

System Software

AS Computer Science ■ Topic 5

A-Level Computer Science

applications

operating system

hardware

What we will cover

5.1 Operating systems: why, and the five management tasks

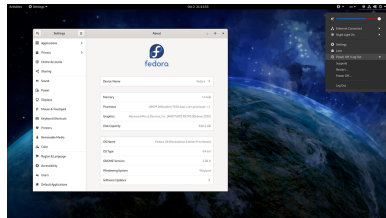
5.1.1 Utility software

5.1.2 Program libraries and DLL files

5.2 Language translators

5.2.1 Partial compilation (Java) and the IDE

Each part of this deck has the number of its exercise sheet and past-paper sheet.





Why a computer needs an OS

The **operating system** is the software layer that:

- manages the hardware for programs
- provides services & a user interface
- lets programs share the hardware **safely**

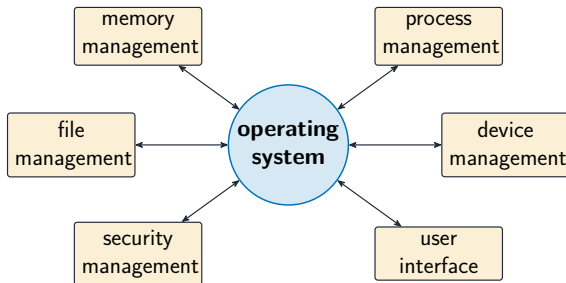


“Describe the purpose of an OS”: the five-mark answer

- 1 it provides an **interface** between the user, the programs and the hardware
- 2 it **hides the complexity** of the hardware
- 3 it **manages the resources**: processor time, memory, storage, input and output
- 4 it **loads and runs** programs, on one platform
- 5 it runs several programs at once (**multitasking**) and keeps them, and the data, **secure**

Five **different** points, one mark each. “It runs the computer” or “it manages resources” on its own earns nothing.

What the OS manages



- **process management** – load, schedule and switch programs
- **memory management** – hand out memory and keep processes apart
- **file management** – folders and permissions
- **device management** – I/O and peripherals
- **security, the user interface, networking**

What each management task actually does

process

a running program is a **process** : load it, schedule it on the CPU, switch between processes, kill the misbehaving

memory

give each process memory and keep them apart; **virtual memory** and **paging** let the working set exceed the physical RAM

file

organise files and folders on **secondary storage** , control permissions, prevent write corruption

device

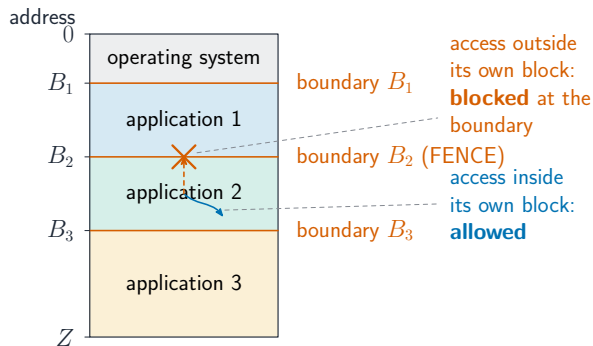
handle I/O through **device drivers** , buffer data, manage interrupts, and give every peripheral a uniform interface

Multitasking: memory and process management together

- 1 memory management loads several programs into memory at once, each in its **own protected area**
- 2 it keeps track of which addresses belong to which program, so one cannot overwrite another
- 3 process management shares the processor: each process runs for a **time slice**, then the OS saves its state and switches to the next
- 4 the switching is so fast that all the programs appear to run together

A four-mark favourite. Name **both** tasks, and say what each one does: two marks for memory, two for the processor.

Memory protection keeps processes apart



Memory protection keeps each application in its own block of memory

Hardware interrupt, or software interrupt?

kind	comes from	examples
hardware interrupt	a device	a key is pressed; the printer is out of paper; a disk transfer has finished; a power failure
software interrupt	a running program	division by zero; an invalid instruction; a program uses memory it does not own; a request for an OS service

Ask: **did a device cause it, or did a program's own instruction cause it?** Hardware management responds to interrupts and passes them to the handler.



Utility software

Antivirus; **backup** software that copies user data elsewhere; compression; a **defragmenter** ; disk analysis and repair for file-system errors and bad sectors; and a **firewall** filtering traffic by rules. Bundling them with the OS saves the user installing each one.

utility programs

antivirus

backup

file
compression

disk
defragmenter

The six utilities the syllabus names

utility	what it does	when it is needed
disk formatter	prepares a disk for use: sets up its file system and partitions, deleting any data	a new disk, or one to be wiped
virus checker	scans files and memory for known virus signatures ; quarantines or deletes what it finds	every download, on a schedule
defragmentation software	moves the scattered blocks of each file so they are stored together	a hard disk that has become slow
disk contents analysis / disk repair	shows what is using the space; finds and repairs bad sectors and file-system errors	a disk that is full, or gives errors
file compression	shrinks a file so it needs less storage and transfers faster	attaching, uploading, archiving
back-up software	copies files to another medium on a schedule, so they can be restored	always: the only recovery from loss

Performance: defragmentation, disk repair, disk analysis, compression. **Protection:** virus checker, backup, formatter.

Which utility would you use?

an old hard disk reads files slowly
– pieces scattered everywhere

→ **disk defragmenter**

an attachment is too big
to email

→ **compression**

one careless delete could
lose the whole project

→ **backup**

a borrowed USB stick
might carry malware

→ **virus checker**

An exam answer names the utility **and** links it to the scenario's need.

Defragmentation for three marks, backup for two

Explain how defragmentation improves performance

- 1 a file is stored in scattered blocks (**fragmented**), so the read/write head moves many times to read it
- 2 the **defragmenter** moves the blocks so each file is **contiguous**, and gathers the free space
- 3 files are read with fewer head movements, so they load faster

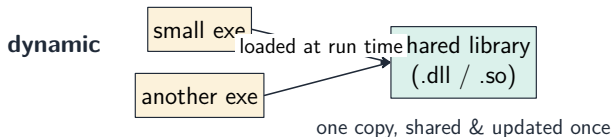
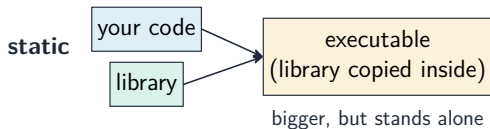
Not needed on an SSD: no moving head.

Explain the need for back-up software

- it copies the data to a **separate medium** on a **schedule**
- so the data can be **restored** after disk failure, deletion, corruption or ransomware
- a **full** backup copies everything; an **incremental backup** only what changed



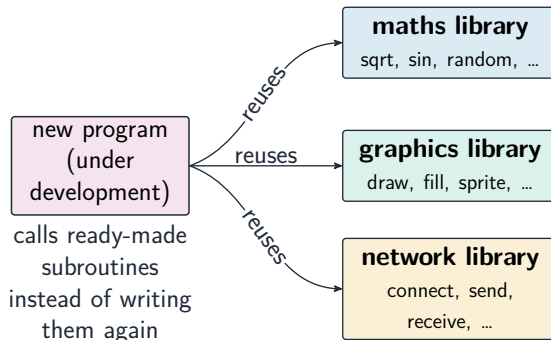
Program libraries



A **program library** is reusable, tested code.

- **static library** : copied into the executable
- **dynamic library** (.dll/.so): loaded at run time, shared

A library is code someone already tested



Reusing ready-made routines **saves time**, is **reliable** (well-tested, widely used) and is **standardised** across every program that calls it.

The benefits to the developer, as the mark scheme lists them

benefit	why
saves time and cost	the routine is already written: nothing to design, write or test
reliable	tested, and used by many programs, so largely error-free
no expertise needed	no need to know how to write encryption, graphics or path-finding
consistent, easy to maintain	every program gets the same result; the code lives in one place, so a fix is made once

Drawbacks: the routine may not do exactly what is needed and cannot be changed; the program depends on the library being available, correct and secure.

DLL files: benefits and drawbacks

A **dynamic library** is loaded at **run time**, when the program calls it, and stays a separate file.

- the executable is **smaller**, and memory is used only while the routine is needed
- one copy is **shared** by every running program that uses it
- a fix or an update is made **once**, without recompiling the programs

Each benefit is a fact **plus its consequence**: two marks. Drawbacks: the program fails if the DLL is **missing** or the wrong version; an update can change behaviour; a fake DLL runs with the program's rights.



The words this topic is built from

machine code

the binary instructions the **processor** actually executes

assembly language

one mnemonic per machine instruction – readable, and still machine-specific

a high-level language

machine-independent, with **subroutines**, data structures and libraries

a compiler

translates the **whole** program once; errors are reported before it runs, and it runs fast

an interpreter

translates and runs **line by line**; errors appear as they are met, and it runs slower

which to use

an interpreter while developing, a compiler for the released product

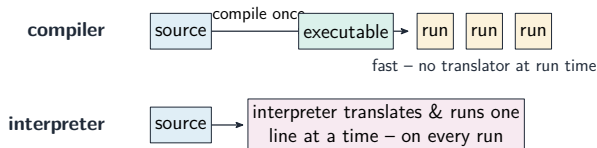
Compilation moves the work **before** running; interpretation moves it **into** running. Everything else about the comparison follows from that.

The need for an assembler

- the processor executes only **machine code**
- **assembly language** writes each instruction as a mnemonic (LDD, ADD, JMP) with labels for addresses
- the **assembler** replaces each mnemonic by its machine-code instruction, **one for one**, and labels by real addresses
- used where the programmer needs direct control of the hardware: embedded systems, device drivers

A compiler must analyse a whole high-level statement and generate several instructions for it. An assembler only **translates one to one**, which is why its job is simpler.

Compiler vs interpreter



- **compiler** : translates **once** into a standalone **executable** . Faster at run time, but tied to **one CPU and OS** – recompile for each platform.
- **interpreter** : line by line, every run. It must be **installed** to run the program and is generally slower, but the same source **ports** anywhere.

The two-mark descriptions, in the examiner's words

A compiler translates the **whole** high-level program into machine code **before** it is run; it produces an **executable** file; it reports **all** the syntax errors together at the end of translation. It does not run the program.

An interpreter translates **one statement** at a time and **executes** it immediately, before moving to the next; no executable file is produced; it stops at the **first** error it meets and reports it.

For “describe the operation of” a translator, give three things: **what** is translated (whole program or one statement), **when** (before running or while running), and **how errors are reported** (all at once or at the first).

Benefits and drawbacks, as the mark scheme lists them

	compiler	interpreter
execution speed	fast: already machine code	slower: translated on every run
what the user needs	only the executable; no translator	the source code and the interpreter
the source code	stays private	must be given to the user
finding errors	all listed together, after translation	reported at the line, as you develop
changing the code	recompile the whole program after every change	edit and run again at once
portability	machine code for one platform; recompile for each	the same source runs wherever an interpreter exists

Every row is one comparison: a benefit of one translator is the drawback of the other.

Choosing between them

Use a compiler when

run-time speed matters
distributing to users without
dev tools
the program runs many times

Use an interpreter when

you want fast edit–run cycles
writing cross-platform scripts
the program is small or run
once
teaching beginners

Neither is “better”. The question is always **how often the program runs** against **how often you change it**.

Interpreter while writing, compiler when finished: four marks

- 1 during development the **interpreter** runs the partly written program at once, with no complete translation
- 2 it stops at an error and reports the line, so the programmer fixes it and runs again immediately
- 3 when the program is finished, the **compiler** translates it once into an executable that runs faster
- 4 the executable needs no translator on the customer's computer, and does not reveal the source code

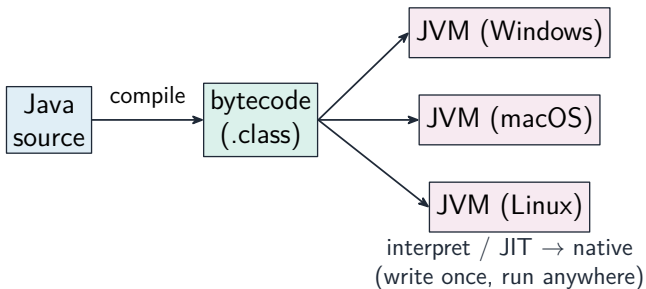
Two marks for how the interpreter is used, two for the compiler. “Interpreter for testing, compiler at the end” with no reasons is one mark at most.



Hybrid: Java bytecode

Java compiles to **bytecode**, run by a **virtual machine** (JVM).

Why both? A compiler alone ties the program to one platform; an interpreter alone is slow. Compiling to **bytecode** does the heavy translation once and catches errors early; the JVM then runs it [anywhere](#), with **JIT** near native speed.



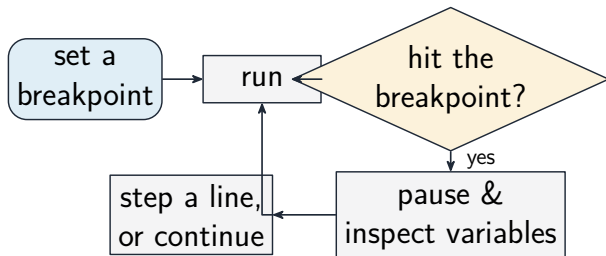
Worked example – why compile to bytecode at all?

Java is compiled to bytecode, which a JVM interprets. Why both, instead of compiling straight to machine code?

- A compiler produces machine code for *one* processor and OS, so the program will not run elsewhere.
- Java targets a **virtual** machine, so the bytecode is identical everywhere and each platform supplies its own JVM – “write once, run anywhere”.
- The price is speed, which is why a real JVM compiles hot paths just in time.

Portability against speed. Naming the trade-off, and the JIT that softens it, is what a full answer needs.

Integrated Development Environment

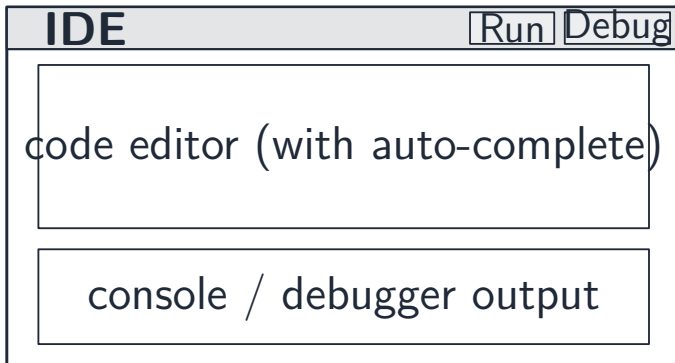


An **IDE** bundles the dev tools:

- editor with **syntax highlighting** , **auto-complete**
- one-key compile/run; inline errors
- **debugger** : **breakpoints** , step, inspect

An IDE, on screen

Editor, Run button and debugger in one window – that is the whole idea.



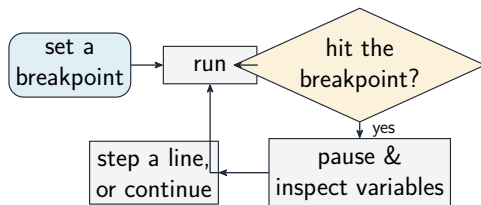
The four feature groups the syllabus names

group	feature	what it does for the programmer
coding	context-sensitive prompts	pops up the identifiers, keywords or parameters that fit here; auto-complete finishes the name
initial error detection	dynamic syntax checks	checks the syntax as you type and underlines a mistake before translation
presentation	prettyprint	keywords, identifiers and comments in colours or fonts, consistent indentation
presentation	expand and collapse code blocks	hides the body of a loop, an IF or a subroutine, so the outline is visible
debugging	breakpoints	the running program pauses at a marked line
debugging	single stepping	from the pause, runs one statement at a time
debugging	variables and expressions window	shows the current values while paused
debugging	report window	lists errors, warnings and output

“One **other** presentation feature” wants the other row of the same group: prettyprint pairs with expand and collapse.

Finding a wrong value with the debugger: four marks

- 1 set a **breakpoint** on the first line of the function, so the program pauses there
- 2 **single-step** through it one statement at a time
- 3 after each step, read the **variables and expressions** window and compare with the expected values
- 4 the first statement after which a value is wrong is where the error is; the **report window** shows any error message and the output so far



Java's hybrid model, and what an IDE gives you

bytecode

Java is compiled once into a **platform-independent** intermediate form

the virtual machine

the JVM then interprets it – so one compiled file runs anywhere a JVM exists

just-in-time compilation

the JVM compiles hot bytecode to native machine code while running, recovering most of the speed

syntax highlighting

keywords, strings and comments in different colours, plus auto-indent and bracket matching

auto-complete

suggests names and shows a function's parameters as you type

and the rest

version control integration, project management, a debugger, **refactoring** tools, and **unit test** runners

An IDE speeds development by putting writing → running → debugging → fixing behind **one interface**. Common ones: Visual Studio, PyCharm, Eclipse.



Exam tips

- name the OS tasks by their syllabus names (memory, file, security, hardware, process) and say what each **does**: “manages resources” alone is too vague
- a utility answer names the utility **and** the need in the scenario; know which ones improve performance and which protect data
- a library or DLL benefit is the developer’s **saving**: time, cost, expertise, reliability, maintenance; a drawback is a **dependence**
- for a translator: **what** is translated, **when**, and **how errors are reported**; never “an interpreter compiles line by line”
- IDE features belong to a syllabus **group**: coding, initial error detection, presentation, debugging

Topic 5 – key takeaways

1

OS

manages CPU, memory, files,
devices, security

2

Libraries

reusable code; static vs dynamic

3

Translators

compiler once vs interpreter line-
by-line

4

IDE

editor, translator & debugger in
one

Check yourself

- 1 Which OS job keeps each program in its own memory?
- 2 Compiler or interpreter: which makes a standalone file?
- 3 Why is Java bytecode portable?
- 4 Name a debugger feature in an IDE.



Answers 1. memory management (protection) 2. compiler 3. any JVM runs it
4. breakpoints / step / inspect