be seen clearly, add a short wait (perhaps 0.2 seconds) before the instruction that changes the costume.

The next task is to implement the movement of the Pac-Man. We start with the simple movement of the Pac-Man in a straight line; later, we will add the additional requirements: detection of collisions with the walls and controlling the direction. The movement has to occur at the same time as the changing of the costumes, so a separate script is used.

Task 2

> Expand the animation so that the Pac-Man moves in a straight line.

Program file name: pacman-moves

Animated movement is implemented by repeatedly moving a small number of steps. For now, we assume that the Pac-Man moves indefinitely:

```
0. when the green flag is clicked
    1. forever
        1.1 move 2 steps
```

The script implementing this description will run in parallel with the script for changing the costumes.

Duplicating the initialization instructions

In Chapter 10, we will explain that whenever two or more scripts run concurrently—at the same time—there are interactions between the scripts that can cause problems. For now, take our word for it and ensure that both scripts contain initialization:

```
0. when the green flag is clicked
    1. initialize the position, direction and costume
    2. forever
        2.1 move 2 steps
```

Construct the Scratch script for this description and run the two scripts concurrently by clicking the green flag.

Task 3

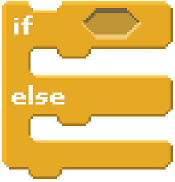
> Extend the game so that the Pac-Man will stop when it hits a wall and say that it is hurt.

Program file name: pacman-moves-and-hits-wall

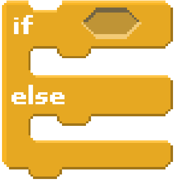
In the description of the movement of the Pac-Man sprite, the `move` instruction in step 2.1 is always run, but now we want the Pac-Man to run a different instruction if it hits the wall. Therefore, we will replace step 2.1 with the following step:

```
2.1 if you hit the wall
    2.1.1 say "Ouch!"
2.2 otherwise
    2.2.1 move 2 steps
```

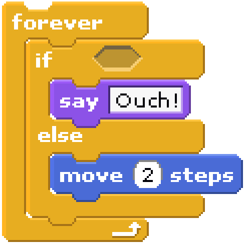
This control structure is called ***conditional run*** because the decision to run certain instructions (`say` and `move`) depends on whether a condition is true or not. First we check a ***condition***: *2.1 if you hit the wall*. If the condition is true, then the `say` step (2.1.1) is run; *2.2 otherwise*—if the condition is false—the `move` step (2.2.1) is run. Each time the run reaches step 2.1, the truth of the condition is checked and a decision is made whether to run step 2.1.1 or step 2.2.1, but never both of them.

The conditional run instruction  is the fifth block from the bottom in the orange Control palette. Like other conditional control structures, there is a window with angled corners where a condition must be added, in our case: *you hit the wall.*

The construction run instruction is sometimes called an ***if-instruction*** because it starts with the word if.

Unlike the other conditional control blocks,  block has *two* "mouths." In the first one, we will place the instructions to be run if the condition is true (in our case, *2.1.1 say "Ouch!"*), and in the second mouth, the instructions to be run if the condition is false (in our case, *2.1.1 move 2 steps*).

Drag this block to the script area and drop it near the script for moving the sprite. Re-arrange the instructions so that the `if-then` block is contained within the "mouth" of the `forever block`, and so that the `say` and `move` instructions are placed within the "mouths" of the `if-then` block:

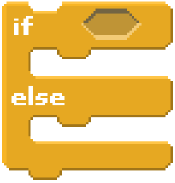


```
forever
  if < >
    say Ouch!
  else
    move 2 steps
```

New concept: conditional run

Conditional run consists of a condition and two sequences of instructions. When it is run, the condition is checked: if it is true, the first sequence is run, while if it is false, the second sequence is run.

New construct in Scratch: conditional run

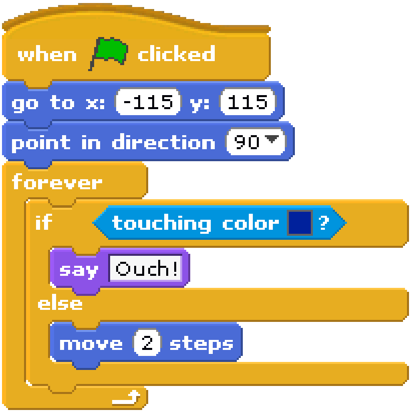
The instruction  is used for conditional run. The window with angled ends contains the condition that is checked. The first "mouth" contains the sequence of instructions that are run if the condition is true, while the second "mouth" (following the word `else`) contains the sequence of instructions that are run if the condition is false.

The condition of hitting the wall

Now we have to add a condition to the conditional run. The condition must be true when the Pac-Man sprite hits the wall of the maze. We use the fact that the maze is drawn using two colors: dark pink for the paths through the maze where the Pac-Man is allowed to move and dark blue for the walls of the maze that surround the paths. We have already encountered situations where we used conditions of the form *"Does the sprite touch something?"* Previously, the conditions were for touching another sprite, the edge of the stage or the mouse cursor. Here, we can use a condition that checks if the sprite is touching an area with a given color: touching color ■ ?, which is the second block in the light blue Sensing palette. The block has a small window for specifying the color that needs to be touched for the condition to be true. To change the color to the color used for the walls of the maze, do the following:

> Click on the small window for the color. The mouse cursor will change to a dropper like the ones used to measure liquid medicines. Move the mouse until the bottom of the dropper is touching a wall of the maze. Click again. The color in the small window become that of the walls of the maze.

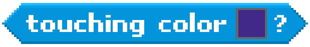
Here is the complete script for moving the Pac-Man:



```
when 🏴 clicked
go to x: -115 y: 115
point in direction 90▾
forever
    if touching color ■ ?
        say Ouch!
    else
        move 2 steps
```

Click on the green flag to run the animation. Add comments and save the project.

New construct in Scratch: touching a color

The condition  touching color ■ ? is true if the sprite is touching an area (the background or another sprite) whose color appears in the window.

The player controls the motion of the Pac-Man by using variables

The next stage of the development of the game is to allow the player to control the Pac-Man sprite so that it can move through the maze without colliding with the walls.

Task 4

> Modify the animation so that the player controls the direction of the Pac-Man using the arrow keys: the up arrow will cause the Pac-Man to turn up 0°, the right arrow will cause it to turn right 90°, the down arrow to turn down 180° and the left arrow to turn left $-90°$.

Program file name: user-control-of-pacman

In Chapter 5 we constructed an animation in which the player modified the motion of the dancing sprites by pressing keys. Conditional repeated run instructions that used the condition key ▾ pressed? caused the repeated motion of a sprite to stop when a key was pressed. However, the dancing sprites had to respond to only two possible key presses at any time, while the Pac-Man sprite needs to respond to four possible key presses. Therefore, a solution with conditional run instructions will be quite complicated. (Exercise 7 asks you to construct a project using conditional run.)

Instead, we will use a new control instruction, one that causes a script to be run when a key is pressed. The block when space ▾ key pressed is found just below the block *when green flag clicked* in the orange Control palette.

We need four scripts, one for each key.

? Where shall we put these scripts?

The obvious place to put them is in the script area for the Pac-Man sprite. However, the script area is quite small and with so many scripts it will be difficult to find them.

Assigning the responsibility for reacting to the keys to the maze

We choose to assign the task of reacting to key presses to the maze sprite, which until now has not had any scripts associated with it. The scripts for the maze sprite will be responsible for reacting to the key presses, while the scripts for the Pac-Man sprite will be responsible for changing direction.

? How are the two sprites going to communicate with each other?

After all, one sprite can not change the direction of another sprite. In Chapter 4 we saw one way that sprites can communicate—by sending and receiving messages—but to do this, we would have to use four messages, one for each key, and the Pac-Man would need four additional scripts, one for receiving each of the messages. This solution will not solve the problem of having too many scripts in the Pac-Man sprite.

Communicating using a variable as a mailbox

The maze sprite and the Pac-Man sprite will communicate using *variables*, which were discussed the previous chapter. The maze will notify the Pac-Man that a key has been pressed by using a ***mailbox***, a variable that is shared by two sprites. Whenever the player presses a key, the maze will change the value in the mailbox to the direction associated with the key. Whenever the Pac-Man needs to change its direction, it will use the direction that is saved in the mailbox.

Since the variable will store the direction to which Pac-Man needs to turn, we will name the variable `turn`. The maze has the responsibility for setting the value of `turn` when the player presses a key. We need four scripts, one for each key; for example, the script for the up arrow is as follows:

```
0. when the up arrow key is pressed
    1. set the variable turn to point up
```

Declare a variable for the mailbox by clicking on the button Make a variable that appears in the red-orange Variables palette. Since this variable will be used by more than one sprite, leave the selection For all sprites checked. There is no need to display the monitor for the variable, because we are not going to read or change its value directly; it is used only for communication between the two sprites.

Exercise 2

> Construct the four scripts in the maze sprite for reacting to the arrow keys.

Exercise 3

> What should the initial value be for the variable `turn`? Add the appropriate initialization instruction to *both* scripts of the Pac-Man sprite.

The Pac-Man sprite uses the values in the variable

The next step is to modify the behavior of the Pac-Man so that it will change its direction depending on the value in the variable `turn`. Currently, the Pac-Man is given an initial direction pointing right and then it repeatedly moves two steps in that direction. We need to change this so that after each movement the Pac-Man will change its direction if needed. This will happen so fast that the player will think that the Pac-Man changed its direction immediately after the key was pressed. Here is a description of the required behavior:

```
0. when the green flag is clicked
    1. initialize the position, direction, costume and the variable turn
    2. forever
        2.1 if you hit the wall
        2.1.1 say "Ouch!"
    2.2 otherwise
        2.2.1 move 2 steps
        2.2.2 change direction according to the value of turn
```

Make the appropriate modification to the script of the Pac-Man sprite, add comments and save the project under a new name. Run the scripts by clicking on the green flag and use the arrow keys to guide the Pac-Man through the maze without hitting the walls.

Setting the pace of the game

It is not at all easy to control the Pac-Man sprite because it is moving so fast. Slow down the movement of the Pac-Man by adding a short wait after each run of the loop:

2.2.3 wait 0.01 secs

Make this change and see if it makes it easier to play the game. Experiment with the length of the wait until you can control the Pac-Man, but it still moves fast enough for the game to be interesting.

Example 2
Pac-Man doesn’t give up—restarting the game

The game terminates when the Pac-Man hits a wall. The sprite stops moving but continues to change costumes and to say “Ouch!” Even dragging the Pac-Man to the initial position with the mouse does not restart the game. Try it; although the Pac-Man moves according to the keys that the player presses, it continues to say “Ouch!” The game can only be restarted by clicking on the green flag.

Task 5

> Modify the animation so that Pac-Man says "Ouch!" for a short period of time after it hits the wall and then the game restarts from the initial conditions.
>
> Program file name: restart-game

What happens in the current script when the Pac-Man sprite hits a wall? The script continues in an infinite loop checking whether Pac-Man is hitting a wall and if so saying "Ouch!" Of course, since the sprite has not moved, it is still touching the color of the wall and so nothing changes.

Let us limit the length of time that Pac-Man says "Ouch!" to two seconds. When the time is up, the initialization steps will be done again, so that when the infinite loop starts, the Pac-Man sprite no longer touches the wall. The behavior of the sprite is now:

```
0. when the green flag is clicked
    1. initialize the position, direction, costume and the variable turn
    2. forever
        2.1 if you hit the wall
            2.1.1 say "Ouch!" for 2 seconds
            2.1.2 initialize the position, direction, costume
                  and the variable turn
```

```
2.2 otherwise
    2.2.1 move 2 steps
    2.2.2 change direction according to the value of turn
    2.2.3 wait for 0.01 seconds
```

Make the appropriate changes to the script. Note that both initialization steps, 1 and 2.1.2, are implemented with four instructions.

Add comments and save the project under a new name. Click the green flag to run the animation and check that it performs as required.

Example 3 Pac-Man turns green—more on conditional run

In most games, when something happens to a character, it changes its appearance. Let us do the same to the Pac-Man. Instead of just saying "Ouch!", it will change its color to green out of shame at crashing into the wall. On the surface, this seems like a simple change to the project, but any change to a computer program must be done carefully, because it is can cause unforeseen problems. The change we are going to make will raise problems that will enable us to learn more about conditional runs.

Task 6

> Modify the animation so that the Pac-Man changes color to green when it hits a wall.

First attempt at a solution

The first step is to create a new costume for the Pac-Man sprite. You can use the file pac-man-green.gif or create it yourself as described below.

The remaining change seems to be very simple. Replace the steps:

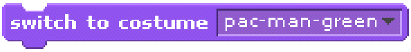
```
2.1.1 say "Ouch!" for 2 seconds
2.1.2 initialize the position, direction, costume and the variable turn
```

by:

```
2.1.1 change costume to pac-man-green
2.1.2 wait for 2 seconds
2.1.3 initialize the position, direction, costume and the variable turn
```

The initialization after changing the costume also initializes the costume (to `pac-man-open`) so we added a `wait` instruction to ensure that we have time to see the green costume.

Use an absolute instruction to change the costume:



switch to costume pac-man-green ▾ .

Exercise 4

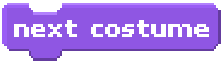
Make the changes described and run the animation. Explain the behavior of the Pac-Man sprite.

Program file name: pacman-change-to-green-bug1

> **Modifying the costume:** First, create a copy of the Pac-Man costume: Click on the Costumes tab, and click on Copy for the costume pac-man-open. A new, third, costume will appear; give it an appropriate name such as `pac-man-green` by clicking in its name field and typing the new name. Now, click on Edit to start the Paint Editor. Change the colors in the costume to some shade of green using the Fill tool (the one that looks like a bucket of paint being poured). Click OK when you are finished.

Analyzing the problem

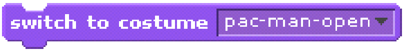
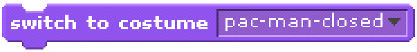
The change that we made did not take into consideration the entire behavior of the sprite. In addition to the script that is responsible for the movement of the sprite, there is a second script that is responsible for causing Pac-Man to open and close its mouth by changing the costumes of the sprite. These two scripts run concurrently. This results in two problems:

- When the Pac-Man starts moving—even before it hits the wall—the green costume is displayed along with the two others `pac-man-open` and `pac-man-closed`. The reason is that the script that changes the costumes uses a relative instruction:  next costume. This worked well when there were just two costumes, but we have added a third costume that should not appear unless the Pac-Man sprite hits the wall. Unfortunately, the relative instruction *next costume* does not know this, and includes the `pac-man-green` costume in the changes it makes while the sprite still moving.

- When the sprite does hit the wall, the script continues to change costumes instead of showing just the green costume for two seconds before restarting.

Second attempt at a solution

To solve the first problem, we have to change the script for opening and closing Pac-Man's mouth so that it no longer uses the relative instruction. Replace the relative instruction with two absolute instructions:

-  switch to costume pac-man-open to change to the open costume
-  switch to costume pac-man-closed to change to the closed costume.

? How do we know when to changed to each of the costumes?

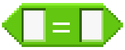
Clearly, when the current costume of the sprite is `pac-man-open` we have to change it to `pac-man-closed` and when it is `pac-man-closed` we have to change it to `pac-man-open`. Let us implement this using a conditional run:

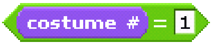
```
4.2 if the current costume is pac-man-open
    4.2.1 switch to the costume pac-man-closed
4.3 otherwise
    4.3.1 switch to the costume pac-man-open
```

Comparing values

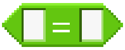
In order to translate this into Scratch, we have to check if the current costume is equal to the open one or to the closed one. That is, we need a condition = (equality) that compares two costumes. Unfortunately, there is no way to do this in Scratch, but it is possible to obtain the *number* of the current costume of a sprite and then to check if this number is equal to some value:

```
4.2 if the number of the current costume = the number of the pac-man
    4.2.1 switch to the costume pac-man-closed
4.3 otherwise
    4.3.1 switch to the costume pac-man-open
```

The condition  can be found in the green Operators palette, the seventh block from the top. The block has angled ends and can be used whenever a condition is needed (such as in an *if-then* block for conditional run or in a *repeat until* block for conditional repeated run). The block for the condition has two small windows in which we enter the values that we want to compare.

The third block from the top the purple Looks palette costume # is the reporter for the number of the costume. The block itself has rounded ends that fit into the small windows of the equality condition. We can drag this block and drop it into one of the two windows. If we click on the other window, we can enter a number. Thus, we can create the condition  costume # = 1 that is true if the current costume is the first costume, which is `pac-man-open`. The condition can then be used in a conditional run.

New construct in Scratch: equality

The block  = is a condition that is true if the value in the left window equals the value in the right window; it is false if the values are different.

New construct in Scratch: costume number

The block is a reporter for the current ***costume number*** of a sprite. Click the Costumes tab for a sprite; the costume number is written to the left of the image of the costume. The numbers are assigned in the order that the costumes are displayed from top to bottom.

We have been very careful to ensure that the shape of a block corresponds exactly to the shape of a window: conditions have angled ends, while rounded windows accept values and reporters. The windows in the equality operator have straight sides, neither rounded nor angled. Such windows are very permissive: we are allowed to enter a number or a string (such as ”Ouch!”), *and* we are allowed to drop blocks that have rounded ends (like reporters) or angled ends (like conditions). In most cases, we will drop reporters with rounded sides when we want the value of a variable, or we will enter numbers. As we have seen, reporters may be for variables that already exist in Scratch (such as `costume#`) or those that we declare (such as `turn`).

Exercise 5

> Change the scripts to use absolute instructions for changing costumes and run the animation by clicking the green flag. Explain what happens.

Program file name: pacman-change-to-green-bug2

Analyzing the problem

We have solved the first problem that we found. The new script for changing costumes does not use the green costume when the Pac-Man is moving because we are using absolute instructions that do not include the green costume. However, we have not solved the second problem: when the Pac-Man hits the wall and stays there for 2 seconds before starting the game again, the costumes continue to change although we want the Pac-Man to remain green.

? Why does this happen?

The idea behind the changes was correct: we want to ensure that the open and closed costumes aren't displayed when the current costume is green. Unfortunately, there is a small but significant problem in the way that we translated this idea into the behavior of the sprite. Here is a description of what we want:

```
if Pac-Man is wearing the open costume
    change it to the closed costume
```

```
otherwise if Pac-Man is wearing the closed costume
    change it to the open costume
```

Instead, our script implemented the following behavior:

```
if Pac-Man is wearing the open costume
    change it to the closed costume
otherwise
    change it to the open costume
```

? Can you find a costume for which the two descriptions are different?

The two descriptions work differently when the current costume is the green one.

- According to the first description, when the costume is green then it is *not* the open costume so it is *not* changed to the closed costume. Moving to the *otherwise*, when the costume is green then it is *not* the closed one, so it is *not* changed to the open costume. The result is correct: the green costume is not changed.

- According to the second description, the first part of the statement works the same (it is *not* changed to the closed costume), but the second part of the statement is different. Since the costume is green, it is *not* the open costume, so the *otherwise* part is run instead and this changes the costume to the open costume. From then on, as the statement is repeated, it will change the costume to the closed costume, then to the open costume and back again.

Nesting conditional run instructions

In order to solve this problem, we need to use conditional run instructions in a more complex manner:

```
4.2 if the current costume is pac-man-open
    4.2.1 switch the costume to pac-man-closed
4.3 otherwise
    4.3.1 if the current costume is pac-man-closed
        4.3.1.1 switch the costume to pac-man-open
```

This structure integrates conditional run instructions in a new way: one conditional run instruction is contained within another. The word *if* appears twice, in step 4.2 and in step 4.3.1 which is part of the behavior defined by the *if* in step 4.2.

When one structure appears within another structure, they are called ***nested structures***. We have already seen an example of nested structures: In Chapter 5, one repeated run instruction was nested within another repeated run instruction. In fact, any control instruction can appear within any control instruction, either the same one or another one. In the first example in this chapter, there was a conditional run instruction contained within an infinite run instruction.

New concept: nested conditional run
Any instructions can be placed within a conditional run, in particular, another conditional run. This is called ***nesting***.

Conditional run without an alternative

There is a difference between the conditional run in step 4.2 and the one in step 4.3.1. The first has an ***alternative*** (*otherwise*) that is run if the condition is not true, while the second one does not have an alternative. If the condition is false, nothing happens. This is called a ***conditional run without an alternative***.

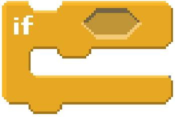
Let us carefully check steps 4.2–4.3 for the three possible values of the current costume:

- The current costume is `pac-man-open`. Steps 4.2 and 4.2.1 cause the costume to be changed to `pac-man-closed`.
- The current costume is `pac-man-closed`. Since this is *not* `pac-man-open`, step 4.2 does not cause step 4.2.1 to be run; instead, step 4.3 *otherwise* causes step 4.3.1 to be run. The current costume is checked *again* and

found to be `pac-man-closed`, so step 4.3.1.1 is run and the costume is changed to `pac-man-open`.

- The current costume is `pac-man-green`. Now, neither the condition at step 4.2 nor the condition at step 4.3 is true and nothing is done.

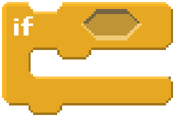
You can see that these steps change costumes exactly as we want.

Let us now implement these instructions in Scratch. The block 

implements *conditional run without an alternative*; it appears just above the block for conditional run (with an alternative) in the orange Control palette. The block also has a window for the condition, but it has only one "mouth" for the instructions that will be run if the condition is true.

New concept: conditional run without an alternative
A conditional run need not have an alternative that is run if the condition is false. In that case, nothing happens.

New construct in Scratch: conditional run without an alternative

The block 

implements conditional run without an alternative. Since nothing happens if the condition is false, there is no second "mouth" labeled `else`.

Exercise 6

Use the conditional run without an alternative to implement steps 4.2–4.3 as described above. Check that the change of costumes is correct during the various stages of the game: when the Pac-Man moves through the maze, when it hits a wall and when the game restarts after it hits a wall.

Program file name:
pacman-change-to-green-correct

Exercise 7

When solving Task 4, we mentioned that there is a different method of responding to the keys pressed by the player, where the responsibility of responding to the keys remains with the Pac-Man sprite and not with the maze. Implement the example using *nested* conditional run instructions, where the condition in each instruction checks if one of the keys has been pressed. The script responsible for the movement of the sprite will also be responsible for checking if a key is pressed and changing the direction accordingly. When your script runs correctly, explain why some conditional run instructions have alternatives and some do not.

Program file name: user-control-nested-if

Example 4
To complete the game, let Pac-Man eat bananas

In the real, Pac-Man scores points whenever it “eats” a dot placed within the maze. In our game, Pac-Man will “eat” bananas; for simplicity, we limit ourselves to two bunches of bananas and we leave it to you to expand the project by keeping score of the number of bananas that are eaten.

Task 7

> Modify the animation so that two bunches of bananas are placed at fixed positions in the maze. When the Pac-Man sprite touches a bunch of bananas, the bananas disappear.

Program file name: pacman-eats-bananas

Since the two banana sprites behave the same, we start by implementing one of them.

? What will happen when the bananas are eaten?

The task of eating the bananas is the responsibility of the Pac-Man. The bananas are passive and all they have to do is to disappear from the stage when eaten. The sprite for the bananas need only initialize its position and disappear from the stage when the Pac-Man sprite touches it:

```
0. when the green flag is clicked
    1. show
    2. wait until touching the Pac-Man
    3. hide
```

We already know how to implement step 2 using a *wait until...* instruction with a condition. The instructions that cause a sprite to appear and disappear are **show** and **hide**; they can be found near the bottom of the purple

Looks palette. The *show* instruction is part of the initialization; it ensures that the bananas are visible after the green flag is clicked. Without this instruction, you will not see the sprite after playing the game for the first time.

New construct in Scratch: hiding and showing a sprite

The instruction show causes the sprite to appear on the stage.
The instruction hide causes the sprite to disappear from the stage.

Initializing the bananas

There is problem with this script. The bananas appear when the green flag is clicked at the beginning of the game, but once they disappear, they do not re-appear when the game restarts after the Pac-Man hits a wall. The Pac-Man sprite has to notify the banana sprites that it is restarting the game. This is easy to do with a message; when the banana sprite receives the message it will reappear:

```
0. when I receive the message Initialize
    1. show
    2. wait until touching the Pac-Man
    3. hide
```

The Pac-Man sprite will broadcast this message before the game is restarted.

When a sprite disappears, it still "remembers" all of its properties like its position, direction and costume, so a subsequent *`show`* instruction will cause it to re-appear exactly as it was when *`hide`* was run.

> There is no need for two almost identical scripts in the banana sprite: one that is run when the green flag is clicked and the other that is run when the game restarts. If the Pac-Man sprite broadcasts the `Initialize` message during its initialization, we can delete the script that is run when the green flag is clicked.

Exercise 8

> Implement this solution:
>
> - Add *`broadcast`* instructions to the script of the Pac-Man.
> - Construct one banana sprite from the image bananas1 from the Things folder. Use the button above the stage to set the mouse to the mode where it can shrink a sprite, and resize the sprite until it can fit comfortably in the paths within the maze. Place it somewhere with in the maze, far

enough from the initial position of the Pac-Man so that it will be challenging for the player to guide the Pac-Man to eat the bananas.

- Write the script for this sprite.
- Duplicate the banana sprite at another location in the maze. Duplicating a sprite was explained in Chapter 2 and we repeat the explanation here for convenience:

  > Click on the leftmost button above the stage . Click on the sprite in the sprite area (a copy of the sprite will appear). Drag the image of the new sprite to an appropriate place.

- Add comments to the project and save it. Play the game and see if you can successfully guide the Pac-Man to eat the two banana sprites.

Let us now leave the Pac-Man game and look at other uses of conditional run.

Example 5 Random numbers

Random numbers are like a lottery: you don't know in advance what numbers will be chosen; all you know is that they will be selected from a certain range such as from 1 to 36. The concept is also familiar from games: in card games you can be dealt any of the 52 cards from the deck and in dice games the numbers 1 through 6 are equally like to appear on each die that is thrown.

A computer can generate sequences of random numbers and they are used in many applications. Computer games use random numbers to make the game unpredictable. Random numbers are used in simulations, for example, of traffic flow, where we can't predict in advance when people will drive their cars. Random numbers are even used in cryptography to keep a credit card number secret when shopping in an online store.

In this section we show how to use the conditional run instruction together with random numbers to create an entertaining version of the animation of a soccer game from Chapter 4. The project in that chapter had three sprites: the referee who gives the signal for the opening kickoff, a soccer player Pele who kicks the ball when he receives the referee's signal and the soccer ball. Here, we replace the referee with a goalie who has the difficult task of trying to block a kick by Pele. Of course, the goalie's job would be simple if he always knew where the ball was going, so we use random numbers to make its path unpredictable.

Task 8

When the green flag is clicked, Pele will kick the

> ball. If the goalie stops the ball, it will bounce back in the direction of Pele. Otherwise, it will stay at the edge of the stage, which we take to mean that the ball entered the goal. Pele will kick the ball in a *different* direction each time that the animation is run and we won't know the direction of the ball until *after* it is kicked.

Program file name: cat-kicks-ball-randomly

Kicking the ball

Let us start with the behavior of Pele. When the green flag is clicked, Pele is positioned at the right edge of the stage, facing left. He says "Let's go" and then kicks the ball:

```
0. when the green flag is clicked
   1. initialize the position and direction
   2. say "Let's go"
   3. kick the ball
```

Pele is *not* responsible for the movement of the ball sprite so he is also not responsible for determining the direction in which the ball moves.

Exercise 9

> Construct the script for Pele to implement the behavior described above.

Guidance: Pele's initial position will be (190, 0) and his initial direction will be pointing left. In order to kick the ball, he moves 40 steps in that direction.

The goalie hopes for the best

The goalie will simply stand at the left edge of the stage and wait:

```
0. when the green flag is clicked
   1. initialize the position and direction
```

Exercise 10

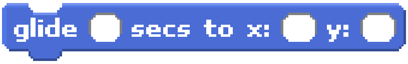
Construct a script for the goalie.

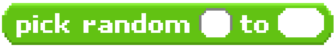
Guidance: Choose one of the People images like amon1 for the goalie. The goalie's initial position will be $(-200, 0)$ facing right.

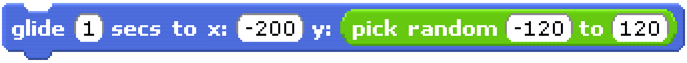
The ball's random behavior

The ball has to do the following: wait until it is kicked by Pele and then move in the general direction of the goalie. If the goalie hits the ball, the ball must turn around and move back in the direction of Pele. Here is a description of the behavior of the ball:

```
0. when the green flag is checked
    1. initialize the position and direction
    2. wait until kicked by Pele
    3. move in a **random** direction
    4. if hit by the goalie
        4.1 turn around and move in the direction of Pele
```

Your knowledge of Scratch is enough to implement this behavior script, except for step 3. This step can be implemented by the instruction  glide ( ) secs to x: ( ) y: ( ). We know what the value of x is supposed to be: the left edge of the stage which is at −200, and it is easy to find the number of seconds for the glide so that the animation can be easily seen. But what about the value of y? We want the ball to *choose* the number by itself just as if it were picking the number from a lottery.

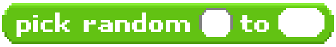
Random numbers are obtained by reading the value of  pick random ( ) to ( ), the fifth block from the top in the light green Operators palette. When its value is read, a number within the range given by the values in the two windows is obtained. Just as in a lottery, if you read its value again and again, you will probably receive different numbers each time. Even if the number does repeat itself, you never know when that is going to happen. This block has rounded ends just like a reporter for a variable. Therefore, it can be used anywhere that a number is expected:



```
glide 1 secs to x: -200 y: pick random -120 to 120
```

New concept: choosing a random number
Given a range of numbers (such as 1 to 100), choosing a random number means that some number will be chosen from the range, but the values of subsequent choices will be unpredictable and a number may be chosen multiple times. We assume that each number within the range has a roughly equal chance of being obtained.

New construct in Scratch: choosing a random number

The block  `pick random ( ) to ( )` is a reporter block whose value can be read. These values are selected randomly from the range of numbers defined by the values in the two windows.

Exercise 11

Construct a script for the ball sprite. Its initial position will be (120, −20) facing left. The ball will move using the instruction *glide* 1 *secs to x:* −120 *y:...*, where the value of y will be a random number between −150 and 150. When the glide is finished, the ball will check if it is touching the goalie; if so it will turn around and move a few steps in the direction of Pele. Add comments and save the project. Run the project many times and check that the ball glides to a different y-position each time.

Program file name: cat-kicks-ball-randomly

Additional material on Scratch: Brightness and color

We will make one more change to the soccer game.

Task 9

If Pele scores a goal (that is, if the ball is not blocked by the goalie), then his image on the stage will change color, become brighter and jump for joy; the goalie, however, will become darker and fall on the ground out of shame for failing to block the ball. On the other hand, if the goalie blocks the ball, he will change color, become brighter and jump for joy, while Pele will become darker and turn his face away.

Program file name: cat-kicks-ball-randomly-change-effect

The ball is responsible for reporting collisions

Although we are enriching the behavior of Pele and the goalie, the only sprite that *knows* whether a goal has been scored or not is the ball. The ball sprite will detect if it collides with the goalie and it has to communicate this fact to the goalie. Let the ball send one of two messages, `Goal` or `Stopped`; the behavior of the goalie sprite will depend on which message it receives.

Exercise 12

> Make the appropriate changes in the script for the ball sprite. In addition to adding ***`broadcast`*** instructions, you will have to change the conditional run instruction. Explain.

Pele and the goalie change their behavior

Let us now change the behavior of Pele and the goalie. We must add two scripts to each sprite, one for each of the messages that can be received. The behavior of the goalie can be described as follows:

```
0. when I receive the message Goal
    1. darken the image
    2. fall down

0. when I receive the message Stopped
```

```
    1. brighten the image
    2. change color
    3. jump up
```

The behavior of Pele is similar with the messages exchanged:

```
0. when I receive the message Stopped
    1. darken the image
    2. turn around

0. when I receive the message Goal
    1. brighten the image
    2. change color
    3. jump up
```

There are many similarities between these descriptions, both in their structure and in the steps that change the appearance of the sprites: brightening or darkening the image, changing color, or moving.

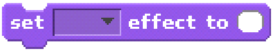
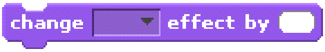
- *Falling down* is done simply by turning to face downwards, while *turning around* is a turn from facing left to facing right.
- *Jumping up* is implemented with a *`glide`* instruction: its x-value is the same as the initial x-value of the sprite, while the y-value is the initial value to which 100 steps have been added.

Exercise 13

> Construct partial scripts for Pele and the goalie. They start with `when I receive...` instructions, followed by the appropriate turn and glide instructions.

Changing effects

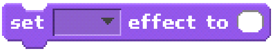
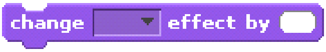
Changing the appearance of a sprite is done by changing graphical *effects* using instructions that appear in the purple Looks palette:

- The absolute instruction ;
- The relative instruction .

Click on the small arrow in the first window in these instructions: you will see a list of effects that can be changed. For this project, change the color and brightness effects, but feel free to experiment with the other effects.

The second window in the instructions controls by how much the effect is changed or the absolute value that is set. In general, these values range from 0 to 100 or from -100 to 100. For example, negative numbers will make the image darker, while positive numbers will make it brighter. Again, experiment with these values to achieve the effect that you want.

New construct in Scratch: setting and changing graphical effects

The block  `set [ ▾] effect to ( )` sets a graphical effect of the image of the sprite. The effect is selected in the first window and the value of the effect is entered in the second window.
The block  `change [ ▾] effect by ( )` changes the value of the selected effect by the value in the second window.

Exercise 14

Complete the scripts for Pele and the goalie using these instructions for setting and changing graphical effects.

Additional Exercises

Exercise 15

a. Construct an animation of a nervous grasshopper. Initially, the grasshopper will be at the center of the stage, facing right. It starts walking four steps at a time, but it is so nervous that after every four-step walk it *randomly* decides if it should reverse its direction, turning 180° from right to left or from left to right. Run the animation many times: does the grasshopper always tend to stay near the middle of the stage or does it wander off to one side?

Guidance: Use the grasshopper1 sprite from the Animals folder. Since it will move only left or right, click the ↔ button next to the sprite's name.

After each four-step walk, the sprite randomly chooses 1 or 2. If 1 is chosen, the grasshopper reverses direction; otherwise, it doesn't.

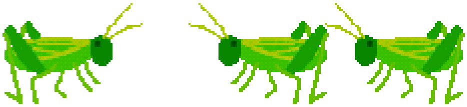
To make it easy to follow the grasshopper's position, display its x-position on the stage, by clicking the small square next to the sprite's reporter for its x-position at the bottom of the Motion palette.

Program file name: grass1

b. Add another grasshopper whose initial position is 100 steps to the right of the initial position of the first one. The second grasshopper doesn't move. The first grasshopper says "Nice to meet you!" if it touches the second one.

Program file name: grass2

c. Add a third grasshopper on the other side of the moving one:



This grasshopper also doesn't move. The first grasshopper says: "Nice to meet you G2!" or "Nice to meet you G3!" depending on which grasshopper it touches. Are both grasshoppers touched equally often?

Program file name: grass3

d. Modify the previous exercise so that the two fixed grasshoppers say "I'm G2. Nice to meet you!" or "I'm G3. Nice to meet you!". The moving grasshopper doesn't say anything.

Program file name: grass4

e. Just for fun, initialize the three grasshoppers with different color effects.

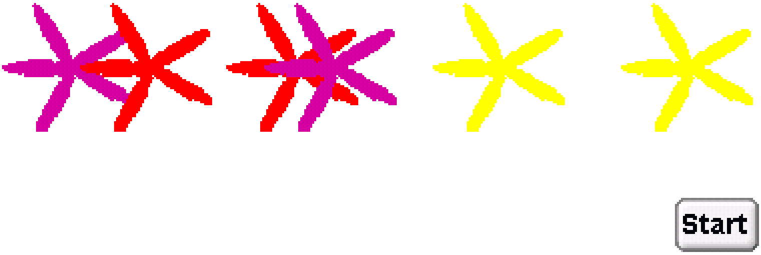
Program file name: grass5

f. Place four stationary grasshoppers surrounding the moving one: above and below, as well as to the left and the right. The moving grasshopper now turns in a random direction from -180 to 180 after each move. Be sure to click the button ↻ next to the sprite's name so that it can rotate in all directions.

Program file name: grass6

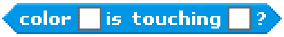
Exercise 16

Flowers of the same color want to be friends and touch each other. The stage starts with three pairs of flowers, one pair of each color: red, yellow and violet. Place a Start button on the stage. (You can find the sprites already defined in the file flowers-costumes.)



In all the following exercises, the six flower sprites all run similar scripts so you can copy a script from one sprite to another and then make the required modifications as was explained in Chapter 2: right click on the script, select duplicate and drag the copy to another sprite in the sprite area below the stage.

a. The flowers are initially placed at *random* positions along the x-axis from −220 to 220, facing right. When the Start button is clicked, they move along the x-axis (bouncing if they hit the edge of the stage) until each flower touches the other flower of its color.

Guidance: Use the condition  color □ is touching □ ? from the Sensing palette, specifying the same color in the two windows.

Program file name: flowers1

b. Modify the previous animation so that the initial direction of each flower sprite is random, either 90° or −90°.

Program file name: flowers2

c. Construct a two-dimensional version of the animations by having the flowers select a random initial direction in the range −180° to 180°.

Program file name: flowers3

d. Modify the previous animation so that the initial y-position of each flower is a random number in the range −150 to 150.

Program file name: flowers4

e. Modify the previous animation so that it runs indefinitely. When a flower of a color meets the other flower of its color, they both say "I found you!!" for 2 seconds; then both flowers reinitialize their position and direction to new random values.

Program file name: flowers5

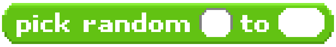
Summary

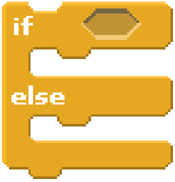
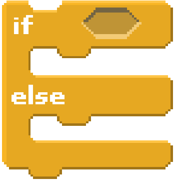
Concepts

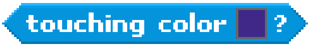
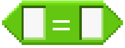
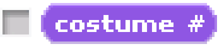
Random numbers: ***Random numbers*** introduce an element of chance into a program. A random number is like a variable, except that every time you use the variable, its value is likely to be different. The values are chosen from a range of numbers.

Conditional run: The control structure ***conditional run*** is used to specify that running a set of instructions depends on whether a condition is true or false. There are two versions of this structure: the first is ***conditional run with an alternative***. If the condition is true, one sequence of instructions is run, while if the condition is false, another sequence of instructions is run. The second form is ***conditional run without an alternative***. This form controls only one sequence of instruction, so if the condition is false, nothing happens. Like all control structures, condition run instructions can be nested.

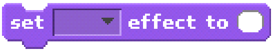
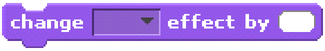
Scratch instructions

Random numbers: The value of the operator  is a random number in the range defined by the values in the two windows. This block has rounded ends and can be used whenever a numerical value is needed.

Conditional run: There are two instructions for conditional run, one with an alternative  and the other without an alternative . The "mouths" contain the sequence or sequences of instruction to run and the condition is place in the window after the if.

Conditions: The block  from the light blue Sensing palette is a condition that is true if the sprite is ***touching*** the specified color in another sprite or in the background. The condition ***equality***  checks if two values are the same. The windows on both sides of the "=" symbol can contain either numbers or reporters that represent the values of variables, either built into Scratch like  or defined by the user.

Hiding and showing: A sprite can ***hide*** itself or cause itself to ***appear*** using the instructions and .

Graphical effects: The ***graphical effects*** used when displaying the images of the sprites on the stage can be set by the absolute instruction  and changed by the relative instruction . The menu in the first window lists the effects that can be modified, while the second window is used to specify the value of the effect or the change in its value.

Chapter 8

Numbers

We have used numbers since the very first example in this book. Many instructions such as `move` require numbers to turn them from general instructions to specific instructions. Numbers have been used to indicate the number of steps to move, the direction to turn, the number of times a loop is to be run, the number of seconds to wait, the identifying number of a costume, and so on. We also used variables to remember values which are numbers such as the size of a dragon. In this chapter we will deepen our knowledge of numbers and see how they can be included in instructions in new ways.

Example 1
Oranges for the prince

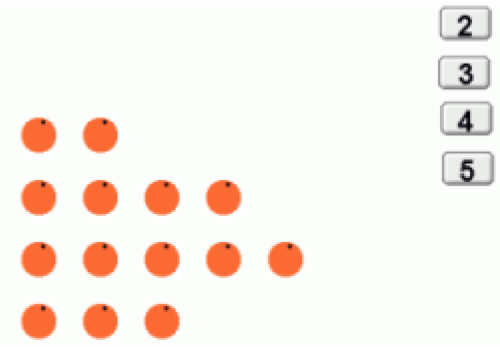
We will construct an interactive animation that could be used to teach young children how to add numbers. The animation will show the prince who wants to receive 12 oranges and who asks the user give them to him. The user will supply the oranges by clicking on buttons, with a separate button for each amount that can be given at one time: 2, 3, 4 or 5 oranges. We will construct the animation in stages: first, we will implement the buttons and later we will add the prince.

Task 1

> Construct an animation for displaying oranges. There will we four buttons, labeled `2`, `3`, `4` and `5`. Clicking on a button labeled with a number will cause that number of oranges to appear on the stage. The oranges will appear on the stage in a pile of rows: Each click on the button will cause the appropriate number of oranges to appear in a new row on the pile.

Program file name: get-oranges

The following image shows the stage after clicking on 3, 5, 4, 2 in that order:



? How many sprites are needed?

Clearly, we need sprites for each of the four buttons.

? What about the oranges?

The oranges are also images that appear on the stage, but we do not know in advance how many oranges there will be (it depends on which one of the buttons the user clicks), so we can't create a separate sprite for each orange. Instead, the oranges will be just images on the stage of a *single* sprite; the images themselves have no scripts.

The orange that duplicates itself—one sprite, many images

In Scratch, a Sprite can create ***images*** of itself on the stage, just like a rubber stamp creates an image of itself on a piece of paper. The image remains even after the stamp is removed from the paper and moved to another position.

This image is not a sprite; it cannot move or respond to messages the way a sprite can. The instruction stamp, which appears as the last block in the dark green palette Pen, creates images.

New construct in Scratch: stamp

The instruction stamp creates an image of the sprite on the stage at the current position of the sprite. The image remains even if the sprite itself moves.

The *stamp* instruction will cause the image of the orange sprite to appear at the current position of the sprite, but the action of stamping will be caused by clicking on one of the buttons. Therefore, a button sprite must respond to a click by notifying the orange sprite that it must stamp an image. Communications between sprites can be done by sending and receiving messages. A partial description of the instructions for the button sprite is as follows (where we give the instructions for the button labeled 3):

```
0. when the button 3 sprite is clicked
    1. inform the orange sprite to add 3 new images
```

The messages passed to the orange sprite

The orange sprite has to respond to the messages that it receives. There are two possible ways of implementing this. One is to have separate messages for each number of oranges: for example, there would be one message for making three images of the orange sprite and the sprite would respond to this message by creating three images; in addition, there would be one message for creating four images and one for five images. The orange sprite needs to have a separate script for each such message because receiving a message always *starts* the run of a script. In our case, there would be four scripts, one for each of 2, 3, 4 and 5 new oranges.

The other way to implement the communications between the button sprites and the orange sprite is to have just one message for making a *single* image of an orange and to require that each button sprite be responsible for sending the correct number of messages. For example, step 1 above would be implemented by sending three messages:

```
0. when the button 3 sprite is clicked
      1. inform the orange sprite to add one new image
      2. inform the orange sprite to add one new image
      3. inform the orange sprite to add one new image
```

The response of the orange sprite is now very simple:

```
0. when you receive the message to add a new image
```

1. stamp a new image

Passing information using variables

These outlines of the behavior of the orange sprite and the button sprites are not complete. In particular, we have not yet shown how to position each new image of an orange: the first image after clicking a button sprite must start a new row, while subsequent images (up to the number that appears on the button) must be in the same row, but not at the same position, so that we can see all the oranges in a row.

Since the orange sprite receives an identical message each time that a new image must be stamped, it does not have the information needed to position the new image at the start of a new row or at a certain position within an existing row. The button sprites will be responsible for computing the position of each orange image, because the button sprites know when an orange is to begin a new row and how many oranges have already been stamped. Information on the position of an orange must be transmitted to the orange sprite and for this we will use variables as mailboxes as we did in the previous chapter.

We will use two variables, `x` to remember the x-position of the orange on the stage and `y` to remember the y-position of the orange on the stage. The orange sprite need only read the values of these variables. The outline of its behavior can be extended as follows:

```
0. when you receive the message to add a new image
    1. go to position (x,y)
    2. stamp a new image
```

The computation of the values of x and y is the responsibility of the button sprites. The behavior of the button labeled 3 is:

```
0. when button 3 sprite is clicked
    1. set the values of x and y to one position before the beginning of a r
    2. run 3 times
        2.1 add a value to x so that it is at the next position in a row
        2.2. inform the orange sprite to add one new image
```

The description of the behavior of the button sprite started with just two steps:

```
0. when your image is clicked upon
    1. inform the orange sprite to add 3 new images
```

The description is now more complex: a repeated run of two steps. Furthermore, step 1 really describes *two* steps: one to set the value of x and one to set the value of y.

Exercise 1

a. Explain why in step 1 the value of x is set to that of a position ***before*** the beginning of a new row of oranges. (Hint: think about the order of steps 2.1 and 2.2.)

b. In step 1, would it be possible to set the value of x to the beginning of a new row? If so, what other changes would you have to make?

Arranging the oranges in rows

We will arrange the rows of oranges as shown in the image of the stage at the beginning of this chapter. The value of x for the first orange in a row will be -150 and there will 50 steps between oranges in a row. There will also be 50 steps between rows. Step 2.1 changes the value of x so that it points at the *next* position in the row and step 1 changes the value of y so that it points to the beginning of the *next* row. Therefore, these steps will be implemented using *relative* instructions: the value of x is changed by 50 in step 2.1 and the value of y is changed by 50 in step 1. Step 1 requires that the value of x be set to point to the start of a *new* row, so it will be implemented by an *absolute* instruction.

Exercise 2

What value should be used in the absolute instruction to set x to a position *before* the start of a row?

Synchronization among the scripts

Since the button sprites and the orange sprite communicate using both messages and variables, we have to ensure that they are correctly synchronized. Step 2 of the description of the button's behavior results in the running of multiple (2, 3, 4 or 5) instructions to send messages, one after the other. However, it is possible that ***before*** the orange sprite has a chance to receive one of the messages and to stamp a new image of an orange, the button sprite will continue with its repeated run instruction and change the value of x in step 2.1. This will cause some of the images to be stamped in an incorrect position.

The solution is to have the button sprites wait until the orange sprite stamps the images before continuing with the repeated run. This can be done in Scratch using the instruction `broadcast ▾ and wait` that causes the script to stop running until the script that is run when the message is received has finished its run. The block appears just below the block for *`broadcast`* in the Control palette.

New construct in Scratch: broadcast and wait

The instruction broadcast ▾ and wait is similar to the broadcast ▾ instruction except that the script that contains it stops running after it sends the message. The scripts that run as a result of receiving the message run to completion and only then is the script containing broadcast ▾ and wait allowed to continue.

Exercise 3

> Open the project costumes which contains sprites for the oranges and the buttons. Make the x and y variables and create the scripts for the orange and for a button sprite as described above. Duplicate the button's script in the other buttons and make the appropriate modifications.

What about initialization?

The project is not yet complete because we have not taken care of the initialization of the sprites. For the buttons, no initialization is needed because they remain in the position where their images were placed when the project was created. For the orange, several initializations must be done:

- We must clear the stage of the images of the orange sprite from the previous run of the project.
- We must place the orange sprite *somewhere*; it doesn't really matter where, since it is used only to stamp images where we tell it to, so we choose to place it in the upper right corner at $(200, 150)$.
- The variable y must be initialized to 50 units below the first row, since the buttons *change* the value of y by 50 before starting a new row. That is, the instructions are relative instructions, so there must be an absolute instruction to initialize the y position.

The variable x need not be initialized since an absolute instruction is used to set its value to -150 at the beginning of each new row. The initialization is therefore as follows:

```
0. when the green flag is clicked
    1. clear the stage
    2. set the value of y below the first row $-200$
    3. go to position $(200, 150)$
```

Step 3 is implemented using the clear instruction from the dark green Pen palette.

New construct in Scratch: clearing an image

The instruction clear erases all images that have been stamped on the stage.

Exercise 4

Create the initialization script for the orange. Check that the animation runs correctly. Add comments and save the project.

Counting the total number of oranges

The game requires that we know the total number of oranges that have been stamped on the stage so that the prince (whom we have not yet created) can report if he has enough oranges already or if the user should click on a button to create more of them.

Task 2

Count and display the total number of oranges that have been created.

Program file name: store-count

The scripts that we have constructed so far look at oranges from a *local* perspective: the orange sprite stamps only one orange at a time, while the button sprites just know how many oranges will be added as a result of clicking on the button. We need to consider the number of oranges from a *global* perspective: how many oranges have been created from the start of the game? Let us define a variable that will remember the total number of oranges that have been created. Whenever a button is clicked, its value will be changed by the number corresponding to that button.

Who is responsible for counting the oranges

? Which sprite or sprites will be responsible for updating the value of this variable?

One possibility is to have the orange sprite add one to the variable each time that it stamps a new orange on the stage. Alternatively, each button sprite could add one to the variable each time that it sends a message to the orange sprite. We prefer a third, higher-level, solution: whenever the script for a button is run (because the sprite was clicked), the value of the variable will be changed by the number corresponding to the button. For example, if we click on the 2 button, the value will be changed by 2, and if we click on the 4 button, the value will be changed by 4.

Since these instructions are relative instructions, it is important that the value of the variable be given an initial value. The initial value should be 0 because initially no oranges have been created. It will be convenient to include this initialization as part of the initialization instructions run by the orange sprite when the green flag is clicked.

Exercise 5

> Modify the project to include a variable `oranges` for the total number of oranges. After creating the variable, click the box by its reporter so that its value is displayed on the stage. Add the initialization instruction to the orange sprite and instructions to the button sprites to change the value of the variable. Check that the project works, update the comments and save.

Accumulators

The variable `oranges` is being used as an ***accumulator***. Accumulators remember the total amount of something. They are very familiar even if the word is not: the odometer in a car remembers the total number of miles or kilometers that the car has been driven. There is an accumulator in your house that remembers the total amount of kilowatt-hours of electricity that you have used. Scoreboards at a sporting event are accumulators that remember the total number of points that each team has scored. In a basketball game, initially each team has zero points and each basket made causes the number of points to increase by 1, 2 or 3, according to the type of the throw.

New concept: accumulator
A variable can be used as an ***accumulator*** in order to remember the sum of a set of values. The pattern of the use of the variable is as follows: The variable is initialized to 0 and whenever an event occurs, a value is added to the current value of the variable.

While the initialization of an accumulator uses an absolute instruction to set its value to zero, the variable is updated using relative instructions which add new values to the current value. In our game, the value added each time to oranges will 2, 3, 4 or 5, depending on which button was pressed.

Counters

A *counter* is an accumulator whose value changes by 1 each time. An example would be a clicker that is used to count the number of people on a bus. It is initialized to zero and it is clicked once for each person on the bus, adding one to the count. We could use a counter in this project to remember the *number of times* that a button is pressed to create oranges rather than the total number of oranges. We could also use a counter to remember the number of times that an orange is stamped instead of using an accumulator that adds the number of oranges created each time a button is pressed.

New concept: counter
A variable can be used as a ***counter*** in order to remember the number of times something happens. The pattern of the use of the variable is as follows: The variable is initialized to 0 and whenever a new event must be counted, 1 is added to the current value of the variable.

The prince arrives: how many oranges are there?

We now add the prince to the game.

Task 3

> Modify the animation by adding a sprite for the prince. He asks for 12 oranges at the start of the game. After each click of a button, the prince will announce the result: he has the right

number of oranges, he has too few oranges or
he has too many oranges.

Program file name: prince-says-when-enough-oranges

The task can be broken down into two parts: asking for oranges and checking the number of oranges. The first part of the task is very simple:

```
0. when the green flag is clicked
    1. say "Please give me 12 oranges" for 2 seconds
```

The second task requires that whenever a button is clicked the prince compares the total number of oranges stored in a variable with the value 12. Again, there is a need for communications between sprites: the button sprites must notify the prince sprite that a button has been clicked so that he can compare of the new total number of oranges with 12. Each button will send a message `New batch` when it has completed changing the number of oranges. The prince will receive this message and perform the comparison:

```
0. when you receive the message New batch
    1. if the total number of oranges is less than 12
        1.1 say "Please give me more oranges" for 2 seconds
    2. otherwise
        2.1 if the number of oranges is greater than 12
        2.1.1 say “I’ve got too many oranges” for 2 seconds
    2.2. otherwise
        2.2.1 say “Thank you for the 12 oranges!!”
```

Comparing numbers

Let us now translate this description into Scratch scripts. You already know enough Scratch constructs to do so, except for the conditions of the conditional run instructions in steps 1 and 2.1. Here the condition is similar to comparing two numbers for equality that we used in the previous chapter, except that we have to check whether one number (the number of oranges) is less than or greater than another (the value 12). Blocks for these comparisons can be found in the light green Operators palette. Above and below the block for equality, you can find operators for ***less than*** ◯<◯ and for ***greater than*** ◯>◯. These blocks have angled ends and can be used as conditions in conditional run instructions. The values to be compared (the variable oranges and the number 12) are placed within the windows of the conditions.

New construct in Scratch: less than and greater than

The condition □ < □ is true if the value in the left window is ***less than*** the value in the right window. The condition □ > □ is true if the value in the left window is ***greater than*** the value in the right window.

Exercise 6

Can we use just one of the conditions *less than* or *greater than* to implement the comparisons in step 1 and 2.1? If so, show how this can be done.

Exercise 7

Complete the project for this game: Create the sprite for the prince (use the costume prince1 from the People folder) and construct its scripts as described above. Add `broadcast` instructions to the sprites for the buttons and use conditional run instructions in the script for the prince. Write comments for these changes, check that the project works and save it.

Example 2
Changing the rules of the game—a surprising button

Our game is rather predictable ... let us now change the rules of the game to add a bit of excitement.

Task 4

> Modify the animation as follows. The prince will ask for 12 oranges, but the number of oranges that will be supplied (2, 3, 4, 5) will be not be chosen by the *user* clicking on different buttons. Instead, there will be only one button labeled `More`. When the user clicks on this button, the computer will *randomly* choose a number between 2 and 5, and this is the number of oranges that will be given to the prince.

Program file name: random-number-of-oranges

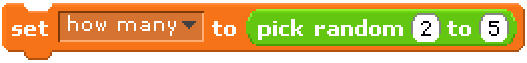
The progress of the game is no longer determined by the user, but by the random selection of numbers by the computer. This is not unfamiliar, because many games use dice to determine the number of steps that a player moves in each turn. Instead of cubes with numbers from 1 through 6, think of our dice as pyramids with the faces labeled with the numbers 2, 3, 4, 5. (Those of you who play the game Dungeons and Dragons will be familiar with dice that are pyramids with four sides.)

The behavior of the single button

In this version of the game, there will be orange and prince sprites as before, but instead of four buttons labeled with numbers, there will be one button labeled `More`. Its script will be similar to that of the previous buttons except for the random choice of the number of oranges:

```
0. when the More sprite is clicked
    1. set the values of x and y to one position before the beginning of a n
    2. choose a random number between 2 and 5
    3. run the following steps this random number
        3.1 add a value to x so that it is at the next position in a row
        3.2. inform the orange sprite to add a new image
    4. add the random number to the total number of oranges
    5. notify (the prince) that he has a new batch of oranges
```

Step 2 is implemented by the operator `pick random ( ) to ( )`, which appears in the light green Operators palette. (The operator was introduced in Example 5 of Chapter 7.) The random value that is read will be remembered in a new variable `how many`:



```
set how many ▾ to pick random 2 to 5
```

Exercise 8

> Create a new project from the previous one by adding a button labeled `More`. Make a new variable `how many` for the random number of times and check the small box so that its value is displayed on the stage. Copy the script from one of the old buttons and make the changes needed so that it implements the above description. Now you can delete the four numbered buttons. Add comments to the new button, check that the project works and save it.

? Do we need to make any changes to the scripts of the prince and orange sprites?

Most of the responsibility for the game rests with the button sprite, while the other two sprites have limited, local, responsibility. The button sprite needs to compute the position of the oranges and update the accumulator that remembers the total number of oranges. The orange sprite simply initializes the game and responds to messages by stamping an image, while the prince simply receives messages and says something. Therefore, the only change to the orange sprite is to add an initialization of the new variable `how many`.

The prince becomes flexible: more than 12 oranges is OK

Let us make a small change to the game.

Task 5

> After playing the game for a while, the prince sees that it is unlikely that random numbers of oranges will add up to exactly 12, so he has agreed to be more flexible and to accept any number of oranges between 12 and 14.
>
> Program file name: prince-says-how-many-oranges

Exercise 9

> Which sprites will be affected by this change?

The prince sprite is the only one that will be affected; his new behavior is as follows:

```
0. when the green flag is clicked
    1. say "Please give me between 12 and 14 oranges" for 2 seconds

0. when you receive the message New batch
    1. if the total number of oranges is less than 12
        1.1 say "Please give me more oranges" for 2 seconds
```

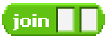
```
2. otherwise
    2.1 if the number of oranges is greater than 14
        2.1.1 say “I’ve got too many oranges” for 2 seconds
    2.2. otherwise
        2.2.1 say “Thank you for giving me the number oranges I rece
```

Saying thank you politely

The changes are very simple, except that we want the prince’s thank-you sentence in step 3.1 to be meaningful; that is, we want him to thank us for exactly the number of oranges he received. Previously, we just used the string “Thank you for giving me 12 oranges!!”, but now we want the sentence to change each time to “Thank you for giving me [`oranges`] oranges!!”, where [`oranges`] is the value of the variable `oranges` that counts the number of oranges he received, whether 12 or 13 or 14.

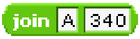
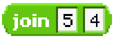
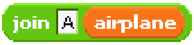
One possibility would be to use a conditional run instruction, checking if the number of oranges is 12, 13 or 14 and running an appropriate say instruction. However, this would make the script very complicated and would not be practical if the range were much larger, say, between 12 to 30 oranges. Instead, we want to *compute* the string “Thank you for giving me [`oranges`] oranges!!” using the value of the variable `oranges`.

Joining two strings into one string

To create the polite thank-you string, we use the operator  which is the fifth block from the bottom in the light green Operators palette. It creates a single string by placing the contents of the two small windows one after another. For example, if we join the string "Good" to the string "morning", we get "Goodmorning". Of course, we should have included a blank letter as the last letter of "Good" or the first letter of "morning" in order to get "Good morning".

? What kind of things can we join together using the *join* operator?

In the block, both windows have *straight* sides like the windows in the conditions "=", "<" and ">". This means that we can put different things in the windows: numbers, strings, and even other blocks with rounded or pointed sides. The values are joined together to make one string. For example:

-  results in the string "A340" (a type of airplane);
-  results in the string "54" that can also be used as the *number* 54
-  results in the string consisting of "A" followed by the value currently contained in the variable `airplane`.

New construct in Scratch: joining strings and numbers

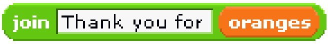
The operator join □□ joins the string (or number) in the first window to the string (or number) in the second window. The result is the same as if the two were written one after the other. If both windows contain numbers, the result can be used as a number.

Joining several strings into one string

We need to join three strings: the string "Thank you for " followed by the value of the variable `oranges`, followed by the string " oranges!!". However, the block join □□ has only two windows.

? How can we join three strings?

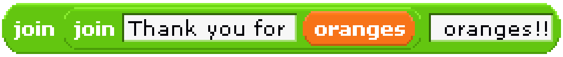
Whenever we want to add three numbers (perhaps using a calculator), we first add two of them and then add the third. That is, 8+3+2 is computed by adding 8+3 and then adding 2 to the result. This is often written (8+3)+2 to emphasize that first we add two numbers and then add the third number to their sum. The *join* operator works like addition: since the block has rounded ends, it can be used itself in one of the windows of *another join* operator. In fact, just as with addition, any number of things can be joined one after another.

We construct the string needed for the project in two steps. First, join "Thank you for " to the value of the variable `oranges`. Drag the block for and drop it in an empty place in the script area. Write "Thank you for " in the first window and drag the reporter for the variable oranges into the second window. Make sure to leave a space after the word "for" so that there will be a space between it and the number of oranges. The block that results should be . Click on the word "join" in the block and you should see the string "Thank you for [`oranges`]", where [`oranges`] is the value of that variable. This block can now be dragged and dropped it into the window of a *say* block. This window has straight edges so the *join* block with rounded edges can be dropped there. Run the project and check that the prince always politely thanks you for the correct number of oranges.

In the following exercise you will complete the construction of the string "Thank you for giving me [`oranges`] oranges!!".

Exercise 10

(a) Drag and drop a new block onto the stage. Into the first window, drag-and-drop the *join* block we constructed previously and in the second window write the word " oranges!!" (again, note the space). The result is:



```
join join Thank you for oranges  oranges!!
```

Finally, drag the nested *join* block into the *say* block. Run the project and check that the prince says what we want him to. Add comments and save the project.

Program file name: prince-says-with-join

(b) Is there another way to create the same sentence? Hint: Is there another way of computing 8+3+2 besides (8+3)+2?

The prince is getting tired and doesn't want to talk so much

The prince has become more flexible but he feels that his answers are too detailed.

Task 6

> Modify the animation so that when oranges are added by clicking the button, the prince will respond in one of two ways: either he will thank the user for giving him the correct number of oranges (between 12 and 14) or he will say "I don't have the correct number of oranges".

Program file name: prince-says-with-compound-condition

The second response is ambiguous because it can result from having too many or too few oranges; the user will have to decide which of the two possibilities is intended.

Compound conditions

This change in the behavior of the prince allows us to simplify his script. Previously, we had to take account of three possibilities: less than 12 oranges, more than 14, or between 12 and 14. The script used a nested alternative run:

```
if the number of oranges is less than 12
...
otherwise
    if the number of oranges is greater than 14
        ...
    otherwise
        ...
```

Now we only have two possibilities: between 12 and 14 oranges and *not* between 12 and 14 oranges; the alternative run needed not be nested:

```
if the number of oranges is between 12 and 14
    ...
otherwise
    ...
```

The description of the behavior of the prince is now:

```
0. when you receive the message "New batch"
    1. if the total number of oranges is between 12 and 14
        1.1 say "Thank you for giving me the number oranges I received!!
    2. otherwise
        2.1 say "I don't have the correct number of oranges"
```

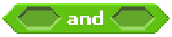
This description is very concise. In order to implement it, we have to learn how to construct the condition “is between 12 and 14” whose meaning can be expressed as “greater than 11 and less than 15”. The compound condition is built from two simpler conditions “greater than 11” **and** “less than 15” and both of then must be true for the compound condition to be true. For example, if the number of oranges is 13, then both the condition “greater than 11” and the condition “less than 15” are true, so the compound condition is true. However, if the number of oranges is 9, the second condition “less than 15” is true, but not the first condition “greater than 11”. Therefore, the compound condition is not true.

Exercise 11

> Is the compound condition true when the number of oranges is 17? Explain your reasoning.

Exercise 12

> Does there exist a number for which both parts of the compound condition are false? If so, give an example of such a number; if not, explain why this cannot happen.

A compound conditions is built using the block 

which has two small windows into which

other conditions can be inserted. It is found in the middle of the light green Operators palette. Both windows have angled ends so that they can only have other conditions inserted. The block itself has angled ends and is a condition.

Exercise 13

> Modify the second script of the prince to implement the new behavior. Add comments and save the project.

New concept: compound condition
A compound condition is formed from simple or (other compound) conditions such as checking equality or touching a sprite.
A compound condition can be created using ***and,*** in which case the condition is true only if both the conditions are true. Otherwise (if only one simple condition is true or neither of them is true), the compound condition is false.
A compound condition can be created using ***or***, in which case the condition is true only if either or both of the conditions are true. Otherwise (if neither of them are true), the compound condition is false.

New construct in Scratch: compound conditions with *and* and *or*

The instruction ⟨⟨ ⟩ and ⟨ ⟩⟩ creates a compound condition from two other conditions that are placed within the windows. The compound condition is true if both conditions are true.
The instruction ⟨⟨ ⟩ or ⟨ ⟩⟩ creates a compound condition from two other conditions that are placed within the windows. The resulting condition is true if either or both conditions are true. (We did not use of this block in our examples.)

Example 3
Arranging the oranges in equal rows

The prince has decided that his pile of oranges should be neat and tidy.

Task 7

Modify the project so all the rows have exactly 4 oranges in them, except for the last row which may have fewer oranges.

Guidance: Start with the project from Example 2 where the prince asks for exactly 12 oranges (file random-number-of-oranges).

Program file name: oranges-in-piles

Let us work through an example: The prince initially receives 3 oranges and they are placed in the first (incomplete) row. Now the prince receives 2 oranges; the first one is used to fill up the first row so that it has exactly 4 oranges, while the second one is used to start the second row. On the third click, the prince receives 5 oranges; of them, three fill up the second row while two appear in the third row.

Computing the place of a new orange

The new requirement will cause changes in calculating the position of the next orange to be stamped on the stage. In our implementation, the responsibility for this calculation belongs to the button sprite. The button started a new row for each batch of oranges by setting the y-position to the new row and the x-position to the start of the row. Consider the description of the computation of the button from Example 2:

```
0. when the More sprite is clicked
    1. set the values of x and y to one position before the beginning of a n
    2. choose a random number between 2 and 5 and store in the variable
    3. run the following steps how many times
        3.1 add a value to x so that it is at the next position in a row
```

```
          3.2. inform the orange sprite to add a new image
     4. add the value of how many to the accumulator variable oranges
     5. notify (the prince) that he has a new batch of oranges
```

Step 1 is no longer needed because we no longer want to start a new row each time the button is clicked. Instead, we must modify Step 3.1 so that it computes (both) the x- and y-positions of the next orange. The y-position is changed only when a row has been filled up and this must be checked for *each* individual orange that is processed, not once at the beginning of a new batch. The computation of the x-position will also change: it will return to the start of a row only when the y-position is changed for that row. The modified description is as follows:

```
0. when the More sprite is clicked
     1. choose a random number between 2 and 5 and store in the variable
     2. run the following steps how many times
          2.1 if the current row is full
               2.1.1 set y to the start of the next row
               2.1.2 set x to one position before the start of a row
          2.2 add a value to x so that it is at the next position in a row
          2.3 inform the orange sprite to add a new image
     3. add the value of how many to the accumulator variable oranges
     4. notify (the prince) that he has a new batch of oranges
```

Is the row full?

? How can the button sprite know that the row is full (step 2.1)?

We need an additional variable to remember the position in the row of the *last* orange that was stamped on the stage. Let us call it `last`. For example, if there are already 3 oranges in the last row, the value of `last` will be 3, and if the row is full, the value of `last` will be 4. The condition:

```
2.1 if the current row is full
```

can be implemented as:

```
2.1 if last is equal to 4
```

Whenever the button sprite informs the orange sprite to stamp a new orange, it must change the value of the variable `last`. If the orange starts a new row, the value of `last` *after* the orange is stamped in the next row must be set to 1; otherwise, its value must increase by 1.

Exercise 14

> Change the description of the behavior of the button sprite to include processing the variable `last`. The value of `last` must be changed whenever the value of `x` is changed, that is, for each new orange to be stamped.

> The absolute instruction used to set x to a new row (step 2.1.2) must be followed by an absolute instruction to set last to its value at the beginning of a row. The relative instruction used to move x to a new position in a row (step 2.2) must be followed by the relative instruction that adds 1 to last.

Checking the initializations

This change in the game will not affect the behavior of the prince sprite, but we have to carefully check the behavior of the orange sprite. In particular, the orange sprite is responsible for initializing the project when the green flag is clicked.

? What should be the initial value of the variable last?

The initial value of last should be 0 since we have not stamped any oranges yet.

? What about the values of x and y?

The value of y is changed (step 2.1.1) only if the row is full which is checked in step 2.1. Since last is initialized to 0, the condition in 2.1 will be false and y will not be changed. It follows that the initial value of y should be the y-position of the first row of oranges -150.

Previously, we did not need to initialize x because its value of always was set just before we stamped an orange. Now, however, its value is only changed is the row is full (step

2.1.2). Since initially the row is empty, this step will not be run initially so we need to give an initial value to `x`: the x-position of the first column -200.

Exercise 15

> Modify the script for the button sprite as described above. Make sure to first define the variable `last` and initialize it. Check that the project works correctly, add comments and save the project.

Cyclical addition using remainder (advanced)

In the following tasks, we will learn a new concept—cyclic addition—but the animation itself will not be changed.

Task 8

> Modify the scripts so that the computation of the positions of the oranges uses cyclic addition implemented with remainder.

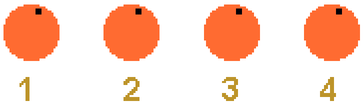
Program file name: oranges-in-pile-with-mod

Cyclic addition

? What are the possible values of the variable `last`?

It is initialized to 4 and becomes 0 immediately. On subsequent clicks, 1 is added until its value become 4, after which it is set to 0 again. Therefore, its value can only be 0, 1, 2, 3 or 4. These numbers represent the positions within a row and since we specified that the number of oranges in a row is at most 4, the variable `last` cannot have a larger value.

In fact, the variable need not ever have the value 4. The following image will help you follow the discussion in the next paragraphs:



The image shows a row of four oranges and possible values of the variable `last`. The meaning of `last` = 1 is that the last orange was placed in the first position in the row and the next orange will be placed in the second position. Similarly, `last` = 2 means that last orange is in position 2 and the next will be in position 3, and `last` = 3 means that last orange is in position 3 and the next will be in position 4.

? What about the values 0 and 4?

`last` will get the value 4 when the orange is stamped in the last position in the row and the next orange will be in position 1 of the next row. The variable will receive the value 0 in two cases: initially (when no orange is displayed) and *immediately* after it receives the value 4:

```
if last is equal to 4
    set last to 0
```

Therefore, the values 0 and 4 really mean the same thing and we can use 0 alone without the value 4.

The values of `last` will change as follows: Initially it will be 0 so that the next (first) orange will be in the first position. When the first orange is stamped, its value will become 1 to indicate that the next orange will be in position 2. The value will increase to 2 and 3 after the second and third oranges are stamped, meaning that the next oranges will be in positions 3 and 4. After the last orange is stamped in position 4, the value of `last` can be directly set to 0 (meaning that the next orange will be in the first position) without first being set to 4.

The values of `last` form a ***cycle*** 0, 1, 2, 3, 0, 1, 2, 3, 0, and so on. ***Cyclical addition*** is like regular addition except that when the length of the cycle is reached the value returns to 0. For a cycle of length 4, cyclical addition starts like regular addition: 0+1=1, 1+1=2, 2+1=3; but for 3+1=4 the value returns to 0 so we write 3+1=0.

Modifying the scripts to use cyclic addition:

Using cyclical addition, we can simplify the description of the behavior of the button sprite. After the change to use the variable `last`, Steps 2.1–2.3 are:

```
2.1 if last is equal to 4
    2.1.1 set y to the start of the next row
    2.1.2 set x to one position before the start of a row
    2.1.3 set last to 0
2.2 add a value to x so that it is at the next position in a row
2.3 add 1 to last
```

Now, since 4 is no longer a possible value, we can simplify them to:

```
2.1 if last is equal to 0
    2.1.1 set y to the start of the next row
    2.1.2 set x to one position before the start of a row
2.2 add a value to x so that it is at the next position in a row
2.3 cyclically add 1 to last
```

Initially, `last` is 0 so the first orange will be stamped at the beginning of the row and `last` will be set to 1. The next two oranges will be stamped in the same row while `last` receives the values 2 and 3. The fourth orange will be stamped in the row and then `last` will receive the value 3+1, which equals 0 since the addition is done cyclically. Now, steps 2.1, 2.1.1 and 2.1.2 ensure that the next orange will be correctly stamped at the beginning of the row.

How is cyclical addition implemented?

The values 0, 1, 2, 3 are exactly the values that are obtained by dividing any number by 4 and looking at the *remainder*. For example, when 16 is divided by 4 the remainder is 0, 17/4 leaves a remainder of 1, 18/4 leaves a remainder of 2, 19/4 leaves a remainder of 3 and 20/4 brings us back to a remainder of 0. Consider now the values of the variable `last`. It is initialized to 0 and 1 is added each time. If we take the remainder after each addition, we obtain exactly the values 0, 1, 2, 3:

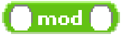
> remainder((0+1) / 4) = remainder(1/4) = 1,
> remainder((1+1) / 4) = remainder(2/4) = 2,
> remainder((2+1) / 4) = remainder(3/4) = 3,
> remainder((3+1) / 4) = remainder(4 / 4) = 0.

We can implement step 2.3:

> *2.3* ***cyclically*** *add 1 to* `last`

by:

> *2.3 set* `last` *to the remainder of* (`last`+*1) divided by 4*

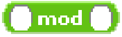
The computation of the remainder is implemented by the operator  which appears as the third block from the bottom in the light green Operators palette. It has two windows: the first for the number being divided (in our case, `last`+1), and the second for the divisor (in our case, 4).

New concept: cyclic addition

When a variable takes a finite set of values 0, 1, 2, ..., N-1 for some N, ***cyclic addition*** can be used. Addition is performed normally for all values except the last one; for N-1, increasing its value by one gives (N-1)+1 = N, which is then changed in 0.

Cyclic addition is implemented by the operator *mod*: $(n+1)$ mod N. For all values of n except N-1, the result of the addition is a number less than N, so its remainder equals n. For N-1, (N-1)+1=N, and the remainder of dividing N by N is 0.

New construct in Scratch: the mod operator

The operator  computes the ***remainder*** of the value in the first window divided by the value in the second window.

Exercise 16

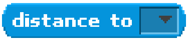
Use the *mod* operator to implement the version of the game with cyclical addition in the script for the button. Check that the project works correctly, add comments and save the project.

Additional Exercises

Exercise 17

This exercise is an extension of Exercise 6 from Chapter 5.

a. Start with the animation of paragraph (b) of that exercise and modify the behavior of the three animals as follows: after an animal stops moving, it says which animal it is closest to (“I am closer to the cow”).

Guidance: Use the operator  distance to ▾ from the Sensing palette to obtain the distance of one sprite to another. This operator is explained in detail in Chapter 11. Display the monitors for the x-positions of the three animals so that you can check the program runs correctly.

Program file name: play-run1

b. The problem with the animation in (a) is that it doesn’t wait until all the animals have stopped moving before reporting which animals are closest to each other. Modify the animation so that the the displays of all the animals occur together when they all stop moving.

Guidance: Define three new variables `elephant-stopped`, `horse-stopped` and `cow-stopped`, which will be initialized to 0 and set to 1 when the motion of the animal has stopped. After an animal has stopped, it waits until both other animals have stopped and only then displays which animal is closer.

Program file name: play-run2

c. Modify the scripts for (b) to use compound conditions (if you didn't do so).

Program file name: play-run3

d. (Advanced) Modify the animation so that one animal will say which pair of animals is closest, for example: "The horse is closest to the cow".

Guidance: One of the animals—let us choose the horse—will be responsible for comparing the three distances (horse ↔ cow, horse ↔ elephant, cow ↔ elephant) to see which is the smallest. If we know those distances, we can use nested conditional run instructions to compare them and to display which is the smallest.

The problem is that the horse cannot know the distance cow ↔ elephant because the operator `distance to...` only gives the distance from the sprite that runs it to another sprite. The solution is to define a new variable `cow-to-elephant` and use it as a mailbox to communicate this distance to the horse sprite. Even though it is not necessary, you might also want to define variables `horse-to-cow` and `horse-to-elephant` to simplify the comparisons.

Program file name: play-run4

Exercise 18

This exercise is an extension of Exercise 3(b) from Chapter 6.

Define a *trip* as the movement of one dancer from its side of the stage to the center or from the center to the side.

a. In that exercise the user controls the number of steps taken by each of the dancers. Modify the program so that the user only controls the number of steps that the dancer Cassy takes in a trip. The number of steps that Jay takes in a trip will be three times whatever you chose for Cassy.

Program file name: dance-steps1

b. Since the dancers dance at the same rate (waiting 0.2 seconds between each movement), if Jay takes three times as many steps per trip as does Cassy, he should take only one-third the number of trips during any period of time. Is this true?

Guidance: You can simply watch the animation and count the trips, but it is easier to add two variables `Cassy-trips` and `Jay-trips` which count the number of trips that each dancer makes. Add monitors to the stage for these two variables. Click the green flag, wait 30 seconds and then click the red flag to stop the animation. The value displayed in the monitor for Jay should be one-third that of the value in the monitor for Cassy.

Program file name: dance-steps2

c. (Advanced) Modify the animation so that the program checks that Cassy takes three times as many trips as does Jay. If not, Jay says "Help, Cassy! I'm out of step!".

Guidance: Write a script for the Jay sprite that is activated whenever Jay completes a pair of trips. The script will check if the number of trips by Cassy equals three times the number of trips by Jay; if not, Jays says that he is out of step. How can you check if the solution works?

Program file name: dance-steps3

d. (Advanced) Jay can't be trusted, so Cassy will say: "Jay! Get in step!" whenever she has not taken three times as many steps as Jay.

Guidance: Your might think that Cassy should compare her number of trips with that of Jay after she finishes a pair of trips. But when that happens Jay is still in the middle of his first trip. Ensure that Cassy does the comparison only *after* Jay has completed each pair of trips, at which time Cassy has completed a multiple of six trips.

Program file name: dance-steps4

Exercise 19

(Advanced) This exercise is an extension of Exercise 4 from Chapter 6. In that exercise, we created animations of rockets and investigated how the animation changed depending on the speed and acceleration of the rocket. Here, we ask you to look at some numerical measures and to explain them.

a. In each of the projects of the exercise count the number of steps taken by the rocket. Why is the number of steps different in each of the projects?

Guidance: Use a variable to count the number of steps and display the monitor for the variable.

Program file name: rocket1, rocket2, rocket3, rocket4

b. Count the number of times that the conditional run instruction is run. Why are these numbers different in each of the projects?

Program file name: rocket1a, rocket2a, rocket3a, rocket4a

Exercise 20

This exercise is an extension of Exercise 15 from Chapter 7.

Expand the projects grass3 through grass6 of that exercise so that they count and display the number of times that the moving grasshopper meets the second grasshopper and the third grasshopper.

Program file name: grass1, grass2, grass3, grass4

Exercise 21

This exercise is an extension of Exercise 15(a) from Chapter 7.

a. The grasshopper moves randomly right and left along the x-axis. In some runs it will move farther away from the center than in others. Modify the project so that it displays the maximum value of the x-position.

Guidance: Create a variable `MaxX` to store the maximum value of the x-position. Initialize it to 0 and set it with the value of `x-position` whenever it is larger than the current value of `MaxX`. Display the value of this variable.

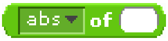
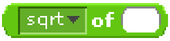
Program file name: grass5

b. Display the maximum ***distance*** of the grasshopper from the center of the stage, whether it is to the left or the right of the center.

Guidance: No matter how far to the left the grasshopper moves, the program in (a) will never consider an x-position to the left of center as the maximum value. The reason is that x-positions to the left of center are negative numbers. Since the current maximum value is initialized to 0 and can only get larger, a negative number will be ignored. For example, if the farthest x-position to the right is 10 and then the grasshopper moves to x-position 30 to the left of the center, 10 is larger than -30 so the value current maximum will remain 10.

We need to compare the ***absolute values*** of the x-positions. The absolute value of a variable is defined by:

> The absolute value of a is: a if a is positive and $-a$ if a is negative.

Thus $abs(10) = 10$ and $abs(-30) = -(-30) = 30$. The absolute value can be computed using the operator  which can be obtained from the last block of the Operators palette . Click on the first window where `sqrt` appears and select `abs` from the menu.

Program file name: grass6

c. Modify the previous project so that after every move the grasshopper turns by a random number of degrees from -180 to 180. Compute and display the maximum *distances* along *both* the x- and y-axes.

Program file name: grass7

d. Modify the previous project so that instead of computing and displaying separately the maximum distances along the two axes, the maximum distance from the *center* of the stage is computed and displayed.

Guidance: Create a very small sprite and place it in the center of the stage. Then use distance to ▾ to compute the distance between the grasshopper and the new sprite.

Alternatively, compute the difference between the position of the center $(0,0)$ and the position of the grasshopper. You will need to use the operator sqrt ▾ of ( ).

Program file name: grass8, grass9

Exercise 22

This exercise is an extension of Exercise 16(d) from Chapter 7.

Modify the project so that it computes and displays how much time it takes for the three pairs of flowers to find each other. The program will be more interesting if the sprites are smaller so that it takes longer for the pairs of flowers to meet.

Guidance: Create two variables `Time`, which counts the amount of time that has passed since the animation starts, and `Found`, which counts the number of pairs that have touched each other. Terminate the animation when `Found` equals three.

Program file name: flowers1

Summary

Concepts

Numbers can be compared for ***equality*** and ***inequality***: ***greater than*** or ***less than***.

Compound conditions can be created from simple conditions (or other compound conditions). Given two conditions `Cond1` and `Cond2`, we can create a new condition that is true only if ***both*** `Cond1` ***and*** `Cond2` are true or a new condition that is true only if ***either*** `Cond1` ***or*** `Cond2` ***(or both)*** are true.

An ***accumulator*** is a variable that is used to remember the sum of a set of values. A ***counter*** is a variable that is used to count the number of times that something happens. In both cases, the initial value is 0 and an instruction changes the value of the variable. For an accumulator, the instruction adds a value, while for a counter, the instruction adds 1 when something happens.

When the values of a variable are limited to a range 0, 1, 2, ..., $N - 1$, for some N, ***cyclical addition*** can be used on the variable to keep the values within this range. Cyclical addition is the same as regular addition except that only the ***remainder*** upon division by N is remembered. In particular, if you add 1 to a value, the addition is as usual except when the value of the variable is $N - 1$; in that case, the value becomes 0 instead of N.

When there are multiple sprites and multiple scripts in sprites, synchronization problems can arise. The problems can often be solved by requiring a script sending a message to ***wait*** (stop running) until the receiving scripts finish running.

An image of a sprite can be ***stamped*** on the stage.

A string can be created by ***joining*** two or more other strings or numbers.

Scratch instructions

The operators for inequality are ***greater than*** ( ) > ( ) and ***less than*** ( ) < ( ).

Compound conditions use the operators ⟨ ⟩ and ⟨ ⟩ (both conditions must be true) and ⟨ ⟩ or ⟨ ⟩ (either or both conditions must be true).

Remainder is implemented by the ( ) mod ( ) operator. (***mod*** is short for modulo, a technical term for remainder.)

The instruction broadcast [▾] and wait is similar to the broadcast [▾] except that the run of its script ***waits*** until the receiving scripts have been completed.

The instruction stamp ***stamps*** an image of the sprite at its current position.

The instruction clear erases *all* images of the sprite from the stage.

The operator join □□ is used to create strings from other strings (and numbers) by placing the strings one after the other.

Chapter 9

Lists—Remembering Complex Information

The project for this chapter is an animation of a waiter taking orders in a restaurant, where the user chooses items from a menu. We will use a new construct in Scratch that is like a variable for storing information but instead it can store an ordered sequence of items, such as the list of entries in a menu or the list that a waiter writes when we order from a menu.

Example 1
What do you want to order?

The prince who asked us for oranges in Chapter 8 suddenly went bankrupt and has to earn a living as a waiter!

You enter the restaurant, examine the menu and call the waiter over. He writes down your order and leaves the stage to go into the kitchen and give them your order. Unfortunately for you, the chef sometimes gets confused and forgets to buy the ingredients for some of the entries on the menu. If an entry you ordered is missing, the waiter comes backs into the restaurant and informs you which entry is not available.

Building the menu

We construct the project in stages, starting with displaying the menu on the stage.

Task 1

> Display three menus on the stage: one for the main course, one for drinks and one for dessert. The main course can be chicken, hamburger, fish or taco; the drink can be cola, lemonade or water; the dessert can be fruit, ice cream or pie. In addition, display a page from the waiter's order book where he can write down your order.

Program file name: import-list-contents

? How can we display the menu on the stage?

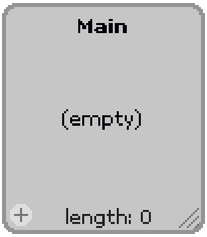
One possibility would be to display images of the various menus like we did for the oranges. We could use the Paint Editor to create images that are displayed on the background. However, this approach is not suitable for two reasons: First, given the confused state of the chef who forgets what she needs to buy, we will have to change the menus frequently during the animation and an image can't be changed. Second, we want to copy the entries that we choose from the menus to the waiter's order book.

What we need is a way of remembering the contents of a menu together with the ability to copy entries from the menu, change them and display them on the stage. This is precisely what can be done with *variables*, but variables are limited to remembering, copying, changing and displaying *single values*, while we need to do these actions for three or four values. Such an ordered sequence of multiple values is called a ***list***.

Creating a new list

Open a new project, initially without any sprites or scripts, and go to the Variables palette. Recall that is not a block that appears in scripts, but a button that creates a variable—an area in memory for remembering a value. Furthermore, when you create a variable for the first time, a set of blocks appropriate for computing with variables appears on the palette.

Just below the button Make a variable is another button Make a list. This button works the same way except that it creates lists instead of variables. Click now on this button. A window will appear asking you for the name of the list; choose an appropriate name, for example, `Main` for the list of entries for the main course in the menu.



Several things now happen. On the stage a rectangle representing the list appears. At the top is the name of the list, in the middle the items in the list, and at the bottom the length, which is the number of items in the list. Since no items have been placed in the list, the word (empty) appears in the middle of the rectangle and the value of length will be zero. In addition, a set of blocks appears in the red-orange Variables palette. The first block is a reporter block Main that can be used to obtain the value of the list. You can check the small square next to the reporter block to display a monitor for the list. Initially, this is checked and we leave it checked because we want the user to see the list of menu entries in the list.

The other instructions for lists will be discussed later in this chapter.

Exercise 1

Create three additional lists that are displayed on the stage: `Drink` and `Dessert` for the menus and `My meal` for a page from the waiter's order book.

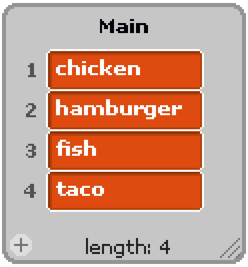
Entering content into a list

Now we want to enter content into the menus: the names of the different entries that can be ordered in each menu. We have prepared three files, one for each of the three menus. Here is how to copy the data from the files into the lists that represent the menus:

- Move the mouse until the mouse cursor appears on a rectangle representing a list, for example, the list for main courses. Click the right mouse button.

- A menu with three items will open. Choose the second item Import.... This causes a window to appear; select a file that contains the entries for the menu.

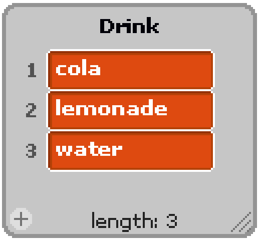
- The data for the file appear in the rectangle for the list. To improve the visual appearance of the rectangles, you can drag-and-drop them and you can resize them by dragging the diagonal marks on the lower right corner.



Main

1 chicken
2 hamburger
3 fish
4 taco

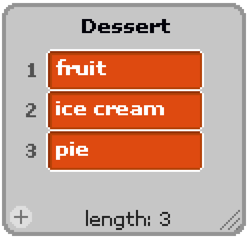
length: 4



Drink

1 cola
2 lemonade
3 water

length: 3



Dessert

1 fruit
2 ice cream
3 pie

length: 3

Look in the files themselves. They are text files with one row for each item in the menu: the file for the main courses (main.txt) has four rows, the file for the drinks (drinks.txt) has three rows, as does the file for the desserts (desserts.txt). If you look closely at these files, you will see that there are two spaces at the end of each row. These spaces will also appear at the end of each row in the list. For example, the first item in the menu for the main courses is "chicken " with nine letters including the two spaces. We have done this on purpose in order to simplify part of the animation as shown later.

Exercise 2

> Import entries for the drinks and dessert menus from the appropriate files.

So far, we have created four lists: a list of length 0 where the waiter will write down the order, a list of length 4 that contains the main courses, and two lists of length 3 with the lists of drinks and desserts. Save this project.

New concept: lists
A ***list*** is a construct for remembering complex data; it allows several related items to be saved together. Each item in the list has a ***position number***, which is its place within the list. An item can be ***added*** to an existing list, items can be ***deleted*** from a list, and it is possible to ***read*** items in the list by giving a position number and asking what item is at that position. The ***length*** of a list is the number of items in the list.

New construct in Scratch: creating a list

The button in the Variables palette brings up a window where you give the name of the list and select if it will be visible to all sprites or only to the current sprite. The reporter block representing the value of the list will appear on the palette. Check the small box next to the reporter to display the monitor for the list on the stage. The monitor is a rectangle that displays the name of the list, the items of the list in order of their position in the list, and the length of the list. If this is the first list that has been created, a set of blocks for the list instructions will appear on the palette.

New construct in Scratch: entering items into a list from a file

Items can be entered into a list from a text file. The file contains the sequence of items, one item on each row. Right click on the rectangular monitor for the list, and choose the entry Import .... A window will appear from which a text file can be selected. The items in the text file will be entered into the list and they will appear in the monitor.

Ordering a meal

Now we are ready to construct the animation by adding a waiter sprite.

Task 2

> Expand the animation by adding a button labeled `Order`. When this button is clicked, the waiter (who used to be a prince . . .) will appear and take our order. First he will ask what we want for our main course, then for our drink and finally for our dessert. The waiter will write the order in his order book and to confirm that it is correct, he will say it back to us.

Program file name: prince-takes-order

We need two sprites for this project: the waiter and the order button. The customer in the restaurant is the user of the animation (that is, you) and need not be represented by a sprite.

The customer calls the waiter

When you are ready to order, click the `Order` button. The button sprite's task is very simple: it just has to call the waiter. We will give it an additional task: to cause a clean page to appear in the waiter's order book. Here is a description of the behavior of the button:

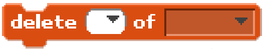
```
0. when Order clicked
    1. clear the order book
    2. call the waiter
```

Step 2 can be done by sending a message.

? How shall we implement step 1?

Deleting items from a list

The current page in the order book is represented by a list. To implement step 1 (*clear the order book*) we have to delete all the items in the list.

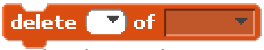
The instruction  ***deletes*** the item described in the first window from the list whose name is given in the second window. Clicking on the arrow in the second window causes a menu of all the known lists in the project to appear; we have only one list called `Menu` so it must be chosen. The first window has more possibilities: you can click on the window and type in the position of the item that you want to delete; alternatively, you can select one of three entries:

- `1`: delete the ***first item*** in the list;
- `last`: delete the ***last item*** in the list;
- `all`: delete ***all the items*** from the list.

Exercise 3

> Create a script for the `Order` button that implements the behavior described above.

New construct in Scratch: deleting an item or items from a list

The instruction  ***deletes*** an item or items from the list whose name appears in the second window. The item(s) to delete are given in the first window: If a number (or any block for a value) is entered, the item deleted is the one with that position number. (If the number is larger than the length of the list, nothing will happen.) The first window also has three menu entries that can be selected by clicking on the small arrow: `1` to delete the first item, `last` to delete the last item, and `all` to delete all the items (after which the list will be empty, of length 0).

The waiter arrives

After the user clicks on the Order button, the waiter sprite receives the message that is broadcast by the button sprite. He asks for the order, writes it down in his order book and repeats the order to the customer. Here is a description of the waiter's behavior:

```
0. when you receive the message that you are called
    1. appear before the customer
```

2. *ask the customer what he would like for his main course*
3. *write his answer in your order book*
4. *ask the customer what he would like to drink*
5. *write his answer in your order book*
6. *ask the customer what he would like for dessert*
7. *write his answer in your order book*
8. *repeat the order back to the customer*
9. *go to the kitchen*

We do not yet know the Scratch instructions for steps 2 through 7, where a dialog is carried out between the customer and the waiter, and the waiter writes down the answers in his order book. In steps 1 and 9, the waiter comes in from the kitchen and returns there. For simplicity, the kitchen is not part of the animation on the stage, so it is enough that the waiter appears and disappears. We can use the instructions hide and show from the Looks palette (Chapter 7).

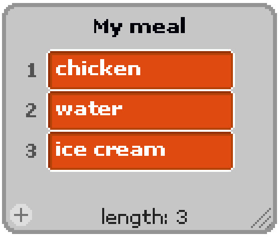
Exercise 4

> Begin to build the script for the waiter with blocks for steps 0, 1, and 9.
>
> The image for the waiter will be that of the prince from Chapter 8. You may want to change the sprite's name from `Prince` to `Waiter`.

There are two more parts to implement in order to obtain a working program: Asking and obtaining a order, represented by the three pairs of steps (2,3), (4,5), (6,7) and repeating the order (step 8). We choose to implement step 8 first; assume for now that steps 2–7 are already implemented so that the list contains three items, one for each course.

Reading the contents of a list

When the customer has finished ordering, the food that he ordered will appear in the list that the waiter write:



My meal

1 chicken

2 water

3 ice cream

length: 3

For step 8, we can use the *say* instruction.

? What should the waiter say?

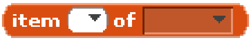
chicken water ice cream

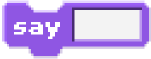
The waiter has to read the values of the items in the list to say the order. This is very similar to reading a variable and using its value, except that the *value* is the *entire list*. For this Example, we could use the reporter for the list `Menu`, but in future projects we will need to obtain the values individual items of the list, so we will implement step 8 in a different way.

In more detail, step 8 will be:

8.1 get the name of the main course

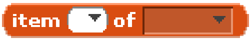
8.2 get the name of the drink
8.3 get the name of the dessert
8.4 compose a sentence from the three names
8.5 say the sentence

The block  that appears in the Variables palette can be used to ***read individual items*** from a list. In the first window, we enter *position number* of the item that we want to read. This can be a number or a value, or it can be the value `1` or the value `last` chosen by clicking on the small arrow. In the second window, we choose the list whose item we want to read.

is a value block whose value is the item at the place in the list given in the first window. Therefore, we can use this block in any instruction that takes a value, including, in particular, the  instruction.

However, step 8.4 asks us to compose a single sentence from the names of the three courses. To obtain this sentence, we use the operator that we learned about in Chapter 8. Use this operator twice, once to join the name of the main course and the name of the drink, and the second time to join the name of the desert to the value that results. Recall, that when we wrote the names of the entries in the files for the lists, we added two spaces at the end of each row; this ensures that when we join the names there will be spaces between each name.

New construct in Scratch: reading the contents of a list

The instruction  reads the value of an item in a list. The list is chosen from the menu in the second window and the value of the block is the item in that list whose position is given in the first window. The position can be:

- `1` or a numeric value: the value of the block is the ***item at that position***, provided that the position number is less than or equal to the length of the list; if the position number is greater than the length, the value of the block is empty;
- `last`: the ***last item*** in the list;
- `any`: a ***randomly chosen item*** from the list.

Exercise 5

Implement step 8 in waiter's script.

A dialogue: the waiter asks and the customer answers

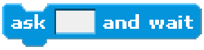
In steps 2 through 7 a dialogue takes place between the waiter and the customer. The waiter asks a question, the customer answers and the waiter writes the answer in his order book. This takes place three times, once for each course.

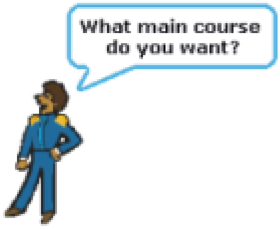
? How can we translate this into instructions in Scratch?

The waiter and the customer could use say instructions to ask the question and give the answer, but this will not give the correct behavior.

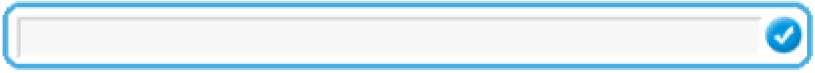
? Why?

First, we don't know in advance what the customer will choose for each course, so we can't write the appropriate instructions. Second, the *say* instruction only affects what is seen on the computer screen by the user, but it has no effect on the computation itself. In our case, we want to use the customer's answers in order to write entries in the waiter's order book.

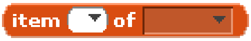
The instruction  allows us to ask a question and receive the user's answer as part of the computation during an animation. It is the fourth block from the top light of the blue Sensing palette. The window is used for the text of the question.



Let us see how this instruction works. Drag it onto the script area but don't connect it to any other blocks. Enter some appropriate text in the window, for example, "What do you want for your main course?" Double-click on the block. You will see the question that you entered appear in a bubble next to the sprite just like in a say instruction. At the bottom of the stage a long narrow frame will appear:



Type in your answer in the frame and then press the Enter key on your keyboard or click the check mark in the frame. The frame will now disappear.

We will be using the customer's answers to choose items within a list, so the answers must be *position numbers* that can be used in the instructions that read items of the list. For  instructions, the position numbers must be values that can be entered into the first window of the block. For example, if you want chicken as your main course, you should type in the number 1, while if you want hamburger, type 2, and so on. Of course, the number must be less than or equal to the length of the list (4 for the main course and 3 for the others).

Exercise 6

> Implement steps 2, 4 and 6 by adding *ask* instructions to the waiter's script.

Obtaining the answers to questions

? What happens to the answer that is typed in when a question is asked?

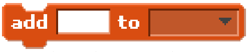
? How can we use the answer in a computation?

Scratch automatically saves the answer to a question into a variable called `answer`. This variable is built into Scratch just like the variables for the size of a sprite or its current costume, so you don't need to create it. The reporter block appears just below the block for the *ask* instruction in the Sensing palette. As with all variables, there is a small box next to the block; if you check it, a monitor for the variable will appear on the stage.

Modifying a list: adding an item to a list

All three of the customer's answers must be written in the waiter's order book, that is, the answers have to be added as items to the list. Steps 3, 5 and 7 can be described as follows:

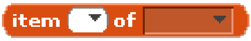
> *add the menu entry indicated by the variable answer to the list for the ord*

This can be implemented using the instruction  which adds the item in the first window to the list whose name appears in the second window. In our case, the second window will contain the name of the list `My meal`.

? What should appear in the first window?

Recall, that the customer answers each of the waiter's questions with a number, which is the *position* of the entry in the menu. Some restaurants have fixed menus, so the waiter need only write down a number and the chef knows what dish that number refers to. However, in this gourmet restaurant, the menu changes frequently, so the waiter needs to write down the entry itself (chicken or hamburger and so on), not just the number.

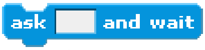
? How can we go from a position number to an entry in the list?

We have just learned the instruction that does this: . In the second window, select the name of the correct list: in step 3 this will be the menu for the main course, in step 5, the menu for the drink and in step 7, the menu for the dessert. In the first window, enter the position number of the entry that the customer selected. But this is exactly what has been remembered in the variable answer each time that the *ask* instruction is run.

Exercise 7

Add to the waiter's script the instructions needed to implement steps 3, 5 and 7.

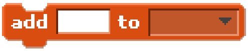
New construct in Scratch: conducting a dialogue

The instruction  ask □ and wait causes the contents of the first window to appear on the stage along with a frame for entering an answer. If there is a sprite on the stage, the contents of the window will appear in a bubble above the sprite; otherwise, the contents will appear in the answer frame.

The animation stops until the user has used the keyboard to enter text in the answer frame and pressed the Enter key or clicked on the check button. Then, the frame disappears and the animation continues.

The text that the user has entered is remembered in the variable □ answer. Each time that an ask instruction is run, the new answer replaces the old contents of the variable.

New construct in Scratch: adding an item to a list

The instruction  causes the value in the first window to be added to the list whose name is given in the second window. The item is added at the end of the list and the length of the list is increased by one.

Initializations

To finish the task, let us check the initialization. The scripts for the `Order` button are run when the button is pressed and the script for the waiter is run when he receives a message from the button.

? Do we need to run any instructions when the green flag is pressed?

The menus, together with the order book, are already on the stage when the animation begins, but if this is not the first time that the animation is run there might be a previous order in the order book (list `My meal`). The `Order` button opens a new page by deleting the items in `My meal`, but we would like `My meal` to be empty when the animation starts. Since the `Order` button is already responsible for the order book, we will also give it the responsibility of clearing `My meal` when the green flag is clicked. Here is the behavior of this script:

```
0. when the green flag is clicked
    1. clear the page in the order book stage
```

or, in more detail:

```
0. when the green flag is clicked
    1. delete all the elements of the list My meal
```

Exercise 8

> Add the initialization script to the `Order` button sprite.

This is not the only initialization that has to be done. When a Scratch animation is run all the sprites appear on the stage initially. However, we have decided that the waiter will not appear on the stage until the customer has looked at the menu and decided that he wants to order by pressing the `Order` button. Therefore, the waiter sprite must perform the following initialization:

```
0. when the green flag is clicked
    1. hide your image
```

Exercise 9

> Add the initialization script to the waiter sprite.

Add comments to the project, save it under a new name and try to run it.

Sorry, sir, we don't have that

As we mentioned at the beginning of the chapter, the chef gets confused sometimes and forgets to buy the necessary ingredients for some of the entries on the menu. Unfortunately, it is the poor waiter who has to deal with this situation. After he gives the order to the chef and finds out that something is missing, he must return to the customer and ask him to choose an alternate entry. Let us assume that every order has something missing so the waiter will always have to return to the customer.

Task 3

> Extend the animation as follows: after the waiter takes the order and disappears, he will reappear and tell the customer that one of his entries is missing. The missing entry (the main course, the drink or the dessert) will be chosen randomly. The customer can now click the `Order` button again and select three new entries.

Program file name: choice-not-available

First solution

From the definition of the task we can extend the behavior of the waiter as follows:

10. *wait a bit in the kitchen*
11. *return to the customer*
12. *randomly choose which entry is missing*
13. *tell the customer that this entry is missing*
14. *open a clean page in the order book*
15. *explain to the customer that he can order again by clicking the* `Or`
16. *return to the kitchen*

Exercise 10

> Extend the waiter's script to implement the behavior described in steps 10-16. Step 12 can be implemented by selecting a random number between 1 and 3.

Exercise 11

> Does the extended script require any changes in the initialization? Explain your answer. If changes are required, modify the script.

Add comments to the project, save it under a new name and try to run it.

Extending the solution—update the menus before taking a new order

It is very thoughtful of the waiter to agree to take a new order, but he would help the customer if the menus were changed so that they do not show the missing entry.

Task 4

> Extend the project as follows: after the waiter tells the customer about the missing entry, and before he asks him to click the `Order` button again, the waiter will delete the missing entry from the appropriate menu.

Program file name: delete-a-menu-element1

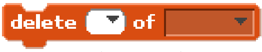
In order to solve this task we have to extend the second part of the description of the waiter's behavior by adding a line that deals with deleting an entry from the menu (line 15):

10. *wait a bit in the kitchen*
11. *return to the customer*
12. *randomly choose which entry is missing*
13. *tell the customer that this entry is missing*
14. *open a clean page in the order book*
15. *delete the missing entry from the appropriate menu*
16. *explain to the customer that he can order again by clicking the* `Or`
17. *return to the kitchen*

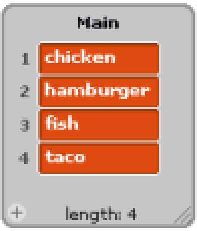
Step 15 is actually quite complex because the menu that has to be modified depends on which entry is missing. In more detail, step 15 is:

```
15.1 if the missing entry is a main course
    15.1.1 delete the entry from the main course menu
15.2 otherwise
    15.2.1 if the missing entry is a drink
        15.2.1.1 delete the entry from the drink menu
    15.2.2 otherwise (the the dessert is the only possibility left)
        15.2.2.1 delete the entry from the dessert menu
```

This behavior can be implemented using a conditional run instructions. The entry which is missing is selected in line 12. The order of the entries in the order book is the same as the order of the menus on the stage, which is also the order in which the waiter asked the customer for his order. That is, if the selected entry is 1 then the main course is missing, if the selected entry is 2 then the drink is missing, and if the selected entry is 3 then the dessert is missing.

A menu is represented by a list, so deleting an entry of the menu means to delete an item from the list. We have already learned the Scratch instruction for deleting from a list , although we used it to delete *all* the items in a list. The same instruction can be used to delete any specific item by giving its ***position number*** in the first window.

For example, if the entry to delete from the main course menu is `taco` whose position number is 4, then *delete 4 of Main* will delete it and



Main

1	chicken
2	hamburger
3	fish
4	taco

the length of the list decreases by one.

? What if the entry to be deleted is from the middle of the list (say hamburger)?

We do not want empty items to appear in the list so the items after the one that was deleted must move up one place. Fortunately, the *`delete`* instruction does this automatically.

We need three *`delete`* instructions, one to implement each of the steps 15.1.1, 15.2.1.1, 15.2.2.1. For each instruction, select the appropriate list in the second window.

We have a problem

Suppose that the customer chose hamburger as the main course and that is the missing entry. The *`delete`* instruction must contain the value 2, which is the *position number* of the item hamburger in the list. When the waiter sprite receives the number of the entry from the customer it is stored in the variable □ answer. The value of this variable is used to insert hamburger in the list representing the order book. However, what is stored in the order book is the entry `hamburger` and not its position. The position renames in the variable `answer`.

Unfortunately, by the time that the waiter needs to delete hamburger from the menu, its position has been erased from □ answer, because that variable was later used to store the positions for the drink and dessert courses when the waiter continued to take the order from the customer.

? Given an entry like hamburger, how can we find its position number?

There are two ways of solving this problem: When the waiter says that an entry like hamburger is missing, we can search for its position number within the list of main courses. Alternatively, we can remember—in additional variables—the position numbers of *each* item that the customer has ordered. Here we implement the second solution and leave the first solution until later in the chapter.

Exercise 12

Modify the description of the waiter′s behavior so that answers are stored in additional variables and then used in step 15.

Modify the waiter′s script to implement the new behavior: make three additional variables for remembering the answers of the customer. Add the appropriate instructions to store the answers and then to use the answer to implement step 15. Write comments and save the project under a new name.

Restoring the menus

Fortunately for the customers of the restaurant, whenever the chef discovers that some ingredient is missing, she sends her assistant to the market to buy it. When the ingredient arrives, we would like the menus to be restored to their original state with all the entries. The owner of the restaurant has invested in a robot whose task is to restore the menus.

> The menus can always be restored by re-importing the text files, just as you did when you created this project. Although we will now develop a program with a button to reset the menus, there is a lot to be said for reseting menus from files. The owners of the restaurant should not have to understand anything about lists or how they are implemented in a program. That is your job as a software developer. It will be much better if they can change the menus just by typing files with the entries of the menu.

Task 5

> Extend the animation as follows: Add a sprite for a robot. Whenever the robot is clicked it rebuilds the menus so that they are restored to the way they were at the start of the animation.

Program file name: restore-all

> Continuing our discussion above, it would be nice if the robot could read the menus from text files; however, there is no instruction in Scratch that allows a sprite to read a text file and initialize a list with the contents of the file. This can only be done by the programmer using `Import` button. That is why we have to add the items one by one to the list.

A first attempt

? What should be the behavior of the robot?

Whenever an item is added to a list, it is added to the list *even if it is already there*. Suppose that the robot tries to add the three entries in the drinks menu and suppose that cola is missing, but water and lemonade are still there. The result will be a list with 5 entries: water, lemonade, cola, water, lemonade.

? How can we prevent this from happening?

Before the robot adds the entries again, it must first clear the list to make sure that there will be no duplicate entries. Here is a description of the behavior of the robot:

```
0. when Robot sprite is clicked
    1. delete all the entries from the main course menu
    2. add the four entries for the main course to the menu
    3. delete all the entries from the drink menu
    4. add the three entries for the drink to the menu
    5. delete all the entries from the dessert menu
    6. add the three entries for the dessert to the menu
```

Exercise 13

Implement the behavior of the robot.

Guidance: Choose an image for the robot sprite (for example robot3 in the folder Fantasy) and give the sprite an appropriate name. Note that steps 2, 4 and 6 must be implemented by more than one instruction. Write comments and save the project under a new name.

A better solution: whatever is there, is there

The robot is doing unnecessary work. Why should it delete entries from that are still on the menu? It makes more sense just to add entries that are actually missing. That is, we want the behavior of the robot to be as follows:

```
0. when Robot sprite is clicked
    1. for each menu entry
```

```
1.1 if this entry is missing from its menu
    1.1.1 add it to the menu
```

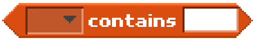
Step 1 is hiding a large number of steps, one for each menu entry, and these have to be written separately. For example, for the entry chicken, the steps have to be:

```
if chicken is missing from the main course menu
    add chicken to the main course menu
```

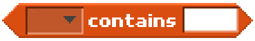
and for hamburger, the steps are:

```
if hamburger is missing from the main course menu
    add hamburger to the main course menu
```

Checking if an item belongs to a list

The condition  checks if an item belongs to a list; it appears as the last block in the Variables palette. The condition is *true* if the list whose name is given in the first window ***contains*** the value given in the second window and the condition is *false* if the the list ***does not contain*** the value. The block can be used whenever a condition is needed, for example, in an `if-then` instruction or a `repeat until` instruction.

New construct in Scratch: checking if an item belongs to a list

The condition 

is true if the list whose name is given in the first window ***contains the value*** given in the second window. If the item is not in the list, the condition is false.

The opposite condition—negating a condition

The robot is interested in finding out in what entries are *not in* the menus, so the descriptions above should be written as follows:

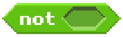
```
if it is not true that chicken is in the main course menu
    add chicken to the main course menu

if it is not true that hamburger is in the main course menu
    add hamburger to the main course menu
```

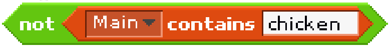
In order to use the condition `contains`, we have to ***negate*** it, that is, we need an operator with the following property:

> Apply the operator `negate` to the condition `Main contains chicken`.

- The result is ***true*** if the condition `Main contains chicken` is **false**;

- The result is ***false*** if the condition `Main contains chicken` is **true**.

The operator  implements negation. If you place a condition in the window of the operator, the resulting condition will be the negation of the one in the window. The operator is itself a condition. The following block implements the condition concerning chicken:

.

Exercise 14

Implement the full behavior of the robot using condition blocks. Check that it runs correctly, write comments and save the project.

Program file name: restock-missing

New construct in Scratch: negation

The condition not is the ***negation*** of the condition given in the window. (*not c*) is true if c is true and it is false if c is true.

Searching for the place of an item in a list

To conclude this chapter, we will improve on one of the projects that we constructed earlier and show how to search for the place of an item in a list. Recall, that when the waiter notifies the customer that a menu entry is missing, the waiter is also responsible for erasing that entry from the menu. The waiter remembers in variables the answers of the customer (the places of the chosen items in the lists), so it is easy for him to delete the missing item in the menu corresponding to the missing entry.

There is another way to solve the problem and that is to search for the item by name in the list representing the menu. This solution does not require that the waiter remember the answers of the customer; instead, we will give the responsibility for deleting items from the menu to the chef. After all, she was the one who ran out the ingredients!

Task 6

> Modify the animation for Task 4 so that deleting a menu entry will be carried out by the chef. She will come onto the stage from the kitchen, search for the missing entry and delete it from the menu. Then she will return to the kitchen.

Program file name: delete-a-menu-element2

What is left for the waiter to do?

Although we place the responsibility for locating the missing entry with the chef, we still need the waiter to tell her to do so. The behavior of the waiter is changed by replacing step 15 with the following steps:

```
15.1 if the missing entry is the main course
     15.1.1 tell the chef to correct the menu for the main course
15.2 otherwise
```

```
*15.2.1 if the missing entry is the drink*
    *15.2.1.1 tell the chef to correct the menu for the drinks*
*15.2.2 otherwise*
    *15.2.2.1. if the missing entry is the dessert*
        *15.2.2.1.1 tell the chef to correct the menu for the desserts*
```

As usual, communications between sprites can be implemented by sending and receiving messages from one sprite to another.

Exercise 15

> Modify the script of the waiter to implement the new steps 15. The variables used to store the customer's answers are no longer necessary and can be deleted.

The tasks of the chef

Since the kitchen is not visible on the stage, we will not animate the tasty cooking that the chef does, just the appearance of the chef on the stage and correction of the menus. The chef will correct the menus when she is notified to do so by the waiter. Since there are three separate notifications, one for each menu, the waiter will send three different messages, and the chef will need three scripts, one for receiving each message. We will discuss how to correct the menu for the main course and leave the other courses for you to do. Here is an outline of the chef's behavior:

```
0. when the message delete an entry from the main course menu
    1. go from the kitchen to the dining room
    2. search for the place of the missing entry in the menu for the main c
    3. delete this entry from the menu
    4. wait two seconds
    5. return to the kitchen
```

You already know enough about programming in Scratch to implement this behavior, except for the search in step 2.

How to search for an item in a list

The chef has to know the name of the menu she has to search and the name of the entry that she is searching for; for example, search for hamburger in the list `Main`. The waiter sprite is the one who knows which entry has to be deleted. If the waiter remembered which course is missing and which entry from that course, the chef can use these values to delete the entry. For now, let us assume that this is so; later you can check if the behavior the waiter has to be modified.

When the chef knows these two pieces of information: the course and the entry within the course, she can go to the menu for that course and search the entries one by one until she finds the missing one. She starts with the first entry, then the second one and so on until she reaches the last entry. We can be sure that the search will be successful because the missing entry is one that the customer chose and it must have been visible in the menu. Here is a preliminary description of the behavior of the chef for the main course:

```
read the first entry for the main course
run until the entry you are reading is the missing one
    read the next entry for the main course
```

If fact, this description will work when you need to search any list, not just the menus in the restaurant in our project:

```
read the first item in the list
run until the item you are reading is the one you are searching for
    read the next item in the list
```

This description is not yet detailed enough to enable us to implement the search. What is missing is an explanation of how to go from reading one item in a list to the next item. To do this, we need a new variable, one that will keep track of how far the search has progressed in the list. Let us call this variable `position`. Its initial value will be 1 since we start the search by reading the first item in the list. By increasing the value of `position` we can read the next item in the list.

```
initialize the value of position to 1
```

```
run until the item at position in the list is the one you are searching
    add 1 to position
```

The repeated run terminates when it finds the position in the list with the item that is searched for. The variable `position` will hold this position number.

Note how the variable is used: it is initialized to 1 and and is increased by 1 each time that the repeated run is run. This pattern of use is characteristic of variables that are used when searching through a list; they are called ***position variables***.

Trace the description of the behavior for each of the entries in the Main menu: chicken, hamburger, fish, taco.

Exercise 16

> In Chapter 6 we learned other patterns of using variables: *accumulators* and *counters* (which are a form of accumulators). Compare the position-variable pattern with these other patterns.

Here is a description of the behavior of the chef that is concerned with the search:

```
initialize the variable position to 1
run repeatedly until the item at the position
        whose value is that of the variable position is the missing en
    increase the value of the variable position by 1
```

When we place these steps within the full description of the behavior of the chef, we get the following list of steps:

```
0. when the message delete an entry from the main course menu
    1. go from the kitchen to the dining room (shown on the stage)
    2. initialize the variable position to 1
    3. run repeatedly until the item at the position
             whose value is that of the variable position is the missing en
        3.1 increase the value of the variable position by 1
    4. delete this entry from the menu
    5. wait two seconds
    6. return to the kitchen
```

Exercise 17

Implement the behavior of the chef for deleting an entry from the the three menus.

Guidance: Choose an appropriate image for the chef sprite from the People folder. The variable `position` need only be visible to the chef sprite, since only the chef performs the search.

Exercise 18

Does the addition of the chef sprite require any changes in the initializations of the other sprites? Explain your answer. If you think that changes are needed, make the changes in the scripts, write comments and save the project under a new name.

Exercise 19

There is another possibility for dividing the responsibility for the menus between the waiter and the chef sprites. Instead of having the waiter decide which menu needs to be changed and sending an appropriate message to the chef, the waiter will simply send a message to the chef that a change is needed. The chef will decide which menu to change. Modify the descriptions of the behaviors of the sprites to fit this approach, and implement the appropriate scripts.

New concept: searching a list
Given a list and a value, the ***search*** problem is to discover if the value exists in the list and, if so, at which position. If the value does not appear in the list, an appropriate message will be displayed.

Additional exercises

Exercise 20

a. Consider the project in file search1. It has one sprite: a math teacher. When the green flag is clicked, a list of ten numbers will be displayed on the stage. The numbers are randomly selected from the range 1 to 10. The teacher asks the user for a number, searches for it in the list and notifies the user of the *position* in the list where that number was found. Run the program several times: for first number in the list, the fifth number in the list and the last number in the list.

What happens if the number in the fifth or last places also appears earlier in the list? What happens if a number does *not* appear in the list?

Guidance: It will be easier to understand the runs of the program if you display a monitor for the position variable.

Program file name: search1

b. It seems that the problem of searching is more complicated than we first thought. When we searched for an item in the menu, we were assured that (i) the item appeared only once and (ii) the item actually appeared. Neither of these conditions is necessarily true.

Modify the project so that the teacher notifies the user if the item does not appear in the list.

Guidance: Use fixed repeated run instead of conditional repeated run to ensure that the teacher does not search beyond the end of the list.

Program file name: search2

c. Run our solution for the previous exercise and enter a number which appears in the list. The teacher notifies the user where the number appears, but he also gives a notification that the number does not appear! Explain why this happens and fix the program.

Program file name: search3

d. Since the numbers in the list are generated randomly, one number might appear several times. Construct a project where the teacher notifies the user of *all* the positions where the number that is entered appears.

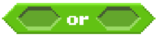
Program file name: search4

e. Run our solution for the previous exercise and enter a number which appears in the list. The teacher notifies the user where the number appears, but he also gives a notification that the number does not appear! Explain why this happens and fix the program.

Guidance: Use an additional variable.

Program file name: search5

f. Construct a different solution for (b). The problem we had resulted from the use of a fixed repeated run instruction instead of a conditional repeated run instruction. Solve the exercise using a conditional repeated run instruction with a compound condition



.

Guidance: One part of the condition will check if the number has been found and the other will check if the end of the list has been reached.

Program file name: search6

g. (Advanced) Here is another very elegant way to search for a value without running into the problem of searching beyond the end of the list. Add *the number you are searching for* as an additional (11th) element in the list. That way, you will always find the number you are looking for! In order to decide what notification to display, all you have to do is to check if you found the number you are searching for at position 11 (meaning that it wasn't in the original list), or at a position that is less than 11 (meaning that it was in the original list). This method of searching is called a ***sentinel search*** and is widely used because it is both efficient and safe.

Program file name: search7

Summary

Concepts

Lists are structures that can remember several values. A list is like an expandable box that is divided into compartments each of which can be used to store a value. We can add compartments or reduce the number of compartments. The current number of compartments at any time is called the ***length*** of the list.

Operators on a list are: ***add*** an item, ***delete*** an item or items, ***read*** an item, check if a list ***contains*** a certain item.

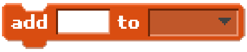
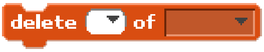
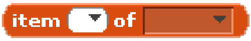
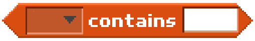
To ***search*** for the position of the specific item that exists in a list, ***position variable*** is used: it is initialized to 1 and then the item at each successive position in the list is read to determine if it is the one being searched for. After checking each item, 1 is added to the position variable so that the next item can be read. If the item is not found, a message is displayed.

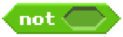
Given a condition `Cond`, the ***negation*** of `Cond` is true if `Cond` is false; the ***negation*** of `Cond` is false if `Cond` is true.

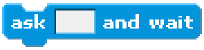
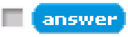
A ***dialogue*** can be carried out: a sprite asks a question and the user returns an answer.

Scratch instructions

The list operators are:

- : ***add*** an item;
- : ***delete*** an item or items;
- : ***read*** an item (this is a value block);
- : check if a list ***contains*** a certain item (this is a condition).

The condition block  gives the ***negation*** of the condition in the window.

A ***dialogue*** can be carried out using the instruction  from the Sensing palette. The text in the window is displayed and the run of the script stops until the user enters an answer. The answer is remembered in the variable .

Scratch techniques

To ***create*** a list in Scratch, click on the button Make a list in the Variables palette and give a name to the list. When the list has been created, a reporter for the list appears in the palette. A monitor can be shown on the stage by clicking the small box next to the reporter. The monitor contains the name of the list, the items in the list and the length of the list. When the first list is created in a program, a set of blocks will appear in the palette; these blocks are used for computing with lists.

Items can be placed in the list by ***importing*** them from a text file: click on the monitor for the list and select Import..., which will bring up a window allowing you to choose a file. Each row of the text file is a separate item to be placed in the list and the items will appear in the list in the same order that they appear in the text file.

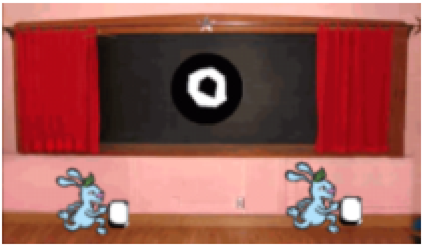
Chapter 10

Concurrent Run

Projects in Scratch are naturally concurrent. Already in Chapter 2 we saw an animation that contained several sprites whose scripts are run concurrently. In Chapter 3, we saw a slightly different form of concurrency—somewhat more difficult to understand at first—where a *single sprite* had several scripts, all of which are run concurrently. It is not surprising that concurrency in Scratch is natural and useful: even in our day to day life, the people and objects that surround us operate concurrently. We ourselves do several things concurrently; for example, we might eat or talk on the telephone at the same time that we are watching TV. We are also very familiar with concurrency in computers: You can start to download a large video and concurrently continue to write a document for your homework or chat with your friends.

We have seen in Chapters 7–8 that when a project has several scripts running concurrently, we have to carefully consider the interaction between the scripts. These interactions can cause the program to behave unexpectedly in ways that are hard to predict when we write the program. In this chapter we focus on the topic of ***concurrency*** and demonstrate how interacting scripts cause problems. We then show techniques for avoiding these problems.

Example 1
Educated rabbits



Two rabbits will participate in the project. The task of the rabbits is to read a number that is written on a blackboard and to write another number in its place. Each rabbit has a small blackboard of its own (called a *slate*) in addition to the large blackboard (called the *board*) on the wall of the room. A rabbit moves to the board, copies the number that is written there to his slate, and returns to his place. There, the rabbit adds one to the number written on its slate, moves back to the board, and copies the number from his slate to the board.

The graphics of the animation

To simplify this project, we have prepared the images that you will need in the file costumes-rabbits, so that all you have to do is provide the scripts. There will be three sprites in the project: `Rabbit1`, `Rabbit2`, and `Number` for the number that is written on the board. The board is not itself a sprite but a part of the background.

Each of the sprites has several costumes. To see the costumes, click on a sprite and select the Costumes tab in the center window. `Number` has several costumes, one for each value of the number that can be written on the board. The first costume is for the value 0, the second for the value 1, and the third for the value 2. Each of the rabbit sprites also has several costumes that correspond to the different values that can be written on the slate during the animation. Here too, the three costumes correspond to the values 0, 1, 2, in that order. The fourth costume for each rabbit represents a blank slate upon which no number is written.

Writing the numbers in order: first one, then the other

We start with a project where the rabbits do *not* run concurrently; rather the script for one rabbit runs to completion before the script for the other rabbit starts.

Task 1

> Initially, each rabbit will be placed in its own corner with a blank slate in its hand. The board in the center of the room has 0 written upon it. `Rabbit1` moves to the board, copies the number written there (initially 0) to its slate, returns to its corner and increases the value of the number on its slate by 1. Now it will move back to the board and copy the number 1 that appears on its slate to the board instead of the value 0 that had been written there. Finally, it returns to its corner and signals `Rabbit2` that its turn has come.
>
> `Rabbit2` will perform the same sequence of actions: it will move to the board, copy the number on the board (which is now 1) to its slate, move back to its corner, increase the value from 1 to 2, move back to the board, copy the number 2 from its slate to the board in place of the 1 that is written there, and move back to its corner.

Program file name: sequential-rabbits

Let us develop the animation in stages, starting with the behavior of Rabbit1.

0. when the green flag is pressed

1. *go to the corner of the room*
2. *clean your slate*
3. *move to the board at the center of the room*
4. *copy the number from the board to your slate*
5. *go back to your corner*
6. *increase the value on your slate by one*
7. *return to the board at the center of the room*
8. *copy the value from your slate to the board*
9. *return to your corner*
10. *notify* `Rabbit2` *that its turn has arrived*

This description is similar to descriptions that we have given before, but it is more general because we have not given details of the positions and motions. There are five steps which require motion: Step 1 where the rabbit is initially placed in its corner, and Steps 3, 5, 7, 9, where the rabbit moves to the center of the room and back, and again moves to the center of the room and back. It is very simple to translate these statements into instructions in Scratch. Step 10 is also easy, because it simply requires broadcasting a message to `Rabbit2`.

Modifying the slates and the board

Steps 2, 4, 6, 8 are more difficult to translate into instructions in Scratch. They all relate to the board on the wall and the slates held by the rabbits. The actions include changing the value written on a board or slate and copying values from the board to a slate and back. How can we represent different values on the board and the slates? The representation has to consider two aspects:

- First, we have to save, remember and read the value written on a board or slate, so that we can, for example, increase the value by 1. We will use variables to remember the number that is written on a board or a slate. It is easy to change the values of the variables to reflect what we intend to display.
- Second, we have to display these values. We prepared several costumes for each sprite that differed only in the numbers that appeared in the images. Therefore, changing the numbers can be done by changing costumes.

We expand the description of `Rabbit1`'s behavior as follows, using the variables `blackboard` to remember the value display on the board and `my slate` to remember to value displayed on this rabbit's slate. The message `your turn` is broadcast to `Rabbit2` when the script is finished.

```
0. when the green flag is pressed
    1. go to the corner of the room
    2. initialize the value of the variable my slate to 0
    3. initialize your costume to the costume with the blank slate
    4. move to the board at the center of the room
    5. copy the value of the variable blackboard to the variable my slat
```

6. *change your costume to display the value of the variable* `my slate`
   (the costume number is `my slate + 1`*)*
7. *go back to your corner*
8. *increase the value of the variable* `my slate` *by 1*
9. *change your costume to display the value of the variable* `my slate`
   (the costume number is `my slate + 1`*)*
10. *return to the board at the center of the room*
11. *copy the value of the variable* `my slate` *to the variable* `blackboa`
12. *return to your corner*
13. *broadcast the message* `your turn`

Exercise 1

In step 11, we changed the value of the variable `blackboard` but we didn't change its costume. Why?

Exercise 2

Translate the description into a script in Scratch. The initial position of `Rabbit1` will be in the lower left corner of the stage (−180, −130) and it will move to the center of the room at (−60, 0).

Guidance: You can use *`glide`* instruction for the rabbit's motion. You may want to use *`wait`* instructions between the *`glide`* instructions and the instructions that change costumes to slow down the animation so that it will be easy to observe.

The variable `blackboard` will be used by all sprites, while the variable `my slate` will be used only by the `Rabbit1` sprite. When creating the variables, check `For all sprites` or `For this sprite` only as appropriate. You should also remove the monitors for all variables by removing the check mark next to the variable's reporter. The reason is that we are representing the values using costumes so we can hide the values of the variables themselves.

The behavior of the second rabbit

How does the second rabbit behave? Its behavior is very similar to that of the first rabbit except that `Rabbit2` should begin to run its script only when it receives the message `your turn`. When it finishes running its script, it does not broadcast a message, because there are no more sprites in this project. Therefore, when the green flag is clicked, `Rabbit2` will only perform its initialization; the rest of its behavior is specified in a second script which will run when it receives the message your turn:

```
0. when the green flag is pressed
    1. go to the corner of the room
```

> *2. initialize the value of the variable* `my slate` *to 0*
> *3. initialize your costume to the costume with the blank slate*

0. when you receive the message your turn
> *(as for* `Rabbit1` *without 13. broadcast the message* `your turn`*)*

Private (local) variables

The scripts for `Rabbit2` also use two variables: `blackboard` and `my slate`. While `blackboard` is the same variable used in the script for `Rabbit1`, the variable `my slate` is a new variable used only by the scripts for `Rabbit2`. It doesn't matter that there are two variables named `my slate`; hopefully, when you created the variables, you told Scratch For this sprite only. Therefore, when the variable name `my slate` appears in the script for `Rabbit1`, Scratch knows that it is referring to the variable that belongs to the sprite `Rabbit1` and, similiarly, the use of the variable name `my slate` in a script for `Rabbit2` can only refer to the variable that belongs to `Rabbit2`. Think of two girls who have the same first name Rachel. No confusion will result if they belong to different families and never leave their houses because in each house there is only one girl named Rachel.

New concept: private (local) variables
A variable can be created so that it is accessible (for reading its value or changing its value) to only one sprite. The variable is said to be ***private*** or ***local*** to the sprite. Since the variable is accessible to only one variable, private variables in different sprites can have the same name and there is no ambiguity.

New construct in Scratch: creating a local variable

Select For this sprite only to obtain a ***local variable***.
A monitor for a local variable displays the name of the sprite that the variable belongs to so that you can distinguish between local variables of the same name.

The name of the sprite in a monitor functions like a family name to distinguish two variables with the same name. Just as Rachel Jones is not the same person as Rachel Smith, `Rabbit1 my slate` is not the same variable as `Rabbit2 my slate`.

Exercise 3

> Translate the description into Scratch. The initial position of `Rabbit2` will be in the lower right corner of the stage (180, −130) and it will move to the center of the room at (60, 0).

The Number sprite

The `Number` sprite represents the number displayed on the board. It must ensure that a change in the value of the variable `blackboard` causes a change in the costume of the sprite. The value of the variable `blackboard` is changed by the rabbit sprites, but in Scratch a sprite cannot change the costume of another sprite, so the number must be a separate sprite that can change its own costume.

How does the `Number` sprite know what its costume should be? This is very easy: the costume is the one whose number is one more than the value of blackboard: the costume that displays 0 is the 1st costume, the one that displays 1 is the 2nd costume, and the one that displays 2 is the 3rd costume. In general, the costume to be displayed is the value of `blackboard + 1`.

? When should the number sprite change its costume?

It should make the change when a rabbit sprite has changed the value of the variable `blackboard`.

? How can the number sprite know when that happens?

One way would be to use messages and to have the rabbit sprites broadcast messages to the number sprite, which would change its costume when it receives the messages. Another way is to use mailboxes (variables) to communicate between sprites. Fortunately, we already have the variable `blackboard` that is shared by all sprites. The number sprite merely has to check repeatedly if this variable has changed and change the costume accordingly. In fact, it need not use a condition to check the value; it can just change the costume according to the current value of the variable.

```
0. when the green flag is clicked
   1. initialize the variable blackboard to 0
   2. initialize your costume to costume 1
   3. repeated indefinitely
   3.1 set your costume to the costume whose value is blackboard + 1
```

Exercise 4

> Translate this description into Scratch. Run the project and check that is does what is required. Write comments on the project and save it.

Example 2
All together now

The rabbits will now run their scripts concurrently.

Task 2

> Modify the project so that the two rabbits move concurrently.
>
> Program file name: concurrent-rabbits

The project for Task 2 is in many ways simpler than the project for Task 1. Since the scripts for both rabbits run concurrently, there is no need for a message to synchronize them, and `Rabbit2` can start its run immediately when the green flag is checked. Remove the *`broadcast`* instruction from the script of `Rabbit1`, and the corresponding `receive` instruction from `Rabbit2` so that it runs all of its instructions after the green flag is clicked. The behaviors for both rabbits are the same, except for the positions in the motion steps.

Exercise 5

> Make these changes in the scripts so that the rabbits will run concurrently. Write comments and save the project. Run the project and see what value is written on the board at the end.

The result of running this animation is that 1 is written on the board at the end of the run. The rabbits move together, arrive at the board at the same time and copy the value of 0 to their slates. Both add 1 to this value, and both arrive again at the board at the same time and copy the value 1 to the board.

We hope that you are amazed at the result. Previously, two rabbits added 1 to the value 0 and the result was 2, as expected. Here, the result of adding 1 to 0 twice is 1!! This happens even though the computation of each individual rabbit is unchanged. The unexpected behavior occurred only because the scripts of the sprites run concurrently.

Example 3
All together now, but at different speeds

Real rabbits don't all move at the same speed! Let us see what happens when the two rabbits move at different speeds.

Task 3

> Modify the project so that the rabbits move at different speeds. We will arbitrarily decide that `Rabbit2` is much slower than `Rabbit1`. `Rabbit1` moves at the same speed as before, but `Rabbit2` takes five times as long as before to reach the board and five times as long to return to its corner.

Program file name: different-speeds

Exercise 6

> Make these changes in the scripts and run the project.

In this version, even though the rabbits run *concurrently*, the value written on the board at the end of the run is now 2, the same as it was when the rabbits ran *sequentially*, one after the other. This happens because `Rabbit1` manages to run through his entire script before `Rabbit2` reaches the board for the first time. `Rabbit1` copies the 0, increases the value to 1 and copies 1 onto the board. Only then does `Rabbit2` arrive at the board, copy the 1, increase it and copy the 2 onto the board. The effect is the same as if `Rabbit2` had waited for a message from `Rabbit1`.

All together now at different speeds, but with a surprise

We do not expect one rabbit to always be faster than the other. `Rabbit1` might start out moving faster, but tire and then move slowly. Let us see what happens when the speeds of the rabbits change randomly. `Rabbit1` will always run faster than `Rabbit2`, but sometimes very much faster and sometimes only a little faster.

Task 4

> Modify the project so that the rabbits run at different, randomly chosen, speeds. Specify the

> ranges of the random speeds so that Rabbit1 is always faster than Rabbit2.

Program file name: random-speed

In the initialization of the scripts, each rabbit will randomly choose the time that it takes to move from its corner to the blackboard. To ensure that there still a significant difference in these times, we decide that Rabbit1 will choose to move this distance in 1, 2 or 3 seconds, while Rabbit2 will chose 6, 7 or 8 seconds.

Each rabbit will save the random value in a new variable and then use the value of the variable in the first window of the *glide* instructions. We can use the same name speed for the variables of the two sprites if they are *private* variables. Select For this sprite only when you create the variables.

Exercise 7

> Make these changes in the scripts so that the rabbits run at different speeds that are chosen randomly. Run the project several times and see what value is written on the board at the end of the run.

Although the rabbits run concurrently, the value on the board at the end of the run can be different each time that we run the project! We cannot know in advance what the result will be because of the randomness. For example, if `Rabbit1` chooses 1 while `Rabbit2` chooses 6, `Rabbit2` will be very slow relative to `Rabbit1` and the final result will be 2, just like in the previous project. However, if the difference is small (`Rabbit1` chooses 3 while `Rabbit2` chooses 6), `Rabbit1` will not be able to return to the board and write 1 there before `Rabbit2` copies the original value of 0. The result will be that first `Rabbit1` writes 1 on the board and then `Rabbit2` also writes 1 on the board.

> ***It is important to understand that the different results are obtained merely by changing the rate at which the scripts run. There is no randomness in the computation of each rabbit.***

Different interleavings in concurrent runs

We have seen that when scripts are run concurrently, the relative speeds of the sprites can influence the result because the instructions are ***interleaved*** in different orders. This is shown in the following tables which have a column for each sprite and where the instructions are run in order from top to bottom:

Rabbit1 runs an instruction	**Rabbit2 runs an instruction**
Go to board and copy value to slate	
Return to corner and add 1 to value	
Go to board and copy value to board	
	Go to board and copy value to slate
	Return to corner and add 1 to value
	Go to board and copy value to board

However, when `Rabbit1` is roughly at the same speed as `Rabbit2`, the following sequence can be obtained that causes the value 1 to be written on the board:

Rabbit1 runs an instruction	**Rabbit2 runs an instruction**
Go to board and copy value to slate	
	Go to board and copy value to slate
Return to corner and add 1 to value	
	Return to corner and add 1 to value
Go to board and copy value to board	
	Go to board and copy value to board

Here is another possibility that leads to a final result of 1:

Rabbit1 runs an instruction	**Rabbit2 runs an instruction**
Go to board and copy value to slate	
Return to corner and add 1 to value	
	Go to board and copy value to slate
Go to board and copy value to board	
	Return to corner and add 1 to value
	Go to board and copy value to board

Each individual rabbit runs its instructions in the same order, but the overall order of the instructions that are run depends on the particular interleaving of the instructions from the two rabbits, and the final result can depend on the interleaving.

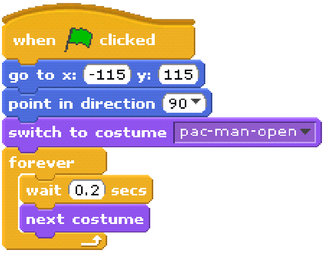
New concept: interleaving
When several scripts are run concurrently, the instructions from each script are ***interleaved*** to obtain the sequence in which the instructions are run. We are assured that each script is run sequentially, but there are many ways to interleave the instructions of different scripts, and the different interleavings may not give the same results.

Exploring concurrency in previous projects

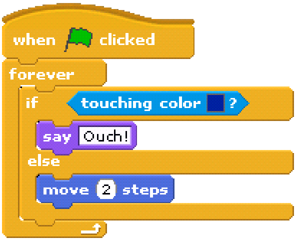
We created rabbit project to demonstrate that concurrent run can result in different results. The project is rather artificial, but the same phenomenon can occur whenever more than one script is run concurrently. In a project with multiple scripts, we must consider all the possible interleavings and ensure they all lead to a correct result.

Consider, for example, the Pac-Man game in Chapter 7. From the beginning, the Pac-Man sprite had two scripts that ran concurrently, one that was responsible for animating the opening and closing of the mouth by changing costumes and a second that was responsible for the motion of the sprite within the maze. Both scripts started with the same initialization instructions: placing the sprite in the top left corner, facing right. It would seem that one sequence of initialization instructions would be sufficient, so why did we “unnecessarily” duplicate these instructions?

Let use consider the version of the game in file pacman-moves-and-hits-the-wall, where the Pac-Man sprite identifies hitting the wall of the maze. The sprite moves as long as it doesn’t hit the wall, but when it does hit the wall, it says “Ouch!” Suppose that we include the initialization instructions *only* in the script that changes the costumes, while the script that is responsible for the movement does not contain initialization instructions.



```
when clicked
go to x: -115 y: 115
point in direction 90
switch to costume pac-man-open
forever
    wait 0.2 secs
    next costume
```



```
when clicked
forever
    if touching color ?
        say Ouch!
    else
        move 2 steps
```

Run the animation by clicking on the green flag. The Pac-Man sprite moves until it hits the wall, at which point it stops, says “Ouch!” repeatedly, and opens and closes its mouth repeatedly. Click again on the green flag. Since the scripts run concurrently, the following interleaving of instructions is possible:

> Immediately after the green flag is clicked (and *before* the script that changes costumes starts running), the script for moving the sprite starts to run. Since the initial position of the Pac-Man sprite has *not* been changed, it is *still* touching the wall, so it will say “Ouch!”. When the script for the costumes starts running, it will initialize the position of the Pac-Man sprite. The sprite will now move correctly but it will still be saying “Ouch!” because the say instruction without a time limit causes its string to appear indefinitely. It is only erased when the green flag is clicked again.

Let us show what happens in more detail. Ignoring the wait instruction, the instructions run by costume script are:

Costume script runs an instruction	
Initialize	
Change costume	
Change costume	
(... repeated indefinitely ...)	

The instructions run by the movement script are:

	Movement script runs an instruction
	if touching wall
	say "Ouch!"
	otherwise move 2 steps
	if touching wall
	say "Ouch!"
	otherwise move 2 steps
	(... repeated indefinitely ...)

We expect the interleaved run of the instructions to be similar to the one shown in the following table, where the initialization instructions are run first:

Costume script runs an instruction	**Movement script runs an instruction**
Initialize	
	if touching wall
	say “Ouch!”
	otherwise move 2 steps
Change costume	
Change costume	
	if touching wall
	say “Ouch!”
	otherwise move 2 steps
Change costume	
Change costume	
(... repeated indefinitely ...)	

But, since the order of interleaving is not known, it could equally well be as follows:

Costume script runs an instruction	**Movement script runs an instruction**
	if touching wall
	say “Ouch!”
	otherwise move 2 steps
Initialize	
Change costume	
Change costume	
	if touching wall
	say “Ouch!”
	otherwise move 2 steps
Change costume	
Change costume	
(... repeated indefinitely ...)	

This interleaving, when it happens after the Pac-Man has hit the wall, will cause it to say “Ouch!” while it is moving through the maze. To avoid such problems we introduced the initialization into both scripts.

Important note

The manner in which Scratch interleaves scripts is not explained in its documentation. There is no point is trying to understand exactly how Scratch works, because it is possible that the method will change in a future version.

If you are chatting on your computer or playing a game, you expect that the computer will work correctly whether or not you are downloading a video at the same time. The computer might run slower because of the extra work needed to download the video, but everything still works correctly.

New concept: correctness of concurrent programs
A concurrent program is ***correct*** if it gives correct results no matter how the instructions are interleaved.

Introducing order

The previous example showed that we have to ensure that unwanted interleavings of instructions do not occur. Let us modify the example with the rabbits to ensure that no interleaving results in an incorrect answer. Since the two rabbits each add 1 to the number on the board that is initialized to 0, the correct answer is 2.

Since the rabbits' movements are random, different answers can be obtained. Real rabbits cannot be forced to run at the same speed, so a program with randomness is a good simulation of the real world. Instead, we need to prevent the situation where one rabbit copies the value on the board to its slate while the other rabbit is in the middle of the process of adding 1 and copying the new value back to the board.

Task 5

> Modify the animation so that one rabbit always updates the value on the board before the other rabbit reads the value, regardless of the speeds at which the rabbits move.

Program file name: key-for-synchronization

? How can we do this?

We will require each rabbit to obtain ***permission*** to update the board. Permission will only be given to one rabbit at a time and the rabbit that doesn't have permission will not be allowed to go to the board in order to copy a number.

Suppose that above the board is a light that can be either red (meaning that the board is in use) or green (meaning that the board is free). A rabbit can go to the board only if the light is green. When the rabbit starts to move towards the board, the light must turn to red to prevent the other rabbit from going to the board. Only when the first rabbit has finished copying a new value to the board will the light become green again, allowing the second rabbit to proceed. The light prevents the unwanted behavior caused by concurrently running certain instructions from two scripts. Here is a description of the behavior of a rabbit:

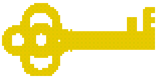
```
0. when the green flag is pressed
    1. go to the corner of the room
```

2. *initialize the value of the variable* `my slate` *to 0*
3. *initialize your costume to the costume with the blank slate*
4. ***request permission to go to the board and wait until you rece***
5. *move to the board at the center of the room*
6. *copy the value of the variable* `blackboard` *to the variable* `my slat`
7. *change your costume to display the value of the variable* `my slate`
8. *go back to your corner*
9. *increase the value of the variable* `my slate` *by 1*
10. *change your costume to display the value of the variable* `my slat`
11. *return to the board at the center of the room*
12. *copy the value of the variable* `my slate` *to the variable* `blackboa`
13. *return to your corner*
14. ***notify that the permission is not needed anymore***

Implementing the permission with a key

The problem with using the red and green light for permission is that we can't see which rabbit has received the permission, because the color red only means that one of the rabbits has received the permission.

We will use a different method:
a rabbit needs to have a ***key*** in order to go to the board. (The concept of a physical object representing permission is often used when a group is discussing something: there is one stick and only the person holding the stick is allowed to talk.) A `Key` sprite will be placed between the two rabbits. Initially it points upwards, signifying that the key has not been taken by



either rabbit. When a rabbit takes the key, the image of the `Key` sprite is rotated to point to that rabbit.

The following two scripts describe the behavior of the `Key` sprite when it receives messages from `Rabbit1`:

```
0. when Rabbit1 requests permission
    1. wait until the board is free
    2. remember that the board is in use
    3. turn towards Rabbit1

    0. when Rabbit1 has finished its task
    1. turn upwards
    2. remember that the board is not in use
```

The behavior when receiving messages from `Rabbit2` is similar.

Messages for carrying out the synchronization

The `Rabbit1` sprite communicates with the `Key` sprite by broadcasting messages (one for requesting and one for releasing). Similarly, `Rabbit2` uses two messages for communicating with the `Key` sprite. The `Key` sprite defines a variable `busy` to remember if the board is in use or not. 0 will indicate that the board is free and 1 that it is in use. The description of the behavior of `Rabbit1` is now:

```
0. when the green flag is pressed
```

1. *go to the corner of the room*
2. *initialize the value of the variable* `my slate` *to 0*
3. *initialize your costume to the costume with the blank slate*
4. ***broadcast message*** `Rabbit1 asks permission and wait`
5. *move to the board at the center of the room*
6. *copy the value of the variable* `blackboard` *to the variable* `my slat`
7. *change your costume to display the value of the variable* `my slate`
8. *go back to your corner*
9. *increase the value of the variable* `my slate` *by 1*
10. *change your costume to display the value of the variable* `my slat`
11. *return to the board at the center of the room*
12. *copy the value of the variable* `my slate` *to the variable* `blackboa`
13. *return to your corner*
14. ***broadcast message*** `Rabbit1 no longer needs permission`

The initial steps of the scripts of the `Key` sprite in terms of messages are:

0. *when the message* `Rabbit1 requests permission` *is received*

0. *when the message* `Rabbit1 no longer needs permission` *is receiv*

Similar descriptions can be given for the behaviors of `Rabbit2` and the `Key`.

Possible interleavings

Consider a possible run of these scripts where `Rabbit2` is the first to run. It sends a request message to the `Key` sprite.

Let us assume that busy is initially 0; then the script for receiving the request message can run to its conclusion. It will set busy to 1 and turn the image of the key towards Rabbit2. When the script has finished, Rabbit2 can continue running its script, going to the boards, copying the number and so on.

Suppose now that while Rabbit2 is at the board, Rabbit1 starts running its script and sends a request message to the Key. The Key will receive the message, but since busy is equal to 1, it will not continue running the rest of its script. Rabbit1 is waiting for this script to terminate, so it will not run anymore instructions. Only when Rabbit2 finishes its entire script (writing 1 on the board) will it send the no longer needs permission message that will cause the Key to reset busy to 0. Now, the script for receiving Rabbit1's request message can continue running. When it terminates, Rabbit1 will be allowed to run its script, eventually writing 2 of the board.

Exercise 8

Write a description of the initialization of the Key sprite.

Exercise 9

Construct a project with all seven scripts corresponding to the above descriptions. There will be one script for each rabbit and five scripts for the key (an initialization script and two scripts for the messages from each rabbit). Use wait instructions freely to make it easy to observe the animation: after a key turns and before a rabbit sends a request.

New concept: preventing unwanted interleaving by forcing sequential run
If some interleavings of a program are unwanted, they can be prevented by ensuring that parts of a script are run sequentially with no possibility of another script running its instructions at the same time. This ***synchronization*** can be implemented by using messages or global variables that are accessible to both scripts.

Exercise 10

Experiment with the use of random numbers for the time that the rabbits move from the corner to the blackboard and convince yourself that the final number displayed on the board is always 2, no matter what speeds are chosen. Change the *wait* instructions in the initialization of the rabbit scripts and convince yourself that the program is correct no matter which rabbit moves first.

Program file name: key-with-random-initialization

Additional Exercises

Exercise 11

Construct a game for launching rockets at an invading alien. You can start from the project costumes-monster, which contains costumes for the rocket (one of the rocket before launching and one with hot exhaust coming out of its nozzle), costumes for the monster and two buttons.

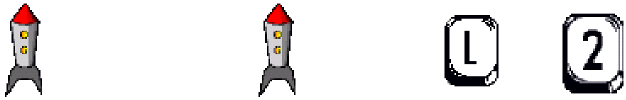
a. Construct an animation of a rocket launch. When the green flag is clicked, a rocket is placed at the bottom of the stage pointing upwards. Click a button labeled L to launch the rocket. After the rocket is launched, its costume changes. The rocket stops when it touches the edge of the stage. The left and right arrows on the keyboard change the direction of the rocket.

Program file name: launch-rocket

b. Add a second rocket that is also launched when the `L` button is clicked. Except for its initial position, the second rocket will use the same script as the first one.

Program file name: launch-two-rockets

c. Unfortunately, in (b) the arrow keys give the same control commands (left or right) to *both* rockets. Modify the animation so that the two arrow keys control only one of the rockets at a time. Add an additional button labeled either `1` or 2:



Initially, the button will randomly display 1 or 2. The arrow keys control only the rocket whose number is displayed on a button.

Guidance: Use a variable to control which rocket responds to the command.

Program file name: button-controls-one-rocket

d. Now add the monster to the project. The monster flies from right to left at the top of the stage. If one of the rockets hits the monster, the monster explodes. When that happens, stop the run of the program.

Guidance: The instruction stop all causes *all* scripts in *all* sprites to stop.

Program file name: rockets-with-monster

Exercise 12

The rockets need to refuel once they reach the top of the stage. Start from the project costumes-rockets, which contains two costumes for the rocket, one of the rocket before launching and one with hot exhaust.

a. When the green flag is clicked, the rocket is launched, but only after a random period has passed. At the top edge of the stage the rocket refuels—the rocket sprite says “Refueling”. Refueling takes a random period. When refueling is completed, the rocket returns to its initial position at the bottom of the stage and is launched again after a random period. This cycle of launch, refuel and return is run forever. Unlike Exercise 12, the rocket is launched automatically, not by the user.

Program file name: rocket1

b. Modify the appearance of the rocket so that when it is in flight a plume of flame appears shooting out of the rear nozzle. The flame will not appear when the rocket is on the launch pad or being refueled.

Program file name: rocket2

c. (Advanced) Modify the project so that four rockets are launched and refueled. They will be positioned at different points along the x-axis. Unfortunately, the space agency only provides two pumps for refueling the rockets. The third and fourth rockets that reach the top of the stage must wait (with their engines running) for a pump to become available. Have the waiting rockets say "Waiting to refuel" so we can see their status.

Guidance: This is a synchronization problem similar to the one with the rabbits who had to share one blackboard. The situation here is more complicated because *two* rockets can refuel at the same time. Use a variable `Refueling` that remembers the number of rockets *currently* refueling. Use different color effects to distinguish the rockets.

Program file name: rocket3

d. (Advanced) The four rocket sprites start at randomly chosen launch pads. When a rocket returns from refueling, it can return to any free pad, not necessarily the one it was launched from.

Guidance: Create a list of unused x-positions for the rockets. A rocket returning to the launch pad takes as its x-position the first element of the list. After launch, the rocket inserts its x-position at a random position in the list (select any in the second window of the block).

Program file name: rocket4

e. (Advanced) In the animation in (d), two rockets can crash into each other at the refueling pump. Modify the project to prevent this.

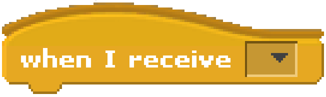
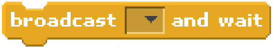
Program file name: rocket5

Summary

Concepts

A program may be composed of more than one component; these components run ***concurrently***, that is, their instructions are ***interleaved.*** We can be sure that the instructions of a single component are run one after the other, but we do not know how the instructions for different components are interleaved. A concurrent program is ***correct*** if and only if it's behavior is correct for every possible `interleaving`.

We must ensure that no interleaving of the instructions of the components of a program gives incorrect results. This is done by ***synchronizing*** the components: forcing some part of a component to run sequentially without interference from instructions in the other components.

Messages can be used for synchronization: a script with  will not start until a message is received and a script with  will not continue until the receiving sprite finishes its script.

Synchronization can also be implemented using a ***global variable,*** which is accessible to all sprites in a project. A variable can be created as a ***private (or local) variable,*** meaning that it is accessible only to one sprite. Since the name of the sprite is effectively part of the name of the variable, different sprites can have variables with the same name.

Chapter 11

Digging Further into Computer Science and Scratch (Optional)

During our journey to learn computer science and Scratch, we have followed a consistent path: we set ourselves the task of solving problems and then writing programs to implement the solutions. As the tasks get more complex, the implementation of programs needed additional instructions in Scratch. However, we have not studied the instructions in a systematic manner that would "cover" all the available instructions.

Having come to the end of our journey, there remain a few instructions that we have not explained. In this chapter, we will explore more concepts of computer science and additional Scratch instructions (though not all of them); by now, you should be enough of a Scratch expert to figure out the meaning of any remaining instructions that aren't explained! In this chapter, we will not give the development of the solutions; instead, we will pose tasks, present the new concepts and the new Scratch instructions that implement them, and leave it to you to develop the solutions.

A list of all the blocks arranged by palettes and by order within each palette can be found in the Scratch *Reference Guide*, which can be downloaded from the webpage that appears when Help entry is selected from the Help menu.

Example 1
Collision-avoidance radar

It is very sad when airplanes collide and crash, killing hundreds of people. In recent years, the number of collisions has been reduced, saving hundreds of lives, by the installation of *collision-avoidance radar* in each airplane. These radars scan the sky around the airplane and warn the pilot if another airplane is coming close. The radar can even suggest to the pilot in which direction to turn the airplane to avoid the collision.

Task 1

> Construct an animation of two airplanes crashing into each other. One sprite is an

airplane that takes off from the bottom left corner of the stage and flies towards the upper right corner. (Make sure that *no part* of airplane touches the edge of the stage.) A second sprite is another airplane that is flying from the upper right of the stage in order to land near the lower left of the stage (again, without touching the edge). When the airplanes fly at the same time, they get very close to each other, crash and fall out of the sky.

Program file name: collision

Task 2

Modify the animation for Task 1 such that when the airplanes get close, they both turn to their right, thus avoiding the collision.

Program file name: collistion-avoidance

Sensing the position of a sprite

We have frequently used instructions that sense a collision. In Chapter 2, we sensed collisions that occur if one sprite is *touching* another, while in Chapter 7, the Pac-Man sprite was considered to have touched the wall of the maze if it touched the *color* of the wall. The requirements of this problem are somewhat different: we need to sense if the an airplane is *close to* the edges of the stage and if it is *close to* the other airplane.

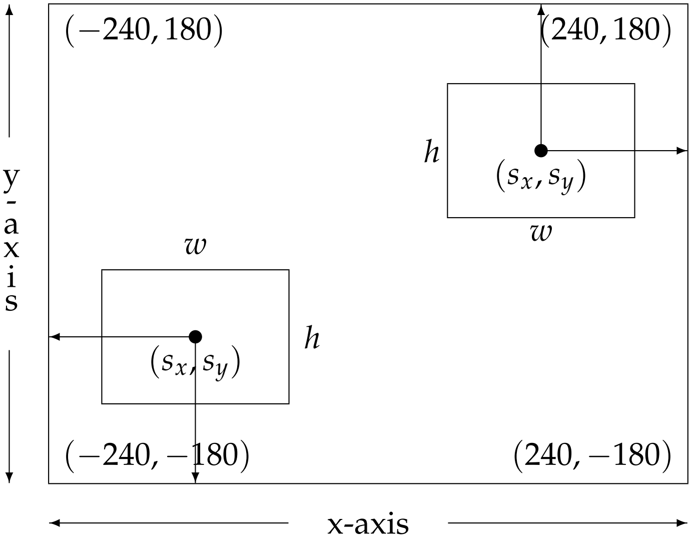
Consider first the requirement that the airplanes do not touch the edges of the stage. We know that the size of the stage is fixed so we know the range of values of the x- and y-positions: -240 to 240 for the x-position and -180 to 180 for the y-position. If the airplane sprites know their actual position, they can stop moving when that position is too close to the wall.

Let (s_x, s_y) be the positions of the center of an airplane sprite and let its width be w and its height be h. Then "too close" can be defined as:

$$s_x + \tfrac{1}{2} \times w < 240 \qquad s_x - \tfrac{1}{2} \times w > -240$$

$$s_y + \tfrac{1}{2} \times h < 180 \qquad s_y - \tfrac{1}{2} \times h > -180$$

as shown in the following diagram:

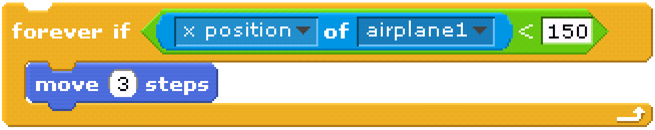


The operator enables a sprite to read properties of *itself or another sprite*; it appears near the bottom of the Sensing palette. Select a property in the first window and select a sprite in the second window. The properties `x-position` and `y-position` correspond to the positions s_x, s_y of the center of the sprite and we have to check that these positions don't get too small or too large:

$$s_x < 240 - \tfrac{1}{2} \times w \qquad s_y > -240 + \tfrac{1}{2} \times w$$

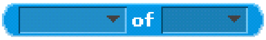
$$s_y < 180 - \tfrac{1}{2} \times h \qquad s_y > -180 + \tfrac{1}{2} \times h$$

For simplicity, we won't actually compute these values; instead, we guess (and then experiment with) values for the limits of x and y. The following instruction moves the airplane to the right until it is “close to” the right edge of the stage:



```
forever if < (x position of airplane1) < 150 >
    move 3 steps
```

New construct in Scratch: sensing a property of a sprite

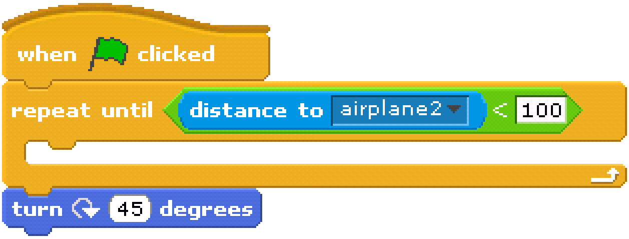
The operator  enables a sprite to *read the value of the property* chosen in the first window of the sprite chosen in the second window. The properties that can be read are the `x-position`, `y-position`, `direction`, `costume#`, `size` and `volume` of its sound.

The Stage can be chosen in the second window. In that case, the properties that can be selected are the `background#` and the `volume` of its sound.

Computing the distance from another sprite

Since the edges of the stage are at fixed positions, if we know the x- and y-positions of a sprite then we can check if it has touched an edge. However, collision avoidance requires that we know the distance *between two sprites*. This is a relative measure since it doesn't matter where the two airplane sprites are, only that they not be close to each other, say within 100 units of each other.

Given the x- and y-positions of both airplanes, we can compute the distance using a mathematical formula; however, this formula is not simple. Since animations frequently need to know distances, Scratch supplies an operator `distance to ▾` which can be found in the Sensing palette. Collision avoidance is now very simple: each airplane continues its movement until the distance to the other sprite is small, at which point, each airplane makes a turn to the right. For the sprite airplane1, an outline of the script is:



```
when green flag clicked
repeat until <(distance to airplane2 ▾) < 100>

turn ↻ 45 degrees
```

New construct in Scratch: distance to

The operator distance to ▾ computes the ***distance to*** another sprite that is selected in the window. The operator can also be used to read the distance of the sprite from the mouse pointer.

New concept: sensing as a way of transferring information

We have frequently used variables and messages as a way of transferring information from one sprite to another. Sensing can also be looked upon as a way of ***transferring information***: one sprite can sense if it is touching another sprite, touching a color that is part of the image of another sprite; it can also sense the distance to another sprite.

The use of the operator distance to ▾ enables us to work with the *concept* of the distance between two sprites, without worrying about how it is implemented. Another example is the instruction `glide`, which is a complex instruction that has to keep track of the position of the sprite and the time that has passed. Like the `distance to` operator, the glide instruction is easy to use without knowing how it is implemented. All software development environments provide ***libraries*** of computations, so that the programmer can easily use them without worrying how they are implemented. The term that is used is ***information hiding***, because the programmer can use a computation without knowing the hidden details of its implementation.

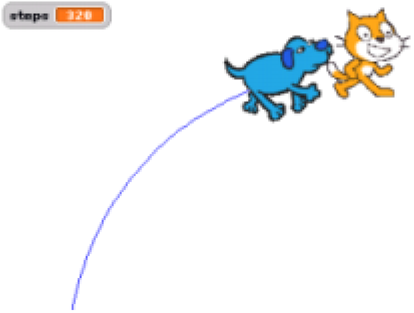
Professional software development environments support many ways of enabling the programmer to define *her own* instructions and libraries that can be used without knowing the details of their implementations. Scratch does not have these features, although you can do something similar by sending and receiving messages. A script that sends a message causes a script to be run when the message is received. Your friend could write that script and you could send it a message without knowing the details of the "hidden" script that your friend wrote.

New concept: information hiding
A computation can be implemented so that a programmer can use the computation without knowing the details of the implementation that are ***hidden*** from the programmer.

Example 2
Guiding a missile is like a dog chasing a cat . . .

A missile aimed at an airplane has to change its direction all the time so that it can hit the moving airplane. The simplest way to do this is called "chasing like a dog," because that is the way a dog runs after something like our poor Scratch cat. In this method, the dog always runs in the current direction of the cat.

Task 3



Construct a project with a cat sprite which runs across the stage from the top left to the top right and a dog sprite which runs after the cat, always running towards the *current position* of the cat. Count the number of steps that the dog runs until it reaches the cat and draw the track of the dog. When the dog reaches the cat, display the number of steps it has taken.

Program file name: stupid dog

There are three problems that we need to solve:

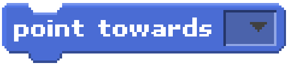
- The dog sprite must point in the direction of the cat sprite;
- The path of the dog must be drawn on the stage;
- The number of steps taken by the dog sprite must be displayed only when the dog reaches the cat.

We will solve these problems one at a time.

Pointing one sprite to another

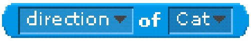
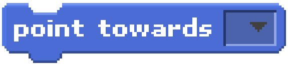
Once a sprite knows the x- and y-positions of another sprite, it is possible to compute the direction of the second sprite relative to the first one. However, this is an advanced computation that uses trigonometry, so Scratch supports this computation with an instruction.

New construct in Scratch: point towards

The instruction  from the Motion palette causes the sprite running the instruction to change its direction until it ***points towards*** the sprite (or the mouse pointer) that is selected in the window.

Like the `distance to` instruction, there are many advantages to using the `point towards` instruction. We don't have to invest the effort to write the mathematical computation and to check that it is correct. The use of a single instruction also makes it easier to write and understand the scripts that use it.

Exercise 1

> The operator  direction of Cat enables the dog sprite to obtain the direction from it to the cat sprite. Can we use this operator in order to point it in the direction of the cat instead of  point towards ? Explain.

Drawing on the stage

The task requires that the path of the dog sprite be traced on the stage. Scratch supports the concept of a *pen* being attached to (the center of) each sprite. As the sprite moves, the pen draws a line on the stage. The pen can be in two positions: *down*, in which case the line the drawn, and *up*, in which cause the sprite can continue to move but nothing is drawn.

New construct in Scratch: pen down and pen up

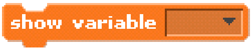
The instruction pen down in the dark green Pen palette causes the pen associated with the sprite to be ***put down*** so that it touches the stage. The movement of the sprite causes a trace to be drawn. The instruction pen up causes the pen associated with the sprite to be ***lifted off*** the stage, so that movement of the sprite does not leave a trace. The Pen palette contains blocks for changing the appearance of the trace.

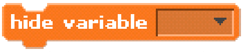
Sprites that draw with a pen are a central concept of the LOGO programming environment that you may have heard of.

Hiding and showing the monitor for a variable

The task requires that the value of the variable that counts the number of steps be displayed only when the dog catches the cat. If the task had allowed the variable to be continuously displayed, we could have simply checked the box in the reporter for the variable and the monitor would be displayed. There are instructions in the Variables palette that enable a script to control whether a monitor is displayed or not.

New construct in Scratch: hiding and displaying a variable

The instruction  causes the monitor for the variable selected in the window to be ***displayed on the stage***.

The instruction  causes the monitor for the variable selected in the window to be ***hidden***.

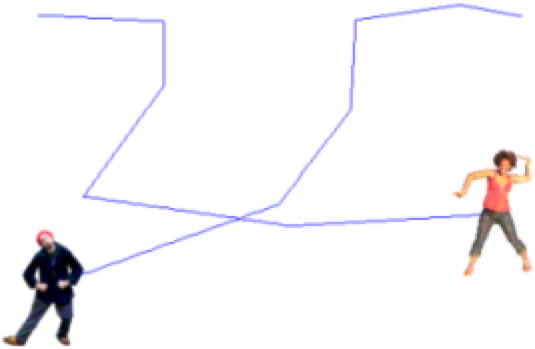
Task 4

A smart dog does not run directly to the *current* position of the cat, but to a point slightly ahead of it, *predicting* where the cat will be. Modify the animation so that after the dog sprite points to the cat sprite, it makes an additional small turn to the right. Experiment with values of this turn and see if the smart dog can catch the cat in fewer steps than the stupid dog.

Program file name: smart-dog

Example 3
Choreography—the depth dimension on the stage

Choreographers are people who design dances. They have special notations that they use to write down the steps that the dancers are to take. Let us construct a Scratch project that will allow us to design a dance, as shown in the following picture.



Task 5

> Let us start by constructing an animation for the single dancer Cassy. After the green flag is clicked, the user will move the mouse to a position on the stage and click the mouse button. When this has been done 5 times, the dancer Cassy will move from one point to another, tracing out the path of the dance.

Program file name: choreography1

Sensing the mouse cursor and clicks on the mouse button

Previously, we have used several forms of interaction between the user of Scratch and the animations that are being run, for example, waiting for keys to be pressed or answering questions being asked. Modern computer systems prefer to use position-oriented interaction, where the user clicks on a mouse or touches the display screen.

? How can we obtain the position where the mouse is clicked?

New concept: sensing the position and status of the mouse

The computer always knows the ***current position of the mouse***, which is defined by the x- and y-positions of the tip of arrow that is the mouse cursor. The computer can also sense when one of the ***mouse buttons has been pressed or released***. Clicking with a mouse button is really two separate events: pressing it and releasing it.

There are operators for sensing the position of the mouse cursor and button actions:

New construct in Scratch: sensing the position of the mouse

The operators mouse x, mouse y in the Sensing palette read the current ***x- and y-positions of the tip of the mouse cursor***.

New construct in Scratch: sensing the mouse button

The condition mouse down? is true when the (left) button of the mouse is pressed and is false when the button is not pressed.

Now that we know how to obtain the mouse positions, we can write a description of the behavior of the Cassy sprite that stores them in lists, one list for the x-positions and one list for the y-positions:

```
0. when the green flag is clicked
    1. initialize
    2. repeat 5 times
        2.1 wait until the mouse is pressed
        2.2 store the x- and y-positions of the mouse in lists
    3. repeat 5 times
        3.1 glide to the next point
```

This description is straightforward to implement but there is a technical problem. Steps 2.1 and 2.2 will be run 5 times, as specified in step 2. In fact, they will be run so fast that you won't have time to move the mouse to a new position of the stage. The result is that all five items in each list will be the same. To solve this, after reading the mouse position wait until the mouse button is released before proceeding:

```
        2.3 wait until the mouse is not pressed
```

Exercise 2

> Write a detailed description of the initialization and the actions on the lists. Implement the project in Scratch.

In front of or behind?

Let us return to the project with the second dancer Jay and make the dance more realistic so that sometimes Cassy passes in front of Jay and sometimes Jay passes in front of Cassy.

Task 6

> Take one of the projects with two dancers and modify it so that when Cassy is going right and Jay is going left, Cassy passes in front of Jay, while when they move in the other direction, Jay passes in front of Cassy.

Program file name: choreography2

Scratch uses two-dimensional graphics so that the images of the sprites do not have true depth. However, a simple concept of depth is supported by Scratch. The sprites don't just move on the stage; instead, they move in many ***layers***. You can think of each layer as a sheet of clear plastic, such that each sprite is drawn on one layer. Whenever sprites on different layers overlap, the sprite on a layer that is closer to the user (*in front*) hides whatever part of the other sprites it covers. By moving individual layers closer to or farther away from the front, the script can control how the sprites are displayed when they cross.

New concept: layers add depth to two-dimensional graphics
Sprites are drawn on separate ***layers***. When sprites do not overlap, all of them are displayed. When they do overlap, they are displayed such that sprites on layers closer to the front hide the parts of the sprites that they cover on the layers behind them.

Cassy will move her layer to the front when she dances from right to left, while she moves her layer behind Jay's layer when she dances from left to right. This is done using the following Scratch instructions from the purple Looks palette.

New construct in Scratch: moving sprites between layers

The instruction go to front moves the layer where the sprite is shown in front of the other layers. This sprite will be completely visible.
The instruction go back ○ layers moves the layer where the sprite is shown back the number of layers given in the window. If the layers in front of it contain sprites whose images overlap this sprite, the covered parts of the sprite are hidden.

Task 7

> Start from the animation for Task 5 and add the second dancer Jay. After the green flag is clicked, the user clicks the mouse ten times. The first five clicks specify Cassy's steps, while the second five clicks specify Jay's clicks. Then, both dancers move concurrently. When they pass each other, Jay acts like a gentleman and lets Cassy pass in front of him.

Program file name: choreography3

We will need four lists: the x- and y-positions for Cassy and the x- and y-positions for Jay. The positions of the mouse clicks must be stored in the proper order, so to prevent problems with the synchronization between Cassy and Jay, we transfer the responsibility for sensing the mouse clicks and storing their positions to the script for the stage. When all ten positions have been stored, the script for the stage notifies both dancers to start dancing at the same time.

Exercise 3

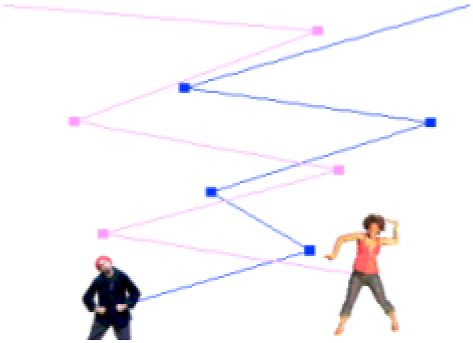
> Modify the animation to make the dance more interesting: Jay dances in front of Cassy on odd-numbered steps, while Cassy dances in front of Jay on even-numbered steps.

Guidance: Use the operator ○ mod ○ to decide if a step is odd or even. Divide the step number by two; if its remainder is 0, the number is even, while if the remainder is 1, the number is odd.

Program file name: choreography4

Exercise 4

Modify the animation of the previous exercise so that the dance is traced out. When the mouse is clicked, a small square is displayed at each point. The trace goes from one point to the next. Use different colors for Cassy and Jay.



Guidance: Add two sprites, a small pink sprite for Cassy and a small blue one for Jay. Use separate scripts for processing the mouse clicks, one for Cassy and one for Jay. Whenever a mouse is clicked, use the `stamp` instruction to place a mark on the stage. The traces are done using the instructions in the Pen palette.

Program file name: choreography5

Summary

Concepts

Information hiding: A computation can be implemented so that a programmer can ***use*** the computation without knowing the details of the implementation that are ***hidden***.

One sprite can ***sense a property*** of another such as the distance to it and its color. This is another way of ***transferring information*** between sprites. A program can sense properties of the ***mouse***: its ***current position*** as well as whether a ***button*** has been pressed or released.

Sprites are drawn on separate ***layers***. When sprites do not overlap, all of them are displayed. When they do overlap, they are displayed such that sprites on layers closer to the front ***hide*** the parts of the sprites that they cover on the layers behind them.

Scratch instructions

The operator (▾ of ▾) enables properties of one sprite to be read by other sprites.

The operator (distance to ▾) gives the distance from one sprite to another (or the mouse cursor).

The instruction point towards ▾ causes a sprite to point in the direction of another sprite (or the mouse cursor).

The instructions pen up and pen down cause the pen associated with the sprite to touch the stage or stop touching the stage, respectively. There are other instructions for setting properties of the pen.

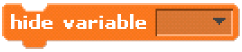
The operators (mouse x) and (mouse y) give the current x- and y-positions of the tip of the mouse cursor.

The condition ⟨mouse down?⟩ is true when the (left) mouse button is pressed and false when the button is released. To perform an action when a button is cliced, wait for ⟨mouse down?⟩ to be true and then wait for it to be false.

The instruction go to front moves the sprite's layer to the front where it is visible.

The instruction go back ○ layers moves the sprite's layer back by a number of layers. A sprite will be hidden if sprites in layers closer to the front cover it.

The instruction show variable ▾ causes the monitor for the variable selected in the window to be displayed on the stage.

The instruction  hide variable causes the monitor for the variable selected in the window to be hidden.

Looking Back

The purpose of this book was to acquaint you with concepts of computer science. You may be asking yourself: Do computer scientists really do the sort of things that you have learned? To a great extent, they do. While Scratch itself is not used by computer scientists, the principles and concepts that you have learned are routinely used by professionals. Computer science deals with ***problem solving***. The problems can be from many different areas and can have various levels of difficulty. However, the process of solving problems and the methods used are very similar to what you did. Although professional programming is usual done using textual languages not graphical languages, the constructs in professional programming languages are very similar to those in Scratch.

Solution by stages

Tasks are divided into several parts that are solved one by one. This ensures that each part is as simple as possible and thus easier to solve. There are several methods of software development that guide the division of a problem into tasks and the order in which they are solved, but the parts need not be solved in any particular order. The person solving the problem may choose to skip over the earlier parts, solve a later part and then return to the earlier ones.

Refining a solution

Solutions to a problem should be written in a general form and then gradually refined by adding more detail. This ensures that we have a good solution without deciding too early on the details. For example, the first step in a project could be just *initialize*; later, we would refine this by listing what variables and properties need to be initialized; finally, we would give the values that are given to each variable and property.

Verbal description

Solving a problem starts by expressing the solution in a natural language like English and only later is the solution translated into a precise form in a programming language that the computer can “understand” and run. (Sometimes, mathematical or graphical languages are used to express the solution to a problem.) In many cases, especially for very complex problems, the design of the solution and a description of its behavior (called an ***algorithm***) is the really difficult aspect of problem solving. Translating the design and description into a programming language is sometimes the simpler part of the task. Some computer scientists specialize in designing algorithms for complex problems even though they may not be the ones who implement the algorithms in computer programs.

Using known patterns

It is often the case that the person solving a problem finds that some parts of the problem have already been solved before, either by herself or by someone else. For example, if the problem involves searching in a list, the patterns for solving this problem are well known and can be used, perhaps with some adaptation. Similarly, counting and accumulating occur frequently and there are patterns for using variables to do this task. Using known patterns can significantly simplify problem solving. These patterns are found in computer science textbooks and in software libraries.

Hiding information

When you solve a problem, you make many decisions. These decisions may involve important aspects of the computation of the solution, but the user of the program need not know them. It is preferable to hide these decisions. An example would be the existence of some of the variables that are used in the solution. Information can also be hidden in the sense that one part of the program does not know how other parts are implemented. We saw an example of this when we used variables that were visible only to one script and not another. The advantage of information hiding is that parts of the solution are independent from one another, so that it is easy to change or improve one part without it affecting another one.

Fixing errors

Errors always occur when writing programs to solve problems. It is important that a computer scientist develop the skills needed to find and fix errors.

Documentation

Hopefully, when a computer scientist solves a problem, the solution is useful over a long period of time. A program frequently needs modification and improvement, and errors must be corrected even if appear only after the program has been used by many people. That is why it is so important to document your programs with comments so that they will be easy to change. You may not be the person making the changes, so you must explain your design in ways that others can understand. Even if you are the one making the changes, you may not remember all the details of your design that you did several months or years ago.

Concepts

Here is a list of important concepts (some of them quite advanced) that you have learned:

- Sequential and concurrent run
- Repeated run (bound and infinite, conditional or fixed)
- Conditional run
- Communications and cooperation by sending messages

- Variables
- Compound variables (lists)
- Random choice

If you have understood these concepts and if you have been able to use them when solving problems in Scratch, then you are familiar with many central ideas of computer science and you are well prepared to continue your studies in computer science.