Thinking about how to program

Dan MacLean

May 2022

Table of contents

What is Programming? **4**

1 Reflection **5**
- 1.1 Reflection and critique as programming tools . . . 5
- 1.2 Scratch - our creativity tool . . . 5
  - 1.2.1 Sharing Scratch Projects with Studios . . . 7
- 1.3 Criticising projects . . . 7
  - 1.3.1 What is appropriate criticism? . . . 7
  - 1.3.2 Delivering (and receiving) criticism well . . . 7
  - 1.3.3 An important note on groups of friends . . . 8
- 1.4 In this section we… . . . 9

2 Sequence **10**
- 2.1 Examining and building a sequence of instructions . . . 10
- 2.2 Building a sequence with constraints . . . 11
- 2.3 Fixing some bugs . . . 12

3 Loops and Conditionals **14**
- 3.1 Fixing some bugs . . . 14
- 3.2 Types of loop blocks . . . 15
- 3.3 Conditionals . . . 16
- 3.4 A creative project . . . 17
- 3.5 In this section we… . . . 17

4 Re-Use **18**
- 4.1 Implementing blocks . . . 18
- 4.2 Fixing some bugs . . . 19
- 4.3 Abstraction . . . 19
- 4.4 A major insight - its blocks all the way down . . . 20
- 4.5 In this section we … . . . 20

5 Variables **21**
- 5.1 Variables - names for changing values . . . 21
- 5.2 Building a game . . . 22

6 Final thought **23**

Appendices **23**

Acknowledgements **24**

What is Programming?

Wisdom

> This is not going to go the way you think.
>
> *Luke Skywalker*

Programming is the task of creating instructions so that a computer can perform a task for you.

At the highest level programming has two steps: design a solution to a problem, then encode the solution so a computer can do the work. The first part is a challenge for creative logic, the second a challenge for language. Programming a computer takes more of our imagination and creativity than it does of our cold, hard logic or mathematics.

This guide is intended to help a beginner reach an understanding of how to design a solution to a problem in such a way that it can later be translated into a programming language. No coding in a programming language is demonstrated in this guide, instead the guide focuses on the building blocks of solutions for programming: sequences, loops, events, conditions, and on the key computational thinking practices: experimentation and iteration, testing, re-use and abstraction. In this guide we use a graphical tool called Scratch to create and apply these concepts in a computer.

The guide aims to teach an understanding of the way that programs are built, before we understand how programs are encoded in a language.

Be prepared, this is not a typical programming course. It will be a largely self led exploration of concepts and ideas, not a slog through syntax and rote copying. If you've seen a programming guide before, forget it. Here your best tools will be your imagination, a notebook, a friend and your powers of reflection and criticism. This guide will be your research partner in the exploration helping you to reveal the fundamental concepts of programming.

For this course you will need.

1. A computer with an internet connection
2. A notebook
3. A friend (actually this is optional, but will be helpful).

1 Reflection

1.1 Reflection and critique as programming tools

The extent to which programming is a slow, difficult and often collaborative effort isn't widely appreciated. Stereotypes of 'hackers' in media abound - and this stereotype suggests that building a program comes from heroic spurts of inspiration and esoteric knowledge. More realistically, building a program is a slow process that involves thinking about a problem, weighing up different potential solutions and expressing them. Reflecting on problems and potential solutions and criticising them (in the strict neutral sense of evaluating and assessing dispassionately) are really important tools for advancing the solution to a problem and its implementation in a program.

The aim of this section is to encourage practice in these skills.

As so much of this guide will involve reflection and criticism, having someone you feel comfortable discussing stuff with will be helpful. It's not absolutely necessary, though. You can do all of this guide by yourself if you'd prefer.

1.2 Scratch - our creativity tool

Scratch is a free graphical computer program for creating media projects. It is available at the MIT Scratch website

With it you can create a wide variety of interactive projects - animations, games etc. Hundreds of thousands of people use Scratch across the world, including primary school children and Harvard computer science undergraduates learning to program. It's designed to be accessible yet complete. It encompasses all the key concepts we'll need to understand programming. Take a look at this introductory video.

Let's investigate Scratch!

For you to do

These tasks may seem trivial, hopefully they will seem playful. Have some fun with them.

1. Sign-up for Scratch https://scratch.mit.edu/
2. Browse some starter projects at https://scratch.mit.edu/starter_projects/ online
3. In your notebook, sketch ideas for three different Scratch projects you would like to create.
4. Go to Scratch and make the Scratch cat do something surprising

!

Wait, what?

Yep, item number four does say to go and do something in Scratch.

!?

But you haven't shown us how to use it? Are we supposed to just go and do it? What are we learning here?

Glad you asked! The object here would be for you to identify the stomach churning desperation that comes from not knowing how to solve something, yet being committed to doing so. And with that burning in your gut you manage not to be paralysed by the darkness of ignorance and still manage to claw your way into the light.

A lot of the time with programming you're not going to know exactly what to do at the outset. This task reaches out to your inquisitiveness and curious spirit. Just give it a go - you can't break anything or go wrong.

Look at this way, if you're sitting there thinking you don't know how to do it, anything you do will be a surprise!

You can do this, I believe in you.

Here's a starter sheet if you would like a *little* hint.

For you to do

When you've built something surprising, reflect on Scratch, perhaps using the points below, jot your responses in your notebook and/or with your friend(s).

1. What did you figure out?
2. What do you want to know more about?

1.2.1 Sharing Scratch Projects with Studios

Scratch Studios are a feature of the Scratch website, basically a sort of gallery to which you can post your creations. Let's use them!

For you to do

1. Find the Scratch Surprise Studio on the website http://scratch.mit.edu/studios/460431. Add your Scratch creation.

Here's a studio sheet for some hints.

1.3 Criticising projects

As you develop programming skills, being able to criticise and find strengths and weaknesses in your and other projects will help you build stronger, better projects.

Let's think now about criticising each others projects. If you are working with a friend (or group of friends) make sure you can find your projects in the Scratch Surprise Studio.

1.3.1 What is appropriate criticism?

This is a tricky subject, anything about *the project* could be criticised legitamately. But much of people's difficulty with giving and receiving criticism stems from the fact different people take criticism differently. But some rules of thumb can help you come up with constructive comments.

- Keep things about the project, not the person!
- Be convinced that there is a need for the point you wish to make - be sure that you believe it really *will* help
- Find and say a positive before you deliver a negative
- Don't say something because you haven't said something for a while.

1.3.2 Delivering (and receiving) criticism well

Some people are totally upset by the slightest perceived negative. These people are awesome to have on a team, they are very valuable as constant re-evaluators: iteration of a project can go much quicker when they're involved. At the other end of the scale some people are impervious to criticism and will take anything you can say and remain upbeat. These people are great at maintaining a positive direction in a team and help stop it from getting mired in details. But such diversity makes it difficult to pitch a point! As you'll have worked out, when delivering a point you need to try to be sensitive about people's views - just because you have a valid point you don't have a right to be rude or insensitive, you still have a responsibility to help maintain the civility of the conversation.

If you know you're the sort of person whom criticism affects strongly, try to accept other people's points with the presumption of good faith from the other side.

If you're the sort of person that criticism just bounces off, try to invest some time in thinking specifically about what you've received, try other people's ideas on for size.

For you to do

1. Fill in the table below for each Surprise Scratch project by each person in your critique group of friends.

Feedback by	What is something that doesn't work or could be improved?	What is something that is confusing or could be done differently?	What is something that works well or you really like about the project?
??	??	??	??
??	??	??	??

It may be helpful to think about:

- Clarity: Did you understand what the project is supposed to do?
- Features: What features does the project have? Does the project work as expected?
- Appeal: How engaging is the project? Is it interactive, original, sophisticated, funny, or interesting? How did you feel as you interacted with it?

1.3.3 An important note on groups of friends

When you choose a group of friends for this work, its going to be best if you can find someone with whom you can exchange ideas frankly. In an academic setting, probably not everyone you could work with will work well as a partner in a critique group. A lot of this will depend on different expectations of roles in people from different social and ethnic backgrounds. Try to work with someone at a similar professional level, it can be hard to tell someone in a more senior role that they're wrong , and what they say can be taken as 'the right answer' - when all contributions are valuable. Be aware too of the influence of a person's background on how they'd express a personal opinion. If you're from a place where it feels more natural to go along with the flow, try to find someone you'd be comfortable to express your ideas with. Experiment with sharing views over written media, rather than face to face.

1.4 In this section we...

Roundup

In this section we have endeavoured to practice the reflection skill so necessary for evaluating your designs and creations in software. The iterative development process relies on incremental improvements. Being able to give and receive new ideas (often from yourself) will help massively.

2 Sequence

Wisdom

A,B,C. It’s easy as 1,2,3. As simple as do-re-mi.

The Jackson 5

A computer program is a sequence of instructions for a computer to execute. Identifying the proper sequence of events is a vital skill. That’s the main aim of this section - breaking things down into sequences. The secondary aim will be to practice iteration, by experimenting with sequences of things we’ll initially get things wrong, but by iterating each time we will get closer to a good solution.

2.1 Examining and building a sequence of instructions

Let’s get straight to it and break down a sequence.

For you to do

1. In pairs work out who doesn't mind being bossed and who doesn't mind being bossy [a]
2. Bossed partner: Close your eyes! (or at least look away from the screen)
3. Bossy partner: Watch one of the videos [b]
    - http://vimeo.com/28612347
    - http://vimeo.com/28612585
    - http://vimeo.com/28612800
    - http://vimeo.com/28612970
4. Bossy partner: describe (using spoken words only) how to perform the sequence of moves in the video. Bossed partner: Do only what the bossy partner tells you, are there any points where you need them to be clearer? If they aren't making sense - say so.
5. Write down the steps as you go. Work from the steps.
6. In your design journal:
    - Reflect on what was easy/difficult about being bossy
    - Reflect on what was easy/difficult about being bossed
    - What was difficult about watching?

[a]As far as you possibly can anyway! If you're doing this alone you'll have to do both parts, which may mean developing temporary amnesia. If you can't decide which of you is which, then choose randomly.
[b]If you have more than one pair in the group, cover as many videos as you can.

The process above may have revealed a few points. The most apparent is that in order to be understood it is vital to be explicit about the action. Another is that the receiver of instruction (here the bossed partner) is not (often) able to 'just do what you meant' and get it right. The whole process probably took a few goes at least while you worked it out, so iteration is important. The list of instructions you created was a prototype program for a dance! This form of program, not real code - but a good description of the important parts is called `pseudocode`. The first draft of a real program will often be pseudocode.

2.2 Building a sequence with constraints

Computer languages don't have a command for each possible action. They instead have very restricted sets of key commands that must be chained and used creatively to make larger effects. The commands they do have are often quite limited and do very specific tasks.

This is a deliberate design feature. Flexibility and power comes from these small units, chained together in novel sequences.

In this section we'll examine using small units in different sequences.

For you to do

1. Start a new Scratch project
2. Using *only* the below ten blocks, make something that interests you
    - `go to`
    - `glide`
    - `say`
    - `show`
    - `hide`
    - `set size`
    - `play sound`
    - `wait`
    - `when this sprite clicked`
    - `repeat`
3. Share your creation in the 10 Blocks Studio http://scratch.mit.edu/studios/475480
4. In your critique groups or your design journal, discuss the following:
    - What was difficult about being able to use 10 blocks?
    - What was easy about only being able to use 10 blocks?
    - How did the constraint make you think of things differently?

2.3 Fixing some bugs

Let's examine some broken sequences and try to fix them. In this section the aim will be to explore the central sequence, and practice iterative testing and de-bugging. This is your first opportunity to try out the major parts of the development cycle.

For you to do

1. Do the five debugging challenges described on this debugging sheet
2. Discuss your testing and debugging practices with a partner. Make note of the similarities and differences in your strategies.

Roundup

In this section we've looked at the importance of sequence. Breaking a problem down in to small parts is a key step to solving it. Programs are built from a sequence of small units executed in a particular order. Applying and replying combinations of commands, testing them and repeating until the combination works is the main development loop of programming.

3 Loops and Conditionals

Wisdom

With just one polka dot, nothing can be achieved.

Yayoi Kusama

In this section we will get to grips with a key control feature of programming - looping. Looping at its simplest is just doing things over and over again until you have some reason to stop. Loops are a way of making the computer really work for you.

A loop is a code construct that contains code. Think of it as a box of code that gets run over and over. In Scratch the graphical metaphor is very simple, you have a block with a mouth, inside that mouth is the loop code, everything in the mouth gets repeated for as long as the loop definition says to. The loop definition is the the bit of writing at the top of the mouth.

These things make a lot more sense when you see them in use. Let's examine some loops by building some that make music!

For you to do

1. Do the build a band worksheet
2. Experiment with timing and the number of repeats in each repeat section to come up with a pleasing musical arrangement.

So that's the basic idea, loops do stuff over and over. They can do set numbers of repeats, keep going forever or keep going until something else happens.

3.1 Fixing some bugs

Let's have a look at some code with loops, some of which needs fixing.

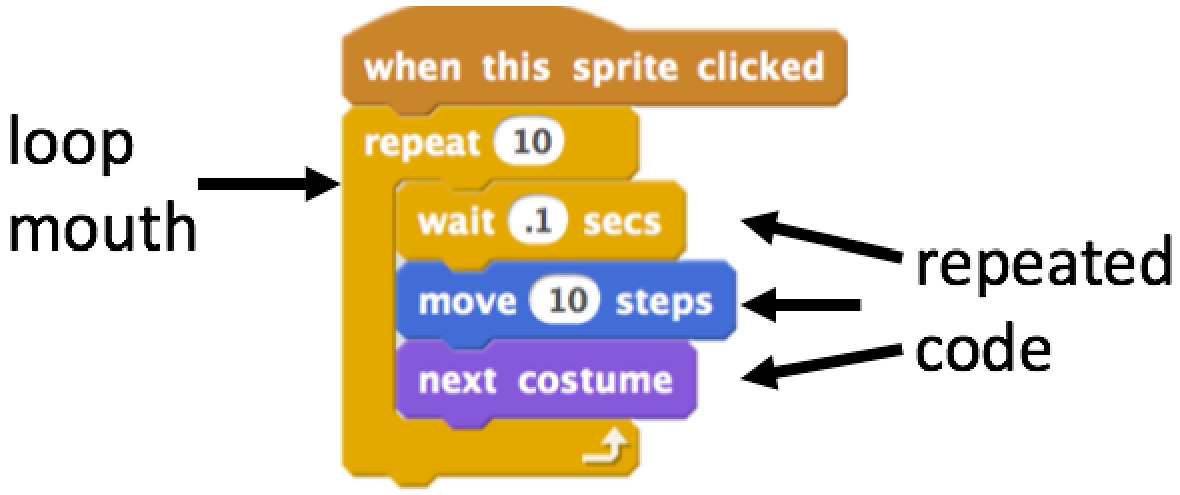

Figure 3.1: A loop block. The yellow mouth section is the definition that describes how often the loop runs. The code inside the loop mouth is the bit that actually gets redone over and over

For you to do

1. Do the loop debugging worksheet
2. In your critique groups or your design journal:
    - Discuss the importance of the counter in the loop? When can it break the code?

3.2 Types of loop blocks

Mostly here we've seen the `repeat` loop block, thats the most obvious sort but there are plenty of variations.

- `until` loop blocks do some code *until* something happens, that something can be specified by a e.g a sprite hits the side, or a number gets too big.
- `while` loop blocks do some code *while* some thing is still in some state,
- `forever` loop blocks do some code forever - this is more common than you think, especially in games for example. With these the main block of code just goes forever and stops only when something exceptional happens, like lives reaching `0`.

For you to do

1. In your critique group or design journal discuss whether all these types of loop are actually interchangeable?
2. Is there some sort of redundancy in all of these types of loop?
3. What might be the reason for this?

3.3 Conditionals

Along the way we've already seen a different type of control block - a conditional. This is a block that contains code that will only be done if something specified by the programmer is true. You'll notice this block because it begins with `if` and then there's some thing that must be tested to see if its true, this is the actual condition to be tested.

You can see the place for the condition in this picture as a gap after the word `if`, that gap is waiting for the programmer to drop in any sort of thing to test.

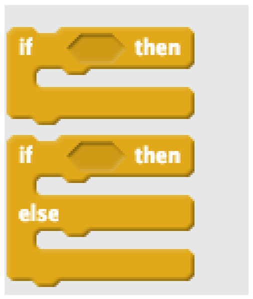

Figure 3.2: (Top) An IF block that only runs code if the condition slotted in the diamond shaped gap passes. (Bottom) An IF-ELSE block, in which the IF section works like the IF block and the ELSE section runs every time the IF section does not

A conditional (or simply just an `if` block) is the easiest way to get a computer program make a decision and only do something in certain cases. They are abundant throughout computer programming.

In the upper form the code in the 'mouth' will only be run if the conditional test passes. If it doesn't pass the test none of that code will be run. The rest of the program will carry on as if that code wasn't even there.

The `if-else` form provides a way to be explicit about what to do if the condition doesn't pass. If the condition for `if` doesn't pass, then the code in the `else` block runs. Note that the two blocks are mutually exclusive, if one block runs, the other won't.

3.4 A creative project

It's time to put some of our skills and creativity to use and build something brand new. For this next section we will build something personalised and/or exciting. Choose one of the activities below and enjoy building something you like!

For you to do

1. Make an interactive collage about yourself [a], using the About Me worksheet for a bit of guidance.

or

2. Make a music video. Use the Music Video worksheet

[a] Or an alter-ego you've constructed for the purposes of this exercise!

3.5 In this section we...

For you to do

We have encountered blocks and conditionals and use them to make stuff happen over and over, and only when we want it to. They save us from having to write out repetitive code.
These two types of blocks are at the heart of controlling the sequence of execution of code in a program. Practice with loops and conditionals will make you a powerful master of the things a computer does.

4 Re-Use

ℹ Wisdom

> We are like dwarfs standing upon the shoulders of giants, and so able to see more and see farther than the ancients.

Bernard of Chartres, circa 1130

Reusing code is the cornerstone of productive programming. Reusing libraries and packages built by other people is very common. Reusing your own code - packing it up into little re-usable blocks is also very handy. Designing your programs such that you can re-use them will bring you great benefits. The code will be easier to build and understand, this means you'll have less trouble building it, take less time and likely have fewer bugs.

In this section we'll explore personal re-use of something you built by making our own Scratch blocks.

4.1 Implementing blocks

Let's begin by making some blocks of our own.

For you to do

1. Do the Characters worksheet.
2. Make sure you define two characters and two behaviours per character
    - can you make `broadcasting` work between your characters? [a]
3. Discuss with a friend (or write in your design journal) about how `Make a Block` works. Come up with a description for a person new to blocks.
4. When might you use `Make a Block`?
5. I asserted that re-using blocks results in fewer code bugs. Discuss how that be justified?

[a]Don't panic if this doesn't work - broadcasting in Scratch highlights a technique in which we make different processes or scripts 'talk' to each other through messages. Its not a core technique by any means but the metaphor of 'talking' is here particularly strong - and it's quite fun - so its worth a go.

Hopefully through these exercises you've seen that code re-use is a timesaver - once you've worked out how to actually implement the code in new block that is.

4.2 Fixing some bugs

Let's get a bit more practice and intuition about how blocks work by fixing some broken ones.

For you to do

1. Do the Blocks debugging worksheet
2. Discuss the following:
    - How big or small do blocks have to be?
    - Would you ever want a huge block that did lots, or are smaller specific task blocks best?
        - Is there an optimal level of re-use?

4.3 Abstraction

A big plus to constructing blocks is that they are a tool for abstraction - they allow us to combine multiple small abilities into one larger, unifiying ability. This means that the amount of thinking we have to do about how our program works is less. In turn this means it is easier for us to progam.

4.4 A major insight - its blocks all the way down

If the concept of blocks seems like it might be a bit generally applicable, you're getting the key lesson of this section. All programming is either defining or re-using blocks of code. In this section with Scratch, its mostly been that our blocks have been a few commands that we want to re-use a couple of times. At different layers the blocks might be different levels of organisation of software. Some blocks with a few commands in each might be strung together into a bigger block called a script or program. A few scripts or programs might be strung together into a workflow, and there are levels under and above each of those. Beginning to think of programming at different levels of abstraction is a significant step in building your conceptual model of how programs and software in general works.

4.5 In this section we ...

Roundup

In this section we built and used some custom blocks. We've seen how they can be reused, and remixed - a process called abstraction and modularisation. These practices vastly reduce work for the programmer and increase integrity of our programs.

5 Variables

Wisdom

> There are only two hard things in Computer Science: cache invalidation and naming things.

Phil Karlton

Programs will often need to keep track of different things that will change during the course of a program. In this final section we'll make use of some simple variables and make a program that truly computes some stuff in response to inputs. We'll spend most of our time coding and end up having produced a simple game.

5.1 Variables - names for changing values

Consider a program that counts some things - minimally it is going to need a tally that it can increase as it finds each new thing. We use code constructs called variables to keep track of changing things. Variables have two parts - a name and a value. So in our counter example, the number of things we've seen is the value, whatever we decide to call that in our program is our name.

?

Hold on, isn't that just algebra? When did this turn into maths? What sort of scam are you pulling here?

Yes, it is quite like algebra in one sense. But in another sense, the most important sense - the practical - it's just names for stuff you don't know the value of at the start. Another distinction is that variables don't need to be just numbers. In other contexts you can keep track of text, pictures, sounds and even code itself. Don't get hung up too much on variables being for numbers.

5.2 Building a game

Let's go crazy and jump straight into building a simple game that will allow us many opportunities for keeping track of internal data computed by your script in variables.

For you to do

1. Do the Maze worksheet
2. Implement some 'collectable' sprites [a] that increment a score variable by one when your player sprite touches them. Check out the 'make a variable' block to do this
3. Implement some moving 'baddie' sprites [b] that decrement a life variable when your player sprite touches them. Add a "game over" process that stops the game when you run out of lives.
4. Consider adding a high score table. Discuss with a friend how you might do this and write your proposal in your design journal [c]?

[a]Like coins or cherries or something.
[b]Like ghosts or turtles or something.
[c]For bonus points, actually implement it.

Roundup

In this section we looked at variables. We saw that these are names for data that our program uses and changes accordingly as the program progresses. We marshalled our programming skills to create a game.

6 Final thought

In this brief guide, we have covered a great deal of ground. The concepts we have discussed are the very core ones of the daily practice of programming. Every time you start to program you'll use reflection, sequences, iteration and re-doing in the problem solving and design phase. When you come to put code down you'll implement it with loops and conditionals, variables and code re-use.

These concepts underpin every programming language. When you move on to build programs in other languages you'll use all of them. The implementation of loops, conditionals, blocks and variables varies from language to language, but underneath they're all the sames. For you, all that remains is the challenge of understanding the specific syntax of the new language.

As I wrote at the beginning the process of making a program is slow, so your first scripts in the new language may not be the most amazing. They'll improve with practice. You don't need to be a genius to program computers, you just need patience with yourself and a little determination to get there.

Good Luck!

Acknowledgements

The exercises, handouts and lesson plans in this document were created by Karen Brennan, Christan Balch and Michelle Chung at Harvard Graduate School of Education as part of the Harvard Graduate School of Education Creative Computing Project.

You can see their full and very exciting project at http://scratched.gse.harvard.edu/guide/.

Aspects of it are reused here, as intended and permitted under the Creative Commons 0 licence.