Table of Contents

THE PROGRAMMING WORKBOOK

22nd Survey Division | PTE WU | Cert IV Programming | Vidimus Omnia

---

PART 0: ORIENTATION

How This Workbook Is Built

Every language in this workbook is broken into three tiers:

Tier	Name	Purpose	When to Use
A	**WHY**	Understand the shape of the language. Why things look the way they do. Mental models.	Read first. Read when confused. Read when nothing makes sense at 0200.
B	**SYNTAX**	What to type. Step-by-step, with examples, for every concept. The dumbcunt guide.	Read when building. Read when you know WHAT you want but not HOW to write it.
C	**QUICK REF**	Lookup tables. Every command, operator, shortcut. No	Print it. Laminate it. Look things up mid-exercise.

Tier	Name	Purpose	When to Use
		explanations, just facts.	

How to Use This Under Pressure

1. If you don't understand WHY something looks like that, read **Tier A** for that language.
2. If you understand the concept but can't remember the syntax, read **Tier B**.
3. If you just need to check a command or operator, hit **Tier C**.

Do not read this workbook cover to cover. It is a reference weapon. Use it like one.

> **The golden rule:** Each piece of information lives in ONE place. WHY sections reference SYNTAX sections. SYNTAX sections reference QUICK REF tables. Nothing is duplicated. If you can't find it, check the other tier.

Table of Contents

PART 1: C#

Part 1A: WHY C# LOOKS LIKE THAT

Read this FIRST. Then the syntax guide. Then everything clicks.

> **You can memorise what to type. But if you don't know WHY it looks the way it does, you forget overnight. This section explains the logic behind the shapes. Once the shapes make sense, you stop memorising and start knowing.**

---

1. C# Is Boxes Inside Boxes

Every C# program is boxes nested inside other boxes. That's the entire structure.

Shipping container. Inside it: crates. Inside each crate: boxes. Inside each box: items.

```
SHIPPING CONTAINER
{
    CRATE
    {
        BOX
        {
            ITEM;    <-- actual instruction
            ITEM;    <-- actual instruction
        }
    }
}
```

Each { opens a box. Each } closes it. Everything between them belongs to that box.

> **{ = open box. } = close box. Every { MUST have a matching }.**

▶ If Visual Studio shows a red error you can't find, count your braces. 5 opens and 4 closes = problem. Click on any brace and VS highlights its partner. Use that constantly.

In older C# tutorials you see code wrapped in `namespace`, `class`, and `Main`:

```
namespace MyProject
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello");
        }
    }
}
```

That's SEVEN lines of wrapper for ONE line of actual code. Modern C# (.NET 6+) lets you skip all that:

```
Console.WriteLine("Hello");
```

One line. Same result. The compiler adds the wrapper invisibly.

▶ Your TAFE projects use .NET 8. You get the clean version. If you see old tutorials with all that namespace/class/Main stuff, don't panic – it's the same thing, just with more wrapping paper. Your code goes where `Console.WriteLine` is in both versions.

> **If your teacher shows code with namespace/class/Main, the stuff you write goes INSIDE the innermost { } block. In modern C# that wrapper is invisible and you just write code directly.**

---

2. Why the Code Looks Like a Wave

Code moves right when it goes deeper, and comes back left when it exits. That's the wave.

```
if (age > 18)               // ground level
{                           // opens a box
    Console.WriteLine();    // ONE level deep
    if (hasLicense)         // still one level
    {                       // opens ANOTHER box
        Drive();            // TWO levels deep
    }                       // close inner box, come back
}                           // close outer box, back to ground
```

The wave is depth. Each indent level = one box deeper. Come back out = one level left.

▶ Think of it like walking through doorways in a building. Each { is walking through a door into a room. Each } is walking back out. The deeper you go, the further right your code sits.

---

3. Can You Indent However You Want?

YES. C# does not give a shit about your indentation. This runs perfectly:

```
if(age>18){Console.WriteLine("Adult");if(hasLicense){Drive();}}
```

It compiles. It's correct C#. But try reading it. Try finding a bug in it. You'd rather eat glass.

Same code, properly indented:

```
if (age > 18)
{
    Console.WriteLine("Adult");
    if (hasLicense)
    {
        Drive();
    }
}
```

> **C# doesn't need indentation. YOUR BRAIN does. Python forces correct indentation. C# doesn't. That makes C# harder for beginners because you can write correct code you can't read.**

▶ Keyboard shortcut that saves your life: select all code (Ctrl+A) then Ctrl+K, Ctrl+D. Visual Studio auto-fixes all your indentation. Use it after every change. Memorise it now.

The Rule: Open { = indent the next lines by one tab. Close } = come back one tab. That's it.

---

4. The Python-to-C# Translation

▶ You already know Python. C# is the same logic wearing a suit. Every Python concept has a C# twin. Here's the translation:

Concept	Python	C#
Print text	`print("Hello")`	`Console.WriteLine("Hello");`
Get user input	`input("Enter: ")`	`Console.ReadLine()`
Variable	`x = 5`	`int x = 5;`
String variable	`name = "Wu"`	`string name = "Wu";`
If statement	`if age > 18:`	`if (age > 18) { }`
Else if	`elif age > 13:`	`else if (age > 13) { }`
Else	`else:`	`else { }`
For loop	`for i in range(10):`	`for (int i = 0; i < 10; i++) { }`
For-each loop	`for item in list:`	`foreach (string item in list) { }`
While loop	`while running:`	`while (running) { }`
Comment	`# comment`	`// comment`
Block comment	`"""multi-line"""`	`/* multi-line */`
String format	`f"Name: {name}"`	`$"Name: {name}"`
AND	`and`	`&&`
OR	`or`	`||`
NOT	`not`	`!`
True/False	`True` / `False`	`true` / `false` (lowercase!)
Null/None	`None`	`null`
Array/List	`list = [1,2,3]`	`int[] arr = {1,2,3};`
Length	`len(list)`	`arr.Length`
Read file	`open(f).readlines()`	`File.ReadAllLines(f)`
End of statement	end of line	semicolon `;`
Code blocks	indentation	curly braces `{ }`

Concept	Python	C#
Types required?	No	Yes, always

> **Same ideas. Different spelling. If you can think it in Python, you can write it in C#.**

---

5. Every Structure Has the Same Shape

Here's the secret. `if`, `for`, `foreach`, `while`, `try`, `catch` – they ALL look the same:

```
KEYWORD (condition or setup)
{
    stuff that happens;
}
```

Watch:

```
if (age > 18)                    { Console.WriteLine("Adult"); }
for (int i = 0; i < 10; i++)     { Console.WriteLine(i); }
foreach (string s in list)        { Console.WriteLine(s); }
while (running)                   { Check(); }
try                               { DoRiskyThing(); }
catch (Exception ex)              { HandleError(); }
```

> **KEYWORD ( ) { stuff; } – Every C# structure. The keyword changes. The contents change. The SHAPE never changes. Ever.**

▶ Next time you learn a new C# feature – methods, classes, switch, whatever – it'll fit this exact shape. You're not learning 20 different things. You're learning ONE shape used 20 different ways.

---

6. When Do You Use a Semicolon?

Statements get semicolons. Structure lines don't. That's the whole rule.

A STATEMENT = one instruction = gets a ;

```
Console.WriteLine("Hi");          // DO this = ;
int age = 29;                      // SET this = ;
name = Console.ReadLine();         // GET this = ;
count++;                           // CHANGE this = ;
```

A STRUCTURE = opens a block = NO ;

```
if (age > 18)                      // opens a block = no ;
{                                  // brace = never ;
}                                  // brace = never ;
```

```
for (int i = 0; i < 10; i++)        // opens a block = no ;
foreach (string s in list)          // opens a block = no ;
try                                 // opens a block = no ;
```

Does the next line have a { brace? Then NO semicolon on this line. Is it a single do-this instruction? Semicolon.

The `for` loop has `;` inside its parentheses – those are separators between three pieces:

```
for ( start ;  condition ;  step )
for (int i=0;  i < 10;     i++  )
      ^^^^^^   ^^^^^^^     ^^^^
      part 1;  part 2;     part 3
```

▶ The semicolons inside `for()` are like the | pipes in a table. They separate fields. The line itself doesn't end with `;` because it opens a `{ }` block.

---

7. What the Dot Means

The dot means "go inside this and use that." Every single time.

```
Console.WriteLine()         // Console is a toolbox. Open it. Grab WriteLine.
Math.Max(5, 10)             // Math is a toolbox. Open it. Grab Max.
File.ReadAllLines()         // File is a toolbox. Open it. Grab ReadAllLines.
name.ToLower()              // name is a string. Ask it to go lowercase.
movies.Length               // movies is an array. Ask it how long it is.
```

DOT = "reach inside this thing and grab something." That's all it ever means.

Why do some have `()` and some don't?

```
movies.Length               // no ()  = a VALUE. Just looking at it.
name.ToLower()              // ()     = an ACTION. Doing something.
```

`()` means "do this." No `()` means "just read this."

Length is a fact about the array – just a number sitting there. ToLower() is an action – it transforms the string. Actions get parentheses.

▶ If you forget `()` on a method, the compiler says "did you intend to invoke this?" Translation: "add the brackets, idiot." If you add `()` on a property, it says "non-invocable." Translation: "remove the brackets."

Chaining dots: You can chain actions together left to right:

```
name.ToLower().Contains("wu")
// Step 1: name          = "Wu"
// Step 2: .ToLower()    = "wu"
// Step 3: .Contains()   = true
```

Each dot takes the result of the previous thing and does the next thing to it. Read it left to right like a sentence.

---

8. Why You Have to Say the Type

In Python you write `x = 5` and Python figures out it's a number. In C# you write `int x = 5` and tell it explicitly.

```
// Python:  Python figures it out
x = 5          # Python: "ok that's a number"
x = "hello"    # Python: "ok now it's text" (no problem)

// C#: You declare it up front, locked in
int x = 5;            // this is a number FOREVER
x = "hello";          // ERROR! x is int, can't hold text
```

Why? Speed and safety. When C# knows the type in advance, it catches mistakes BEFORE you run the program. Python lets mistakes through until runtime.

▶ Think of it like labelling boxes in a warehouse. Python lets you shove anything in any box. C# makes you label each box first – "NUMBERS ONLY" – and stops you from putting shoes in the numbers box. More work up front, fewer surprises later.

> **C# is strict. You pick a type, you're stuck with it. int holds numbers. string holds text. You can't swap them without explicitly converting.**

The types you'll actually use:

```
string name = "Wu";         // text
int count = 42;             // whole number
double price = 19.99;       // decimal number
bool alive = true;          // true or false
string[] list = {"a","b"}; // array of strings
int[] nums = {1, 2, 3};     // array of numbers
```

▶ That's 90% of what you'll use in the first semester. Don't stress about the other types yet.

What's the `[]` in `string[]`?

`[]` means "array of." An array is a list.

```
string    = one piece of text
string[]  = a LIST of text pieces
int       = one number
int[]     = a LIST of numbers
```

The `[]` is like adding an 's' to make it plural. `string` = one. `string[]` = many.

For more on declaring vs initialising, and the `var` keyword: –> See Part 1B, Section 16.

For converting between types: –> See Part 1B, Section 17.

---

9. Why Parentheses Everywhere (Conditions vs Arguments)

Parentheses ( ) do TWO different jobs in C#. Knowing which one you're looking at is key:

Job 1: Conditions – wrapping the question

```
if (age > 18)              // the QUESTION goes in ()
while (running)            // the CONDITION goes in ()
for (int i=0; i<10; i++)   // the SETUP goes in ()
```

Python doesn't require these. C# does. Every condition MUST be wrapped in parentheses.

```
// Python:  if age > 18:       (no parens needed)
// C#:      if (age > 18)      (parens REQUIRED)
```

▶ Why? Because C# uses curly braces for blocks, not colons. The ( ) tell the compiler "everything inside here is the condition, everything after is the action." It's how C# separates the question from the answer.

Job 2: Arguments – passing information in

```
Console.WriteLine("Hello")     // passing "Hello" to the method
Math.Max(5, 10)                // passing 5 and 10 to the method
int.Parse("42")                // passing "42" to convert
name.Contains("wu")            // passing "wu" to search for
```

When a method needs information to do its job, you put that information in the parentheses.

```
Console.ReadLine()             // empty () = needs no information
```

▶ Empty ( ) means "do the thing, you don't need any input from me." ReadLine() just waits for keyboard input – it doesn't need you to tell it anything.

> **( ) after a KEYWORD = condition/setup. ( ) after a METHOD NAME = passing data in. No ( ) = it's a property/value, not an action.**

---

10. Double Quotes vs Single Quotes vs Prefixes ($, @)

Double quotes = strings (text)

```
string name = "Wu";          // double quotes = text
Console.WriteLine("Hello"); // double quotes = text
```

Single quotes = single character only

```
char letter = 'A';          // single quotes = ONE character
char letter = 'AB';         // ERROR! Too many characters
```

▶ You'll almost never use single quotes. Stick with double quotes for everything text-related.

$ prefix = interpolation (embed variables)

```
string msg = $"Name: {name}, Age: {age}";
```

$ says "I'm going to embed variables in this string using { }." Without $ the curly braces print as literal text. With $ they get replaced with values.

@ prefix = literal path (ignore backslashes)

```
string path = @"C:\Users\gwu07\file.csv";
```

@ says "don't treat backslashes as special. They're just backslashes." Without @ you'd need to double every backslash: `"C:\\Users\\gwu07\\file.csv"`

Can you combine them?

```
string msg = $@"File at C:\Users\{username}\data.csv";
```

$@ = both interpolation AND literal backslashes. For file paths with variables.

> **" " = text. ' ' = single character. $"" = text with {variables}. @"" = text with literal backslashes. $@"" = both.**

For the full table of backslash escape codes: –> See Quick Ref, Table 15.

---

11. The = vs == Trap

This is the mistake that will burn you multiple times before it sticks:

```
int x = 5;        // SETTING x to 5 (assignment)
if (x == 5)       // ASKING if x is 5 (comparison)

=   means SET. Put this value in this box.
==  means ASK. Is this value equal to that value?

if (x = 5)        // WRONG! You just SET x to 5 inside the if
if (x == 5)       // RIGHT! You're ASKING if x equals 5
```

> **= is a command ("make it so"). == is a question ("is it so?"). Inside `if()`, you're asking a question. Use ==.**

▶ C# actually catches this one – if you write = instead of == inside an if, the compiler usually throws an error. But not always. And in some languages (C, JavaScript) it silently does the wrong thing. Build the habit now.

For the full set of comparison operators: –> See Quick Ref, Table 5.

---

12. How to Read C# Like English

▶ If you can read this section out loud, you can read any C# code. Every line translates to a normal sentence.

Declarations:

```
int age = 29;
```

“Create a whole-number box called age and put 29 in it.”

```
string name = "Wu";
```

“Create a text box called name and put Wu in it.”

```
string[] movies = { "Aliens", "Predator" };
```

“Create a list-of-text box called movies with Aliens and Predator in it.”

Conditions:

```
if (age > 18)
```

“If age is greater than 18...”

```
if (name.ToLower().Contains("wu"))
```

“If name, made lowercase, contains wu...”

```
if (age >= 18 && hasLicense == true)
```

“If age is 18 or more AND hasLicense is true...”

Loops:

```
for (int i = 0; i < movies.Length; i++)
```

“Starting at 0, while i is less than the number of movies, count up by 1 each time.”

```
foreach (string movie in movies)
```

“For each text item, which I’m calling movie, in the movies list...”

Method calls:

```
Console.WriteLine($"Hello {name}");
```

"Go to Console, use WriteLine, and print Hello plus whatever name is."

```
string[] lines = File.ReadAllLines(@"C:\data.csv");
```

"Go to File, use ReadAllLines on that path, and store the result in a list-of-text called lines."

Try-catch:

```
try { File.ReadAllLines("x.csv"); }
catch (FileNotFoundException ex) { Console.WriteLine(ex.Message); }
```

"Try to read the file. If it doesn't exist, catch that specific error and print what went wrong."

> **Every line of C# is an English sentence wearing a costume. Strip the costume and you can read any code.**

Naming conventions – why some words are Capitalised and some not:

Thing	Convention	Example	Why
Classes & Methods	PascalCase	`Console.WriteLine`	They're "big" things
Variables & parameters	camelCase	`myAge`, `firstName`	They're "small" things
Constants	PascalCase or SCREAMING	`MaxSize` or `MAX_SIZE`	They NEVER change
Properties	PascalCase	`.Length`	They belong to classes

> **C# is case-sensitive. `Console` is not `console`. `Math` is not `math`. Spelling AND capitalisation matter.**

---

13. Scope – Where Variables Live and Die

▶ This one catches EVERYONE. A variable only exists inside the { } block where it was created. Outside those braces, it's dead. Gone. The compiler pretends it never existed.

```
if (age > 18)
{
    string status = "Adult";     // born here
    Console.WriteLine(status);    // alive here
}
// status is DEAD here. It doesn't exist anymore.
Console.WriteLine(status);        // ERROR! "status does not exist"
```

The variable "status" was born inside the if block's { }. When the } closes, the variable dies.

A variable lives ONLY inside the { } where it was declared. One brace deeper = fine. Outside those braces = dead.

Fix: declare it BEFORE the block

```
string status = "";                // born OUTSIDE the if

if (age > 18)
{
    status = "Adult";              // changed inside (alive here)
}

Console.WriteLine(status);         // still alive here (born outside)
```

If you need a variable after a block closes, declare it before the block opens.

▶ Think of { } like a room with a door. Anything created inside the room stays in the room. When the door closes, it's gone. If you want to carry it out, create the box outside the room first, then fill it inside.

Loop scope trap:

```
for (int i = 0; i < 10; i++)
{
    Console.WriteLine(i);          // i exists here
}
Console.WriteLine(i);              // ERROR! i is dead here
```

The counter variable `i` was created in the for loop's ( ). It dies when the loop's } closes. If you need `i` after the loop, declare it before:

```
int i;                             // born outside
for (i = 0; i < 10; i++) { }       // used inside
Console.WriteLine(i);              // still alive (i is now 10)
```

---

14. Null – The Invisible Nothing

`null` means "nothing." Not zero. Not empty text. Not false. NOTHING. The box exists but there's absolutely nothing in it.

```
string name = null;     // the box exists, but it's completely empty
string name = "";       // the box has an empty string IN it
int count = 0;          // the box has zero in it
```

These are three different things:

Value	Analogy	What It Means
`null`	An empty shelf. No box.	Variable exists but points to NOTHING

Value	Analogy	What It Means
`""`	A box with nothing inside	Variable holds an empty string (still a string)
`0`	A box with a zero inside	Variable holds the number zero (still a number)

The NullReferenceException:

If you try to USE a null variable, C# explodes:

```
string name = null;
Console.WriteLine(name.ToLower());   // CRASH! NullReferenceException
// You asked null to go lowercase. Null is nothing. Nothing can't do things.
```

Fix: check for null first:

```
if (name != null)
{
    Console.WriteLine(name.ToLower());
}
```

▶ NullReferenceException is C#'s most common runtime crash. It means "you tried to use a dot on something that's null." Every time you see this error, some variable you thought had a value is actually null. Find it, check it, fix it.

> **`null` = nothing exists. `""` = something exists but it's empty. `0` = something exists and it's zero. Three different things.**

---

15. The Complete Mental Model

▶ If you remember nothing else from this document, remember this section.

Every C# program is:

```
1.  DECLARE things          int age = 29;
                            string name = "Wu";

2.  DECIDE things           if (age > 18) { }
                            else { }

3.  REPEAT things           for (...) { }
                            foreach (...) { }

4.  READ/WRITE things       File.ReadAllLines()
                            File.WriteAllLines()

5.  PROTECT things          try { } catch { }
```

```
6.  WAIT                    Console.ReadLine();
```

That's it. Declare, decide, repeat, read/write, protect, wait. Every program you wrote in Week 1 is those six actions in different combinations.

Every C# structure looks like:

```
KEYWORD (setup) { stuff; }
```

Every dot means:

```
THING.action()    or    THING.property
```

Every error says:

```
WHAT went wrong  +  WHERE (line number)
```

C# is Python in a suit. Same brain. More punctuation.

Vidimus Omnia.

Part 1B: C# SYNTAX – THE DUMBCUNT GUIDE

Every C# concept from Week 1. One page per idea. No waffle.

> **C# looks like someone took English and ran it through a paper shredder. It's not. It's the same logic as SQL, Python, or any language – just with more curly braces and semicolons. This guide is your decoder ring.**

---

1. Your First Program

Every C# program starts the same way. You type code in a `.cs` file, hit F5, and Visual Studio compiles it into a `.exe` that runs.

```
Console.WriteLine("Hello World");
```

That's it. One line. It prints text to the screen.

▶ `Console.WriteLine` is C#'s version of Python's `print()`. Same job, different spelling. You'll type it a thousand times.

Console.WriteLine vs Console.ReadLine:

```
Console.WriteLine("I print text OUT to the screen");
Console.ReadLine();  // I wait for the user to type something IN
```

`WriteLine` = output. `ReadLine` = input. That's the pair.

▶ Always put `Console.ReadLine()` as your last line while learning. Without it, the console window flashes and closes before you can read anything. It just waits for you to press Enter.

The Semicolon ;

Every statement in C# ends with a semicolon. It's a full stop. Forget it and the compiler screams at you with a red squiggly line.

```
Console.WriteLine("This works");
Console.WriteLine("This breaks")   // <-- missing ; = ERROR
```

If you get a weird error you can't understand, check if you forgot a semicolon. It's the #1 beginner mistake in C#.

For the full explanation of WHEN to use semicolons: –> See Part 1A, Section 6.

Curly Braces { }

Curly braces group code together. Everything inside { } belongs to that block.

```
if (age > 18)
{
    Console.WriteLine("Adult");   // inside the block
}
```

▶ Python uses indentation to group code. C# uses curly braces. The indentation in C# is just for readability – the braces are what actually matters.

---

2. Variables and Data Types

A variable is a labelled box that holds a value. You declare it by saying what TYPE it is, then what it's called, then what's in it.

```
string name = "Wu";          // text
int age = 29;                // whole number
double price = 19.99;        // decimal number
bool isAdmin = true;         // true or false
const int YEAR = 2026;       // locked, can never change
```

▶ The TYPE goes first. That's C#'s thing. Python doesn't care (`x = 5` works). C# needs to know up front: "is this text? a number? true/false?" You tell it with the type keyword.

The Type Cheat Sheet:

Type	What It Holds	Example	Python Equivalent
`string`	Text (letters, words, sentences)	`"Wu"`	`str`
`int`	Whole numbers (no	`29`	`int`

Type	What It Holds	Example	Python Equivalent
	decimals)		
`double`	Decimal numbers	`19.99`	`float`
`bool`	True or false only	`true`	`bool` (True/False)
`char`	Single character	`'A'`	no direct equivalent
`const`	Any type, but LOCKED forever	`const int X = 5;`	no direct equivalent

–> See Quick Ref, Table 2 for the full variables & types table.

String Interpolation (the $ trick):

Instead of gluing strings together with +, put a $ before the quote and use { } to embed variables:

```
// The old way (concatenation):
Console.WriteLine("Name: " + name + ", Age: " + age);

// The good way (interpolation):
Console.WriteLine($"Name: {name}, Age: {age}");
```

> **$ goes ONCE, BEFORE the opening quote mark. Then every `{variable}` inside gets replaced with the value. No $ inside the string. No spaces inside the braces.**

▶ You WILL forget the $ and see literal `{name}` printed instead of Wu. It happened in class. It'll happen again. When your output looks wrong, check for the $.

@ for File Paths:

Backslashes in C# are escape characters. To use file paths, put @ before the quote:

```
// Wrong -- C# thinks \U and \s are escape codes:
"C:\Users\gwu07\source\test.csv"

// Right -- @ tells C# to treat backslashes as literal:
@"C:\Users\gwu07\source\test.csv"
```

▶ The @ goes OUTSIDE the quotes. Not inside. You made this mistake in class with the survey CSV. Remember: $ for interpolation, @ for paths. You can even combine them: `$@"path\{variable}"`

---

3. Operators

Arithmetic (Maths):

Operator	What It Does	Example	Result
+	Add	`5 + 3`	8
-	Subtract	`5 - 3`	2
*	Multiply	`5 * 3`	15
/	Divide	`10 / 3`	3 (integer division!)
%	Remainder (modulo)	`10 % 3`	1

▶ Integer division catches everyone. `10 / 3 = 3`, not 3.33. Because both numbers are `int`, C# gives you an `int` back. Use `10.0 / 3.0` for the decimal answer (makes them doubles).

Comparison (Returns true or false):

–> See Quick Ref, Table 5 for the full comparison operators table.

> **== is comparison (asking). = is assignment (setting). `if (x = 5)` is WRONG. `if (x == 5)` is RIGHT. This is the second most common beginner mistake after missing semicolons.**

Math Class:

```
Math.Max(13, 20);     // returns 20 (the bigger one)
Math.Abs(-5);         // returns 5 (removes the negative)
Math.Min(13, 20);     // returns 13 (the smaller one)
Math.Round(3.7);      // returns 4
Math.Sqrt(25);        // returns 5 (square root)
```

▶ It's `Math`, not `Maths`. American spelling. You got burned by this in class. C# is American English throughout.

–> See Quick Ref, Table 11 for the full Math class table.

---

4. Arrays

An array is a numbered list of items. All the same type. Fixed size once created.

```
string[] movies = { "Aliens", "Terminator", "Predator", "Matrix" };

// Access by position number (starts at 0, not 1):
Console.WriteLine(movies[0]);    // "Aliens"
Console.WriteLine(movies[1]);    // "Terminator"
Console.WriteLine(movies[3]);    // "Matrix"
```

> **Arrays start at 0. The first item is `[0]`, not `[1]`. This will trip you up. Every. Single. Time.**

▶ If you have 4 items, the last one is `[3]`, not `[4]`. `movies[4]` crashes your program because there is no fifth item. This is called an `IndexOutOfRangeException` and you WILL see it.

Parallel Arrays:

Two arrays where position `[0]` in both refers to the same thing:

```
string[] candy = { "Reeses", "KitKat", "Snickers" };
double[] prices = { 2.50, 1.80, 3.00 };

// candy[0] costs prices[0]
// candy[1] costs prices[1]
// Same index = same item
```

▶ Think of parallel arrays like two columns in a spreadsheet. Row 0 in both is the same product. The index ties them together.

.Length:

`.Length` tells you how many items are in the array:

```
Console.WriteLine(movies.Length);    // 4

// Last item is always [Length - 1]
Console.WriteLine(movies[movies.Length - 1]);    // "Matrix"
```

–> See Quick Ref, Table 3 for the full arrays table.

---

5. Loops

▶ Loops repeat code. That's it. Two types: `for` (when you need the counter number) and `foreach` (when you just want each item).

for Loop:

```
for (int i = 0; i < movies.Length; i++)
{
    Console.WriteLine($"{i + 1}. {movies[i]}");
}
// Output:
// 1. Aliens
// 2. Terminator
// 3. Predator
// 4. Matrix
```

Breaking it down:

```
int i = 0            // start counter at 0
i < movies.Length     // keep going while i is less than 4
i++                   // add 1 to counter each loop
```

▶ `i++` means `i = i + 1`. It's shorthand. The loop runs with i=0, then i=1, then i=2, then i=3, then stops because 4 is NOT less than 4. For the full breakdown of `i++` and other shorthand operators: –> See Part 1B, Section 13.

foreach Loop:

```
foreach (string movie in movies)
{
    Console.WriteLine(movie);
}
```

Same output, simpler syntax. You don't get the index number though.

> **Use FOR when you need the counter (i) – like accessing parallel arrays. Use FOREACH when you just want each item and don't care about position.**

Filtering with if Inside a Loop:

```
for (int i = 0; i < candy.Length; i++)
{
    if (prices[i] < 100)
    {
        Console.WriteLine($"{candy[i]} is affordable");
    }
}
```

▶ This is the bread and butter. Loop through data, check a condition, act on matches. Same pattern in SQL: `SELECT * FROM candy WHERE price < 100`. Same logic, different syntax.

–> See Quick Ref, Table 8 for the full loops table.

---

6. Reading and Writing Files

Reading a File:

```
string[] lines = File.ReadAllLines("data.csv");
```

Opens the file, reads every line into an array. Line 1 = `lines[0]`, line 2 = `lines[1]`, etc.

Splitting a CSV Line:

```
string line = "Wu,0412,Sydney";
string[] fields = line.Split(',');

// fields[0] = "Wu"
```

```
// fields[1] = "0412"
// fields[2] = "Sydney"
```

`Split(',')` chops the string at every comma and gives you an array of pieces.

The Full Pattern:

```
string[] lines = File.ReadAllLines("survey.csv");
foreach (string line in lines)
{
    string[] fields = line.Split(',');
    Console.WriteLine($"Point: {fields[0]}, East: {fields[1]}");
}
```

> **Read all lines –> Loop through each line –> Split at commas –> Access pieces by index. That pattern processes ANY CSV file.**

▶ This is exactly how you parsed your survey data in class. Same pattern works for customer lists, inventory files, log files – anything comma-separated.

Writing a File:

```
// Simple -- dump entire array to file:
File.WriteAllLines("output.txt", candy);

// Advanced -- write line by line with filtering:
using (StreamWriter writer = new StreamWriter("filtered.csv"))
{
    for (int i = 0; i < candy.Length; i++)
    {
        if (prices[i] < 100)
            writer.WriteLine($"{candy[i]},{prices[i]}");
    }
}
```

What's `using`?

`using` automatically closes the file when the block ends. Without it, the file stays locked and other programs can't access it. Always wrap `StreamWriter` in `using`.

For the full explanation of both meanings of `using`: –> See Part 1B, Section 18.

–> See Quick Ref, Table 12 for the full file reading & writing table.

---

7. Try-Catch (Error Handling)

▶ Without try-catch, one error kills your entire program. With it, you catch the error and decide what to do instead.

```
try
{
    // attempt the risky thing
    string[] lines = File.ReadAllLines("data.csv");
}
catch (FileNotFoundException ex)
{
    // runs ONLY if file wasn't found
    Console.WriteLine("File missing: " + ex.Message);
}
catch (Exception ex)
{
    // runs for ANY other error
    Console.WriteLine("Error: " + ex.Message);
}
```

How it works:

1. Code inside `try` runs normally
2. If something breaks, execution JUMPS to the matching `catch`
3. Specific catches go first (`FileNotFoundException`)
4. General catch (`Exception`) goes last as a fallback
5. `ex.Message` gives you the error description as text

Without try-catch, a missing file crashes your program. With it, you get "REEEE!!!!: Could not find file" and the program keeps running.

▶ In class you used "REEEE!!!!" as your error message. That's actually fine – it tells you which catch block fired. In production code you'd use proper messages, but for learning, make them distinctive so you know exactly what happened.

When to Use Try-Catch:

Wrap anything that could fail: file reading, user input conversion, network calls, database queries. If the data comes from outside your code, it can break.

```
// User types "abc" when you expect a number:
try
{
    int age = int.Parse(Console.ReadLine());
}
catch (FormatException)
{
    Console.WriteLine("That's not a number.");
}
```

–> See Quick Ref, Table 13 for the full error handling table.

---

8. String Formatting Tricks

.PadRight() and .PadLeft():

Makes columns line up in console output:

```
"Wu".PadRight(10)      // "Wu        " (padded to 10 chars)
"Wu".PadLeft(10)       // "        Wu" (right-aligned)

// Table-like output:
Console.WriteLine($"| {name.PadRight(15)} | {age.ToString().PadRight(5)} |");
```

.ToLower() and .ToUpper():

For case-insensitive searching:

```
"Wu".ToLower()          // "wu"
"wu".ToUpper()          // "WU"

// Search that ignores case:
if (input.ToLower() == "quit")
```

.Contains():

```
if (movie.ToLower().Contains("alien"))
{
    Console.WriteLine("Found it!");
}
```

string.IsNullOrWhiteSpace():

Checks if a string is empty, null, or just spaces:

```
if (string.IsNullOrWhiteSpace(line)) continue;  // skip blank lines
```

int.Parse() and Convert.ToInt32():

`Console.ReadLine()` always returns a string. To use it as a number, convert it:

```
string input = Console.ReadLine();    // user types "25"
int age = int.Parse(input);            // now it's the number 25
```

▶ If the user types “abc” and you call `int.Parse` on it, your program explodes with `FormatException`. That’s why you wrap it in try-catch.

–> See Quick Ref, Table 9 for the full string operations table.

---

9. If / Else If / Else

```
if (age >= 18)
{
```

```
    Console.WriteLine("Adult");
}
else if (age >= 13)
{
    Console.WriteLine("Teenager");
}
else
{
    Console.WriteLine("Child");
}
```

C# checks top to bottom. First true condition wins. Everything else is skipped.

▶ Same as Python's `if/elif/else`. Just replace `elif` with `else if` and add curly braces.

Combining Conditions:

```
if (age >= 18 && hasLicense == true)    // AND - both must be true
if (age < 13 || isStudent == true)      // OR  - either can be true
if (!isAdmin)                           // NOT - flips true/false
```

–> See Quick Ref, Table 6 for the logical operators table. –> See Quick Ref, Table 7 for the full conditions table.

---

10. Error Messages Decoded

▶ Every C# error message is trying to tell you something specific. Here's the translation of the ones you'll see most.

Error	Translation	Fix
`CS0103: Name does not exist`	You used a variable that doesn't exist or is misspelled	Check spelling. `Maths` –> `Math`
`CS1002: ; expected`	Missing semicolon	Add `;` at end of line
`CS0029: Cannot convert type`	Wrong type assignment (putting text in an int)	Use the right type or convert
`IndexOutOfRangeException`	You accessed `array[5]` but only 4 items exist	Check `.Length`, remember arrays start at 0
`FormatException`	`int.Parse("abc")` – can't convert text to number	Wrap in try-catch, validate input
`FileNotFoundException`	File path is wrong or file doesn't exist	Check path, use `@` before quotes

Error	Translation	Fix
`NullReferenceException`	You tried to use something that's null (empty)	Check if variable is null first
`CS0019: Operator can't be applied`	Using wrong operator for the type	== not = for comparison
Red line under everything	One error above is breaking all below	Fix the FIRST error. Others may vanish.

Every error message has three parts:

```
CS0103  |  The name 'Maths' does not exist  |  Line 7
^^^^^^     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^     ^^^^^^
error      what's wrong                        where
code
```

Step 1: Look at the LINE NUMBER. Go to that line. Step 2: Read the MESSAGE. What's it complaining about? Step 3: Fix the thing at the place.

▶ Don't read the error code (CS0103). Read the English message. "The name Maths does not exist" tells you exactly what's wrong. You spelled something wrong. In this case, it's `Math` not `Maths`.

> **When you get an error: read the line number first, then the message. The line number tells you WHERE. The message tells you WHAT. Fix the what at the where.**
>
> **Fix the FIRST error in the list. Often one error causes a cascade of others. Fix the first one, re-run, and half the others vanish.**

---

11. The C# Pattern

Every exercise you did in Week 1 follows the same skeleton:

```
// 1. DECLARE variables
string name = "Wu";
int age = 29;

// 2. DO something with them
if (age >= 18)
    Console.WriteLine($"{name} is an adult");

// 3. LOOP through collections
foreach (string item in array)
    Console.WriteLine(item);
```

```
// 4. READ/WRITE files
string[] lines = File.ReadAllLines("data.csv");

// 5. HANDLE errors
try { /* risky code */ }
catch (Exception ex) { /* backup plan */ }

// 6. WAIT before closing
Console.ReadLine();
```

▶ That's Week 1. Declare, do, loop, read, catch, wait. Every program you wrote – from "Hello World" to the survey data parser – is just those six steps in different combinations.

Same logic as every other language. Just more curly braces.

---

12. Comments

Comments don't run. They're notes for humans. The compiler pretends they don't exist.

Single-line comment:

```
// This is a comment. C# ignores everything after //

int age = 29; // You can put comments at end of lines too
```

Block comment:

```
/* This is a block comment.
   Everything between the stars is ignored.
   Use for longer explanations. */
```

▶ Use them. Especially while learning. When you write code that works but you're not sure WHY it works, add a comment explaining it. When you come back in a week, you'll thank yourself.

Commenting out code:

You can "disable" code without deleting it by turning it into a comment:

```
// Console.WriteLine("This line won't run");
Console.WriteLine("But this one will");
```

Shortcut: select lines and press **Ctrl+K, Ctrl+C** to comment them out. **Ctrl+K, Ctrl+U** to uncomment.

▶ You did this in class – commented out old exercises so they don't run, but you can still read them. Good habit. Keep doing it.

13. break and continue

break = stop the loop entirely

```
for (int i = 0; i < 100; i++)
{
    if (i == 5)
        break;                      // STOP. Exit the loop NOW.
    Console.WriteLine(i);
}
// Output: 0, 1, 2, 3, 4 (stops before printing 5)
```

continue = skip this one, keep going

```
for (int i = 0; i < 10; i++)
{
    if (i == 5)
        continue;                   // SKIP this iteration. Jump to next.
    Console.WriteLine(i);
}
// Output: 0, 1, 2, 3, 4, 6, 7, 8, 9 (5 is skipped)
```

> **break = emergency exit. Loop is OVER. continue = skip this one item and move on to the next.**

▶ You used `continue` already with `if (string.IsNullOrWhiteSpace(line)) continue;` in the CSV reader. That line says "if this line is blank, skip it and move to the next line." Now you know why it works.

Shorthand operators (i++ and friends):

C# has shorthand for common math operations:

```
i++          means    i = i + 1       // add 1
i--          means    i = i - 1       // subtract 1
i += 5       means    i = i + 5       // add 5
i -= 3       means    i = i - 3       // subtract 3
i *= 2       means    i = i * 2       // multiply by 2
i /= 4       means    i = i / 4       // divide by 4
```

They all do the same thing: take the current value, do maths to it, put the result back.

▶ `i++` is the one you'll see the most. It's in every for loop: `for (int i = 0; i < 10; i++)`. That `i++` is just "add 1 to i each time around the loop." That's all it does.

Why not just write `i = i + 1`? You can. It works fine. `i++` is just shorter. Programmers are lazy. That's the only reason.

`i++` = add one. `i--` = subtract one. `i += n` = add n. They're just shortcuts for maths you'd do anyway.

---

14. Console.Write vs Console.WriteLine

```
Console.Write("Hello ");
Console.Write("World");
// Output: Hello World     (same line)

Console.WriteLine("Hello");
Console.WriteLine("World");
// Output:
// Hello                  (line 1)
// World                  (line 2)
```

`WriteLine` adds a new line (Enter) at the end. `Write` doesn't.

`WriteLine` = print + press Enter. `Write` = print + stay on same line.

▶ Use `WriteLine` for most things. Use `Write` when you want to build up a line piece by piece, like a progress bar or a prompt that stays on the same line as the user's input.

---

15. The "new" Keyword

You've seen this:

```
using (StreamWriter writer = new StreamWriter("file.csv"))
```

What's new?

new creates a fresh instance of something. Think of it like ordering from a factory:

```
new StreamWriter("file.csv")
^^^                          "Build me a new one"
    ^^^^^^^^^^^^             "specifically a StreamWriter"
                ("file.csv") "pointed at this file"
```

`StreamWriter` is a blueprint. `new StreamWriter()` builds a real one from that blueprint.

▶ You don't use new for basic types like `int`, `string`, `bool`. Those are simple enough that C# handles them directly. new is for complex tools – file writers, database connections, lists. Think of simple types as picking up a pen. new is ordering a power tool from the catalogue.

new = "build me a fresh one of these." You'll see it more in Week 2+. For now just know it creates something.

---

16. Declaring vs Initialising (and var)

```
int age;                // DECLARING: "I need an int box called age"
age = 29;               // INITIALISING: "Put 29 in the age box"

int age = 29;           // BOTH AT ONCE: "Make the box and fill it"
```

Declaring = creating the box. **Initialising** = putting something in it. You can do them separately or together.

Why would you declare without initialising?

Sometimes you don't know the value yet:

```
string result;                  // make the box

if (score > 50)
    result = "Pass";            // fill it based on condition
else
    result = "Fail";

Console.WriteLine(result);      // use it after
```

▶ If you try to use a variable you declared but never gave a value to, C# throws "use of unassigned local variable." It's protecting you from reading an empty box. Give it a value before using it, or set it to something default when you declare it: `string result = "";`

var – let C# figure out the type:

Instead of writing the type yourself, you can use `var` and let C# guess:

```
var name = "Wu";            // C# sees the string, knows it's string
var age = 29;               // C# sees the number, knows it's int
var price = 19.99;          // C# sees the decimal, knows it's double
```

`var` only works if you initialise at the same time (C# needs a value to guess from).

```
var x;                      // ERROR! C# can't guess the type from nothing
```

▶ `var` is lazy shorthand. The variable still HAS a type – C# just figures it out for you. Some teachers prefer you write the explicit type while learning. Either way works.

---

17. Converting Between Types

▶ `Console.ReadLine()` ALWAYS returns a string. Even if the user types 42, you get the TEXT "42", not the NUMBER 42. You have to convert it.

String to Number:

```
string input = Console.ReadLine();   // user types "42"
int age = int.Parse(input);            // now it's the number 42
double price = double.Parse(input);    // for decimals
```

If the user types "abc" and you call `int.Parse` on it – CRASH. `FormatException`. Always wrap in try-catch.

Number to String:

```
int age = 29;
string text = age.ToString();          // now it's the text "29"
```

Why would you do this? When you need to use `.PadRight` or concatenation or anything string-specific on a number.

Double to Int (losing decimals):

```
double price = 19.99;
int rounded = (int)price;              // 19 (CHOPS the decimals off)
int proper = (int)Math.Round(price);   // 20 (rounds first, then converts)
```

`(int)` is called a "cast." It forces the value into a different type. Casting a double to int CHOPS the decimal – it doesn't round. Use `Math.Round` first if you want proper rounding.

> **`ReadLine` returns string. `int.Parse()` makes it a number. `.ToString()` makes it text again. `(int)` chops a double into a whole number. Always try-catch user input conversions.**

–> See Quick Ref, Table 10 for the full type conversion table.

---

18. "using" Keyword (Imports vs Auto-Close)

"using" at the top of files – imports:

Sometimes you see lines like this at the very top of a `.cs` file:

```
using System;
using System.IO;
```

These are IMPORTS. They tell C# which toolboxes you want to use.

```
using System;       // gives you Console, Math, Convert, etc.
using System.IO;    // gives you File, StreamWriter, StreamReader
```

Think of it like bringing tools to a job site. `using System` means "bring the basic toolkit." `using System.IO` means "bring the file-handling toolkit."

▶ In .NET 6+ (your TAFE projects), most common ones are imported automatically. You probably won't need to add them yourself. But if the compiler says "File does not exist" or

"StreamWriter does not exist" and you know you spelled it right – you're probably missing a `using` line at the top.

> **"using" at the top = importing a toolkit. Modern .NET does this automatically for common ones. If something should work but doesn't exist, add the using line.**

"using" in the middle of code – auto-close:

```
using (StreamWriter w = new StreamWriter("f"))  // auto-close when done
{
    w.WriteLine("data");
}
```

This `using` automatically closes the resource (file, connection) when the block ends. Without it, the file stays locked.

Don't confuse the two:

```
using System.IO;                                // TOP OF FILE: importing a t
oolkit

using (StreamWriter w = new StreamWriter("f"))  // IN CODE: auto-close when d
one
{
    w.WriteLine("data");
}
```

Same word, two different jobs. At the top of the file = import. In the middle of code with ( ) = auto-close a resource. Context tells you which one.

▶ Yeah, C# reuses the word `using` for two completely different things. It's confusing. Just remember: top of file = import tools. Inside code with parentheses = auto-close when done.

Same logic as every other language. Just more curly braces.

Vidimus Omnia.

Part 1C: C# QUICK REFERENCE

Every command from Week 1. Print this. Laminate it. Keep it next to your keyboard.

Table 1: Output & Input

Code	What It Does	Notes
`Console.WriteLine("Hello");`	Print text + new line	Most common output

Code	What It Does	Notes
		method
`Console.Write("Hello");`	Print text, stay on same line	No Enter at end
`Console.ReadLine();`	Wait for user to type + press Enter	ALWAYS returns a string, even if they type a number
`Console.WriteLine($"Hi {name}");`	Print with variable embedded	$ before the quote activates { } interpolation
`Console.WriteLine("A: " + x);`	Print by joining strings with +	Concatenation – works but $ is cleaner
`Console.WriteLine($"{i + 1}. {name}");`	Print with maths inside { }	You can do maths inside the braces

Table 2: Variables & Types

Code	What It Does	Notes
`string name = "Wu";`	Create a text variable	Double quotes always
`int age = 29;`	Create a whole number	No decimals allowed
`double price = 19.99;`	Create a decimal number	For money, measurements
`bool alive = true;`	Create true/false	Lowercase `true`/`false` in C#
`char letter = 'A';`	Create single character	Single quotes, one char only
`const int YEAR = 2026;`	Create a constant (locked)	Cannot change after creation
`var x = 5;`	Let C# figure out the type	Only works if you assign a value immediately
`string name;`	Declare without a value	Must assign before using or compiler error
`int age = 0;`	Declare with default value	Safe – always has a value

Table 3: Arrays

Code	What It Does	Notes
`string[] movies = {"A","B","C"};`	Create an array (list)	`[]` = array of. Fixed size once created.
`movies[0]`	Access first item	Arrays start at 0, NOT 1
`movies[movies.Length - 1]`	Access last item	Last item is always Length minus 1
`movies.Length`	How many items in array	No `()` – it's a property not a method
`int[] nums = {1, 2, 3};`	Array of numbers	All items must be same type
`string[] lines = File.ReadAllLines("f");`	File lines into an array	Each line becomes one array item
`movies[5]`	Access item 6	If only 4 items exist = `IndexOutOfRangeException` CRASH

Table 4: Arithmetic Operators

Code	What It Does	Notes
`5 + 3`	Add –> 8	Also joins strings: `"Hi" + " " + "Wu"` = `"Hi Wu"`
`5 - 3`	Subtract –> 2	
`5 * 3`	Multiply –> 15	Asterisk, not x
`10 / 3`	Divide –> 3 (not 3.33!)	int / int = int (chops decimal). Use `10.0/3.0` for 3.33
`10 % 3`	Remainder –> 1	Modulo. 10 divided by 3 = 3 remainder 1
`i++`	Add 1 to i	Same as `i = i + 1`
`i--`	Subtract 1 from i	Same as `i = i - 1`
`i += 5`	Add 5 to i	Same as `i = i + 5`
`i -= 3`	Subtract 3 from i	Same as `i = i - 3`
`i *= 2`	Multiply i by 2	Same as `i = i * 2`
`i /= 4`	Divide i by 4	Same as `i = i / 4`

Table 5: Comparison Operators

Code	What It Does	Notes
`x == 5`	Is x equal to 5?	TWO equals signs = asking. ONE = setting.
`x != 5`	Is x NOT equal to 5?	
`x < 5`	Is x less than 5?	
`x > 5`	Is x greater than 5?	
`x <= 5`	Is x less than or equal to 5?	
`x >= 5`	Is x greater than or equal to 5?	

Table 6: Logical Operators

Code	What It Does	Notes
`a && b`	AND – both must be true	Python: `and`
`a || b`	OR – either can be true	Python: `or`
`!a`	NOT – flips true to false	Python: `not`

Table 7: Conditions

Code	What It Does	Notes
`if (condition) { }`	Run code only if true	Parentheses around condition REQUIRED
`else if (condition) { }`	Check next condition	Only checked if all above were false
`else { }`	Run if nothing above was true	No condition – it's the fallback
`if (x > 5 && x < 10) { }`	Both conditions must be true	AND logic
`if (x < 5 || x > 10) { }`	Either condition can be true	OR logic
`if (!isAdmin) { }`	If isAdmin is false	NOT flips the boolean

Table 8: Loops

Code	What It Does	Notes
`for (int i = 0; i < 10; i++) { }`	Loop with counter	i starts at 0, runs while < 10, adds 1 each time

Code	What It Does	Notes
`foreach (string s in list) { }`	Loop through each item	No counter – just gives you each item
`while (condition) { }`	Loop while true	Careful: infinite loop if condition never becomes false
`break;`	Exit the loop entirely	Emergency stop – loop is OVER
`continue;`	Skip this iteration, next item	Jump to next loop cycle
`for (int i = 0; i < arr.Length; i++)`	Loop through array by index	Use when you need the position number (i)
`foreach (string line in lines)`	Loop through array by value	Use when you just want each item

Table 9: String Operations

Code	What It Does	Notes
`$"Hello {name}"`	Interpolation – embed variables	$ BEFORE the opening quote, variables in { }
`@"C:\Users\file"`	Literal string – backslashes as-is	@ BEFORE the opening quote
`$@"C:\{user}\file"`	Both interpolation AND literal	For file paths with variables
`line.Split(',')`	Chop string at every comma	Returns an array of pieces
`name.ToLower()`	Make all lowercase	`"Wu"` –> `"wu"`
`name.ToUpper()`	Make all uppercase	`"Wu"` –> `"WU"`
`name.Contains("wu")`	Does it contain this text?	Returns true or false. Case-sensitive!
`name.ToLower().Contains("wu")`	Case-insensitive search	Chain: lowercase first, then search
`"Wu".PadRight(10)`	Pad to 10 chars with spaces	`"Wu        "` – for column alignment
`"Wu".PadLeft(10)`	Right-align to 10 chars	`"        Wu"`
`name.Length`	Number of characters	Property, no `()`
`name.Trim()`	Remove spaces from both ends	`"  Wu  "` –> `"Wu"`
`name.Replace("a"`	Replace all a with b	Returns new string

Code	What It Does	Notes
`, "b")`		(doesn't modify original)
`string.IsNullOrWhiteSpace(s)`	Is it null, empty, or just spaces?	Static method – use for blank line checks
`name.ToString()`	Convert to string	Every type has this

Table 10: Type Conversion

Code	What It Does	Notes
`int.Parse("42")`	String to int	Crashes with `FormatException` if not a number
`double.Parse("19.99")`	String to double	Crashes if not a valid decimal
`age.ToString()`	Number to string	29 –> `"29"`
`(int)19.99`	Double to int (chop decimals)	19.99 –> 19 (does NOT round – just cuts)
`(int)Math.Round(19.99)`	Double to int (proper rounding)	19.99 –> 20 (rounds first, then converts)
`Convert.ToInt32("42")`	String to int (alternative)	Handles null better than `int.Parse`

Table 11: Math Class

Code	What It Does	Notes
`Math.Max(5, 10)`	Returns the larger –> 10	Not Maths. American spelling.
`Math.Min(5, 10)`	Returns the smaller –> 5	
`Math.Abs(-5)`	Removes negative –> 5	Absolute value
`Math.Round(3.7)`	Rounds –> 4	Banker's rounding (2.5 rounds to 2, 3.5 rounds to 4)
`Math.Sqrt(25)`	Square root –> 5	Returns a double
`Math.Pow(2, 3)`	Power –> 8	2 to the power of 3

Table 12: File Reading & Writing

Code	What It Does	Notes

Code	What It Does	Notes
`File.ReadAllLines("data.csv")`	Read file into string array	Each line = one array item
`File.ReadAllText("data.txt")`	Read entire file as one string	Good for small files
`File.WriteAllLines("out.txt", arr)`	Write array to file	Each array item = one line
`File.WriteAllText("out.txt", str)`	Write string to file	Overwrites if file exists
`new StreamWriter("out.csv")`	Create file writer (line by line)	Wrap in `using()` to auto-close
`writer.WriteLine($"{a},{b}")`	Write one line to file	Inside a `using(StreamWriter)` block
`File.Exists("data.csv")`	Does the file exist?	Returns true/false – check before reading

Table 13: Error Handling (Try-Catch)

Code	What It Does	Notes
`try { }`	Attempt risky code	If it fails, jumps to catch
`catch (FileNotFoundException ex) { }`	Catch specific error	Only runs if file not found
`catch (FormatException ex) { }`	Catch format error	Only runs if type conversion fails
`catch (IndexOutOfRangeException ex) { }`	Catch array bounds error	Only runs if array index invalid
`catch (Exception ex) { }`	Catch ANY error	General fallback – put LAST
`ex.Message`	Get the error description	Human-readable error text

Table 14: Keywords & Special Words

Code	What It Does	Notes
`null`	Nothing. Variable points nowhere.	Not 0, not "", not false. NOTHING.
`new`	Create a fresh instance of something	`new StreamWriter()`, `new List<string>()`
`using System.IO;`	Import a toolkit (top of file)	Gives access to File, StreamWriter etc
`using (resource) { }`	Auto-close when done (in code)	For files, connections – auto-cleanup
`const`	Lock a variable forever	`const int X = 5;` – can never change
`var`	Let C# guess the type	Only works with immediate assignment
`true` / `false`	Boolean values	Lowercase in C# (Python uses `True/False`)
`return`	Exit a method and give back a value	You'll use this more in Week 2+

Table 15: Escape Characters

Code	What It Does	Notes
`\n`	New line (Enter)	`"Line1\nLine2"` prints on two lines
`\t`	Tab (indent)	`"A\tB"` = `"A  B"`
`\"`	Literal quote mark	`"She said \"hi\""` prints: She said "hi"
`\\`	Literal backslash	Without `@`, you need `\\` for one `\`

Table 16: Structure Rules

Rule	Example	Why
Every { must have a matching }	`if () { }`	Compiler error if mismatched
Statements end with ;	`Console.WriteLine("Hi");`	Semicolon = full stop
Structure lines do NOT	`if (age > 18)`	The { block follows

Rule	Example	Why
end with `;`		instead
Conditions need `( )`	`if (x == 5)`	Required in C#, optional in Python
Arrays start at 0	`arr[0]` = first item	NOT `arr[1]`
C# is case-sensitive	`Console` is not `console`	Spelling + capitalisation matter
= sets, == asks	`x = 5` vs `x == 5`	Assignment vs comparison
Variables die at their }	`{ int x = 5; }` // x dead	Scope: declare outside if needed after
`for()` has 3 parts separated by `;`	`for (start; condition; step)`	Those `;` are separators, not endings
Indent after every {	`if { \n code`	For YOUR brain, not the compiler
Fix the FIRST error	Ignore cascade errors	One error often causes 10 more

Table 17: Visual Studio Shortcuts

Shortcut	What It Does	When
F5	Run program (with debugging)	Testing your code
Ctrl+F5	Run without debugging	Just want to see output
Ctrl+A, Ctrl+K, Ctrl+D	Auto-fix all indentation	After EVERY edit
Ctrl+K, Ctrl+C	Comment out selected lines	Disable code temporarily
Ctrl+K, Ctrl+U	Uncomment selected lines	Re-enable code
Ctrl+Z	Undo	When you break something
Ctrl+S	Save	Constantly
Ctrl+. (dot)	Quick fix / suggestions	Red squiggly line? Try this
Click a { or }	Highlights matching brace	Finding missing braces
F12	Go to definition	See what a

Shortcut	What It Does	When
		method does
Hover over error	Shows error message	Understanding what's wrong

---

KEYWORD (setup) { stuff; } – That's every C# structure. Same shape. Every time.

Vidimus Omnia.

PART 2: HTML / CSS / JAVASCRIPT

22nd Survey Division | PTE WU | Gelos Task Manager | Vidimus Omnia

---

PART 2A: WHY HTML/CSS/JS LOOK LIKE THAT

Three languages, one page, zero mystery. A web page is three things working together. HTML is the skeleton. CSS is the skin and clothes. JavaScript is the muscles and brain. You need all three but they each do ONE job.

---

1. HTML Is Just Labelled Boxes

Every HTML element is a box with a label on it. The label tells the browser what kind of content is inside.

```
<h1>This is a heading</h1>
<p>This is a paragraph</p>
<button>Click me</button>
```

The `<h1>` label says "this is a heading." The `</h1>` says "heading ends here." Everything between them is the heading content.

> `< >` = opening label. `</ >` = closing label. Everything between them = the content. That's ALL HTML is.

Why do tags come in pairs?

Because the browser needs to know WHERE your content starts AND where it ends. Without the closing tag, the browser doesn't know where to stop. It'll swallow everything after it into the same element.

```
<p>This paragraph starts here and ends here.</p>
```

```
<p>This paragraph starts here
   and keeps going
   until the browser finds this:</p>
```

> It's like not closing a radio transmission. "Bravo team moving to" ... to where? Close the transmission. Over. The `</p>` is your "over."

Self-closing tags

A few tags don't have content inside them, so they don't need a closing tag:

```
<img src="photo.jpg">          <!-- an image, no content inside -->
<br>                           <!-- a line break, nothing inside -->
<input type="text">            <!-- an input box, nothing inside -->
<hr>                           <!-- a horizontal line, nothing inside -->
```

These are empty elements. They DO something (show an image, make a break) but they don't wrap around content.

---

2. Nesting – Boxes Inside Boxes

HTML nests elements inside other elements. Just like C# nests code inside braces.

```
<div>                           <!-- outer box -->
  <h1>Title</h1>               <!-- box inside the outer box -->
  <p>Some text here</p>        <!-- another box inside -->
</div>                          <!-- close outer box -->
```

The `<div>` is a generic container. It holds other elements. Think of it as a shipping container that holds crates.

> C# nests with { }. HTML nests with `<tag></tag>`. Same concept. Different punctuation.

Why does indentation matter?

Same answer as C#: it doesn't matter to the computer. The browser doesn't care. YOUR BRAIN cares.

```
<!-- This works but is unreadable: -->
<div><h1>Title</h1><p>Text</p><ul><li>Item</li></ul></div>

<!-- Same thing, readable: -->
<div>
  <h1>Title</h1>
  <p>Text</p>
  <ul>
    <li>Item</li>
```

```
  </ul>
</div>
```

> Every time you open a tag that contains other tags, indent the contents. When you close it, come back out. Same rule as C#. Same reason.

---

3. The HTML Skeleton – Every Page Has This

Every single web page in the world has this structure:

```
<!DOCTYPE html>              <!-- tells browser: this is HTML5 -->
<html>                       <!-- the entire page -->
  <head>                     <!-- invisible setup stuff -->
    <title>Page Title</title><!-- browser tab name -->
    <link rel="stylesheet" href="style.css">
  </head>
  <body>                     <!-- everything you SEE -->
    <h1>Hello</h1>
    <p>Content goes here</p>
    <script src="script.js"></script>
  </body>
</html>
```

> Think of a body. The `<head>` is the brain – it controls settings, loads resources, but you don't see it. The `<body>` is everything visible. The `<html>` tag is the skin holding it all together. The `DOCTYPE` is the birth certificate – tells the system what it's dealing with.

What goes in `<head>`?

Invisible setup: `<title>` (browser tab name), `<link>` (connects CSS file), `<meta>` (page settings like character encoding and viewport), `<script>` (can go here but better at bottom of body).

What goes in `<body>`?

Everything the user sees and interacts with. Text, images, buttons, forms, links, tables, videos. All of it. Plus the `<script>` tag at the very bottom.

> `<head>` = invisible setup (title, CSS links, meta info). `<body>` = visible content. Everything you build goes in `<body>`.

---

4. Attributes – What Are Those Things Inside the Tag?

Tags can have extra information inside them called attributes:

```
<img src="photo.jpg" alt="A photo">
     ^^^^^^^^^^^^^^  ^^^^^^^^^^^
     attribute 1     attribute 2

<a href="https://google.com">Click here</a>
   ^^^^^^^^^^^^^^^^^^^^^^^
   attribute (the URL)

<input type="text" id="taskName" placeholder="Enter task">
       ^^^^^^^^^  ^^^^^^^^^^^^  ^^^^^^^^^^^^^^^^^^^^^^^
       what kind  unique ID     hint text
```

Attributes are always in the OPENING tag, never the closing tag. They follow the pattern: `name="value"`.

> Attributes are like specifications on a survey mark. The mark (tag) exists, but the attributes tell you what kind it is, where it goes, and how to identify it. PM 12345 isn't just a mark – it's a Permanent Mark with ID 12345 at specific coordinates. Same concept.
>
> Attributes go INSIDE the opening < > tag. Format: `name="value"`. They add extra info about the element.

For the full list of common attributes, see Part 2C Quick Ref Table: HTML Common Attributes.

---

5. ID vs Class – Service Number vs Rank

This trips up every beginner. Here's the rule:

```
<div id="mainHeader">       <!-- UNIQUE. Only ONE element can have this ID -->
<div class="task-item">     <!-- SHARED. Many elements can have this class -->

<!-- WRONG: -->
<div id="task">Task 1</div>
<div id="task">Task 2</div>  <!-- ERROR! Two elements same ID -->

<!-- RIGHT: -->
<div class="task">Task 1</div>
<div class="task">Task 2</div>  <!-- Fine! Class is shared -->
```

> ID = service number. Every soldier has ONE unique service number. No two soldiers share it. CLASS = rank. Many soldiers share the rank "Private." You target an individual with their service number. You target a group with their rank.

Why does this matter?

Because CSS and JavaScript use IDs and classes to find elements:

```
/* CSS: */
#mainHeader { color: black; }     /* # = find by ID */
.task-item { border: 1px solid; }  /* . = find by class */

// JavaScript:
document.getElementById("mainHeader")           // find ONE by ID
document.querySelectorAll(".task-item")          // find ALL by class
```

The # and . prefixes are how CSS and JS tell the difference. # means ID. . means class.

> ID = unique, one per page, for targeting ONE specific element. CLASS = shared, for targeting GROUPS of similar elements.

---

6. Semantic Tags – Why Use <header> Instead of ?

You could build an entire website with nothing but tags. It would work. But it would be like labelling every evidence bag "THING."

```
<!-- Bad (but works): -->
<div>Header stuff</div>
<div>Navigation links</div>
<div>Main content</div>
<div>Footer stuff</div>

<!-- Good (tells you what things ARE): -->
<header>Header stuff</header>
<nav>Navigation links</nav>
<main>Main content</main>
<footer>Footer stuff</footer>
```

Both versions look identical in the browser. The difference is meaning. Semantic tags tell the browser, screen readers, and search engines what each section IS.

> It's like the difference between a blank plan and a plan with proper notation. Both show the same boundaries. But the properly notated one tells anyone reading it what each element means. Title block, north point, scale bar, boundary marks – each has its own symbol for a reason.

For the full list of semantic tags, see Part 2C Quick Ref Table: HTML Semantic Structure.

---

7. Forms – Getting Data from the User

Your Gelos Task Manager needs to collect task information. That's a form. Each input element collects one piece of data. The label tells the user what to type. The button triggers an action.

```
<form>
  <label for="taskName">Task Name:</label>
  <input type="text" id="taskName" placeholder="Enter task">

  <label for="priority">Priority:</label>
  <select id="priority">
    <option value="low">Low</option>
    <option value="medium">Medium</option>
    <option value="high">High</option>
  </select>

  <button type="button" id="addTask">Add Task</button>
</form>
```

> Think of it as field notes. Each field has a label ("Bearing:") and a space to write (the input). The button is "SUBMIT" – it processes the data. Your Gelos form collects task name, priority, due date, and description.
>
> CRITICAL: Use `type="button"` on buttons in JS-handled forms. `type="submit"` reloads the page and you lose everything.

For the full list of input types and form syntax, see Part 2B, Section 5 and Part 2C Quick Ref Table: HTML Form Inputs.

---

8. CSS Is Instructions for How Things Look

HTML says WHAT is on the page. CSS says HOW it looks. Two separate jobs.

```
/* CSS rule: */
h1 {
  color: black;
  font-size: 24px;
  text-align: center;
}

/* Read it as: */
/* "All h1 elements should be black, 24px, centred" */
```

Every CSS rule has the same shape:

```
SELECTOR {
  property: value;
  property: value;
}
```

> SELECTOR = what to style. PROPERTY = what to change. VALUE = change it to what. That's every CSS rule ever.

It's `KEYWORD { stuff; }` again. Same shape as C#. The selector picks the target. The properties are the instructions. The values are the settings. You've seen this pattern before.

Where does CSS go?

Three places, from worst to best:

```
<!-- 1. INLINE (worst -- messy, hard to maintain): -->
<h1 style="color: black; font-size: 24px;">Title</h1>

<!-- 2. INTERNAL (ok for learning -- in <head>): -->
<style>
  h1 { color: black; font-size: 24px; }
</style>

<!-- 3. EXTERNAL (best -- separate file): -->
<link rel="stylesheet" href="style.css">
```

For Gelos, use external CSS. One file for structure (HTML), one file for looks (CSS). Separation of concerns. Same reason you keep field notes separate from the plan – different jobs, different documents.

---

9. How CSS Finds Elements – Selectors

The selector is how CSS targets the elements it wants to style. Three main types:

```
/* 1. TAG selector -- targets ALL elements of that type */
p { color: black; }           /* every <p> on the page */
h1 { font-size: 32px; }       /* every <h1> on the page */

/* 2. CLASS selector -- targets elements with that class */
.task-item { border: 1px solid; }  /* anything with class="task-item" */
.urgent { color: red; }            /* anything with class="urgent" */

/* 3. ID selector -- targets ONE specific element */
#mainHeader { background: grey; }  /* the element with id="mainHeader" */
#taskList { padding: 20px; }       /* the element with id="taskList" */
```

Tag = no prefix. Class = dot prefix. ID = hash prefix. Tag targets all. Class targets groups. ID targets one.

It's like addressing a radio message. "ALL CALL" = tag selector (everyone hears it). "ALPHA SECTION" = class selector (just that group). "BRAVO 2-1 ACTUAL" = ID selector (one specific person).

Combining selectors:

```css
/* Element with a class: */
p.urgent { color: red; }         /* only <p> with class urgent */

/* Element inside another: */
#taskList .task-item { margin: 5px; }  /* .task-item INSIDE #taskList */

/* Multiple selectors, same style: */
h1, h2, h3 { font-family: Arial; }    /* all three get same style */
```

The space between selectors means “inside.” The comma means “and also.” The dot stuck to a tag means “this tag with this class.”

For the full selector reference, see Part 2C Quick Ref Table: CSS Selector Types.

---

10. The Box Model – Every Element Is a Box

Every HTML element is a rectangular box. Every single one. Even circles (the box is invisible, the circle is inside it).

Every box has four layers from inside to outside:

```
+-------------------------------------+
|             MARGIN                  |  <-- space OUTSIDE the border
|  +--------------------------------+|
|  |           BORDER               ||  <-- the visible edge
|  |  +---------------------------+||
|  |  |         PADDING           |||  <-- space INSIDE the border
|  |  |  +---------------------+  |||
|  |  |  |       CONTENT       |  |||  <-- your actual text/image
|  |  |  +---------------------+  |||
|  |  +---------------------------+||
|  +--------------------------------+|
+-------------------------------------+
```

> Think of a framed photo on a wall. CONTENT = the photo. PADDING = the matting around the photo. BORDER = the frame. MARGIN = the space between this frame and the next one on the wall.

```css
.task-item {
  padding: 10px;            /* 10px breathing room inside */
  border: 1px solid black;  /* thin black frame */
  margin: 5px;              /* 5px gap between items */
}
```

Why does my layout look wrong?

90% of layout problems are box model problems. The element is bigger than you expected because padding and border ADD to the size. Fix it:

```css
* { box-sizing: border-box; }
```

Put this at the top of every CSS file. It makes `width` mean total width including padding and border. Without it, a 200px box with 10px padding becomes 220px. With it, it stays 200px.

> Always add `box-sizing: border-box` to your reset. It's the first line in every professional CSS file. Without it, you'll spend hours wondering why your boxes are too big.

---

11. Flexbox – How to Arrange Things

Flexbox is how you lay things out side by side or stack them. Before flexbox, this was a nightmare. Now it's three lines.

```css
/* Put items in a row: */
.container {
  display: flex;              /* turn on flexbox */
  flex-direction: row;        /* items go left to right */
  gap: 10px;                  /* space between items */
}

/* Put items in a column: */
.container {
  display: flex;
  flex-direction: column;     /* items go top to bottom */
  gap: 10px;
}
```

> `display: flex` is the ON switch. `flex-direction` is the layout axis. `gap` is the spacing. That's 80% of flexbox right there. Row = parade line. Column = single file.

Centering things – the thing everyone Googles:

```css
.container {
  display: flex;
  justify-content: center;    /* centre on main axis */
  align-items: center;        /* centre on cross axis */
}
```

> `justify-content` = main axis (horizontal in row, vertical in column). `align-items` = cross axis (the other one).

For detailed flexbox syntax and patterns, see Part 2B, Section 11. For the full property reference, see Part 2C Quick Ref Tables: CSS Flexbox.

---

12. Responsive Design – Why @media Exists

Your Gelos site needs to work on phones AND desktops. That's responsive design.

```
/* Default styles (mobile first): */
.task-list {
  display: flex;
  flex-direction: column;    /* stack on mobile */
}

/* When screen is wider than 768px: */
@media (min-width: 768px) {
  .task-list {
    flex-direction: row;      /* side by side on desktop */
  }
}
```

`@media` is a condition. "When the screen is at least this wide, apply these styles." It's an if-statement for CSS.

> It's literally: `IF (screen > 768px) { apply these styles }`. Same logic as C#'s if statement, just CSS syntax. You're not learning something new. You're translating something you already know.

Mobile-first = write styles for small screens by default. Then add `@media` queries to EXPAND the layout for larger screens. Don't do it backwards.

For responsive syntax and breakpoints, see Part 2B, Section 14.

---

13. Why JavaScript Exists

HTML is a corpse. CSS is the makeup. JavaScript makes it move.

Without JS, a web page is static. It sits there. Click a button, nothing happens. Type in a form, nothing processes. JavaScript adds behaviour.

```
<!-- Without JavaScript: -->
<button>Add Task</button>     <!-- does nothing -->

<!-- With JavaScript: -->
<script>
  document.getElementById("addBtn").addEventListener("click", addTask);
</script>
```

> HTML builds the barracks. CSS paints them. JavaScript is the soldiers inside doing the work. Your Gelos Task Manager without JS is a picture of a task manager. With JS, it actually manages tasks.
>
> HTML = structure. CSS = appearance. JavaScript = behaviour. You need all three.

14. JS Looks Like C# – The Comparison

Good news. You already learned most of this in C#. JavaScript is C# in a hoodie.

Concept	C#	JavaScript
Variable	`int x = 5;`	`let x = 5;`
Constant	`const int X = 5;`	`const X = 5;`
String	`string name = "Wu";`	`let name = "Wu";`
Array	`string[] arr = {"a","b"};`	`let arr = ["a","b"];`
If	`if (x > 5) { }`	`if (x > 5) { }`
For	`for (int i=0; i<10; i++)`	`for (let i=0; i<10; i++)`
For-each	`foreach (var s in list)`	`for (let s of list)`
While	`while (running) { }`	`while (running) { }`
Function/Method	`void DoThing() { }`	`function doThing() { }`
Print	`Console.WriteLine(x);`	`console.log(x);`
String format	`$"Hello {name}"`	`` `Hello ${name}` ``
Comment	`// comment`	`// comment`
AND / OR / NOT	`&& / || / !`	`&& / || / !`
Equal check	`==`	`===` (strict)
Semicolons	Required	Optional (but use them)
Types	Required (`int`, `string`)	Not required (`let`, `const`)

> Same braces. Same semicolons. Same if/for/while. JS just uses `let` instead of `int`/`string`, and `function` instead of `void`. The SHAPE is identical to C#.

`let` vs `const` vs `var`:

```
let name = "Wu";        // can change later
const MAX = 100;        // can NEVER change
var old = "don't use"; // old way, avoid it
```

> `let` = a whiteboard you can erase and rewrite. `const` = a plaque bolted to the wall. `var` = the old whiteboard that has weird scoping bugs. Pretend `var` doesn't exist.

Why no types? JavaScript figures it out automatically. Like Python. Unlike C#. This makes JS faster to write but easier to break. C# catches type errors before running. JS catches them WHILE running (or doesn't catch them at all).

15. The DOM – How JS Talks to HTML

The DOM (Document Object Model) is how JavaScript sees your HTML. It's a tree of objects that JS can read and modify.

```
// FINDING elements:
document.getElementById("taskName")       // find by ID
document.querySelector(".task-item")       // find first match (CSS selector)
document.querySelectorAll(".task-item")    // find ALL matches

// CHANGING elements:
element.textContent = "New text";          // change text
element.innerHTML = "<b>Bold text</b>";    // change HTML inside
element.style.color = "red";               // change CSS
element.classList.add("urgent");            // add a class

// CREATING elements:
let div = document.createElement("div");   // make a new <div>
div.textContent = "New task";              // put text in it
document.getElementById("list").appendChild(div); // add it to the page
```

> The DOM is the construction site. HTML is the blueprint. JS is the builder who can read the blueprint and modify the building in real time. `getElementById` is like saying "go find the wall with tag #47 and paint it red."

For the full DOM method reference, see Part 2C Quick Ref Tables: JS DOM Finding/Reading/Changing Elements.

---

16. Events – When Things Happen

Events are how your page responds to user actions. Click, type, hover, scroll, submit.

```
// The pattern:
element.addEventListener("EVENT", functionToRun);

// Examples:
addBtn.addEventListener("click", addTask);       // when clicked
input.addEventListener("keyup", filterTasks);    // when key released
form.addEventListener("submit", handleSubmit);   // when form submitted
```

The pattern is always: WHAT element, WHAT event, WHAT function to run.

> `addEventListener` = "when THIS happens, do THAT." Three pieces: element, event name, function.

For detailed event listener syntax and event delegation, see Part 2B, Section 18.

---

17. Local Storage – Saving Data in the Browser

Your Gelos tasks need to survive a page refresh. Local Storage saves data in the browser permanently (until cleared).

```
// SAVE data:
localStorage.setItem("tasks", JSON.stringify(taskArray));

// LOAD data:
let tasks = JSON.parse(localStorage.getItem("tasks"));

// DELETE data:
localStorage.removeItem("tasks");
```

`localStorage` only stores strings. That's why you need `JSON.stringify` (convert array to string) and `JSON.parse` (convert string back to array).

> It's a filing cabinet in the browser. `setItem` = put a file in the cabinet. `getItem` = pull a file out. The cabinet survives closing the browser. It only gets cleared if the user manually clears it or you call `removeItem`/`clear`.
>
> Always `JSON.stringify` when saving arrays/objects, `JSON.parse` when loading.

For the full localStorage syntax, see Part 2B, Section 20 and Part 2C Quick Ref Table: JS localStorage.

---

18. The Three-Language Mental Model

If you remember nothing else from this entire document, remember this section.

A web page is three files working together:

```
index.html      ->  WHAT is on the page      (structure)
style.css       ->  HOW it looks             (appearance)
script.js       ->  WHAT it does             (behaviour)
```

They connect like this:

```
<html>
  <head>
    <link rel="stylesheet" href="style.css">     <!-- CSS -->
  </head>
  <body>
    <h1 id="title" class="header">Gelos</h1>    <!-- HTML -->
    <script src="script.js"></script>            <!-- JS (at bottom) -->
  </body>
</html>
```

CSS finds elements using selectors:

```
h1 { }          /* tag name */
.header { }     /* class (dot) */
#title { }      /* ID (hash) */
```

JS finds elements using methods:

```
document.getElementById("title")
document.querySelector(".header")
document.querySelectorAll("h1")
```

JS responds to actions using events:

```
element.addEventListener("click", doSomething)
```

JS saves data using localStorage:

```
localStorage.setItem("key", JSON.stringify(data))
let data = JSON.parse(localStorage.getItem("key"))
```

HTML is the bones. CSS is the skin. JS is the brain. Three files. Three jobs. One page.

PART 2B: HTML/CSS/JS SYNTAX (THE DUMBCUNT GUIDE)

Line-by-line syntax. What to type, where, and why.

> The WHY Guide (Part 2A) explained the logic. This section shows you exactly what to type. Every code block is annotated line-by-line. If you don't understand WHY something looks the way it does, go back to Part 2A.

HTML SYNTAX

1. The Starter Template – Type This First

Every HTML file starts with this. Type it once, save it as a template, copy it for every new project. VS Code shortcut: type ! then hit Tab.

```
<!DOCTYPE html>
<!-- ^ Tells the browser: "This is HTML5." Always the first line. -->

<html lang="en">
<!-- ^ Opens the entire page. lang="en" = English content. -->

  <head>
  <!-- ^ Invisible setup section. Browser sees it, user doesn't. -->

    <meta charset="UTF-8">
    <!-- ^ Character encoding. UTF-8 handles emojis, accents, symbols. -->
```

```
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <!-- ^ Makes mobile devices render at correct width. Always include this.
 -->

    <title>Gelos Task Manager</title>
    <!-- ^ Text that appears in the browser tab. -->

    <link rel="stylesheet" href="style.css">
    <!-- ^ Connects your CSS file. Path is relative to this HTML file. -->

  </head>

  <body>
  <!-- ^ Everything visible goes in here. -->

    <!-- Your content goes here -->

    <script src="script.js"></script>
    <!-- ^ Connects your JS file. AT THE BOTTOM of body so HTML loads first.
-->

  </body>
</html>
```

Three files: `index.html`, `style.css`, `script.js`. The HTML file links to the other two. CSS in <head>. JS at bottom of <body>.

---

2. Text Elements

Headings

```
<h1>Main Page Title</h1>          <!-- biggest, one per page -->
<h2>Section Heading</h2>           <!-- sub-section -->
<h3>Sub-sub Heading</h3>           <!-- deeper -->
<h4>Even Deeper</h4>               <!-- you get the idea -->
<h5>Rarely Used</h5>
<h6>Almost Never Used</h6>         <!-- smallest -->
```

Use ONE <h1> per page. Multiple <h2>s under it. Multiple <h3>s under each <h2>. It's a chain of command.

h1 = CO. h2 = Platoon Commander. h3 = Section Commander. Don't skip levels. Don't use h4 before you've used h3. Keep the chain of command clean.

Paragraphs and basic text

```
<p>This is a paragraph of text.</p>
<p>Each paragraph gets its own line with spacing above and below.</p>
```

```
<strong>Bold text</strong>          <!-- semantic: important -->
<em>Italic text</em>              <!-- semantic: emphasis -->
<br>                              <!-- line break (no closing tag) -->
<hr>                              <!-- horizontal rule / divider -->

<span>Inline wrapper</span>       <!-- style a piece of text within a line -->
```

> is like a highlighter pen. It wraps around text inside a paragraph so you can style just that piece without breaking the flow. breaks to a new line. doesn't.

Lists

```
<!-- Unordered list (bullet points): -->
<ul>
  <li>First item</li>
  <li>Second item</li>
  <li>Third item</li>
</ul>

<!-- Ordered list (numbered): -->
<ol>
  <li>Step one</li>
  <li>Step two</li>
  <li>Step three</li>
</ol>
```

 = bullets. = numbers. = each item. The list items go INSIDE the list container.

---

3. Links and Images

Links

```
<a href="https://google.com">Go to Google</a>
<!-- ^ href = where to go. Text between tags = what user clicks. -->

<a href="page2.html">Go to Page 2</a>
<!-- ^ Relative link to another file in your project. -->

<a href="https://google.com" target="_blank">Open in new tab</a>
<!-- ^ target="_blank" opens in a new browser tab. -->
```

Images

```
<img src="logo.png" alt="Company Logo">
<!-- ^ src = file path. alt = text if image fails to load (accessibility). -->

<img src="images/photo.jpg" alt="Team photo" width="300">
<!-- ^ width attribute sets size in pixels. Can also use CSS instead. -->
```

Always include `alt` text on images. Screen readers read it aloud. Also shows if the image breaks. It's accessibility AND debugging.

---

4. Containers – div, section, header, etc.

```
<div>Generic container</div>
<!-- ^ A box with no meaning. Use when no semantic tag fits. -->

<header>
  <h1>Gelos Task Manager</h1>
  <nav>
    <a href="#">Home</a>
    <a href="#">About</a>
  </nav>
</header>
<!-- ^ <header> = top of page/section. <nav> = navigation links. -->

<main>
  <section id="taskSection">
    <h2>My Tasks</h2>
    <div id="taskList">
      <!-- Tasks inserted here by JavaScript -->
    </div>
  </section>
</main>
<!-- ^ <main> = primary content. <section> = grouped chunk. <div> = JS target.
 -->

<footer>
  <p>&copy; 2026 Gelos Task Manager</p>
</footer>
<!-- ^ <footer> = bottom of page. &copy; = copyright symbol. -->
```

Use semantic tags (`<header>`, `<main>`, `<footer>`, `<section>`) for the big structural blocks. Use `<div>` for smaller utility containers that JS needs to target or CSS needs to style.

---

5. Forms – Collecting User Input

This is the core of Gelos. Your task creation form collects: task name, description, priority, due date. Here's every input type you'll need.

Text input

```
<label for="taskName">Task Name:</label>
<!-- ^ for="taskName" links this label to the input with id="taskName". -->
```

```
<input type="text" id="taskName" placeholder="Enter task name" required>
<!-- ^ type="text" = single line text box. -->
<!-- ^ id = JS uses this to read the value. -->
<!-- ^ placeholder = grey hint text that disappears when you type. -->
<!-- ^ required = form won't submit without this filled in. -->
```

Textarea (multi-line)

```
<label for="taskDesc">Description:</label>
<textarea id="taskDesc" rows="3" placeholder="Describe the task"></textarea>
<!-- ^ <textarea> = multi-line input. rows = visible lines. -->
<!-- ^ Note: textarea has a closing tag. Input doesn't. -->
```

Date input

```
<label for="dueDate">Due Date:</label>
<input type="date" id="dueDate">
<!-- ^ Browser shows a date picker calendar. Value format: YYYY-MM-DD. -->
```

Select dropdown

```
<label for="priority">Priority:</label>
<select id="priority">
  <option value="">Select priority</option>
  <!-- ^ Empty value = placeholder option. -->
  <option value="low">Low</option>
  <option value="medium">Medium</option>
  <option value="high">High</option>
</select>
```

Checkbox

```
<label>
  <input type="checkbox" id="taskComplete">
  Mark as complete
</label>
<!-- ^ Wrapping input in label makes the text clickable too. -->
```

Button

```
<button type="button" id="addTaskBtn">Add Task</button>
<!-- ^ type="button" prevents form submission. JS handles the click instead.
-->
<!-- ^ type="submit" would reload the page. Use "button" for JS-handled forms.
 -->
```

> CRITICAL: Use `type="button"` on buttons in JS-handled forms. `type="submit"` reloads the page and you lose everything.

---

6. Tables

You might display tasks in a table format. Here's the structure:

```html
<table>
  <thead>
  <!-- ^ Table header section. -->
    <tr>
    <!-- ^ Table Row. -->
      <th>Task</th>
      <!-- ^ Table Header cell (bold, centred by default). -->
      <th>Priority</th>
      <th>Due Date</th>
      <th>Actions</th>
    </tr>
  </thead>
  <tbody id="taskTableBody">
  <!-- ^ Table body. JS will insert rows here. -->
    <tr>
      <td>Buy groceries</td>
      <!-- ^ Table Data cell (normal text). -->
      <td>Medium</td>
      <td>2026-03-15</td>
      <td><button>Delete</button></td>
    </tr>
  </tbody>
</table>
```

Table hierarchy: `<table>` contains `<thead>` and `<tbody>`. Each contains `<tr>` (rows). Each row contains `<th>` (headers) or `<td>` (data cells). It's a grid. Rows across, cells down.

Tag	Meaning	Contains
`<table>`	The whole table	`<thead>`, `<tbody>`
`<thead>`	Header section	`<tr>` with `<th>` cells
`<tbody>`	Data section	`<tr>` with `<td>` cells
`<tr>`	Table Row	`<th>` or `<td>` cells
`<th>`	Header Cell	Text (bold by default)
`<td>`	Data Cell	Text, buttons, inputs

---

7. Gelos HTML Structure – The Full Skeleton

Here's the full HTML skeleton for a Gelos-style task manager. Copy this, then build on it.

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Gelos Task Manager</title>
  <link rel="stylesheet" href="style.css">
```

```
</head>
<body>

  <header>
    <h1>Gelos Task Manager</h1>
  </header>

  <main>
    <!-- TASK CREATION FORM -->
    <section id="addTaskSection">
      <h2>Add New Task</h2>
      <form id="taskForm">
        <label for="taskName">Task Name:</label>
        <input type="text" id="taskName" placeholder="Enter task" required>

        <label for="taskDesc">Description:</label>
        <textarea id="taskDesc" rows="2" placeholder="Details..."></textarea>

        <label for="priority">Priority:</label>
        <select id="priority">
          <option value="low">Low</option>
          <option value="medium" selected>Medium</option>
          <option value="high">High</option>
        </select>

        <label for="dueDate">Due Date:</label>
        <input type="date" id="dueDate">

        <button type="button" id="addTaskBtn">Add Task</button>
      </form>
    </section>

    <!-- TASK DISPLAY AREA -->
    <section id="taskDisplaySection">
      <h2>My Tasks</h2>
      <input type="text" id="searchBox" placeholder="Search tasks...">
      <select id="filterPriority">
        <option value="all">All Priorities</option>
        <option value="high">High</option>
        <option value="medium">Medium</option>
        <option value="low">Low</option>
      </select>
      <div id="taskList">
        <!-- Tasks rendered here by JavaScript -->
      </div>
    </section>
  </main>

  <footer>
```

```
    <p>&copy; 2026 Gelos Task Manager</p>
  </footer>

  <script src="script.js"></script>
</body>
</html>
```

This is a scaffold. Every element has an ID for JS to grab. The `taskList` div is empty – JS fills it. The form uses `type="button"` so it doesn't reload.

CSS SYNTAX

8. CSS Reset – Always Start With This

Browsers add their own default styling. A reset removes it so everything starts clean. Type this at the top of every `style.css` file.

```
/* === RESET & BASE === */
* {
  /* ^ * = select EVERY element on the page. */
  margin: 0;
  /* ^ Remove default spacing outside elements. */
  padding: 0;
  /* ^ Remove default spacing inside elements. */
  box-sizing: border-box;
  /* ^ Padding and border go INSIDE the width, not outside. */
}

body {
  font-family: Arial, Helvetica, sans-serif;
  /* ^ Set the default font for the entire page. Falls back if Arial unavaila
ble. */
  line-height: 1.6;
  /* ^ Space between lines of text. 1.6 = 60% more than font size. */
  background-color: #f5f5f5;
  /* ^ Light grey background. Professional, easy on the eyes. */
  color: #333;
  /* ^ Dark grey text. Not pure black -- easier to read on screens. */
}
```

`* { box-sizing: border-box; }` is NON-NEGOTIABLE. Put it in every CSS file. Without it, your layouts will be off by mysterious amounts and you'll lose hours debugging.

---

9. Colours – Three Ways to Write Them

```css
/* 1. Named colours (limited selection): */
color: black;
color: white;
color: red;

/* 2. Hex codes (most common): */
color: #333333;        /* dark grey */
color: #333;           /* shorthand (same as above) */
color: #f5f5f5;        /* light grey */
color: #ff0000;        /* red */
color: #ffffff;        /* white */
color: #000000;        /* black */

/* 3. RGB/RGBA (with transparency): */
color: rgb(51, 51, 51);           /* dark grey */
color: rgba(0, 0, 0, 0.5);        /* black at 50% opacity */
background: rgba(255, 0, 0, 0.2); /* red at 20% opacity */
```

> For Gelos, stick with hex codes. `#333` for text, `#f5f5f5` for backgrounds, `#fff` for white. Use `rgba` when you need transparency (overlays, shadows).

For the full greyscale reference, see Part 2C Quick Ref Table: CSS Greyscale Quick Ref.

---

10. Typography

```css
h1 {
  font-size: 2rem;
  /* ^ 2rem = 2x the root font size (usually 32px). */
  font-weight: bold;
  /* ^ bold, normal, or a number like 700. */
  color: #333;
  margin-bottom: 10px;
}

h2 {
  font-size: 1.5rem;
  color: #444;
  margin-bottom: 8px;
}

p {
  font-size: 1rem;
  /* ^ 1rem = base size (16px by default). */
  margin-bottom: 10px;
}
```

```css
a {
  color: #333;
  text-decoration: none;
  /* ^ Removes the underline. Add your own styles instead. */
}
a:hover {
  text-decoration: underline;
  /* ^ :hover = when mouse is over it. */
}
```

> Use `rem` for almost everything. It scales with the browser's font settings, which is better for accessibility. Use `px` for tiny fixed things like `1px` borders.

For the full units reference, see Part 2C Quick Ref Table: CSS Units.

---

11. Layout with Flexbox

Flexbox is how you arrange elements side by side or in columns. It replaces all that float nonsense from 2010.

Horizontal layout (row)

```css
.nav-links {
  display: flex;
  /* ^ Turns this container's children into flex items. */
  flex-direction: row;
  /* ^ Items flow left to right. (This is the default, can be omitted.) */
  gap: 20px;
  /* ^ 20px space between each item. */
  align-items: center;
  /* ^ Vertically centre items within the row. */
}
```

Vertical layout (column)

```css
.form-group {
  display: flex;
  flex-direction: column;
  /* ^ Items stack top to bottom. */
  gap: 8px;
}
```

Centring everything

```css
.centre-me {
  display: flex;
  justify-content: center;
  /* ^ Centre along the MAIN axis (horizontal in row). */
  align-items: center;
  /* ^ Centre along the CROSS axis (vertical in row). */
  height: 100vh;
```

```
  /* ^ 100% of viewport height. Needed so there's space to centre in. */
}
```

Wrapping items (card grid)

```
.task-grid {
  display: flex;
  flex-wrap: wrap;
  /* ^ Items wrap to next line when they run out of space. */
  gap: 15px;
}
.task-grid .task-card {
  flex: 1 1 250px;
  /* ^ flex: grow shrink basis. */
  /* ^ grow: 1 = expand to fill space. */
  /* ^ shrink: 1 = shrink if needed. */
  /* ^ basis: 250px = ideal width before growing/shrinking. */
}
```

> `flex: 1 1 250px` = "try to be 250px wide, but grow or shrink to fit." Perfect for responsive card grids.

Space between items

```
justify-content: flex-start;   /* pack left (default) */
justify-content: flex-end;     /* pack right */
justify-content: center;       /* centre */
justify-content: space-between; /* first and last at edges, even gaps */
justify-content: space-around;  /* equal space around each item */
justify-content: space-evenly;  /* perfectly even gaps everywhere */
```

> `space-between` is the most useful for navbars (logo left, links right). `center` for centring. `flex-start` for most other layouts.

Common flexbox patterns:

```
/* Centre everything: */
.x { display: flex; justify-content: center; align-items: center; height: 100
vh; }

/* Navbar (logo L, links R): */
.nav { display: flex; justify-content: space-between; align-items: center; }

/* Form column: */
.form { display: flex; flex-direction: column; gap: 8px; }

/* Card grid (wrap): */
.grid { display: flex; flex-wrap: wrap; gap: 15px; }
.card { flex: 1 1 250px; }
```

---

12. Styling Forms

Making inputs not look like 1995:

```
input[type="text"],
input[type="date"],
textarea,
select {
  width: 100%;
  /* ^ Fill the full width of the parent container. */
  padding: 10px 12px;
  /* ^ 10px top/bottom, 12px left/right. Breathing room inside. */
  border: 1px solid #ccc;
  /* ^ Thin light grey border. */
  border-radius: 4px;
  /* ^ Slightly rounded corners. */
  font-size: 1rem;
  font-family: inherit;
  /* ^ Use same font as the page. Inputs don't inherit by default. */
}

input:focus,
textarea:focus,
select:focus {
  outline: none;
  /* ^ Remove the browser's default blue outline. */
  border-color: #333;
  /* ^ Darken the border when focused instead. */
  box-shadow: 0 0 0 2px rgba(0, 0, 0, 0.1);
  /* ^ Subtle glow effect around the input. */
}

button {
  padding: 10px 20px;
  background-color: #333;
  color: #fff;
  border: none;
  border-radius: 4px;
  font-size: 1rem;
  cursor: pointer;
  /* ^ Shows hand icon on hover. Tells user it's clickable. */
  transition: background-color 0.2s ease;
  /* ^ Smooth colour change when hovered. */
}
button:hover {
  background-color: #555;
}
```

`:focus` = when the element is selected/active. `:hover` = when mouse is over it. `transition` = animate the change smoothly.

13. Styling Task Cards

Each task in Gelos should be a card. Here's the CSS pattern:

```css
.task-card {
  background: #fff;
  border: 1px solid #ddd;
  border-radius: 8px;
  padding: 15px;
  margin-bottom: 10px;
  box-shadow: 0 1px 3px rgba(0, 0, 0, 0.1);
  /* ^ Subtle shadow: 0 horizontal, 1px down, 3px blur, 10% black. */
  display: flex;
  justify-content: space-between;
  /* ^ Content pushed to opposite ends. */
  align-items: center;
  transition: box-shadow 0.2s ease;
}

.task-card:hover {
  box-shadow: 0 2px 8px rgba(0, 0, 0, 0.15);
  /* ^ Bigger shadow on hover = looks elevated. */
}

/* Priority indicators: */
.task-card.high {
  border-left: 4px solid #333;
  /* ^ Thick left border = visual priority indicator (works in greyscale). */
}
.task-card.medium {
  border-left: 4px solid #999;
}
.task-card.low {
  border-left: 4px solid #ddd;
}

/* Completed task styling: */
.task-card.completed {
  opacity: 0.6;
  /* ^ Makes the card semi-transparent. */
  text-decoration: line-through;
}
```

> `border-left` for priority works in greyscale printing. Don't rely on colour alone – use thickness and shade.

14. Responsive Layout (@media queries, mobile-first)

```css
/* Mobile-first base styles */
.container {
  max-width: 900px;
  /* ^ Content won't stretch wider than 900px. */
  margin: 0 auto;
  /* ^ 0 top/bottom, auto left/right = horizontally centred. */
  padding: 15px;
}

/* Tablet and up */
@media (min-width: 768px) {
  .task-form {
    display: flex;
    flex-wrap: wrap;
    gap: 10px;
  }
  .task-form .form-group {
    flex: 1 1 200px;
    /* ^ Form fields go side by side on wider screens. */
  }
}

/* Desktop */
@media (min-width: 1024px) {
  .container {
    max-width: 1100px;
  }
}
```

> Mobile first = write styles for small screens by default. Then add `@media` queries to EXPAND the layout for larger screens. Don't do it backwards.

Common breakpoints:

Width	Device	What Changes
< 480px	Small phone	Single column, stacked everything
480-768px	Large phone / small tablet	Wider cards, maybe 2 columns
768-1024px	Tablet / small laptop	Side navigation appears
> 1024px	Desktop	Full layout, multiple columns

The `@media` syntax:

```
@media (min-width: 768px) { /* tablet+ styles */ }
@media (min-width: 1024px) { /* desktop styles */ }
```

JAVASCRIPT SYNTAX

15. Variables and Data Types

```
// DECLARING VARIABLES
let taskName = "Buy groceries";    // text (string) -- can change
let taskCount = 5;                  // number -- can change
let isComplete = false;             // boolean -- can change
const MAX_TASKS = 100;              // constant -- CANNOT change

// ARRAYS (lists)
let tasks = [];                     // empty array
let colours = ["red", "blue"];      // array with items

// OBJECTS (like a mini-database record)
let task = {
  name: "Buy groceries",
  priority: "high",
  dueDate: "2026-03-15",
  completed: false
};
// ^ Each task in Gelos will be an object like this, stored in an array.

// ACCESSING object properties
console.log(task.name);             // "Buy groceries"
console.log(task.priority);         // "high"
task.completed = true;              // change a property
```

An array of objects is your task database. Each object is a row. Each property is a column. `tasks = [{name: ..., priority: ...}, {name: ..., priority: ...}]`. That's a table in memory.

Use `const` by default. Only use `let` if the value needs to change. Never use `var`.

---

16. Functions

```
// Function declaration (traditional)
function addTask() {
  // code goes here
}

// Function with parameters
function greet(name) {
  console.log(`Hello ${name}`);
```

```
  // ^ Backticks ` ` with ${} = template literal. Same as C#'s $"".
}

// Function that returns a value
function getTaskCount() {
  return tasks.length;
}

// Arrow function (shorter syntax, common in modern JS)
const addTask = () => {
  // code goes here
};

// Arrow function with parameter
const greet = (name) => {
  console.log(`Hello ${name}`);
};
```

`function doThing() { }` and `const doThing = () => { }` do the same thing. Use whichever your teacher prefers. The arrow => version is newer and shorter.

For Gelos, you'll write functions for: `addTask()`, `deleteTask()`, `editTask()`, `filterTasks()`, `searchTasks()`, `saveTasks()`, `loadTasks()`, `renderTasks()`. Each one does ONE job.

---

17. DOM Manipulation – Finding, Reading, Changing, Creating Elements

Finding elements

```
// By ID (returns ONE element):
const taskList = document.getElementById("taskList");
const nameInput = document.getElementById("taskName");

// By CSS selector (returns FIRST match):
const firstCard = document.querySelector(".task-card");

// By CSS selector (returns ALL matches as NodeList):
const allCards = document.querySelectorAll(".task-card");
// ^ NodeList is like an array. Use forEach to loop it.
```

Reading values from inputs

```
const name = document.getElementById("taskName").value;
// ^ .value reads what the user typed in the input.

const priority = document.getElementById("priority").value;
// ^ .value reads the selected option's value attribute.

const date = document.getElementById("dueDate").value;
// ^ .value returns the date as YYYY-MM-DD string.
```

```
const isChecked = document.getElementById("taskComplete").checked;
// ^ .checked returns true/false for checkboxes.
```

Changing elements

```
element.textContent = "New text";
// ^ Changes the visible text. Safe (no HTML injection).

element.innerHTML = "<b>Bold text</b>";
// ^ Changes HTML inside. Powerful but dangerous with user input.

element.style.display = "none";
// ^ Hides the element.

element.style.display = "flex";
// ^ Shows it again.

element.classList.add("completed");
// ^ Adds a CSS class to the element.

element.classList.remove("completed");
// ^ Removes a CSS class.

element.classList.toggle("completed");
// ^ Adds if missing, removes if present. Perfect for checkboxes.
```

Creating new elements

```
const card = document.createElement("div");
// ^ Creates a new <div> in memory (not on page yet).

card.className = "task-card";
// ^ Sets its CSS class.

card.innerHTML = `
  <span class="task-name">${task.name}</span>
  <span class="task-priority">${task.priority}</span>
  <button class="delete-btn">Delete</button>
`;
// ^ Sets the HTML content using a template literal.

document.getElementById("taskList").appendChild(card);
// ^ Adds the card to the page inside the taskList container.
```

> `createElement` + `innerHTML` + `appendChild` = the JS pattern for dynamically building HTML. You'll use this in `renderTasks()`.

---

18. Event Listeners

The pattern

```
// PATTERN: element.addEventListener("event", function);
```

Click

```
document.getElementById("addTaskBtn").addEventListener("click", addTask);
// ^ When addTaskBtn is clicked, run the addTask function.
```

Keyup (live search)

```
document.getElementById("searchBox").addEventListener("keyup", filterTasks);
// ^ When a key is released in searchBox, run filterTasks.
```

Change (dropdown)

```
document.getElementById("filterPriority").addEventListener("change", filterTa
sks);
// ^ When the dropdown selection changes, run filterTasks.
```

DOMContentLoaded (page load)

```
document.addEventListener("DOMContentLoaded", () => {
  loadTasks();
  renderTasks();
});
// ^ When page finishes loading, load saved tasks and display them.
```

Event delegation (for dynamically created elements)

```
document.getElementById("taskList").addEventListener("click", (e) => {
  if (e.target.classList.contains("delete-btn")) {
    // ^ e.target = the exact element that was clicked.
    const taskId = e.target.dataset.id;
    // ^ data-id attribute on the button = custom data storage.
    deleteTask(taskId);
  }
});
```

> Event delegation = put ONE listener on the parent, check what was actually clicked. This works for elements that didn't exist when the page loaded (because JS created them later). Essential for Gelos task cards.
>
> For dynamically created elements, use event delegation on the parent. Don't attach listeners to elements that get recreated – they'll lose their listeners.

---

19. Array Methods – CRUD Operations

Your `tasks` array is your database. These methods are your CRUD operations.

CREATE – adding tasks

```
tasks.push(newTask);
// ^ Add to END of array. Like INSERT in SQL.

// Building a task object:
const newTask = {
  id: Date.now(),
  // ^ Unique ID using current timestamp. Good enough for local storage.
  name: document.getElementById("taskName").value,
  priority: document.getElementById("priority").value,
  dueDate: document.getElementById("dueDate").value,
  completed: false
};
```

READ – finding tasks

```
// Find one by ID:
const task = tasks.find(t => t.id === targetId);
// ^ find() returns FIRST match or undefined. Like SELECT WHERE id = x.

// Find all matching a condition:
const highPriority = tasks.filter(t => t.priority === "high");
// ^ filter() returns ALL matches. Like SELECT WHERE priority = 'high'.

// Find the index of an item:
const index = tasks.findIndex(t => t.id === targetId);
// ^ Returns position number or -1 if not found.
```

UPDATE – changing tasks

```
// Find and modify:
const task = tasks.find(t => t.id === targetId);
if (task) {
  task.completed = !task.completed;
  // ^ Toggle: true becomes false, false becomes true.
  task.name = newName;
}
```

DELETE – removing tasks

```
// Filter out the deleted task:
tasks = tasks.filter(t => t.id !== targetId);
// ^ Keep everything that DOESN'T match the ID. Like DELETE WHERE id = x.
```

LOOP – processing all tasks

```
// forEach (do something to each):
tasks.forEach(task => {
  console.log(task.name);
});

// map (transform each into something new):
const names = tasks.map(t => t.name);
```

```
// ^ Returns a new array of just the names.

// sort (reorder):
tasks.sort((a, b) => new Date(a.dueDate) - new Date(b.dueDate));
// ^ Sort by due date, earliest first.
```

push = add. find = get one. filter = get matching. filter(!=) = delete. forEach = loop. map = transform. sort = reorder. That's your whole CRUD toolkit.

---

20. Local Storage – The saveTasks/loadTasks Pattern

```
// SAVE tasks to browser storage:
function saveTasks() {
  localStorage.setItem("gelosTasks", JSON.stringify(tasks));
  // ^ Convert the tasks array to a JSON string and store it.
  // ^ "gelosTasks" is the key -- like a filename.
}

// LOAD tasks from browser storage:
function loadTasks() {
  const stored = localStorage.getItem("gelosTasks");
  // ^ Retrieve the JSON string. Returns null if nothing saved.
  if (stored) {
    tasks = JSON.parse(stored);
    // ^ Convert JSON string back to an array.
  }
}

// WHEN to save:
// Call saveTasks() after EVERY change:
function addTask() {
  // ... create task object, push to array ...
  saveTasks();    // Save immediately after adding.
  renderTasks();  // Redraw the task list.
}
function deleteTask(id) {
  tasks = tasks.filter(t => t.id !== id);
  saveTasks();    // Save immediately after deleting.
  renderTasks();
}
```

Save after every change. Load on page load. That's the entire persistence strategy for Gelos. Every addTask, deleteTask, editTask, and toggleComplete should end with saveTasks() then renderTasks().

---

21. The Render Function – Clear, Filter, Loop, Create, Append

This is the most important function in Gelos. It clears the display, loops through tasks, and builds fresh HTML for each one.

```
function renderTasks() {
  const taskList = document.getElementById("taskList");
  taskList.innerHTML = "";
  // ^ Clear everything currently displayed.

  // Get filter/search values
  const searchTerm = document.getElementById("searchBox").value.toLowerCase();
  const filterPriority = document.getElementById("filterPriority").value;

  // Filter tasks
  let filtered = tasks.filter(task => {
    const matchesSearch = task.name.toLowerCase().includes(searchTerm);
    const matchesPriority = filterPriority === "all" || task.priority === fil
terPriority;
    return matchesSearch && matchesPriority;
  });

  // Build HTML for each task
  filtered.forEach(task => {
    const card = document.createElement("div");
    card.className = `task-card ${task.priority}${task.completed ? " complete
d" : ""}`;
    // ^ Sets classes: task-card + priority + completed if true.
    card.innerHTML = `
      <div class="task-info">
        <strong>${task.name}</strong>
        <small>${task.priority} | ${task.dueDate || "No date"}</small>
      </div>
      <div class="task-actions">
        <button class="complete-btn" data-id="${task.id}">
          ${task.completed ? "Undo" : "Done"}
        </button>
        <button class="delete-btn" data-id="${task.id}">Delete</button>
      </div>
    `;
    taskList.appendChild(card);
  });

  // Show message if no tasks
  if (filtered.length === 0) {
    taskList.innerHTML = "<p class='empty-msg'>No tasks found.</p>";
  }
}
```

The pattern: clear > filter > loop > createElement > innerHTML > appendChild.
This is the core loop that makes your UI update.

---

22. Gelos script.js Skeleton – Full Annotated Structure

Here's the full structure of your `script.js`. Each function does ONE job. Fill in the details as you build features.

```
// === DATA ===
let tasks = [];

// === LOAD & SAVE ===
function loadTasks() {
  const stored = localStorage.getItem("gelosTasks");
  if (stored) tasks = JSON.parse(stored);
}

function saveTasks() {
  localStorage.setItem("gelosTasks", JSON.stringify(tasks));
}

// === CRUD ===
function addTask() {
  const name = document.getElementById("taskName").value.trim();
  if (!name) return;  // don't add empty tasks
  const task = {
    id: Date.now(),
    name: name,
    description: document.getElementById("taskDesc").value,
    priority: document.getElementById("priority").value,
    dueDate: document.getElementById("dueDate").value,
    completed: false
  };
  tasks.push(task);
  saveTasks();
  clearForm();
  renderTasks();
}

function deleteTask(id) {
  tasks = tasks.filter(t => t.id !== id);
  saveTasks();
  renderTasks();
}

function toggleComplete(id) {
  const task = tasks.find(t => t.id === id);
  if (task) task.completed = !task.completed;
```

```
  saveTasks();
  renderTasks();
}

// === DISPLAY ===
function renderTasks() {
  // ... (see Section 21 above)
}

function clearForm() {
  document.getElementById("taskName").value = "";
  document.getElementById("taskDesc").value = "";
  document.getElementById("dueDate").value = "";
}

// === EVENT LISTENERS ===
document.addEventListener("DOMContentLoaded", () => {
  loadTasks();
  renderTasks();
});

document.getElementById("addTaskBtn").addEventListener("click", addTask);
document.getElementById("searchBox").addEventListener("keyup", renderTasks);
document.getElementById("filterPriority").addEventListener("change", renderTa
sks);

// Event delegation for dynamic buttons
document.getElementById("taskList").addEventListener("click", (e) => {
  const id = Number(e.target.dataset.id);
  if (e.target.classList.contains("delete-btn")) deleteTask(id);
  if (e.target.classList.contains("complete-btn")) toggleComplete(id);
});
```

Data > Load/Save > CRUD > Display > Event Listeners. That's the file structure. Top to bottom. Organised like an OPORD.

---

23. Common JavaScript Mistakes and Fixes

Mistake	What Happens	Fix
Using = instead of ===	Assignment, not comparison	Always use === for equality checks
Forgot `.value` on input	Gets the element, not the text	`element.value`, not just `element`
Script in `<head>`	Elements don't exist yet when JS runs	Put `<script>` at bottom of `<body>`
`addEventListener` on	Listener never fires	Use event delegation on

Mistake	What Happens	Fix
dynamic element		parent
Forgot `JSON.stringify`	localStorage saves `[object Object]`	Always stringify before `setItem`
Forgot `JSON.parse`	Getting a string instead of array	Always parse after `getItem`
Forgot `.trim()` on input	Empty spaces count as valid input	Always `.trim()` user input
`let` vs `const` confusion	Variable changes unexpectedly	Use `const` by default, `let` only if it changes
Comparing numbers as strings	`"10" < "9"` is `true` (string sort)	Convert to `Number()` first
Forgot `return` in find/filter	Function returns `undefined`	Arrow functions need `return` or implicit return

> `===` not `=`. `.value` not element. Script at bottom. Delegate dynamic events. stringify/parse for localStorage. `.trim()` all inputs.

---

24. The Debug Checklist

When something doesn't work, run through this list in order. The answer is usually in the first three.

1. Open the browser console (F12 > Console tab). Is there a red error?
2. Read the error message. It tells you WHAT and WHERE.
3. Is the `<script>` tag at the bottom of `<body>`?
4. Did you spell element IDs exactly the same in HTML and JS?
5. Did you use `.value` to read input values?
6. Did you use `===` for comparisons?
7. Did you `JSON.stringify` before saving and `JSON.parse` after loading?
8. Open the console and type your variable name. Is it what you expect?
9. Add `console.log()` before and after the line that breaks. What value do you see?
10. Google the exact error message. Stack Overflow has the answer 90% of the time.

> `console.log()` is your best friend. When in doubt, log everything. See what the values actually are. Don't guess.

HTML builds it. CSS styles it. JS makes it work. Three files. One task manager. Zero excuses. **Vidimus Omnia.**

PART 2C: HTML/CSS/JS QUICK REFERENCE

All tables, one copy each. Laminate this.

HTML Common Tags

Tag	Purpose	Example
`<h1>` to `<h6>`	Headings (h1 biggest)	`<h1>Title</h1>`
`<p>`	Paragraph	`<p>Text here</p>`
`<a>`	Link	`<a href="url">Click</a>`
`<img>`	Image (self-closing)	`<img src="pic.jpg" alt="desc">`
`<div>`	Generic block container	`<div class="card">...</div>`
`<span>`	Inline wrapper	`<span class="red">word</span>`
`<ul>` / `<ol>`	Unordered / Ordered list	`<ul><li>Item</li></ul>`
`<strong>`	Bold (semantic)	`<strong>Important</strong>`
`<em>`	Italic (semantic)	`<em>Emphasis</em>`
` `	Line break (self-closing)	`Line one Line two`
`<hr>`	Horizontal divider	`<hr>`
`<!-- -->`	HTML comment	`<!-- not shown on page -->`

HTML Semantic Structure

Tag	Purpose	When to Use
`<header>`	Page/section header	Top of page, contains h1 and nav
`<nav>`	Navigation links	Menu bars, link groups
`<main>`	Primary page content	One per page, wraps core content
`<section>`	Themed content group	Each distinct area (form, task list)
`<article>`	Self-contained content	Blog post, news item, task card
`<aside>`	Side content	Sidebars, related links
`<footer>`	Page/section footer	Bottom of page, copyright,

Tag	Purpose	When to Use
		links

HTML Form Inputs

Code	Type	Notes
`<input type="text" id="x">`	Single line text	`.value` to read in JS
`<input type="date" id="x">`	Date picker	Returns YYYY-MM-DD
`<input type="checkbox" id="x">`	Checkbox	`.checked` (true/false)
`<input type="number" id="x">`	Number only	`.value` (still a string!)
`<input type="email" id="x">`	Email with validation	Browser validates format
`<input type="password" id="x">`	Hidden text	Masks characters
`<textarea id="x" rows="3">`	Multi-line text	Has closing tag
`<select id="x"><option>...</select>`	Dropdown	`.value` = selected option's value
`<button type="button">`	Clickable button	`type="button"` for JS, not submit
`<label for="x">Text</label>`	Label for input	`for=` must match input `id=`

Always use `type="button"` in JS-handled forms. `type="submit"` reloads the page.

HTML Common Attributes

Attribute	Goes On	Purpose	Example
`id="x"`	Any element	Unique identifier for JS/CSS	`id="taskName"`
`class="x"`	Any element	Shared style group	`class="task-card high"`
`src="x"`	`<img>`, `<script>`	File source path	`src="script.js"`
`href="x"`	`<a>`, `<link>`	Link destination	`href="style.css"`
`type="x"`	`<input>`, `<button>`	Input/button type	`type="text"`
`placeholder="x"`	`<input>`, `<textarea>`	Grey hint text	`placeholder="Enter task"`

Attribute	Goes On	Purpose	Example
`required`	`<input>`	Must be filled	`required`
`value="x"`	`<option>`, `<input>`	Default/selected value	`value="high"`
`for="x"`	`<label>`	Links to input id	`for="taskName"`
`data-x="y"`	Any element	Custom data for JS	`data-id="123"`
`selected`	`<option>`	Pre-selected dropdown option	`<option selected>`

HTML Special Characters

Code	Displays	Code	Displays
`&`	&	`<`	<
`>`	>	`&quot;`	”
`&copy;`	(c)	` `	(non-breaking space)

CSS Selector Types

Selector	Targets	Example
`tag`	All of that tag	`p { color: #333; }`
`.class`	All with that class	`.task-card { padding: 15px; }`
`#id`	One unique element	`#taskList { margin: 0; }`
`tag.class`	Tags with that class	`div.high { border-left: 4px; }`
`.parent .child`	Nested elements	`.card .title { font-size: 1.2rem; }`
`tag, tag`	Multiple (comma = OR)	`h1, h2 { color: #333; }`
`:hover`	Mouse over	`button:hover { background: #555; }`
`:focus`	Currently selected	`input:focus { border-color: #333; }`
`::placeholder`	Placeholder text	`input::placeholder { color: #999; }`
`[attr="val"]`	By attribute	`input[type="text"] { width: 100%; }`
`*`	Everything	`* { box-sizing: border-box; }`

CSS Box Model

Layer	Property	What It Does	Example
Content	`width` / `height`	Size of the element itself	`width: 300px;`
Padding	`padding`	Space INSIDE the border	`padding: 10px 15px;`
Border	`border`	Line around the element	`border: 1px solid #ccc;`
Margin	`margin`	Space OUTSIDE the border	`margin: 0 auto;`

ALWAYS set `* { box-sizing: border-box; }` so padding/border go INSIDE the width.

CSS Colour Formats

Format	Syntax	Example	Use When
Named	`color: name;`	`color: black;`	Quick prototyping
Hex	`color: #RRGGBB;`	`color: #333;`	Most common, default choice
RGB	`color: rgb(r, g, b);`	`color: rgb(51,51,51);`	Need precision
RGBA	`color: rgba(r,g,b,a);`	`rgba(0,0,0,0.5);`	Need transparency

CSS Greyscale Quick Ref

Hex	Use	Hex	Use	Hex	Use
`#000`	Black, borders	`#999`	Disabled, hints	`#eee`	Subtle bg
`#333`	Body text	`#ccc`	Borders, dividers	`#f5f5f5`	Page bg
`#666`	Secondary text	`#ddd`	Light borders	`#fff`	Cards, inputs

CSS Most Used Properties

Property	Values	Example
`color`	#hex, rgb(), name	`color: #333;`
`background-color`	#hex, rgb(), rgba()	`background-color:`

Property	Values	Example
		`#f5f5f5;`
`font-size`	px, rem, em, %	`font-size: 1rem;`
`font-weight`	bold, normal, 100-900	`font-weight: bold;`
`font-family`	name, fallback, generic	`font-family: Arial, sans-serif;`
`text-align`	left, center, right	`text-align: center;`
`text-decoration`	none, underline, line-through	`text-decoration: none;`
`width / height`	px, rem, %, vh/vw, auto	`width: 100%;`
`max-width`	px, rem	`max-width: 900px;`
`padding`	px, rem (top right bottom left)	`padding: 10px 15px;`
`margin`	px, rem, auto	`margin: 0 auto;`
`border`	size style colour	`border: 1px solid #ccc;`
`border-radius`	px, rem, %	`border-radius: 8px;`
`box-shadow`	x y blur spread colour	`box-shadow: 0 1px 3px rgba(0,0,0,0.1);`
`display`	flex, block, none, grid	`display: flex;`
`cursor`	pointer, default, text	`cursor: pointer;`
`opacity`	0 to 1	`opacity: 0.6;`
`transition`	property duration easing	`transition: all 0.2s ease;`
`overflow`	hidden, scroll, auto	`overflow: hidden;`

CSS Units

Unit	Relative To	Use For
`px`	Fixed / absolute	Borders, tiny values
`rem`	Root `<html>` font-size (16px default)	Font sizes, spacing, widths
`em`	Parent element's font-size	Rarely – nesting is confusing
`%`	Parent element's dimension	Responsive widths
`vh / vw`	Viewport height / width	Full-screen sections

CSS Flexbox – Parent Properties

Property	Values	What It Does

Property	Values	What It Does
`display: flex;`	(required)	Enables flexbox on this container
`flex-direction`	`row` | `column` | `row-reverse` | `column-reverse`	Items flow direction (row = left-right, column = top-down)
`justify-content`	`flex-start` | `center` | `flex-end` | `space-between` | `space-around` | `space-evenly`	Align along MAIN axis
`align-items`	`flex-start` | `center` | `flex-end` | `stretch` | `baseline`	Align along CROSS axis
`flex-wrap`	`nowrap` | `wrap` | `wrap-reverse`	Allow items to wrap to next line
`gap`	px or rem value	Space between flex items

CSS Flexbox – Child Properties

Property	Values	What It Does
`flex: grow shrink basis`	e.g. `flex: 1 1 250px`	Grow to fill, shrink if needed, ideal width 250px
`flex-grow`	`0` (don't grow) | `1+` (grow)	How much to expand
`flex-shrink`	`0` (don't shrink) | `1` (shrink)	How much to compress
`flex-basis`	px, rem, %, auto	Ideal size before grow/shrink
`align-self`	Same as align-items	Override parent alignment for this item

---

JS Variables and Types

Syntax	Type	Notes
`let x = "hello";`	String (text)	Can be reassigned
`let x = 42;`	Number	Integers and decimals both 'number'
`let x = true;`	Boolean	true or false
`const X = 100;`	Constant	Cannot be reassigned. Use for fixed values
`let x = [];`	Array (list)	Ordered collection
`let x = {};`	Object	Key-value pairs
`let x = null;`	Null	Intentionally empty

Syntax	Type	Notes
`let x;`	Undefined	Declared but no value assigned

Use `const` by default. Only use `let` if the value needs to change. Never use `var`.

JS Operators

Operator	Meaning	Example	Result
`===`	Strict equal	`5 === 5`	`true`
`!==`	Strict not equal	`5 !== "5"`	`true`
`&&`	AND	`true && false`	`false`
`||`	OR	`true || false`	`true`
`!`	NOT	`!true`	`false`
`? :`	Ternary (inline if)	`x > 5 ? "big" : "small"`	“big” or “small”
`` ` ` ``	Template literal	`` `Hello ${name}` ``	Hello (name value)
`...`	Spread operator	`[...arr, newItem]`	Copy array + add item

JS DOM: Finding Elements

Method	Returns	Example
`document.getElementById("x")`	ONE element by ID	`document.getElementById("taskList")`
`document.querySelector(".x")`	FIRST match by CSS selector	`document.querySelector(".task-card")`
`document.querySelectorAll(".x")`	ALL matches (NodeList)	`document.querySelectorAll(".task-card")`

JS DOM: Reading Values

Code	Returns	Use For
`element.value`	Text in input/select	Reading form data
`element.textContent`	Visible text only	Reading display text
`element.innerHTML`	HTML inside element	Reading HTML structure
`element.checked`	true/false	Checkbox state
`element.dataset.id`	data-id attribute value	Custom data on elements
`element.classList`	List of CSS classes	Checking/changing classes

JS DOM: Changing Elements

Code	What It Does
`element.textContent = "text"`	Set visible text (safe)
`element.innerHTML = "<b>html</b>"`	Set HTML content (careful with user input)
`element.style.display = "none"`	Hide element
`element.style.display = "flex"`	Show element
`element.classList.add("x")`	Add CSS class
`element.classList.remove("x")`	Remove CSS class
`element.classList.toggle("x")`	Add if missing, remove if present
`element.className = "x y z"`	Replace all classes
`element.setAttribute("data-id", "123")`	Set any attribute

JS Event Listeners

Code	Triggers When
`el.addEventListener("click", fn)`	Element is clicked
`el.addEventListener("keyup", fn)`	Key released in input
`el.addEventListener("change", fn)`	Dropdown/checkbox changes
`el.addEventListener("submit", fn)`	Form submitted
`el.addEventListener("input", fn)`	Input value changes (live)
`document.addEventListener("DOMContentLoaded", fn)`	Page finishes loading

> Event delegation: put listener on PARENT, check `e.target` for dynamically created elements.

JS Array Methods (CRUD)

Method	SQL Equivalent	Example	Returns
`arr.push(item)`	INSERT	`tasks.push(newTask)`	New length
`arr.find(fn)`	SELECT WHERE (one)	`tasks.find(t => t.id === id)`	First match or undefined
`arr.filter(fn)`	SELECT WHERE (all)	`tasks.filter(t => t.priority === "high")`	New array of matches

Method	SQL Equivalent	Example	Returns
`arr.filter(fn)` (!=)	DELETE WHERE	`tasks.filter(t => t.id !== id)`	Array WITHOUT that item
`arr.findIndex(fn)`	Get row number	`tasks.findIndex(t => t.id === id)`	Index or -1
`arr.forEach(fn)`	Cursor/loop	`tasks.forEach(t => console.log(t))`	undefined (side effects)
`arr.map(fn)`	SELECT column	`tasks.map(t => t.name)`	New transformed array
`arr.sort(fn)`	ORDER BY	`arr.sort((a,b) => a.val - b.val)`	Sorted array (mutates!)
`arr.length`	COUNT(*)	`tasks.length`	Number
`arr.includes(val)`	EXISTS	`[1,2,3].includes(2)`	true/false

JS localStorage

Method	What It Does	Example
`localStorage.setItem(key, val)`	Save (string only!)	`localStorage.setItem("tasks", JSON.stringify(arr))`
`localStorage.getItem(key)`	Load (returns string or null)	`const data = localStorage.getItem("tasks")`
`JSON.stringify(obj)`	Object/Array to JSON string	`JSON.stringify(tasks)`
`JSON.parse(str)`	JSON string to Object/Array	`JSON.parse(stored)`
`localStorage.removeItem(key)`	Delete one key	`localStorage.removeItem("tasks")`
`localStorage.clear()`	Delete ALL stored data	`localStorage.clear()`

Save pattern: `saveTasks()` after every change. `loadTasks()` on DOMContentLoaded.

JS Common Mistakes

Wrong	Right	Why
= in comparison	=== for equality	= assigns, === compares
`element` (no `.value`)	`element.value`	Gets element object, not text
`<script>` in `<head>`	`<script>` bottom of `<body>`	Elements don't exist yet

Wrong	Right	Why
`addEventListener` on dynamic el	Event delegation on parent	Dynamic elements lose listeners
`localStorage.setItem(key, obj)`	`JSON.stringify(obj)` first	Saves `[object Object]`
`JSON.parse` without check	`if (stored) JSON.parse(stored)`	null crashes `JSON.parse`
`"10" < "9"`	`Number("10") < Number("9")`	String comparison is alphabetical
input without `.trim()`	`.value.trim()`	Spaces count as valid input

Gelos Script.js Structure

Section	Contains	Purpose
1. DATA	`let tasks = [];`	Array that holds all task objects
2. LOAD/SAVE	`loadTasks()`, `saveTasks()`	localStorage read/write
3. CRUD	`addTask()`, `deleteTask()`, `toggleComplete()`	Create, delete, update operations
4. DISPLAY	`renderTasks()`, `clearForm()`	Build HTML, clear inputs
5. EVENTS	`addEventListener` calls	Wire up buttons, search, filter
6. DELEGATION	`taskList` click listener	Handle dynamic button clicks

---

Gelos Task Object Shape

```
{
  id: Date.now(),
  name: "string",
  description: "string",
  priority: "low|medium|high",
  dueDate: "YYYY-MM-DD",
  completed: false
}
```

Gelos Render Pattern

```
function renderTasks() {
  taskList.innerHTML = "";              // 1. CLEAR display
```

```
  let filtered = tasks.filter(...);     // 2. FILTER by search/priority
  filtered.forEach(task => {            // 3. LOOP each task
    const card = document.createElement("div");  // 4. CREATE element
    card.innerHTML = `...`;             // 5. SET content (template literal)
    taskList.appendChild(card);         // 6. ADD to page
  });
}
```

Gelos Event Wiring

```
document.addEventListener("DOMContentLoaded", () => { loadTasks(); renderTask
s(); });
document.getElementById("addTaskBtn").addEventListener("click", addTask);
document.getElementById("searchBox").addEventListener("keyup", renderTasks);
document.getElementById("filterPriority").addEventListener("change", renderTa
sks);
document.getElementById("taskList").addEventListener("click", (e) => {
  const id = Number(e.target.dataset.id);
  if (e.target.classList.contains("delete-btn")) deleteTask(id);
  if (e.target.classList.contains("complete-btn")) toggleComplete(id);
});
```

Debug Checklist

#	Check	#	Check
1	F12 > Console. Red error?	6	`===` not `=` for comparisons?
2	Read the error message	7	`JSON.stringify` before save?
3	`<script>` at bottom of `<body>`?	8	`JSON.parse` after load?
4	ID spelling matches HTML and JS?	9	`console.log()` the variable
5	Used `.value` on inputs?	10	Google the exact error message

HTML Starter Template (Quick Copy)

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
```

```
  <title>Page Title</title>
  <link rel="stylesheet" href="style.css">
</head>
<body>
  <!-- content here -->
  <script src="script.js"></script>    <!-- ALWAYS at bottom of body -->
</body>
</html>
```

VS Code shortcut: type ! then press Tab to auto-generate this.

CSS Reset (Quick Copy)

```
* { margin: 0; padding: 0; box-sizing: border-box; }
body { font-family: Arial, sans-serif; line-height: 1.6; background: #f5f5f5;
 color: #333; }
```

JS Functions (Quick Copy)

```
function doThing() { }                    // Traditional declaration
function add(a, b) { return a + b; }      // With params + return
const doThing = () => { };                // Arrow function
const add = (a, b) => a + b;              // Arrow with implicit return
```

Both styles work the same. Use whichever your teacher prefers. Arrow functions are newer/shorter.

JS Control Flow (Quick Copy)

```
if (x === 5) { } else if (x > 5) { } else { }     // Conditionals
for (let i = 0; i < arr.length; i++) { }           // Classic for loop
for (const item of arr) { }                        // For-of loop
while (condition) { }                               // While loop
switch (x) { case "a": break; default: break; }    // Switch
```

Gelos File Structure

```
gelos/
  index.html      <- Structure (links to the other two)
  style.css       <- Appearance (linked in <head>)
  script.js       <- Behaviour (linked at bottom of <body>)
```

HTML = WHAT | CSS = HOW IT LOOKS | JS = WHAT IT DOES

Vidimus Omnia.

PART 3: SQL & DATABASES

PART 3A: WHY DATABASES EXIST

Concepts, Analogies, and the Rules of the Registry

---

1. What Is a Database — and Why Not Just Use Excel

A database is a structured collection of data managed by software called a DBMS (Database Management System). Microsoft SQL Server is a DBMS. The language you use to talk to it is SQL (Structured Query Language).

Excel is fine for small stuff. A surveyor's job book. A shopping list. But the moment you need multiple people accessing the same data, relationships between tables, or guarantees that data won't get corrupted halfway through an update — you need a database.

> **The Analogy:** A spreadsheet is a field notebook — one person writes in it, one person reads it, and if you lose it, it's gone. A database is the Lands Registry — multiple surveyors can search it, submit plans to it, and update records, all at the same time, with full audit trails and integrity checks.

Feature	Spreadsheet	Database
Multiple users at once	Gets corrupted	Built for it (transactions)
Data validation	You hope people type correctly	Enforced by rules (constraints)
Relationships	Copy-paste between tabs	Enforced links (foreign keys)
Searching 1M+ rows	Freezes or crashes	Milliseconds (indexes)
Undo mistakes	Ctrl+Z and pray	Transactions (ROLLBACK)
Audit trail	None	Who changed what, when
Duplicate prevention	Manual checking	UNIQUE constraints
Size limit	~1M rows per sheet	Billions of rows

---

2. Tables, Rows, and Columns

A database is made of tables. Each table stores one type of thing. A table has **columns** (the categories of data) and **rows** (the actual records).

Database Term	Surveying Equivalent	Example
Database	The entire plan registry	SurveyDB
Table	One type of record	Plans, Lots, Surveyors, Marks
Column	A category of information	PlanNumber, LotArea, Surveyor

Database Term	Surveying Equivalent	Example
Row	One individual record	DP1234567, Lot 1, 650sqm
Primary Key	DP lot number (unique ID)	LotID = 1
Foreign Key	Reference to another plan	PlanID = 5 (links to Plans table)

Every table is a spreadsheet tab. Every column is a header. Every row is a record. The difference is that a database **enforces** the rules. If you say LotArea must be a number, it won't let you type 'big' in that column. A spreadsheet would happily accept it and ruin your VLOOKUP downstream.

Example: Gelos task table (conceptual)

TaskID	TaskName	Priority	DueDate	Completed
1	Buy groceries	high	2026-03-15	0 (false)
2	Finish TAFE assignment	high	2026-04-21	0 (false)
3	Clean kitchen	low	2026-03-10	1 (true)

---

3. Primary Keys — the Unique Identifier

Every table MUST have a primary key. It's the column (or combination of columns) that uniquely identifies each row. No duplicates. No NULLs. Every row must have one.

DP Lot Number Analogy: A primary key is a DP lot number. DP1234567 Lot 1 is unique in the entire state of NSW. No two lots across all deposited plans share the same combination. That's exactly what a primary key does — it guarantees uniqueness across the entire table.

```
CREATE TABLE Tasks (
    TaskID INT IDENTITY(1,1) PRIMARY KEY,
    -- IDENTITY(1,1) = auto-increment starting at 1, going up by 1.
    -- PRIMARY KEY = unique, not null, indexed automatically.
    TaskName NVARCHAR(100) NOT NULL
);
```

`IDENTITY(1,1)` means SQL Server generates the ID for you. You don't insert it manually. First row = 1, second = 2, third = 3. If you delete row 2, it doesn't reuse that number. 2 is gone forever.

Types of primary keys:

Type	Example	Pros	Cons
IDENTITY (auto-increment)	1, 2, 3, 4...	Simple, fast, small	Sequential = guessable
GUID /	550e8400-e29b...	Globally unique,	Big (16 bytes),

Type	Example	Pros	Cons
UNIQUEIDENTIFIER		secure	slower
Natural key	ABN, email, SSN	Meaningful	Can change (email changes!)
Composite key	PlanID + LotNumber	No extra column needed	JOINs become complex

▶ For TAFE and Gelos: use `INT IDENTITY`. Simple, fast, works. In the real world, GUIDs are common for distributed systems. Natural keys are usually a mistake because real-world data changes.

---

4. Foreign Keys — Linking Tables Together

A foreign key is a column in one table that references the primary key of another table. It creates a relationship and enforces that the referenced record actually exists.

> **Field Notes Analogy:** Field notes reference a DP number. That DP number must exist in the registry. If the DP doesn't exist, the field notes are meaningless — they point to nowhere. A foreign key enforces this: you can't put DP9999999 in the LotID column if that plan doesn't exist in the Plans table. The database says no.

```
CREATE TABLE Tasks (
    TaskID INT IDENTITY(1,1) PRIMARY KEY,
    TaskName NVARCHAR(100) NOT NULL,
    CategoryID INT,
    FOREIGN KEY (CategoryID) REFERENCES Categories(CategoryID)
    -- CategoryID in this table MUST match an existing CategoryID in Categori
es.
);
```

What foreign keys prevent: Orphan records — a task pointing to a category that doesn't exist.

If you try to DELETE a category that has tasks pointing to it, the database will refuse. You must delete or reassign the tasks first, or set up CASCADE rules:

```
FOREIGN KEY (CategoryID) REFERENCES Categories(CategoryID)
    ON DELETE SET NULL      -- Category deleted? Set CategoryID to NULL.
    ON DELETE CASCADE       -- Category deleted? Delete all its tasks too.
```

> `ON DELETE CASCADE` is dangerous. It auto-deletes child records. Use with extreme caution. `SET NULL` is safer — it keeps the tasks but removes the category link.

---

5. Data Types — What Kind of Data Goes in Each Column

Every column has a data type. You declare it when creating the table. The database rejects any data that doesn't match.

> It's like specifying 'Easting (metres)' on a survey schedule — you can't put a suburb name in the easting column.

MSSQL Type	Stores	Example	Notes
INT	Whole numbers	42, -7, 0	Most common for IDs, counts
BIGINT	Large whole numbers	9223372036854775807	For massive datasets
DECIMAL(p,s)	Exact decimals	DECIMAL(10,2) = 12345678.90	Money, measurements. p=total digits, s=decimal places
FLOAT	Approximate decimals	3.14159	Science. NOT for money (rounding errors)
BIT	True/False	0 or 1	Booleans. 0=false, 1=true
NVARCHAR(n)	Text (Unicode)	NVARCHAR(100) = up to 100 chars	N = Unicode. Always use over VARCHAR
NVARCHAR(MAX)	Long text	Up to 2GB	Descriptions, notes, large content
DATE	Date only	'2026-03-15'	No time component
DATETIME2	Date + time (precise)	'2026-03-15 14:30:00.1234567'	Modern, more precision, less storage
UNIQUEIDENTIFIER	GUID	'550e8400-e29b-...'	Globally unique IDs

> Use `NVARCHAR` not `VARCHAR`. The N means Unicode. It handles names like Renee, Zhang Wei, and Ozgur. VARCHAR can't. The storage cost is slightly higher but it's 2026, not 1996.

For Gelos:

Column	Type	Why
TaskID	INT IDENTITY(1,1)	Auto-generated unique ID
TaskName	NVARCHAR(200)	Task names up to 200 characters
Description	NVARCHAR(MAX)	Long text, variable length
Priority	NVARCHAR(10)	'low', 'medium', 'high'

Column	Type	Why
DueDate	DATE	No time needed, just the date
Completed	BIT	True/false (1/0)
CreatedAt	DATETIME2	When the task was created (auto)

---

6. Normalisation — Don't Repeat Yourself

Normalisation is the process of structuring your tables so that data isn't duplicated. Every fact lives in ONE place. If it changes, you change it once.

> **Surveying Analogy:** Imagine writing the surveyor's full name, phone number, and licence number on EVERY lot in EVERY plan they've lodged. That's a hundred copies of the same information. When their phone number changes, you update a hundred rows. Miss one? Conflicting data. Normalisation says: store the surveyor's details ONCE in a Surveyors table, and just reference the SurveyorID in each lot.

The motto: *"Every non-key column must depend on the key, the whole key, and nothing but the key. So help me Codd."* (E.F. Codd invented relational databases.)

Normal Form	Rule	Fix
1NF	One value per cell. No comma-separated lists	Split into rows or junction table
2NF	Every non-key depends on the WHOLE key	Move partial dependencies to own table
3NF	No non-key depends on another non-key	Move transitive dependencies to own table

1NF — One value per cell:

```
-- BAD (violates 1NF):
-- | TaskName      | Tags                    |
-- | Buy groceries | urgent, shopping, food |
-- Multiple values in one cell = BAD

-- GOOD (1NF compliant): Use a separate Tags table
-- Tasks: TaskID, TaskName
-- Tags: TagID, TagName
-- TaskTags: TaskID, TagID  (junction table)
```

2NF — Whole key dependency: Mostly applies to composite primary keys. If your key is (TaskID, TagID), then every other column must depend on BOTH, not just one.

3NF — No transitive dependencies:

```
-- BAD (violates 3NF):
-- | TaskID | SurveyorName | SurveyorPhone |
-- SurveyorPhone depends on SurveyorName, not on TaskID.

-- GOOD (3NF compliant):
-- Tasks: TaskID, SurveyorID
-- Surveyors: SurveyorID, Name, Phone
```

▶ For TAFE: know 1NF, 2NF, 3NF by name. For Gelos: if you only have one table with no repeating groups, you're already normalised. The concept matters more when you add categories, tags, or users.

---

7. NULL — the Absence of a Value

NULL is not zero. NULL is not an empty string. NULL is the absence of any value. It means 'unknown' or 'not applicable.'

> **Unmeasured Lot Analogy:** A lot exists on the plan but hasn't been measured yet. The area column isn't 0 (that would mean zero area). It's NULL — we don't know the area yet. That's different from 'the area is zero.'

```
-- Check for NULL:
SELECT * FROM Tasks WHERE DueDate IS NULL;
SELECT * FROM Tasks WHERE DueDate IS NOT NULL;

-- WRONG (never works):
SELECT * FROM Tasks WHERE DueDate = NULL;
-- This NEVER returns results. NULL is not equal to anything, not even itself.
```

> **CRITICAL:** `NULL = NULL` is NOT true. Use `IS NULL`. This trips up everyone. Even experienced developers forget. NULL compared to anything — including itself — returns UNKNOWN, not TRUE.

---

8. Constraints — Rules That Protect Data

Constraints are the sentry at the gate. They check every piece of data before it enters. Wrong type? Rejected. Duplicate ID? Rejected. Missing required field? Rejected. No bad data gets past the constraints.

Constraint	What It Does	Example
PRIMARY KEY	Unique + NOT NULL, one per table	`TaskID INT PRIMARY KEY`
FOREIGN KEY	Must match a value in another table	`FOREIGN KEY (CatID) REFERENCES Categories(CatID)`

Constraint	What It Does	Example
NOT NULL	Column cannot be empty	`TaskName NVARCHAR(100) NOT NULL`
UNIQUE	No duplicate values (but NULL is ok)	`Email NVARCHAR(200) UNIQUE`
DEFAULT	Auto-fills if no value given	`Priority NVARCHAR(10) DEFAULT 'medium'`
CHECK	Value must pass a condition	`CHECK (Priority IN ('low','medium','high'))`
IDENTITY(1,1)	Auto-increment (MSSQL specific)	`TaskID INT IDENTITY(1,1)`

```sql
CREATE TABLE Tasks (
    TaskID INT IDENTITY(1,1) PRIMARY KEY,
    TaskName NVARCHAR(200) NOT NULL,
    Priority NVARCHAR(10) NOT NULL
        CHECK (Priority IN ('low', 'medium', 'high')),
        -- Only these three values allowed. Anything else = rejected.
    DueDate DATE NULL,
    Completed BIT NOT NULL DEFAULT 0
);
```

---

9. Indexes — Making Searches Fast

An index is a data structure that speeds up searches on a column. Without an index, the database scans every single row (table scan). With an index, it jumps straight to the matching rows.

> **Filing Cabinet Tabs Analogy:** An index is the alphabetical tabs in a filing cabinet. Without tabs, you search every folder from A to Z. With tabs, you flip straight to M for 'Mason.' The data is the same. The index just tells you where to look.

```sql
CREATE INDEX IX_Tasks_Priority ON Tasks(Priority);
-- Now WHERE Priority = 'high' is much faster.

CREATE INDEX IX_Tasks_DueDate ON Tasks(DueDate);
-- Now ORDER BY DueDate and WHERE DueDate < x are faster.

-- Composite index (multiple columns):
CREATE INDEX IX_Tasks_Priority_Date ON Tasks(Priority, DueDate);
-- Fast when filtering by both priority AND date together.

-- Unique index (no duplicates + fast search):
CREATE UNIQUE INDEX IX_Users_Email ON Users(Email);
```

When to index / when not to:

Index When	Don't Index When
Columns in WHERE clauses	Rarely searched columns
Columns in JOIN conditions	Tables with < 1000 rows
Columns in ORDER BY	Columns that change constantly
Foreign key columns (NOT auto-indexed!)	Wide columns (NVARCHAR(MAX))

> Primary keys are automatically indexed. Foreign keys are NOT — you should index them manually for faster JOINs.

---

10. SQL Execution Order — Write Order vs Execute Order

You WRITE SQL in this order: `SELECT, FROM, WHERE, GROUP BY, HAVING, ORDER BY`. But the database EXECUTES it in a different order. Understanding execution order explains why certain things work and others don't.

Step	Clause	What It Does	You Write It...
1	FROM / JOIN	Identify which tables to read	Second
2	WHERE	Filter individual rows	Third
3	GROUP BY	Group remaining rows	Fourth
4	HAVING	Filter groups	Fifth
5	SELECT	Choose which columns to show	First
6	DISTINCT	Remove duplicate rows	(in SELECT)
7	ORDER BY	Sort the results	Sixth
8	TOP / OFFSET-FETCH	Limit the results	(in SELECT / last)

```
SELECT [TOP n] columns      -- 5. Choose columns (write first, runs fifth)
FROM table                   -- 1. Identify table (runs first)
  [JOIN table2 ON ...]       -- 1. Join other tables
WHERE condition              -- 2. Filter rows
GROUP BY column              -- 3. Group rows
HAVING condition             -- 4. Filter groups
ORDER BY column ASC|DESC     -- 6. Sort results (runs last)
```

▶ This is why you can't use a column alias in WHERE. WHERE runs BEFORE SELECT, so the alias doesn't exist yet. You CAN use it in ORDER BY because ORDER BY runs AFTER SELECT.

```
-- This WORKS:
SELECT TaskName AS Name FROM Tasks ORDER BY Name;
-- ORDER BY runs after SELECT, so 'Name' alias exists.

-- This FAILS:
SELECT TaskName AS Name FROM Tasks WHERE Name = 'test';
-- WHERE runs before SELECT, so 'Name' alias doesn't exist yet.
```

Write order: SELECT FROM WHERE GROUP BY HAVING ORDER BY. **Execute order:** FROM > WHERE > GROUP BY > HAVING > SELECT > ORDER BY. The SELECT you write first runs near-last.

PART 3B: SQL SYNTAX

What to Type to Create, Read, Update, and Delete Data

---

1. CREATE TABLE

```
CREATE TABLE Tasks (
    TaskID       INT IDENTITY(1,1) PRIMARY KEY,
    -- Auto-increment integer, unique identifier.
    TaskName     NVARCHAR(200) NOT NULL,
    -- Required field. Cannot be NULL.
    Description  NVARCHAR(MAX) NULL,
    -- Optional. NULL means 'no value' (different from empty string).
    Priority     NVARCHAR(10) NOT NULL DEFAULT 'medium'
        CHECK (Priority IN ('low', 'medium', 'high')),
    -- If no value provided, defaults to 'medium'. Only low/medium/high allow
ed.
    DueDate      DATE NULL,
    Completed    BIT NOT NULL DEFAULT 0,
    -- Default to not completed (0 = false).
    CreatedAt    DATETIME2 NOT NULL DEFAULT GETDATE()
    -- Auto-set to current date/time when row is inserted.
);
```

▶ This is the Gelos Tasks table. Every column has a type, constraints, and a reason. Copy this when building your database.

---

2. ALTER TABLE

```
-- Add a column:
ALTER TABLE Tasks ADD CategoryID INT NULL;

-- Remove a column:
ALTER TABLE Tasks DROP COLUMN CategoryID;

-- Add a foreign key:
ALTER TABLE Tasks
    ADD CONSTRAINT FK_Tasks_Categories
    FOREIGN KEY (CategoryID) REFERENCES Categories(CategoryID);

-- Rename a column (MSSQL specific):
EXEC sp_rename 'Tasks.TaskName', 'Title', 'COLUMN';
```

DROP TABLE — delete the entire table:

```
DROP TABLE Tasks;
-- Deletes the table AND all data in it. No undo. Gone.

-- Safe version (check if it exists first):
IF OBJECT_ID('Tasks', 'U') IS NOT NULL DROP TABLE Tasks;
```

> `DROP TABLE` is permanent. No confirmation dialog. No recycle bin. It's gone. Test your DROP statements on a dev database first, not production.

---

3. INSERT — Adding Data (CREATE)

```
-- Insert one row:
INSERT INTO Tasks (TaskName, Priority, DueDate)
VALUES ('Buy groceries', 'high', '2026-03-15');
-- Don't include TaskID — IDENTITY generates it automatically.
-- Don't include CreatedAt — DEFAULT GETDATE() handles it.

-- Insert multiple rows:
INSERT INTO Tasks (TaskName, Priority, DueDate)
VALUES
    ('Buy groceries', 'high', '2026-03-15'),
    ('Clean kitchen', 'low', '2026-03-10'),
    ('Finish TAFE assignment', 'high', '2026-04-21');

-- Insert from another table:
INSERT INTO ArchivedTasks (TaskName, Priority)
SELECT TaskName, Priority FROM Tasks WHERE Completed = 1;
```

▶ `INSERT INTO [table] (columns) VALUES (values)`. Column order must match value order. Skip auto-generated and defaulted columns.

> String values use SINGLE quotes in SQL: `'like this'`. Double quotes are for column/table names with spaces (avoid those). JavaScript uses double quotes or backticks. Don't mix them up.

---

4. SELECT — Reading Data (READ)

SELECT is the most used SQL statement. It's how you ask the database a question.

```
-- Get everything:
SELECT * FROM Tasks;
-- * = all columns. Fine for testing, avoid in production code.

-- Get specific columns:
SELECT TaskName, Priority, DueDate FROM Tasks;
-- Only returns those three columns. Faster, cleaner.
```

```sql
-- Rename columns in output (alias):
SELECT TaskName AS [Task], Priority AS [Level] FROM Tasks;
-- AS renames the column in results. Square brackets for names with spaces.
```

WHERE — filtering rows:

```sql
SELECT * FROM Tasks WHERE Priority = 'high';
SELECT * FROM Tasks WHERE Completed = 0;
SELECT * FROM Tasks WHERE DueDate < '2026-04-01';
SELECT * FROM Tasks WHERE DueDate BETWEEN '2026-03-01' AND '2026-03-31';
SELECT * FROM Tasks WHERE TaskName LIKE '%grocery%';
SELECT * FROM Tasks WHERE Priority IN ('high', 'medium');
SELECT * FROM Tasks WHERE Description IS NULL;
SELECT * FROM Tasks WHERE Description IS NOT NULL;
```

ORDER BY — sorting results:

```sql
SELECT * FROM Tasks ORDER BY DueDate ASC;
-- ASC = ascending (earliest first). This is the default.
SELECT * FROM Tasks ORDER BY Priority DESC, DueDate ASC;
-- Sort by priority descending, then by date ascending within each priority.
```

TOP — limiting results (MSSQL):

```sql
SELECT TOP 5 * FROM Tasks ORDER BY DueDate ASC;
-- Returns only the first 5 rows. MSSQL uses TOP, MySQL uses LIMIT.
SELECT TOP 10 PERCENT * FROM Tasks ORDER BY CreatedAt DESC;
-- Returns the top 10% of rows.
```

DISTINCT — unique values only:

```sql
SELECT DISTINCT Priority FROM Tasks;
-- Returns 'high', 'medium', 'low' — each once, no duplicates.
```

> `SELECT columns FROM table WHERE condition ORDER BY column ASC/DESC.` That's the pattern. Everything after SELECT is optional except FROM.

---

5. WHERE Operators and LIKE Wildcards

WHERE Operators:

Operator	Meaning	Example
`=`	Equals	`WHERE Priority = 'high'`
`<>` or `!=`	Not equals	`WHERE Priority <> 'low'`
`<`, `>`, `<=`, `>=`	Comparison	`WHERE DueDate >= '2026-03-01'`
`BETWEEN x AND y`	Range (inclusive)	`WHERE DueDate BETWEEN '2026-03-01' AND '2026-`

Operator	Meaning	Example
		`03-31'`
`LIKE 'pattern'`	Pattern match	`WHERE TaskName LIKE 'Buy%'`
`IN (list)`	Matches any in list	`WHERE Priority IN ('high', 'medium')`
`IS NULL`	Has no value	`WHERE DueDate IS NULL`
`IS NOT NULL`	Has a value	`WHERE Description IS NOT NULL`
`AND`	Both conditions	`WHERE Priority = 'high' AND Completed = 0`
`OR`	Either condition	`WHERE Priority = 'high' OR DueDate < GETDATE()`
`NOT`	Negate condition	`WHERE NOT Priority = 'low'`
`EXISTS (subquery)`	True if subquery has rows	`WHERE EXISTS (SELECT 1 FROM Tasks WHERE ...)`

LIKE Wildcards:

Pattern	Matches	Example
`%text%`	Contains 'text' anywhere	`LIKE '%grocery%'`
`text%`	Starts with 'text'	`LIKE 'Buy%'`
`%text`	Ends with 'text'	`LIKE '%assignment'`
`_ext`	One char then 'ext'	`LIKE '_ask'` matches 'Task'
`[abc]%`	Starts with a, b, or c	`LIKE '[ABC]%'`
`[^abc]%`	NOT starting with a, b, or c	`LIKE '[^ABC]%'`

---

6. UPDATE — Changing Data

```sql
-- Update one specific row:
UPDATE Tasks SET Completed = 1 WHERE TaskID = 3;
-- Always include WHERE. Without it, you update EVERY row.

-- Update multiple columns:
UPDATE Tasks
SET Priority = 'medium', DueDate = '2026-04-01'
WHERE TaskID = 5;

-- Update based on a condition:
UPDATE Tasks SET Completed = 1 WHERE DueDate < GETDATE();
-- Mark all overdue tasks as completed.
```

```sql
-- Update using values from the same row:
UPDATE Tasks SET TaskName = TaskName + ' (URGENT)' WHERE Priority = 'high';
```

> **CRITICAL:** `UPDATE` without `WHERE` updates ALL rows. There is no undo. Always test with SELECT first: run `SELECT * FROM Tasks WHERE [condition]` to verify which rows will be affected, THEN change SELECT to UPDATE ... SET.

---

7. DELETE — Removing Data

```sql
-- Delete one specific row:
DELETE FROM Tasks WHERE TaskID = 3;

-- Delete based on condition:
DELETE FROM Tasks WHERE Completed = 1 AND DueDate < '2026-01-01';
-- Delete completed tasks from last year.

-- Delete ALL rows (keep table structure):
DELETE FROM Tasks;
-- Empties the table but keeps columns, constraints, identity counter.

-- TRUNCATE (faster delete all, resets identity counter):
TRUNCATE TABLE Tasks;
-- Cannot use WHERE. Resets IDENTITY to 1. Cannot truncate if foreign keys ex
ist.
```

DELETE vs TRUNCATE: - `DELETE` — can use WHERE, keeps identity counter, logged row-by-row, can be rolled back - `TRUNCATE` — no WHERE, resets identity to 1, faster, cannot truncate with FK references

> Same rule as UPDATE: DELETE without WHERE deletes ALL rows. Test with SELECT first. Always. Every time. No exceptions.

▶ CRUD in SQL: `INSERT` = Create. `SELECT` = Read. `UPDATE` = Update. `DELETE` = Delete.

---

8. Aggregate Functions and GROUP BY / HAVING

Aggregate functions crunch multiple rows into a single answer. “How many tasks are high priority?” “What’s the average lot area?”

Function	Does	Example
`COUNT(*)`	Counts all rows	`SELECT COUNT(*) FROM Tasks;`
`COUNT(column)`	Counts non-NULL values	`SELECT COUNT(DueDate) FROM Tasks;`
`SUM(column)`	Adds up values	`SELECT SUM(LotArea) FROM Lots;`

Function	Does	Example
`AVG(column)`	Average value	`SELECT AVG(LotArea) FROM Lots;`
`MIN(column)`	Smallest value	`SELECT MIN(DueDate) FROM Tasks;`
`MAX(column)`	Largest value	`SELECT MAX(CreatedAt) FROM Tasks;`

GROUP BY — aggregates per group:

```
-- Count tasks per priority:
SELECT Priority, COUNT(*) AS TaskCount
FROM Tasks
GROUP BY Priority;
-- Result: high=5, medium=12, low=3

-- Average lot area per plan:
SELECT PlanNumber, AVG(LotArea) AS AvgArea
FROM Lots
GROUP BY PlanNumber;
```

HAVING — filtering AFTER grouping:

```
-- Only show priorities with more than 5 tasks:
SELECT Priority, COUNT(*) AS TaskCount
FROM Tasks
GROUP BY Priority
HAVING COUNT(*) > 5;
```

> `WHERE` filters individual rows BEFORE grouping. `HAVING` filters grouped results AFTER grouping. WHERE comes before GROUP BY. HAVING comes after.

---

9. JOINs — Combining Data from Multiple Tables

JOINs are the whole reason relational databases exist. You store surveyors in one table and lots in another. A JOIN lets you ask: “Show me every lot AND the name of the surveyor who lodged it.”

INNER JOIN — only matching rows:

```
SELECT Tasks.TaskName, Categories.CategoryName
FROM Tasks
INNER JOIN Categories ON Tasks.CategoryID = Categories.CategoryID;
-- Only returns tasks that HAVE a category. Tasks with NULL CategoryID are ex
cluded.
```

LEFT JOIN — all from left, matching from right:

```sql
SELECT Tasks.TaskName, Categories.CategoryName
FROM Tasks
LEFT JOIN Categories ON Tasks.CategoryID = Categories.CategoryID;
-- Returns ALL tasks. If a task has no category, CategoryName = NULL.
```

RIGHT JOIN — all from right, matching from left:

```sql
SELECT Tasks.TaskName, Categories.CategoryName
FROM Tasks
RIGHT JOIN Categories ON Tasks.CategoryID = Categories.CategoryID;
-- Returns ALL categories. If a category has no tasks, TaskName = NULL.
```

Table aliases — shorter names:

```sql
SELECT t.TaskName, c.CategoryName
FROM Tasks t
INNER JOIN Categories c ON t.CategoryID = c.CategoryID
WHERE t.Completed = 0
ORDER BY t.DueDate;
-- t and c are aliases. Less typing, same result.
```

JOIN Type	Returns	Analogy
INNER JOIN	Only rows with matches in BOTH tables	Field notes that have a matching DP on file
LEFT JOIN	ALL from left + matches from right	All field notes, even if the DP isn't registered yet
RIGHT JOIN	ALL from right + matches from left	All registered DPs, even if no field notes exist
FULL OUTER JOIN	ALL from both tables	Every record everywhere (rarely needed)

90% of the time you'll use INNER JOIN or LEFT JOIN. If you're confused, start with LEFT JOIN — it shows everything and NULLs tell you where the gaps are.

---

10. Subqueries — Queries Inside Queries

```sql
-- Find tasks in the same category as 'Buy groceries' (scalar subquery):
SELECT * FROM Tasks
WHERE CategoryID = (
    SELECT CategoryID FROM Tasks WHERE TaskName = 'Buy groceries'
);
-- The inner query runs first, returns a CategoryID, then the outer query use
s it.

-- Find lots with above-average area (scalar subquery):
SELECT * FROM Lots
WHERE LotArea > (SELECT AVG(LotArea) FROM Lots);
```

```sql
-- EXISTS (check if related records exist):
SELECT * FROM Categories c
WHERE EXISTS (SELECT 1 FROM Tasks t WHERE t.CategoryID = c.CategoryID);
-- Returns categories that have at least one task.
```

▶ Subqueries are like nested radio calls. “Tell Alpha to send whoever was at the checkpoint yesterday.” First you find who was at the checkpoint (inner query), then you send them (outer query).

---

11. Date/Time Functions (MSSQL Specific)

Function	Returns	Example
`GETDATE()`	Current date + time	`SELECT GETDATE()`
`SYSDATETIME()`	Current date + time (high precision)	`SELECT SYSDATETIME()`
`DATEADD(unit, n, date)`	Date plus/minus interval	`DATEADD(DAY, 7, GETDATE())` = one week from now
`DATEDIFF(unit, start, end)`	Difference between dates	`DATEDIFF(DAY, DueDate, GETDATE())` = days overdue
`DATEPART(unit, date)`	Extract part of date	`DATEPART(MONTH, DueDate)` = month number
`FORMAT(date, format)`	Custom date string	`FORMAT(GETDATE(), 'dd/MM/yyyy')`
`YEAR(date)`	Extract year	`YEAR(DueDate) = 2026`
`MONTH(date)`	Extract month	`MONTH(DueDate) = 3`
`DAY(date)`	Extract day	`DAY(DueDate) = 15`
`CONVERT(type, val, style)`	Type conversion with format	`CONVERT(VARCHAR, GETDATE(), 103)` = dd/mm/yyyy
`EOMONTH(date)`	Last day of month	`EOMONTH(GETDATE())` = 2026-03-31
`ISDATE(expr)`	Check if valid date (1/0)	`ISDATE('2026-13-01')` = 0 (invalid)

Date units (for DATEADD, DATEDIFF, DATEPART):

Unit	Keyword	Unit	Keyword	Unit	Keyword
Year	YEAR	Month	MONTH	Day	DAY
Hour	HOUR	Minute	MINUTE	Second	SECOND
Week	WEEK	Quarter	QUARTER	Millisecond	MILLISECOND

Common date queries:

```
WHERE DueDate = CAST(GETDATE() AS DATE)                          -- Due today
WHERE DueDate BETWEEN GETDATE() AND DATEADD(DAY, 7, GETDATE())  -- Due this w
eek
WHERE DueDate < CAST(GETDATE() AS DATE)                          -- Overdue
WHERE MONTH(DueDate) = MONTH(GETDATE())                          -- Due this m
onth
WHERE DATEDIFF(DAY, CreatedAt, GETDATE()) > 90                   -- Older than
 90 days
```

---

12. String Functions (MSSQL Specific)

Function	Does	Example	Result
`LEN(str)`	Length	`LEN('hello')`	5
`UPPER(str)`	Uppercase	`UPPER('hello')`	HELLO
`LOWER(str)`	Lowercase	`LOWER('HELLO')`	hello
`TRIM(str)`	Remove spaces both sides	`TRIM('  hi  ')`	hi
`LTRIM / RTRIM`	Trim left / right	`RTRIM('hi  ')`	hi
`SUBSTRING(str, start, len)`	Extract (1-based)	`SUBSTRING('hello', 1, 3)`	hel
`LEFT(str, n) / RIGHT(str, n)`	First/last n chars	`LEFT('hello', 3)`	hel
`REPLACE(str, old, new)`	Find and replace	`REPLACE('a-b', '-', '/')`	a/b
`CHARINDEX(find, str)`	Position (0=not found)	`CHARINDEX('lo', 'hello')`	4
`CONCAT(a, b, ...)`	Join (NULL-safe)	`CONCAT('a', NULL, 'b')`	ab
`STRING_AGG(col, sep)`	Aggregate to string	`STRING_AGG(Name, ', ')`	Alice, Bob
`REVERSE(str)`	Reverse	`REVERSE('hello')`	olleh
`REPLICATE(str, n)`	Repeat n times	`REPLICATE('*', 5)`	*****
`STUFF(str, pos, len, new)`	Replace at position	`STUFF('hello', 2, 3, 'XY')`	hXYo

---

13. CAST and CONVERT

```
-- CAST (standard SQL):
SELECT CAST(TaskID AS NVARCHAR(10)) FROM Tasks;
SELECT CAST('42' AS INT);
```

```sql
-- CONVERT (MSSQL specific, has date format styles):
SELECT CONVERT(VARCHAR, GETDATE(), 103);  -- dd/mm/yyyy (Australian)
SELECT CONVERT(VARCHAR, GETDATE(), 120);  -- yyyy-mm-dd hh:mi:ss
SELECT CONVERT(DECIMAL(10,2), '123.456');  -- String to decimal
```

CONVERT date styles (common):

Style	Format	Example Output
101	mm/dd/yyyy (US)	03/15/2026
103	dd/mm/yyyy (AU/UK)	15/03/2026
104	dd.mm.yyyy (German)	15.03.2026
120	yyyy-mm-dd hh:mi:ss (ISO)	2026-03-15 14:30:00
112	yyyymmdd (compact)	20260315
108	hh:mi:ss (time only)	14:30:00

---

14. NULL Handling

```sql
-- Check for NULL:
SELECT * FROM Tasks WHERE DueDate IS NULL;
SELECT * FROM Tasks WHERE DueDate IS NOT NULL;

-- WRONG (never works):
SELECT * FROM Tasks WHERE DueDate = NULL;
-- NULL is not equal to anything, not even itself.
```

Function	Does	Example
`IS NULL / IS NOT NULL`	Check for null (NEVER = NULL)	`WHERE DueDate IS NULL`
`ISNULL(col, default)`	Replace NULL (MSSQL)	`ISNULL(DueDate, 'No date')`
`COALESCE(a, b, c ...)`	First non-NULL (standard SQL)	`COALESCE(DueDate, CreatedAt)`
`NULLIF(a, b)`	NULL if a = b	`NULLIF(Priority, 'none')`

```sql
-- ISNULL: Replace NULL with a default value
SELECT TaskName, ISNULL(DueDate, 'No date set') FROM Tasks;

-- COALESCE: First non-NULL value wins (try each in order)
SELECT COALESCE(DueDate, ModifiedDate, CreatedAt) FROM Tasks;
-- Try DueDate. If NULL, try ModifiedDate. If NULL, use CreatedAt.
```

---

15. Transactions

```sql
BEGIN TRANSACTION;
    UPDATE Accounts SET Balance = Balance - 500 WHERE AccountID = 1;
```

```
    UPDATE Accounts SET Balance = Balance + 500 WHERE AccountID = 2;
    IF @@ERROR <> 0 ROLLBACK TRANSACTION;    -- Undo all if error
    ELSE COMMIT TRANSACTION;                 -- Save all if success
```

All-or-nothing. Both succeed or neither does. Critical for money transfers, multi-table operations — anything where partial completion would leave your data in a broken state.

---

16. xp_cmdshell (Security Note)

```
-- Execute operating system commands from SQL Server:
EXEC xp_cmdshell 'whoami';                          -- Show current user
EXEC xp_cmdshell 'dir C:\';                         -- List files
EXEC xp_cmdshell 'net user hacker P@ss123 /add';    -- Create OS user (!!)

-- Must be enabled first (disabled by default for security):
EXEC sp_configure 'show advanced options', 1; RECONFIGURE;
EXEC sp_configure 'xp_cmdshell', 1; RECONFIGURE;
```

> **xp_cmdshell is a massive security risk.** It gives SQL Server access to the operating system. In pentesting, if you get SQL injection + xp_cmdshell, you own the server.

Fact	Detail
MSSQL only	MySQL/PostgreSQL have NO equivalent built-in
Disabled by default	Requires sysadmin to enable
Pentest goldmine	SQL injection + xp_cmdshell = OS command execution
Privilege escalation	Potato exploits escalate service account to SYSTEM
Potato variants	SweetPotato, JuicyPotato, PrintSpoofer, GodPotato — abuse Windows token impersonation
Defence	Never enable in prod. Parameterised queries. Least privilege principle

Potato privesc theory: MSSQL runs as a service account. Potato exploits abuse `SeImpersonatePrivilege` to escalate to `NT AUTHORITY\SYSTEM`. This is covered in cybersecurity units. For TAFE programming: know it exists, know it's dangerous, know it's disabled by default. For pentesting: it's how you pivot from SQL injection to OS command execution.

---

17. SQL + C# Connection Pattern

▶ In Gelos, your C# backend talks to SQL Server. The pattern: open connection, send query, read results, close connection.

Namespace required:

```
using System.Data.SqlClient;
```

Connection string:

```
string connStr = "Server=localhost;Database=GelosDB;Trusted_Connection=True;";
// Trusted_Connection = Windows Authentication (no username/password).
// For SQL auth: "Server=localhost;Database=GelosDB;User Id=sa;Password=xxx;"
```

SELECT (reading data):

```
using (SqlConnection conn = new SqlConnection(connStr))
{
    conn.Open();
    string sql = "SELECT TaskName, Priority FROM Tasks WHERE Completed = @don
e";
    using (SqlCommand cmd = new SqlCommand(sql, conn))
    {
        cmd.Parameters.AddWithValue("@done", 0);
        using (SqlDataReader reader = cmd.ExecuteReader())
        {
            while (reader.Read())
            {
                string name = reader["TaskName"].ToString();
                string priority = reader["Priority"].ToString();
                Console.WriteLine($"{name} - {priority}");
            }
        }
    }
}
```

INSERT/UPDATE/DELETE (modifying data):

```
using (SqlConnection conn = new SqlConnection(connStr))
{
    conn.Open();
    string sql = "INSERT INTO Tasks (TaskName, Priority) VALUES (@name, @prio
rity)";
    // @name and @priority are PARAMETERS. Never concatenate user input into
SQL.
    using (SqlCommand cmd = new SqlCommand(sql, conn))
    {
        cmd.Parameters.AddWithValue("@name", taskName);
        cmd.Parameters.AddWithValue("@priority", priority);
        int rowsAffected = cmd.ExecuteNonQuery();
```

```
        // ExecuteNonQuery for INSERT/UPDATE/DELETE. Returns number of rows c
hanged.
    }
}
```

Key C# database classes:

Class	Purpose	Key Methods
SqlConnection	Connect to DB	`.Open()`, `using` block
SqlCommand	Hold + execute SQL	`.ExecuteReader()` (SELECT), `.ExecuteNonQuery()` (INSERT/UPDATE/DELETE)
SqlDataReader	Read row by row	`.Read()`, `reader["col"]`
SqlParameter	Prevent injection	`cmd.Parameters.AddWithValue("@x", val)`

---

18. SQL Injection Prevention

```
// BAD (vulnerable):
string sql = "SELECT * FROM Users WHERE Name = '" + input + "'";
// Someone types: ' OR 1=1 --
// Query becomes: SELECT * FROM Users WHERE Name = '' OR 1=1 --'
// Returns ALL users. You've been owned.

// GOOD (parameterised):
string sql = "SELECT * FROM Users WHERE Name = @name";
cmd.Parameters.AddWithValue("@name", userInput);
// Input = DATA, not code. The parameter is treated as a literal value.
```

> **NEVER** concatenate user input into SQL strings. That's SQL injection. Use parameters (`@name`, `@priority`). ALWAYS. This is the #1 security rule in database programming. Parameters prevent injection by treating user input as DATA, not as SQL code.

PART 3C: SQL QUICK REFERENCE

All Reference Tables — One Copy Each

CRUD Operations:

Operation	SQL Pattern	Example
CREATE	`INSERT INTO table (cols) VALUES (vals)`	`INSERT INTO Tasks (TaskName, Priority) VALUES ('Buy milk',`

Operation	SQL Pattern	Example
		'high')
READ	SELECT cols FROM table WHERE condition	SELECT * FROM Tasks WHERE Completed = 0 ORDER BY DueDate
UPDATE	UPDATE table SET col = val WHERE condition	UPDATE Tasks SET Completed = 1 WHERE TaskID = 5
DELETE	DELETE FROM table WHERE condition	DELETE FROM Tasks WHERE TaskID = 5

WHERE Operators:

Operator	Meaning	Example
= / <> / != / < / > / <= / >=	Comparison	WHERE Priority = 'high'
BETWEEN x AND y	Range (inclusive)	WHERE DueDate BETWEEN '2026-03-01' AND '2026-03-31'
IN (list)	Matches any in list	WHERE Priority IN ('high', 'medium')
LIKE 'pattern'	Pattern matching	WHERE TaskName LIKE '%grocery%'
IS NULL / IS NOT NULL	Null check (never use = NULL)	WHERE DueDate IS NULL
AND / OR / NOT	Logical operators	WHERE Priority = 'high' AND Completed = 0
EXISTS (subquery)	True if subquery has rows	WHERE EXISTS (SELECT 1 FROM Tasks WHERE ...)

LIKE Wildcards:

Pattern	Meaning	Pattern	Meaning
%text%	Contains	text%	Starts with
%text	Ends with	_ext	One char + 'ext'
[abc]%	Starts with a, b, or c	[^abc]%	NOT starting with a, b, or c

Data Types:

Type	Stores	Example	Notes
INT	Whole numbers	42, -7	IDs, counts
BIGINT	Large integers	9223372036854775807	Massive datasets

Type	Stores	Example	Notes
DECIMAL(p,s)	Exact decimals	DECIMAL(10,2)	Money, areas. p=total, s=decimal
FLOAT	Approximate decimals	3.14159	Science. NOT for money
BIT	Boolean	0 or 1	True/false
NVARCHAR(n)	Unicode text	NVARCHAR(100)	Always use over VARCHAR
NVARCHAR(MAX)	Long Unicode text	Up to 2GB	Descriptions, notes
DATE	Date only	'2026-03-15'	No time
DATETIME2	Date + precise time	'2026-03-15 14:30:00'	Preferred over DATETIME
UNIQUEIDENTIFIER	GUID	NEWID()	Globally unique

Constraints:

Constraint	Does	Syntax
PRIMARY KEY	Unique + NOT NULL, one per table	`col INT PRIMARY KEY`
FOREIGN KEY	Must match value in another table	`FOREIGN KEY (col) REFERENCES Table(col)`
NOT NULL	Cannot be empty	`col NVARCHAR(100) NOT NULL`
UNIQUE	No duplicates (NULL ok)	`col NVARCHAR(200) UNIQUE`
DEFAULT	Auto-fills if omitted	`col BIT DEFAULT 0`
CHECK	Value must pass condition	`CHECK (col IN ('a','b','c'))`
IDENTITY(1,1)	Auto-increment (MSSQL)	`col INT IDENTITY(1,1)`

Aggregate Functions:

Function	Does	Example
`COUNT(*)`	Count all rows	`SELECT COUNT(*) FROM Tasks`
`COUNT(col)`	Count non-NULL values	`SELECT COUNT(DueDate) FROM Tasks`
`SUM(col)`	Total	`SELECT SUM(LotArea) FROM Lots`
`AVG(col)`	Average	`SELECT AVG(LotArea) FROM Lots`
`MIN(col)`	Smallest	`SELECT MIN(DueDate) FROM`

Function	Does	Example
		`Tasks`
`MAX(col)`	Largest	`SELECT MAX(CreatedAt) FROM Tasks`

JOIN Types:

Type	Returns	Use When
INNER JOIN	Only rows matching in BOTH tables	You only want records with matches
LEFT JOIN	ALL from left + matches from right (NULLs if no match)	Want all from first table regardless
RIGHT JOIN	ALL from right + matches from left	Rarely used; swap table order + LEFT JOIN
FULL OUTER JOIN	ALL from both tables	Want everything (rare)
CROSS JOIN	Every row paired with every other row	Cartesian product (very rare)

Date/Time Functions:

Function	Returns	Example
`GETDATE()`	Current datetime	`SELECT GETDATE()`
`SYSDATETIME()`	Current datetime (high precision)	`SELECT SYSDATETIME()`
`DATEADD(unit, n, date)`	Date plus/minus interval	`DATEADD(DAY, 7, GETDATE())`
`DATEDIFF(unit, start, end)`	Difference between dates	`DATEDIFF(DAY, DueDate, GETDATE())`
`DATEPART(unit, date)`	Extract part of date	`DATEPART(MONTH, DueDate)`
`YEAR / MONTH / DAY`	Extract year/month/day	`YEAR(GETDATE())`
`FORMAT(date, fmt)`	Custom format string	`FORMAT(GETDATE(), 'dd/MM/yyyy')`
`CONVERT(type, val, style)`	Type cast with format code	`CONVERT(VARCHAR, GETDATE(), 103)`
`EOMONTH(date)`	Last day of month	`EOMONTH(GETDATE())`
`ISDATE(expr)`	Check if valid date (1/0)	`ISDATE('2026-13-01') = 0`

Date Units:

Unit	Keyword	Unit	Keyword	Unit	Keyword
Year	YEAR	Month	MONTH	Day	DAY

Unit	Keyword	Unit	Keyword	Unit	Keyword
Hour	HOUR	Minute	MINUTE	Second	SECOND
Week	WEEK	Quarter	QUARTER	Millisecond	MILLISECOND

CONVERT Date Styles:

Style	Format	Example Output
101	mm/dd/yyyy (US)	03/15/2026
103	dd/mm/yyyy (AU/UK)	15/03/2026
104	dd.mm.yyyy (German)	15.03.2026
120	yyyy-mm-dd hh:mi:ss (ISO)	2026-03-15 14:30:00
112	yyyymmdd (compact)	20260315
108	hh:mi:ss (time only)	14:30:00

String Functions:

Function	Does	Example	Result
`LEN(str)`	Length	`LEN('hello')`	5
`UPPER(str)`	Uppercase	`UPPER('hello')`	HELLO
`LOWER(str)`	Lowercase	`LOWER('HELLO')`	hello
`TRIM(str)`	Remove spaces both sides	`TRIM('  hi  ')`	hi
`LTRIM / RTRIM`	Trim left / right	`RTRIM('hi  ')`	hi
`SUBSTRING(str, start, len)`	Extract (1-based)	`SUBSTRING('hello', 1, 3)`	hel
`LEFT / RIGHT`	First/last n chars	`LEFT('hello', 3)`	hel
`REPLACE(str, old, new)`	Find and replace	`REPLACE('a-b', '-', '/')`	a/b
`CHARINDEX(find, str)`	Position (0=not found)	`CHARINDEX('lo', 'hello')`	4
`CONCAT(a, b, ...)`	Join (NULL-safe)	`CONCAT('a', NULL, 'b')`	ab
`STRING_AGG(col, sep)`	Aggregate to string	`STRING_AGG(Name, ', ')`	Alice, Bob
`REVERSE(str)`	Reverse	`REVERSE('hello')`	olleh
`REPLICATE(str, n)`	Repeat n times	`REPLICATE('*', 5)`	*****
`STUFF(str, pos, len, new)`	Replace at position	`STUFF('hello', 2, 3, 'XY')`	hXYo

NULL Handling:

Function	Does	Example
`IS NULL / IS NOT NULL`	Check for null (NEVER = NULL)	`WHERE DueDate IS NULL`
`ISNULL(col, default)`	Replace NULL (MSSQL)	`ISNULL(DueDate, 'No date')`
`COALESCE(a, b, c ...)`	First non-NULL (standard)	`COALESCE(DueDate, CreatedAt)`
`NULLIF(a, b)`	NULL if a = b	`NULLIF(Priority, 'none')`

Index Guidelines:

Index When	Don't Index When
Columns in WHERE clauses	Rarely searched columns
Columns in JOIN conditions	Tables with < 1000 rows
Columns in ORDER BY	Columns that change constantly
Foreign key columns (NOT auto-indexed!)	Wide columns (NVARCHAR(MAX))

Normalisation Rules:

Form	Rule	Fix
1NF	One value per cell. No comma-separated lists	Split into rows or junction table
2NF	Every non-key depends on the WHOLE key	Move partial dependencies to own table
3NF	No non-key depends on another non-key	Move transitive dependencies to own table

xp_cmdshell Facts:

Fact	Detail
MSSQL only	MySQL/PostgreSQL have NO equivalent built-in
Disabled by default	Requires sysadmin to enable
Pentest goldmine	SQL injection + xp_cmdshell = OS command execution
Privilege escalation	Potato exploits escalate service account to SYSTEM
Potato variants	SweetPotato, JuicyPotato, PrintSpoofer, GodPotato
Defence	Never enable in prod. Parameterised queries. Least privilege

C# Database Classes:

Class	Purpose	Key Methods
SqlConnection	Connect to DB	`.Open()`, `using` block
SqlCommand	Hold + exec SQL	`.ExecuteReader()` (SELECT), `.ExecuteNonQuery()` (INSERT/UPDATE/DELETE)
SqlDataReader	Read row by row	`.Read()`, `reader["col"]`
SqlParameter	Prevent injection	`cmd.Parameters.AddWithValue("@x", val)`

MSSQL vs MySQL:

Feature	MSSQL	MySQL
Limit results	`SELECT TOP 10 *`	`SELECT * LIMIT 10`
Auto-increment	`INT IDENTITY(1,1)`	`INT AUTO_INCREMENT`
Current datetime	`GETDATE()` / `SYSDATETIME()`	`NOW()` / `CURRENT_TIMESTAMP`
String concat	`CONCAT()` or +	`CONCAT()` only (+ = addition)
If NULL	`ISNULL(col, default)`	`IFNULL(col, default)`
Boolean	`BIT (0/1)`	`BOOLEAN / TINYINT(1)`
Unicode strings	`NVARCHAR`	`VARCHAR` with `utf8mb4`
Rename column	`sp_rename`	`ALTER TABLE RENAME COLUMN`
Pagination	`OFFSET x ROWS FETCH NEXT y ROWS ONLY`	`LIMIT y OFFSET x`
OS commands	`xp_cmdshell`	No equivalent
String length	`LEN()` (trims trailing spaces)	`LENGTH()` / `CHAR_LENGTH()`

Safety Rules:

#	Rule
1	Always WHERE on UPDATE/DELETE. Test with SELECT first.
2	Always parameterise. NEVER concatenate user input.
3	NVARCHAR over VARCHAR. Unicode matters.
4	IS NULL not = NULL. NULL = NULL returns UNKNOWN.

#	Rule
5	Back up before DROP, TRUNCATE, mass DELETE.
6	Index foreign keys manually (NOT auto-indexed).
7	Use transactions for multi-step operations.
8	IDENTITY gaps are normal. IDs never reuse.
9	Single quotes for strings. Double quotes for identifiers (avoid).
10	DROP TABLE is permanent. No recycle bin.

CRUD Cross-Reference (SQL / C# / JS):

Op	SQL	C# Method	JS Array
C	`INSERT INTO ... VALUES ...`	`ExecuteNonQuery()`	`arr.push(item)`
R	`SELECT ... WHERE ...`	`ExecuteReader()`	`arr.find() / arr.filter()`
U	`UPDATE ... SET ... WHERE ...`	`ExecuteNonQuery()`	`item.prop = val`
D	`DELETE FROM ... WHERE ...`	`ExecuteNonQuery()`	`arr.filter(x => x.id !== id)`

The database is the truth. Parameterise everything. Back up before you break things.

Vidimus Omnia.

PART 4: GELOS BATTLE PLAN

Assessment Operations Order (OPORD)

Gelos Task Manager | Build Order | Assessment Criteria Mapping

22nd Survey Division | PTE WU | Vidimus Omnia

Due: 21 April 2026 | Cert IV Programming | Tuesday Class

> This is the operations order. Follow the phases in sequence. Each phase is a self-contained deliverable that builds on the last. Don't skip ahead. Don't optimise before it works. Get it functional first, then make it pretty.

---

1. SITUATION

Assessment: Build a task management web application called Gelos using HTML, CSS, and JavaScript. The application must demonstrate CRUD operations, data persistence, responsive design, and clean code practices.

Mission: Deliver a fully functional single-page task manager that creates, reads, updates, and deletes tasks, persists data in localStorage, and works on both mobile and desktop browsers. Due 21 April 2026.

File structure:

```
gelos/
    index.html       -- Structure and content
    style.css        -- All styling
    script.js        -- All functionality
    README.md        -- Optional but impressive: project description
```

Three files. One folder. No frameworks, no libraries, no npm, no build tools. Pure HTML + CSS + JS. That's what they're assessing.

What you're assessed on:

Criteria	What They Want to See	Where in Your Code
HTML structure	Semantic tags, valid markup, accessibility	index.html
CSS styling	Professional layout, responsive, consistent	style.css
JavaScript functionality	CRUD operations, event handling, DOM manipulation	script.js
Data persistence	localStorage save/load, survives page refresh	script.js (save/load functions)
User interface	Forms, inputs, feedback, intuitive flow	All three files
Code quality	Comments, naming, indentation, DRY principle	All three files
Responsive design	Works on mobile and desktop	style.css (@media queries)
Error handling	Validates input, handles edge cases	script.js (trim, empty checks)

---

2. EXECUTION – 7 PHASES

Seven phases. Each phase ends with a working product. If you run out of time, stop after any phase and you still have something to submit. Phase 1-4 = minimum viable. Phase 5-7 = polish and bonus marks.

PHASE 1: HTML SKELETON (30 mins)

Build the complete HTML structure. No styling, no functionality. Just the bones.

Steps: 1. Create `index.html` with the starter template. –> Copy the HTML skeleton from Part 2B, Section 7 (Gelos HTML Structure). 2. Add `<header>` with `<h1>Gelos Task Manager</h1>`. 3. Add `<main>` with two `<section>` elements: `addTaskSection` and `taskDisplaySection`. 4. In `addTaskSection`: build the form with all inputs (TaskName, Description, Priority, DueDate, Add button). 5. In `taskDisplaySection`: add search input, priority filter dropdown, empty `taskList` div. 6. Add `<footer>` with copyright. 7. Add `<link>` to `style.css` in `<head>`, `<script>` to `script.js` at bottom of `<body>`. 8. Create empty `style.css` and `script.js` files.

Checkpoint: Open `index.html` in browser. You should see an ugly but functional form with all inputs visible. All IDs match your planned JS targets.

> –> Reference: Part 2B, Section 7 for the complete Gelos HTML structure. Copy it. Modify as needed.

PHASE 2: CORE JAVASCRIPT – ADD + DISPLAY (1 hour)

Get tasks appearing on screen. This is the minimum viable product.

Steps: 1. Declare `let tasks = [];` at the top of `script.js`. 2. Write `addTask()` function: read inputs, create task object with `Date.now()` ID, push to array, clear form, call `renderTasks()`. 3. Write `renderTasks()` function: clear `taskList` innerHTML, loop tasks with `forEach`, create div for each, `appendChild`. 4. Write `clearForm()` function: reset all input values to empty. 5. Add `DOMContentLoaded` event listener that calls `renderTasks()`. 6. Add click event listener on `addTaskBtn` that calls `addTask()`.

Checkpoint: Type a task name, click Add Task. Task card appears on page. Add three tasks. All three visible. Refresh page – they disappear (no persistence yet). That's expected.

> –> Reference: Part 2B, Section 22 for the Gelos script.js skeleton. The structure is all there.

PHASE 3: LOCALSTORAGE – PERSISTENCE (30 mins)

Tasks survive page refresh. This is the data persistence requirement.

Steps: 1. Write `saveTasks()`: `localStorage.setItem('gelosTasks', JSON.stringify(tasks))`. 2. Write `loadTasks()`: get from localStorage, parse, assign to tasks array. 3. Add `saveTasks()` call at the end of `addTask()`. 4. Add `loadTasks()` call in `DOMContentLoaded` listener, before `renderTasks()`.

Checkpoint: Add three tasks. Refresh the page. Tasks are still there. Open F12 > Application > Local Storage. See 'gelosTasks' key with JSON data.

> After Phase 3 you have a submittable product. It creates and displays tasks that persist. Everything from here is improvement.
>
> –> Reference: Part 2B, Section 20 for localStorage patterns.

PHASE 4: DELETE + COMPLETE (45 mins)

Full CRUD minus edit. Delete tasks and toggle completion.

Steps: 1. Write `deleteTask(id)`: filter out the task, `saveTasks()`, `renderTasks()`. 2. Write `toggleComplete(id)`: find task, flip `completed` boolean, `saveTasks()`, `renderTasks()`. 3. Add `data-id='${task.id}'` attributes to delete and complete buttons in `renderTasks()`. 4. Add event delegation on `taskList`: listen for clicks, check `e.target.classList`, call `deleteTask` or `toggleComplete` with `Number(e.target.dataset.id)`. 5. Add `'completed'` class to task cards where `task.completed` is true.

Checkpoint: Add task > click Done > card shows strikethrough/faded > click Undo > card returns to normal. Click Delete > card disappears. Refresh > changes persist.

> Event delegation is critical here. You're putting ONE listener on `taskList` (the parent), not individual listeners on each button. Because the buttons are created dynamically by `renderTasks()`, direct listeners would disappear every time you re-render.
>
> –> Reference: Part 2B, Section 18 for event delegation patterns. Part 2B, Section 19 for array CRUD methods.

PHASE 5: SEARCH + FILTER (30 mins)

Live search and priority filtering. Makes the app feel professional.

Steps: 1. In `renderTasks()`, BEFORE the `forEach` loop, get the search term and filter priority values. 2. Filter the tasks array: `task.name.toLowerCase().includes(searchTerm)` AND `(filterPriority === 'all' || task.priority === filterPriority)`. 3. Add `keyup` event listener on `searchBox` that calls `renderTasks()`. 4. Add `change` event listener on `filterPriority` that calls `renderTasks()`. 5. Add 'No tasks found' message when filtered array is empty.

Checkpoint: Add five tasks with mixed priorities. Type in search box – list filters live as you type. Select 'High' from dropdown – only high priority tasks show. Clear both – all tasks return.

PHASE 6: CSS STYLING (1-2 hours)

Make it look professional. This is where you go from 'it works' to 'it looks good.'

Steps (in order): 1. CSS Reset: `* { margin:0; padding:0; box-sizing:border-box; }` + body defaults. 2. Layout: `.container` with `max-width` + `margin: 0 auto`. Header styling. 3. Form styling: inputs get padding, border, border-radius, `font-family:inherit`. Focus states. 4. Button styling: `background #333`, `color #fff`, `border-radius`, `cursor:pointer`, hover state. 5. Task cards: white background, border, border-radius, box-shadow, flex layout (space-between). 6. Priority borders: `.task-card.high` = `border-left 4px solid #333`, medium = `#999`, low = `#ddd`. 7. Completed state: `.task-card.completed` = `opacity 0.6` + `text-decoration line-through`. 8. Responsive: `@media (min-width: 768px)` for tablet form layout. Test on mobile.

Checkpoint: Open in Chrome. Looks professional, clean, consistent. Open DevTools (F12) > toggle device toolbar (Ctrl+Shift+M). Test on mobile width. Form stacks vertically. Cards are full width. No horizontal scrolling.

> –> Reference: Part 2B, Sections 8-14 for every CSS pattern. Copy the card styling from Section 13, form styling from Section 12, reset from Section 8. You've already written it all.

PHASE 7: POLISH + BONUS (remaining time)

Extra features that separate a pass from a distinction.

Feature	How	Time	Impact
Edit task inline	Populate form with task data, update instead of add	30 min	High – completes CRUD
Sort by due date	`tasks.sort()` before rendering	10 min	Medium
Task count display	'Showing 3 of 7 tasks'	10 min	Medium
Confirmation on delete	`if(confirm('Delete this task?'))`	5 min	Low but safe
Input validation messages	Show 'Task name required' below input	15 min	High – shows error handling
Clear all completed	Button to batch-delete completed tasks	15 min	Medium
Dark mode toggle	Toggle a `.dark-mode`	20 min	Low priority, high

Feature	How	Time	Impact
	class on body		impression
Keyboard shortcuts	Enter to add task	10 min	Low effort, good UX
Code comments	Every function gets a `// purpose` comment	15 min	High – explicit assessment criteria
README.md	Project description, how to run, features list	15 min	Professional touch

Edit task is the highest-value bonus feature. Without it, you have CRD (Create, Read, Delete) but not full CRUD. The U matters. See Section 6 below for the exact implementation.

Comments are free marks. Every function gets a one-line comment explaining what it does. If the assessor reads your code and understands it without running it, you've won.

---

3. TIMELINE – Build Schedule

Week	Date Range	Phase	Deliverable
Week 3	Feb 17-20	Phase 1: HTML skeleton	All HTML written, form and display areas
Week 4	Feb 24-27	Phase 2: Core JS	Add tasks + display working
Week 5	Mar 3-6	Phase 3: localStorage	Tasks persist on refresh
Week 6	Mar 10-13	Phase 4: Delete + Complete	Full delete and toggle complete
Week 7	Mar 17-20	Phase 5: Search + Filter	Live search and priority filter
Week 8-9	Mar 24 - Apr 3	Phase 6: CSS	Full responsive styling
Week 10-11	Apr 7-17	Phase 7: Polish	Edit, comments, validation, extras
Week 12	Apr 21	SUBMIT	Final test, zip, submit

▶ This gives you 9 weeks of build time with 2 weeks of buffer. If you fall behind, Phase 4+ can be compressed. Phases 1-3 are non-negotiable – they're the foundation. Never skip ahead to CSS before the JS works.

4. EMERGENCY SCENARIO – 48 Hours Before Deadline

If you're panicking with 48 hours left, do this:

1. **Phase 1-3 in one sitting (2 hours).** Copy HTML from Part 2B, Section 7. JS skeleton from Part 2B, Section 22.
2. **Add delete and complete (Phase 4, 45 mins).**
3. **CSS reset + basic card styling (1 hour).** Don't try to make it beautiful. Make it clean.
4. **Add comments to every function (15 mins).**
5. **Test on mobile.** Fix any overflow issues.
6. **Submit.** You'll pass.

A working ugly app beats a beautiful broken one. Functionality first. Always.

---

5. PRE-SUBMISSION CHECKLIST

Run through this list before you submit. Every item should be a yes.

Functionality: - [ ] Can you add a task with name, description, priority, and due date? - [ ] Does the task appear on screen after adding? - [ ] Can you delete a task? - [ ] Can you mark a task as complete? - [ ] Can you undo a completed task? - [ ] Do tasks persist after page refresh? - [ ] Does search filter tasks as you type? - [ ] Does the priority dropdown filter correctly? - [ ] Does an empty task name get rejected? - [ ] Is the form cleared after adding a task?

Visual / Responsive: - [ ] Does it look professional on desktop (>1024px)? - [ ] Does it look good on tablet (768px)? - [ ] Does it work on mobile (375px)? - [ ] No horizontal scrolling on any screen size? - [ ] Priority indicators visible (border-left colours)? - [ ] Completed tasks visually different (strikethrough/opacity)? - [ ] Buttons have hover states? - [ ] Inputs have focus states?

Code Quality: - [ ] Every function has a comment explaining its purpose? - [ ] Variable names are descriptive (not x, y, temp)? - [ ] Consistent indentation (2 or 4 spaces, not mixed)? - [ ] No `console.log()` debug statements left in? - [ ] No unused code or commented-out blocks? - [ ] HTML validates (no unclosed tags)? - [ ] CSS is organised (reset > layout > components > responsive)? - [ ] JS is organised (data > functions > event listeners)?

Edge Cases: - [ ] What happens with 0 tasks? (Should show 'No tasks' message) - [ ] What happens with 50+ tasks? (Should still be usable) - [ ] What happens if localStorage is full? (Unlikely but good to consider) - [ ] What if someone enters `<script>` in the task name? (Use `textContent`, not `innerHTML` for user data) - [ ] What if due date is in the past? (Should still work, maybe show as overdue)

The XSS check matters: if you use `innerHTML` with user input, someone can inject JavaScript. Use `textContent` for user-provided data. Use `innerHTML` only for HTML you build yourself with template literals.

---

6. BONUS: EDIT TASK IMPLEMENTATION

This is the most valuable bonus feature. It completes CRUD. Here's the exact pattern:

Step 1: Add a global edit state

```
let editingTaskId = null;
// null = adding new task. Number = editing existing task.
```

Step 2: Write editTask() function

```
function editTask(id) {
    const task = tasks.find(t => t.id === id);
    if (!task) return;

    // Populate form with existing data
    document.getElementById('taskName').value = task.name;
    document.getElementById('taskDesc').value = task.description || '';
    document.getElementById('priority').value = task.priority;
    document.getElementById('dueDate').value = task.dueDate || '';

    // Switch button text
    document.getElementById('addTaskBtn').textContent = 'Update Task';

    // Set edit mode
    editingTaskId = id;
}
```

Step 3: Modify addTask() to handle both add and edit

```
function addTask() {
    const name = document.getElementById('taskName').value.trim();
    if (!name) return;

    if (editingTaskId !== null) {
        // UPDATE existing task
        const task = tasks.find(t => t.id === editingTaskId);
        if (task) {
            task.name = name;
            task.description = document.getElementById('taskDesc').value;
            task.priority = document.getElementById('priority').value;
            task.dueDate = document.getElementById('dueDate').value;
        }
        editingTaskId = null;
        document.getElementById('addTaskBtn').textContent = 'Add Task';
```

```
    } else {
        // CREATE new task
        const task = {
            id: Date.now(),
            name: name,
            description: document.getElementById('taskDesc').value,
            priority: document.getElementById('priority').value,
            dueDate: document.getElementById('dueDate').value,
            completed: false
        };
        tasks.push(task);
    }

    saveTasks();
    clearForm();
    renderTasks();
}
```

Step 4: Add edit button to renderTasks()

```
// In the card innerHTML template, add:
<button class="edit-btn" data-id="${task.id}">Edit</button>
```

Step 5: Add edit handler to event delegation

```
// In the taskList click listener:
if (e.target.classList.contains('edit-btn')) editTask(Number(e.target.dataset.
id));
```

Step 6: Add cancel edit option

```
function cancelEdit() {
    editingTaskId = null;
    document.getElementById('addTaskBtn').textContent = 'Add Task';
    clearForm();
}
```

Add a 'Cancel' button next to 'Add Task' that's only visible when `editingTaskId` is not null.

> With edit implemented, you have full CRUD: **Create** (addTask), **Read** (renderTasks), **Update** (editTask + addTask), **Delete** (deleteTask). That's the whole assessment.

---

7. REFERENCE MAP – Where to Find What in This Workbook

I Need To...	Go To	Section
Understand WHY HTML/CSS/JS are separate	Part 2A	Sections 1-3, 14, 20

I Need To...	Go To	Section
Copy the HTML starter template	Part 2B	Section 1
Build the Gelos HTML structure	Part 2B	Section 7
Style form inputs	Part 2B	Section 12
Style task cards	Part 2B	Section 13
Make it responsive	Part 2B	Section 14
Write addTask() function	Part 2B	Section 22
Write renderTasks() function	Part 2B	Section 21
Handle events and delegation	Part 2B	Section 18
Use localStorage	Part 2B	Section 20
Use array methods (CRUD)	Part 2B	Section 19
Debug something broken	Part 2B	Section 24
Look up a CSS property	Part 2C	CSS Quick Reference
Look up flexbox	Part 2C	Flexbox Quick Reference
Look up a JS DOM method	Part 2C	JS DOM Quick Reference
Look up array methods	Part 2C	JS Array Quick Reference
Understand database concepts	Part 3A	Sections 1-10
Write SQL queries	Part 3B	Sections 1-15
Connect C# to SQL	Part 3B	Section 17
Understand SQL injection	Part 3B	Section 18
Compare syntax across languages	Appendix A	Rosetta Stone
Decode an error message	Appendix B	Error Decoder
Look up keyboard shortcuts	Appendix C	Shortcuts & Tools
Look up a term	Appendix D	Glossary

Plan the work. Work the plan. Phase by phase. A working ugly app beats a beautiful broken one.

Vidimus Omnia.

APPENDIX A: CROSS-LANGUAGE ROSETTA STONE

One table. Four languages. Same concepts.

Concept	Python	C#	JavaScript	SQL
Variable declaration	`x = 10`	`int x = 10;`	`let x = 10;`	`DECLARE @x INT = 10;`
Constant	`X = 10` (convention, not enforced)	`const int X = 10;`	`const X = 10;`	N/A
String / text	`name = "Alice"`	`string name = "Alice";`	`let name = "Alice";`	`NVARCHAR(100)` / `'Alice'`
Number types	`x = 10` (int), `x = 3.14` (float)	`int x = 10;` `double x = 3.14;` `decimal x = 3.14m;`	`let x = 10;` `let x = 3.14;` (both `number`)	`INT`, `DECIMAL(10,2)`, `FLOAT`
Boolean	`True` / `False`	`bool x = true;`	`let x = true;`	`BIT` (0/1)
Array / list	`items = [1, 2, 3]`	`int[] items = {1, 2, 3};` `List<int> items = new List<int>();`	`let items = [1, 2, 3];`	Table rows (no arrays)
Print / output	`print("hello")`	`Console.WriteLine("hello");`	`console.log("hello");`	`PRINT 'hello'` / `SELECT 'hello'`
User input	`x = input("Enter: ")`	`string x = Console.ReadLine();`	`let x = prompt("Enter: ");`	N/A (parameters from app)
String interpolation	`f"Hello {name}"`	`$"Hello {name}"`	`` `Hello ${name}` ``	`CONCAT('Hello ', @name)`
If statement	`if x > 5:`	`if (x > 5) { }`	`if (x > 5) { }`	`IF @x > 5 BEGIN ... END`
Else if / elif	`elif x > 3:`	`else if (x > 3) { }`	`else if (x > 3) { }`	`ELSE IF @x > 3 BEGIN ... END`
For loop	`for i in range(10):`	`for (int i = 0; i < 10; i++) { }`	`for (let i = 0; i < 10; i++) { }`	`WHILE @i < 10 BEGIN ... SET @i += 1 END`
For-each loop	`for item in items:`	`foreach (var item in items) { }`	`items.forEach( item => { });`	`CURSOR` (avoid if possible)
While loop	`while x > 0:`	`while (x > 0) { }`	`while (x > 0) { }`	`WHILE @x > 0 BEGIN ... END`
Function / method	`def greet(name):`	`void Greet(string name) { }`	`function greet(name) { }`	`CREATE PROCEDURE Greet @name NVARCHAR(50) AS ...`

Concept	Python	C#	JavaScript	SQL
Comment	`# comment`	`// comment`	`// comment`	`-- comment`
Block comment	`""" block """`	`/* block */`	`/* block */`	`/* block */`
AND / OR / NOT	`and` / `or` / `not`	`&&` / `||` / `!`	`&&` / `||` / `!`	`AND` / `OR` / `NOT`
Equal check	`==`	`==`	`===` (strict)	`=`
Null / None / undefined	`None`	`null`	`null` / `undefined`	`NULL`
True / False	`True` / `False`	`true` / `false`	`true` / `false`	`1` / `0` (BIT)
Read file	`open("f.txt").read()`	`File.ReadAllText("f.txt");`	`fetch("f.txt")` (async)	`BULK INSERT` / `OPENROWSET`
Create (CRUD)	`items.append(x)`	`list.Add(x);` / `ExecuteNonQuery()`	`arr.push(x);`	`INSERT INTO t (col) VALUES (val)`
Read (CRUD)	`items[0]` / `for i in items:`	`list[0]` / `ExecuteReader()`	`arr.find()` / `arr.filter()`	`SELECT * FROM t WHERE ...`
Update (CRUD)	`items[0] = newVal`	`list[0] = newVal;` / `ExecuteNonQuery()`	`item.prop = newVal;`	`UPDATE t SET col = val WHERE ...`
Delete (CRUD)	`items.remove(x)`	`list.Remove(x);` / `ExecuteNonQuery()`	`arr.filter(x => x.id !== id)`	`DELETE FROM t WHERE ...`

APPENDIX B: ERROR DECODER (ALL LANGUAGES)

One table. Every error you'll see. What it means. How to fix it.

C# Errors

Language	Error	Translation	Fix
C#	`CS0103: The name 'x' does not exist in the current context`	You used a variable that hasn't been declared, or it's declared in a different scope (inside another `{ }` block).	Declare the variable before using it, or move the declaration to a scope that's accessible. Check spelling.
C#	`CS1002: ; expected`	You forgot a semicolon at the end of a statement.	Add `;` at the end of the line. Every statement in C#

Language	Error	Translation	Fix
			ends with a semicolon.
C#	`CS0029: Cannot implicitly convert type 'x' to 'y'`	You're trying to put a value of one type into a variable of another type. Like putting text into an `int`.	Use explicit conversion: `int.Parse()`, `Convert.ToInt32()`, or cast `(int)`. Check your types match.
C#	`IndexOutOfRangeException`	You tried to access an array index that doesn't exist. Array has 5 elements (0-4) but you asked for index 5.	Check your loop bounds. Arrays are 0-indexed. `arr.Length` is the count, `arr[arr.Length - 1]` is the last element.
C#	`FormatException`	You tried to convert a string to a number but the string isn't a valid number. `int.Parse("hello")` fails.	Use `int.TryParse()` instead of `int.Parse()`. Validate user input before converting.
C#	`FileNotFoundException`	The file path you specified doesn't exist. Wrong path, wrong filename, wrong extension.	Check the full path. Use `File.Exists()` before reading. Check for typos. Use `@"C:\path"` to avoid escape issues.
C#	`NullReferenceException`	You tried to use a variable that is `null` – it points to nothing. Like trying to open a door that doesn't exist.	Check for `null` before using: `if (x != null)`. Find where the variable should have been assigned and fix it.
C#	`CS0019: Operator '==' cannot be applied to operands of type 'x' and 'y'`	You're comparing two things that can't be compared with `==`. Like comparing a string to an int.	Convert one side to match the other, or use the correct comparison method (`.Equals()` for objects).

JavaScript Errors

Language	Error	Translation	Fix
JS	`TypeError: Cannot read properties of null`	You called `.something` on a variable that is `null`. Usually means `document.getElementById()` returned `null` because the element doesn't exist yet or the ID is wrong.	Check the element ID matches exactly (case-sensitive). Make sure the script runs after the DOM loads (`DOMContentLoaded`). Check spelling.
JS	`TypeError: x is not a function`	You tried to call something as a function but it's not one. Maybe you overwrote a function with a variable, or spelled the method wrong.	Check the function name spelling. Make sure you haven't assigned a non-function value to that name. Check `()` placement.
JS	`ReferenceError: x is not defined`	You used a variable name that doesn't exist. Not declared, or declared in a different scope, or typo.	Declare with `let/const` before use. Check spelling. Check scope (variables inside `{ }` don't exist outside).
JS	`SyntaxError: Unexpected token`	Your code has a structural problem. Missing bracket, extra comma, unclosed string, wrong quote type.	Check for matching `{ }`, `( )`, `[ ]`. Check string quotes match. Look at the line number in the error.
JS	`SyntaxError: Unexpected end of input`	You're missing a closing `}`, `)`, or `]` somewhere. The file ended before the code was complete.	Count your opening and closing braces. Use VS Code's bracket matching (Ctrl+Shift+).
JS	DOM: Element not found / null	`document.getElementById('myId')` returns `null` because the ID doesn't exist in the	Verify the ID in your HTML. Put `<script>` at the bottom of `<body>`, or wrap code in

Language	Error	Translation	Fix
		HTML, or the script runs before the HTML loads.	`DOMContentLoaded`.
JS	DOM: Event listener not working	Added event listener to a dynamically created element. The element didn't exist when the listener was attached.	Use event delegation: listen on the parent element that exists in the HTML. Check `e.target.classList.contains()`. –> See Part 2B, Section 18.
JS	`localStorage` returning `null`	First time running the app, or the key name doesn't match between `setItem` and `getItem`.	Check key name spelling matches exactly. Handle the `null` case: `JSON.parse(localStorage.getItem('key')) || []`.

SQL Errors

Language	Error	Translation	Fix
SQL	`Incorrect syntax near 'x'`	You have a typo, missing comma, missing keyword, or wrong quote type in your SQL.	Check for missing commas between columns, missing `FROM`, wrong quote type (use single quotes `'` for strings).
SQL	`Invalid column name 'x'`	The column name doesn't exist in the table you're querying. Typo or wrong table.	Check column names with `SELECT * FROM table`. Check spelling. Remember: aliases from SELECT can't be used in WHERE.
SQL	`Cannot insert the value NULL into column 'x'`	You tried to insert a row without providing a value for a NOT NULL column that has no DEFAULT.	Provide a value for that column in your INSERT, or add a DEFAULT constraint to the column.
SQL	`The UPDATE/DELETE`	You tried to	Delete or reassign

Language	Error	Translation	Fix
	`statement conflicted with the FOREIGN KEY constraint`	delete/update a row that other rows reference via foreign key. Can't delete a category if tasks point to it.	the child records first. Or set up `ON DELETE SET NULL` / `ON DELETE CASCADE` on the FK.
SQL	`UPDATE/DELETE without WHERE`	Not an error – it succeeds. That's the problem. It affects ALL rows.	Always include WHERE. Always test with SELECT first. This is the #1 SQL safety rule.
SQL	`NULL = NULL returns no results`	You used `WHERE col = NULL` instead of `WHERE col IS NULL`. NULL is not a value, so = doesn't work.	Use `IS NULL` or `IS NOT NULL`. Never `= NULL` or `<> NULL`.
SQL	`Conversion failed when converting the varchar value 'x' to data type int`	You're comparing or inserting a string into an integer column, or vice versa. Type mismatch.	Use `CAST()` or `CONVERT()` to explicitly convert types. Check your WHERE conditions match column types.
SQL	`String or binary data would be truncated`	The value you're inserting is longer than the column allows. NVARCHAR(50) but your string is 75 characters.	Increase the column size with `ALTER TABLE`, or truncate the input with `LEFT(value, 50)`.
SQL	`Violation of PRIMARY KEY constraint`	You tried to insert a row with a primary key value that already exists. Duplicate ID.	If using IDENTITY, don't specify the ID in your INSERT – let SQL generate it. If manual, check for existing values first.
SQL	SQL Injection (not an error, a vulnerability)	User input is being concatenated into SQL strings instead of using parameters. Attacker can run	Use parameterised queries. ALWAYS. `cmd.Parameters.AddWithValue("@name", input)`. –> See Part

Language	Error	Translation	Fix
		arbitrary SQL.	3B, Section 18.

APPENDIX C: KEYBOARD SHORTCUTS & TOOLS

Visual Studio (C#)

Shortcut	What It Does
`F5`	Run with debugging (starts the app, stops at breakpoints)
`Ctrl+F5`	Run without debugging (just run it, faster)
`Ctrl+A`, then `Ctrl+K, Ctrl+D`	Select all, then auto-format/indent the entire file
`Ctrl+K, Ctrl+C`	Comment out selected lines
`Ctrl+K, Ctrl+U`	Uncomment selected lines
`Ctrl+Z`	Undo
`Ctrl+S`	Save
`Ctrl+.`	Quick actions / show suggestions (lightbulb menu)
`F12`	Go to definition (jump to where a method/class is defined)
`Ctrl+Shift+B`	Build the solution (compile without running)
`F9`	Toggle breakpoint on current line
`F10`	Step over (debugging: execute line, don’t go into methods)
`F11`	Step into (debugging: go inside the method call)

VS Code (HTML/CSS/JS)

Shortcut	What It Does
`! + Tab`	Generate HTML5 boilerplate (in an .html file)
`Shift+Alt+F`	Format document (auto-indent everything)
`Ctrl+/`	Toggle comment on selected lines
`Ctrl+D`	Select next occurrence of current selection (multi-cursor)
`Ctrl+Shift+K`	Delete entire line
`Alt+Up/Down`	Move current line up or down
`Ctrl+```	Toggle integrated terminal
`Ctrl+B`	Toggle sidebar

Shortcut	What It Does
`Ctrl+P`	Quick open file by name
`Ctrl+Shift+P`	Command palette (search all VS Code commands)
`Ctrl+Space`	Trigger IntelliSense (auto-complete suggestions)
`F2`	Rename symbol (renames everywhere it's used)

Chrome DevTools

Shortcut	What It Does
`F12`	Open/close DevTools
`Ctrl+Shift+I`	Open DevTools (alternative)
`Ctrl+Shift+J`	Open DevTools directly to Console tab
`Ctrl+Shift+M`	Toggle device toolbar (responsive mode)
Console tab	View errors, run JavaScript live, test expressions
Elements tab	Inspect HTML structure, edit CSS live
Application tab	View localStorage, cookies, session storage
Network tab	See all HTTP requests (useful for debugging fetch/API calls)

SSMS (SQL Server Management Studio)

Shortcut	What It Does
`F5`	Execute selected query (or all if nothing selected)
`Ctrl+E`	Execute query (alternative)
`Ctrl+R`	Toggle results pane
`Ctrl+Shift+U`	Make selected text UPPERCASE
`Ctrl+Shift+L`	Make selected text lowercase
`Ctrl+K, Ctrl+C`	Comment selected lines
`Ctrl+K, Ctrl+U`	Uncomment selected lines
Select query + `F5`	Execute only the highlighted portion

APPENDIX D: GLOSSARY

Quick-lookup definitions. Alphabetical. No waffle.

Term	Definition
API	Application Programming Interface. A set of

Term	Definition
	rules for how software components talk to each other. Like a waiter between you and the kitchen – you don't cook, you just order.
Array	An ordered collection of values stored in a single variable. Accessed by index (0-based). Python calls them lists. C# has both arrays (fixed size) and Lists (dynamic).
Boolean	A value that is either `true` or `false`. Named after George Boole. Used in conditions, flags, and logic. In SQL, stored as `BIT` (0 or 1).
CRUD	Create, Read, Update, Delete. The four basic operations for any data system. SQL: INSERT, SELECT, UPDATE, DELETE. JS arrays: push, find/filter, assignment, filter.
CSS	Cascading Style Sheets. Controls how HTML looks: colours, layout, fonts, spacing, responsive design. Separate file from HTML because structure and presentation are different concerns.
DOM	Document Object Model. The browser's live representation of your HTML as a tree of objects. JavaScript manipulates the DOM to change what the user sees without reloading the page.
Event Delegation	Attaching one event listener to a parent element instead of individual listeners on each child. The event bubbles up from the clicked child to the parent. Essential for dynamically created elements.
Flexbox	A CSS layout system for arranging items in a row or column. `display: flex` on the parent, then control alignment with `justify-content`, `align-items`, `gap`, and `flex-direction`.
Foreign Key	A column in one table that references the primary key of another table. Creates a relationship. Prevents orphan records. Like field notes referencing a registered DP number.

Term	Definition
Function	A reusable block of code that does one thing. Called by name, optionally takes parameters, optionally returns a value. Python: `def`. C#: return type + name. JS: `function` keyword or arrow `=>`.
HTML	HyperText Markup Language. The structure of a web page. Tags define elements: headings, paragraphs, forms, images, links. The skeleton that CSS dresses and JS animates.
IDE	Integrated Development Environment. A code editor with built-in tools: compiler, debugger, IntelliSense. Visual Studio for C#. VS Code for web dev. SSMS for SQL.
Index	A database structure that speeds up searches on a column. Like alphabetical tabs in a filing cabinet. Primary keys are auto-indexed. Foreign keys are NOT – index them manually.
JavaScript	The programming language of the web browser. Handles interactivity, DOM manipulation, events, and data logic on the client side. Dynamically typed. Runs in the browser, no compilation needed.
JOIN	A SQL operation that combines rows from two or more tables based on a related column. INNER JOIN = matching rows only. LEFT JOIN = all from left table + matches from right.
localStorage	A browser API that stores key-value pairs as strings. Persists across page refreshes and browser restarts. `setItem()` to save, `getItem()` to load, `JSON.stringify()`/`JSON.parse()` for objects.
Method	A function that belongs to an object or class. `console.log()` – `log` is a method of the `console` object. `arr.push()` – `push` is a method of the array. Same as function, just attached to something.

Term	Definition
MVC	Model-View-Controller. An architecture pattern. Model = data. View = what the user sees. Controller = logic that connects them. Keeps concerns separated.
Normalisation	Structuring database tables to eliminate data redundancy. Every fact stored once. 1NF: one value per cell. 2NF: depend on whole key. 3NF: no transitive dependencies. "The key, the whole key, nothing but the key."
NULL	The absence of a value. Not zero, not empty string, not false. Means "unknown" or "not applicable." In SQL, use `IS NULL` to check – never `= NULL`. In C#: `null`. In JS: `null` and `undefined`.
Object	A collection of key-value pairs (properties). In JS: `{ name: "Alice", age: 30 }`. In C#: an instance of a class. In SQL: tables, views, stored procedures are all "objects."
Parameter	A variable in a function definition that receives a value when the function is called. `function greet(name)` – `name` is the parameter. The value you pass in is the argument. Also used in SQL to prevent injection.
Primary Key	A column (or combination) that uniquely identifies each row in a table. No duplicates. No NULLs. One per table. Automatically indexed. Like a DP lot number – unique across the entire registry.
Property	A named attribute of an object. `task.name` – `name` is a property of the `task` object. In C#, properties use get/set accessors. In JS, they're just keys on an object.
Query	A request for data from a database. A SELECT statement is a query. "Show me all high-priority tasks due this week" translates to `SELECT * FROM Tasks WHERE Priority = 'high' AND DueDate BETWEEN ....`
Responsive Design	Building web pages that adapt to different

Term	Definition
	screen sizes. Uses `@media` queries, flexible units (`%`, `vw`, `rem`), and flexbox/grid. Test at 375px (mobile), 768px (tablet), and 1024px+ (desktop).
Scope	Where a variable is accessible. Variables declared inside `{ }` (block scope) only exist inside that block. `let` and `const` are block-scoped. `var` is function-scoped (avoid `var`).
Selector	The CSS pattern that targets HTML elements. `h1` = tag. `.className` = class. `#idName` = ID. `div > p` = direct child. Specificity: ID > class > tag.
Semantic HTML	Using HTML tags that describe their meaning: `<header>`, `<nav>`, `<main>`, `<section>`, `<article>`, `<footer>`. Better for accessibility and SEO than generic `<div>` everywhere.
SQL	Structured Query Language. The language for talking to relational databases. CRUD operations: INSERT, SELECT, UPDATE, DELETE. Not a programming language – it's a query language.
SQL Injection	A security vulnerability where user input is treated as SQL code instead of data. Prevented by parameterised queries. Never concatenate user input into SQL strings.
String Interpolation	Embedding variables inside a string. Python: `f"Hello {name}"`. C#: `$"Hello {name}"`. JavaScript: `` `Hello ${name}` `` (template literals with backticks). SQL: `CONCAT('Hello ', @name)`.
Template Literal	JavaScript strings using backticks (`` ` ``) instead of quotes. Support `${expression}` interpolation and multi-line strings. Essential for building HTML strings in JS.
Transaction	A group of SQL operations that either ALL succeed or ALL fail. BEGIN TRANSACTION > do stuff > COMMIT (save all) or ROLLBACK (undo all). Prevents partial updates that corrupt data.

Term	Definition
Type Casting	Converting a value from one type to another. C#: `int.Parse("42")`, `Convert.ToInt32()`. JS: `Number("42")`, `String(42)`. SQL: `CAST('42' AS INT)`, `CONVERT(INT, '42')`.
Variable	A named container for a value. Declared with `let`/`const` (JS), type + name (C#), just name (Python), `DECLARE @name TYPE` (SQL). Name should describe what it holds.
Viewport	The visible area of the browser window. `100vw` = full viewport width. `100vh` = full viewport height. The `<meta name="viewport">` tag is required for responsive mobile rendering.

The database is the truth. Everything else is a copy. Plan the work. Work the plan. Phase by phase. A working ugly app beats a beautiful broken one.

Vidimus Omnia.

22nd Survey Division | PTE WU | Cert IV Programming