Computer languages

1. Introduction

An instruction running inside a CPU is in the form of binary.

But people find writing instructions as pure binary numbers too difficult and so more people-friendly computer languages were developed to help them write programs.

In this section we will not be discussing individual languages, but rather the different *types* of languages used in programming and how we translate person-written source code into a computer-readable binary file.

The types of language we will be covering are:

- Machine code
- Low-level language (Assembly code)
- High-level language (Source code)

As well as the tools used to write and convert them:

- Translators
- IDEs

- # Computer languages
- ## 2. Machine code

- A CPU can only process binary data, this includes the program instructions themselves.
- Therefore a typical instruction will look like this inside the CPU:

Typical Binary instruction							
0	1	0	1	1	0	1	1

- This is definitely not a person-friendly format. We cannot easily see what this instruction is saying.
- But we can improve readability slightly by representing is as a hexadecimal number, like this
    - 01011011 is **5B** Hex
- If you have read the section on binary and hexadecimal numbers (here), then you will know that a binary number is usually presented to users in hex format.

- An instruction that can run inside a CPU is called **machine code** and is in binary
- If you think of a CPU as a machine, then it should be no surprise that these instructions are called machine code.
- Each type of CPU uses different binary numbers to represent particular instructions, so the machine code written for one type of CPU will not run on another type of CPU. It will have to be re-written first.
- Programming in pure machine code is very difficult and time consuming. And so the CPU engineers offer programmers a slightly more person-friendly way of coding called assembly language.
- This is discussed on the next page.

Computer languages

3. Assembly language (low level language)

Machine code is made up of a set of binary instructions like this

```
Line 1:  10101010    (99 in Hex)
Line 2:  01011100    (5C in Hex)
Line 3:  00111010    (3A in Hex)
```

Each line is a single instruction acting on some data. We have shown them as binary and hex numbers above.

Assembly language is made up of a predetermined set of commands that can be read by a person (such as ADD). Each command translates directly into a machine code number. So these commands are basically mnemonics used by people to more easily remember, read, and write instructions for the computer.

Assembly language is called a **low level language** because it is almost the same as machine code. It has a low level of abstraction i.e. there is a direct link between an instruction in assembly and the machine code which it represents. The complete set of commands available is called the **instruction set** of the CPU. And each type of CPU has its own instruction set.
The instructions within an assembly language are called '**mnemonics**' as they are much easier to remember than a binary number.

Assembly language consists of a set of mnemonic instructions, each of which has a machine code equivalent.

The table below shows the machine code and the equivalent mnemonic in assembly language. You won't need to know these - it's just to give an example of what they might look like.

MACHINE CODE	MNEMONIC CODE
2403	ADD AL,3
4000	INC AX
2C02	SUB AL,2

Programming in assembly

Programming in assembly language is still quite difficult as the programmer needs to know a lot about the hardware details inside the CPU. Another issue is that the code only works on a specific CPU family. If you use a different CPU, you need to rewrite the code using a different version of assembly language.

The main advantage of assembly language is that programs in Assembly run *fast* compared to programs written in a high level language, because there is less code overall.

An example of a program that is typically written in assembly is a device driver, used to directly control hardware such as a graphics card. Drivers need to be as fast and efficient as possible and so assembly language is often used to create it.

Computer languages

4. High level language

High level languages were developed in order to make programming a lot easier for people, compared to coding in assembly language.

A high level language has two main features

- It has a set of **keywords** such as PRINT or IF
- It has a set of grammar rules (**syntax**) which define how to combine them correctly

A programmer writes **source code** (programming code) using the allowed set of keywords and its syntax rules.

There are many high level languages. For example Java, Python, C++, C and so on. High-level languages are "machine-independent". That means that they can be ported to different computers and still run.

Some Python source code is shown below

```
print("This program is going to state the person \n")

# set a variable for what each person is like
john = input ("What is John like? ")
sue = input ("What is Sue like? ")
derek = input ("What is Derek like? ")
paul = input ("What is paul like? ")
print("\n")

# Now print them out

print ("John is ",john,"\n")
print ("Sue is ",sue,"\n")
print ("Derek is ",derek,"\n")
print ("paul is ",paul,"\n")
```

It shows typical keywords such as 'print' and 'input'. They are easier to read and understand compared to a low level language.

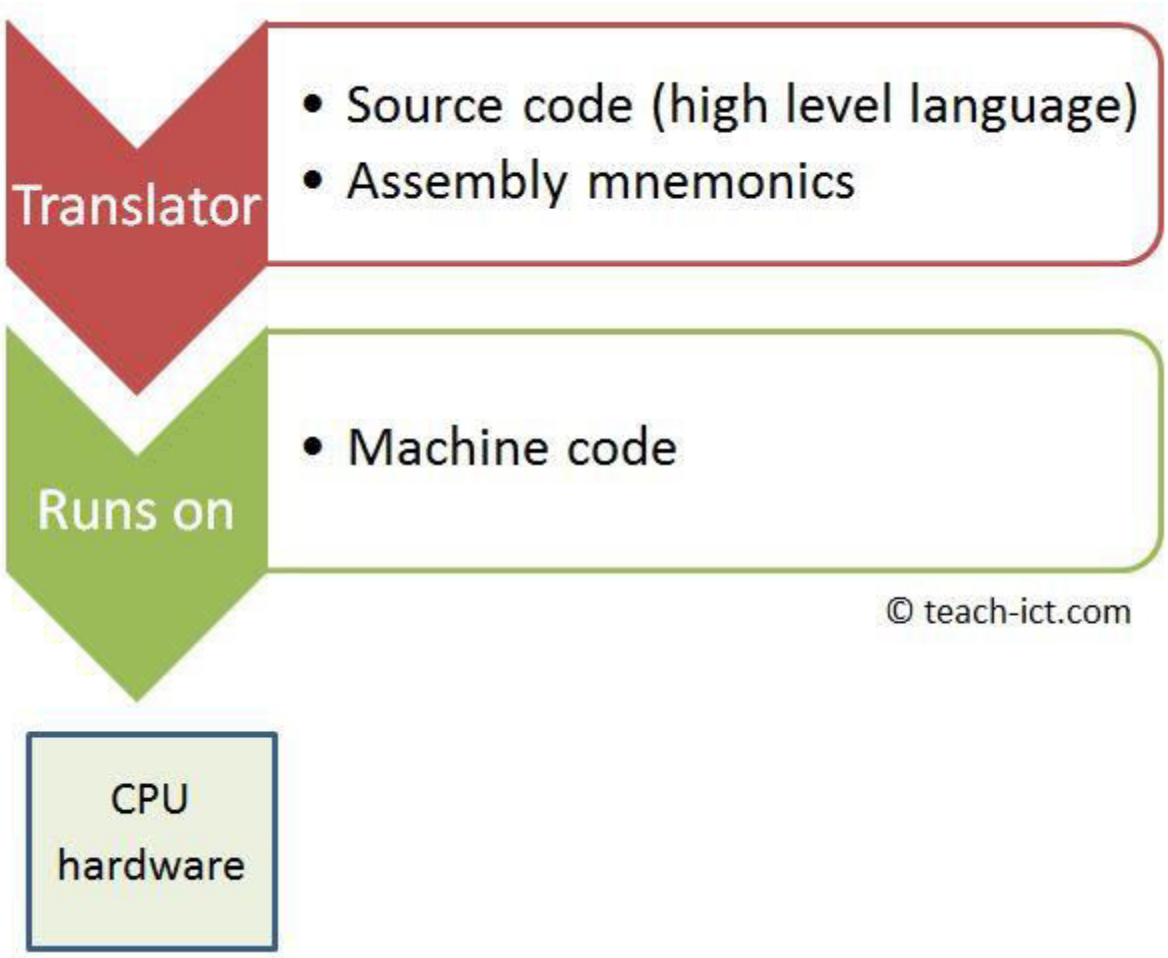
However, source code needs to be converted into machine code to run in a CPU.

This is the task of a piece of software called a 'translator' which we discuss next.

Computer languages

5. Translators

A **translator** is a piece of software that converts programming code into machine code. The programming code could be high level source code or low level assembly instructions.



The machine code then runs in the CPU hardware - or it is stored as a file and later loaded in the CPU to run.

There are three different types of translators which are:

- **Assembler** - converts assembly language instruction into machine code
- **Compiler** - converts high level source code into machine code and stores it in an executable file
- **Interpreter** - converts a single line of high level source code into machine code and then immediately runs it on the CPU.

These are discussed on more detail on the next page

Computer languages

6. Compilers

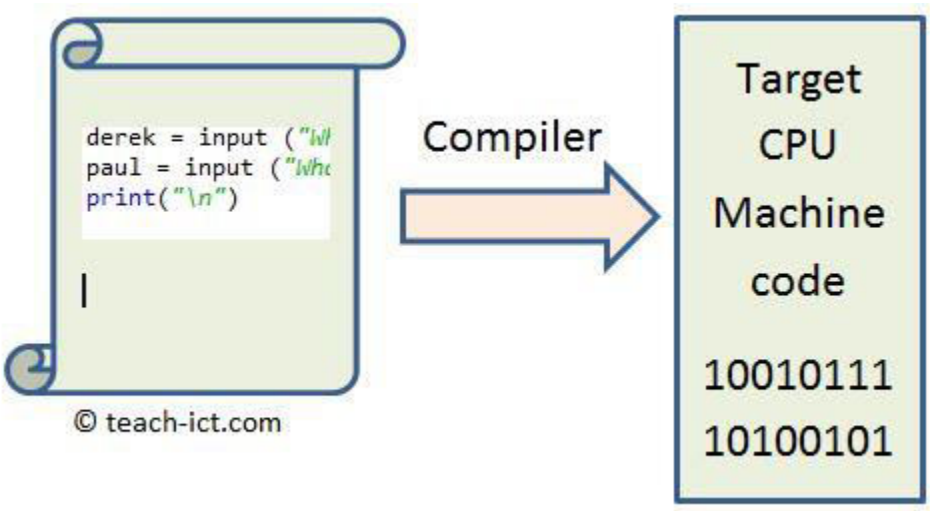
Compilers translate programs written in a high-level language into machine code instructions.

Compilers convert programs written in a high-level language into machine code instructions.

A compiler reads *all* the source code in one go, examining each keyword and their arguments. It translates those keywords into a set of machine code instructions. Each keyword often translates into a lot of machine code instructions.

Often, the compiler will check the code for basic syntax errors, and notify the user if it finds any.

Once the code has been successfully "compiled", it is stored in an executable file. This executable file can then run without either the compiler or the source code being present.



Linker

Sometimes a large program is broken down into several software modules. Each module is compiled into an **object file** which has machine code. Then these object files are linked together to form a single executable file. The software that combines object files is called the linker.

Computer languages

7. Interpreters

The final type of translator we will cover is the **interpreter**.

Much like a compiler, an interpreter translates source code written in a high-level language into machine code. Where it differs from a compiler is that it doesn't translate the entire source code in one go.

Instead, an interpreter translates a single line of source code and then immediately carries out the resulting machine code instructions. Then it reads the next line of source code and does the same. It carries on doing this until the program is terminated.

Interpreters convert source code into machine code instructions and execute those instructions immediately, line by line.

Because of this source code -> translate -> run loop, an interpreter tends to be a bit slower compared to compiled code.

The source code can be stored in a text file which the interpreter reads and translates line by line.

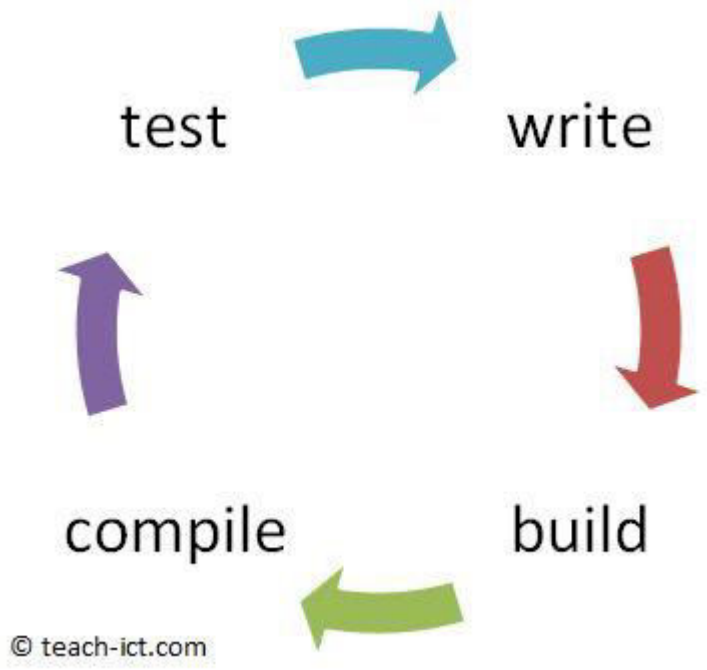
The advantage of an interpreter is that it is excellent for developing and testing code as any errors show up immediately.

The disadvantage is that the interpreter must always be present to run the source code. And the source code itself also needs to be available.

Computer languages

8. IDE - Integrated Development Environment

Writing a computer program involves a number of stages



A typical code project involves going around a write -> build -> compile -> test loop as the programmer builds up the program. A small chunk of the code is written then tested. Once that bit works another chunk of code is written and tested.

For larger programs, there may be many source code files, the 'build' phase combines them together ready for the next stage. The source code is compiled and then tested. Then the cycle starts again for the next chunk of programming.

Each stage needs a different set of tools and it can be really awkward and slow to move from one tool to another. The solution is an application called an **IDE** - Integrated Development Environment.

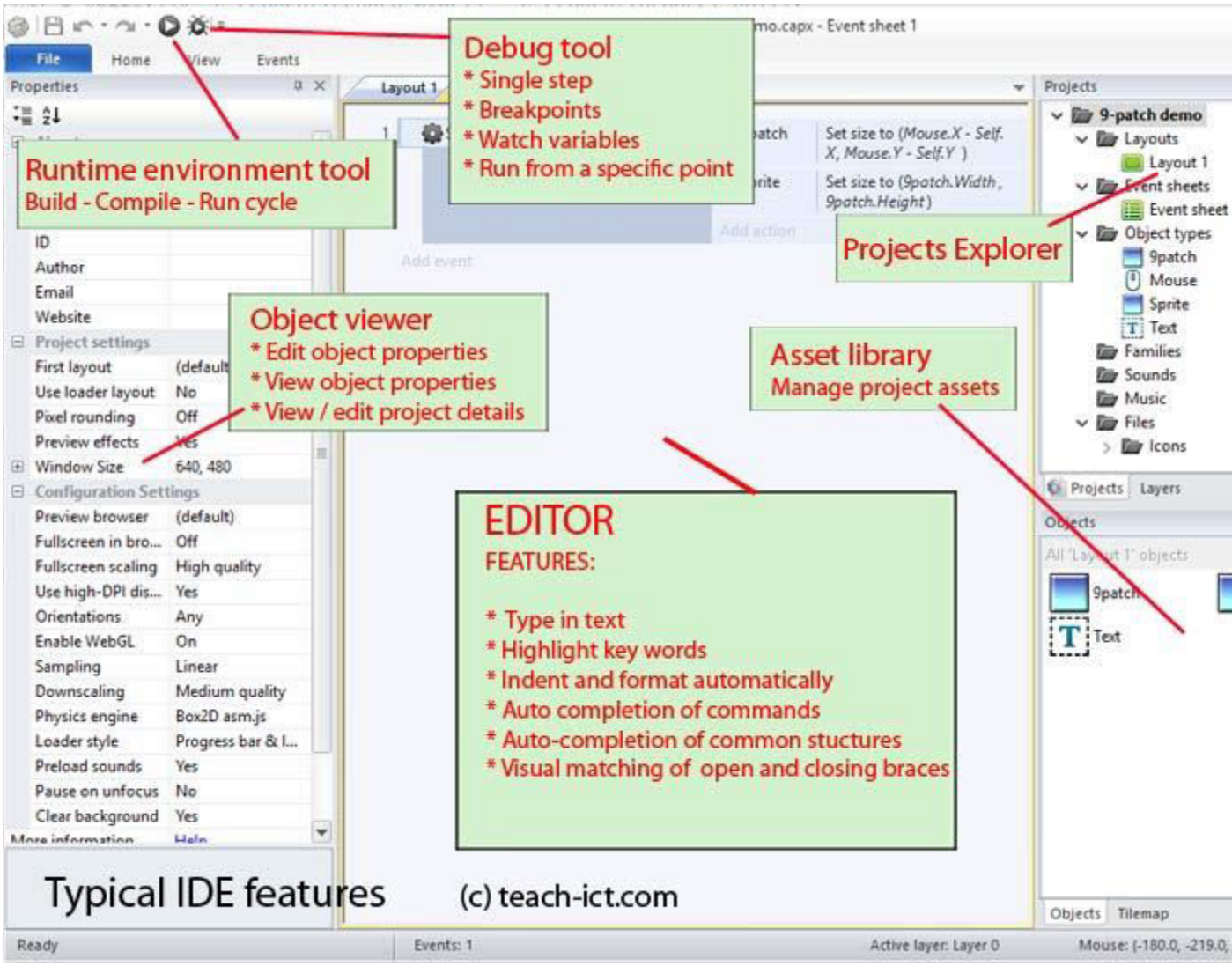
An **IDE** brings all the tools necessary for coding into one application. The tools are designed to work seamlessly together.

Computer languages

9. IDE features

Integrated Development Environment packages are not all the same as each one has some extra features. Some are open source (such as Eclipse) and others are commercial.

Here is a typical IDE and the features it offers



Editor:

Help you to write code efficiently and easily with aids such as colour coded highlighting, auto-completion and brace matching.

Project(s) explorer

A window that shows project files and folders `

Asset library

A window that lists all the assets such as images, sound, media files in the project. It may also allow you to duplicate, edit and delete assets or drag them in from another project

Runtime Environment

This means it allows you to build the project from its files, compile it all and then run it as if the code was running on the target CPU. It will not run as fast as the real thing, but it really helps complete the project faster and easier.

Object viewer

Many languages allow you to create 'objects' visually, and these objects have 'attributes' or properties that you can view and alter from this window. For example some game-making IDEs allow you to create game sprites

Debug tools

These allow you to spot and correct programming errors.

A basic feature of debugging is to be able to single-step through the code. In this way you can see how the variables are changing and how the program flows. To make it a bit more flexible you can choose to 'step-into' a subroutine or you can 'step-over' it if you know it is working.

'Breakpoints' can be placed in the code and when that point is reached in the program as it runs, it stops and allows you to see variable values at that instant. Another tool is 'Watch variables' with this one you can tell the program to stop at the instant a variable has a certain value.

And of course it will highlight any syntax errors that are preventing the program from compiling.

Translator

This is either a built-in compiler or a built-in interpreter depending on the language. The compiler IDE will create the final executable code and also be used within the run-time environment to execute the source code.

Output screen

The runtime environment includes an output screen that lets you see what the program is showing to the user as it runs. This is often another window that pops up outside the main IDE once the runtime starts.

Computer languages

10. Summary

- Every CPU runs binary instructions called machine code.
- Machine code is difficult to produce by hand.

- Assembly language provides mnemonics instead of raw machine code.
- A program written in assembly mnemonics is easier to understand.
- Assembly language is a low level language because each instruction has a machine code equivalent.
- High level languages were developed to make it easier to write programs.
- There are many high level languages - C, Python, Javascript, Fortran.
- A program in a high level language is written as source code.
- Source code is made up of a set of keywords and arguments written down according to the syntax rules of the language.
- A translator converts source code or assembly mnemonics into machine code.
- Compiler and Interpreter are high level language translators.
- A compiler converts all the source code in one pass to create an executable file containing machine code.
- An interpreter converts a single line of source code into machine code and runs it.
- An IDE combines a set of standard development tools into one application.
- The main tools of an IDE are Project Manager, Code editor, Real-Time environment, Debugger and Translator.