How software works

- Basically, computers (plus phones, smart devices, etc) are just circuitry designed to recognize specific patterns of 0’s and 1’s as instructions to carry out certain tiny actions
- Software programs are just thousands (or millions) of these instructions, arranged in a sequence that generates the behaviour we want

Simplified model of the hardware

- We can regard a computer as a processor (the brain), memory (holding the data we’re using), peripherals (all the other devices that capture, display, or store data), and busses (communication links connecting them all together)
- This is called the von Neumann model – most modern computer architectures are just variants on this (adding extra processors, caches, busses etc)
- (for a bit more background try von Neumann architecture on wikipedia)

Simplified model of the software

- When we type commands, tap on the screen, click the mouse, etc, we are issuing commands that must somehow eventually be acted upon by the computer
- The software that captures/interprets these commands is called the command shell, and is the outer layer of the operating system
- The operating system is the collection of software programs that control the computer actions
- The innermost layer of the operating system is the kernel: a specialized program that controls/guides/interacts with the processor itself
- Aside from the operating system, we have the various applications – all the extra software installed to do what we want (games, calendars, browsers, etc)
- (for a bit more background try kernel on wikipedia)

Running a program

- When we run (execute) a program, the operating system first finds it in storage, copies it into memory, and locates the first instruction in the program
- As the program runs, the processor repeatedly looks up (fetches) the next instruction from memory, figures out what the instruction means (decodes its bit pattern), and carries it out (executes it) ... the fetch-decode-execute cycle
- The cycle runs millions of times per second (essentially your processor clock speed)

Lots of programs run at once

- On most devices, we have multiple programs open and running simultaneously – but we only have 1 (or 2/4/8/etc) processor
- The operating system gives each program access to the processor for a short burst (tiny fraction of a second), then pauses that one and gives another program access for a short burst, then another, etc ... each program takes turns
- Because there are millions of cycles per second, it looks to the user like the programs are all continually running simultaneously (really they’re each being paused repeatedly)

Programs as bit patterns

- So programs are just thousands (or millions) of patterns of 0’s and 1’s, each with a tiny specific meaning
- Early (1940’s) programmers actually had to work out and program the right sequences of 0’s and 1’s to get the desired behaviour
- TEDIOUS AND ERROR PRONE
- 1947 Kathleen Booth (with von Neumann) got idea to give the instructions names (e.g. ADD, MOVE, etc) and write code using these, then use a program to translate them to bit patterns – thus the first assembly language/assembler created
- Huge improvement – faster/easier/safer development

From assembly to high level

- Assembly language way better than bit patterns, but takes an incredible number of assembly language instructions to do anything useful
- Grace Hopper (1952) got the idea to write code in higher-level, more human-friendly, instructions and use a program (compiler) to translate these instructions into sequences of lower level assembly instructions
- Thus the first high level language and compiler created
- Now there are thousands of languages/compilers, specializing in different kinds of application and/or for different platforms/devices

The programming cycle

- Most modern programming is done in a high level language (C++, Java, etc etc etc)
- Developer writes the code in chosen language, runs it through a compiler to produce an executable (and/or a lot of error messages indicating things that have to be fixed first), then runs the program to test it (and then back to writing/changing the code when it misbehaves)
- Interpretters are much like compilers, except that they decode and run the high level language statements directly, instead of generating the executable as an intermediate step (generally at the cost of run-time speed)