Course Companion

AP Computer Science A

Every topic handout in one volume

全部专题讲义合集

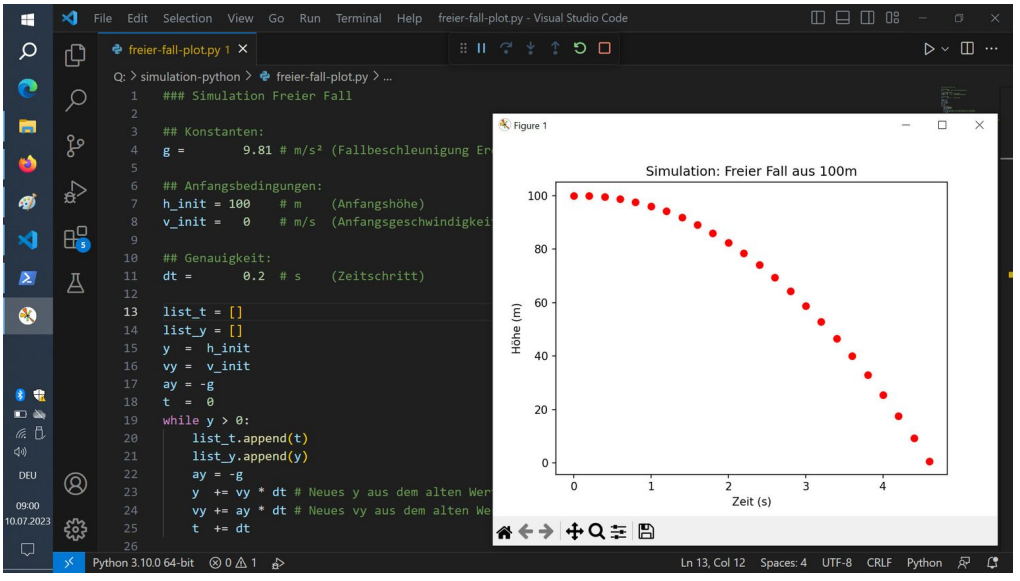
exercises.lol

Contents 目录

Using Objects and Methods

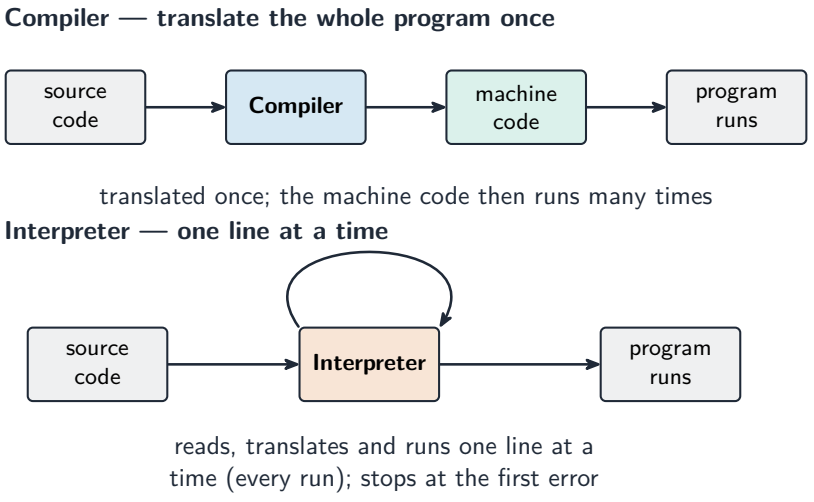
AP Computer Science A

Introduction to Algorithms, Programming, and Compilers

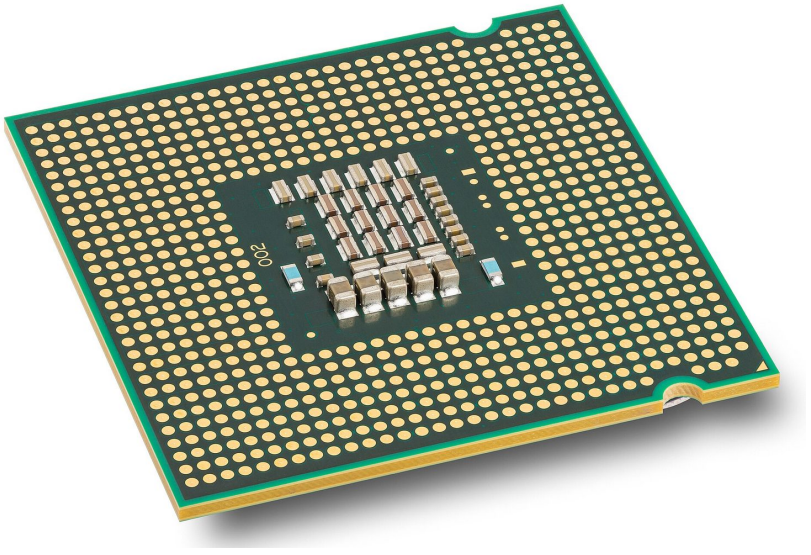


Source code on a workstation — programs are written, compiled, and run as precise instructions

An **algorithm** 算法 is a finite, step-by-step procedure that solves a problem. A **program** 程序 expresses an algorithm in a language a computer can run. Java is **compiled** 编译: the **compiler** 编译器 translates your source code into **bytecode**, which the **Java Virtual Machine (JVM)** runs. A **syntax error** 语法错误 (breaking the grammar) is caught by the compiler; a **logic error** 逻辑错误 (wrong result) is not – the program runs but misbehaves.



A compiler translates the whole program at once; an interpreter runs it line by line



Your Java program is compiled to instructions a CPU like one of these actually runs

Variables and Data Types

A **variable** 变量 is a named box that stores a value of a fixed **type** 类型. Java's main **primitive types** 基本类型 are `int` (whole numbers), `double` (decimals), and `boolean` (`true`/`false`). Declare with the type first:

type	*stores*	*example*
`int`	whole number	`7`
`double`	decimal number	`3.5`
`char`	one character	`'A'`
`String`	text (a sequence of chars)	`"hello"`
`boolean`	true or false	`true`

primitives: int, double, char, boolean · String is an object

Java's basic data types, each storing a different kind of value

```
int score = 90;
double price = 4.99;
boolean passed = true;
```

Expressions and Output

An **expression** 表达式 combines values and **operators** to compute a result: `+ - * / ` and `%` (**modulus** 取模, the remainder). **Integer division** truncates: `7 / 2` is `3`, while `7 % 2` is `1`. **Operator precedence** follows math (`*`,`/`,`%` before `+`,`-`). Print with:

```
System.out.print("no newline");
System.out.println("with newline");
```

Dividing an integer by the integer `0` (like `7 / 0`) is not allowed and crashes at run time with an `ArithmeticException`. Inside a string, a backslash marks an **escape sequence** 转义序列: `\"` prints a double quote, `\\` a single backslash, and `\n` starts a new line – so `System.out.println("She said \"hi\"");` prints `She said "hi"`.

Assignment Statements and Input

An **assignment** 赋值 `x = expr;` evaluates the right side and stores it in the left variable. Read input with a `Scanner`:

```
Scanner in = new Scanner(System.in);
int age = in.nextInt();
String name = in.next();
```

Casting and Range of Variables

Each type has a fixed range; an `int` overflows past about 2.1 billion. **Casting** 类型转换 converts between types. Widening (`int` to `double`) is automatic; narrowing needs an explicit cast, which **truncates** (does not round):

```
double avg = (double) total / count;   // force real division
int whole = (int) 3.9;                 // 3, truncated
```

Exam skill: watch for integer division producing a truncated result when a decimal was expected – cast one operand to `double` first.

Worked example. Trace each expression:

- `7 / 2` → `3` (both `int`, so division **truncates**);
- `7.0 / 2` → `3.5` (one `double` forces real division);
- `7 % 2` → `1` (the remainder);
- `(double) 7 / 2` → `3.5` (the cast binds tighter than `/`, so it is `7.0 / 2`);
- `(double) (7 / 2)` → `3.0` (the parentheses compute `7 / 2 = 3` in `int` **first**, then widen).

The last two look alike but differ – the position of the cast decides whether the truncation happens.

Compound Assignment Operators

Shorthands combine an operation with assignment: `x += 5` means `x = x + 5`; likewise `-=`, `*=`, `/=`, `%=`. The **increment** and **decrement** operators `x++` and `x--` add or subtract one.

Application Program Interface (API) and Libraries

An **API** (Application Programming Interface) 应用程序接口 is the published list of classes and methods you may use. A **library** 库 is a collection of ready-made classes (like `Math`, `String`, `Scanner`). You read the API documentation to learn what a method needs (its parameters) and returns, without seeing its inner code – an example of **abstraction** 抽象.

Documentation with Comments

Comments 注释 are ignored by the compiler but explain code to humans: `//` for a single line, `/* ... */` for a block, and `/** ... */` for a **Javadoc** comment that documents a method's purpose, parameters, and return value. Precise **preconditions** and **postconditions** are written here.

Method Signatures

A **method signature** 方法签名 is a method's name plus its parameter types, e.g. `nextInt()` or `substring(int, int)`. To call a method you must supply **arguments** 实参 that match the parameters in number, type, and order. The method header (the full declaration) also states the **return type** – the type of value the method gives back (`void` if none) – but the return type is *not* part of the signature, which is why two methods cannot differ by return type alone.

Calling Class Methods

A **class (static) method** 类方法 belongs to the class itself, so you call it on the **class name**: `ClassName.method(args)`. No object is needed.

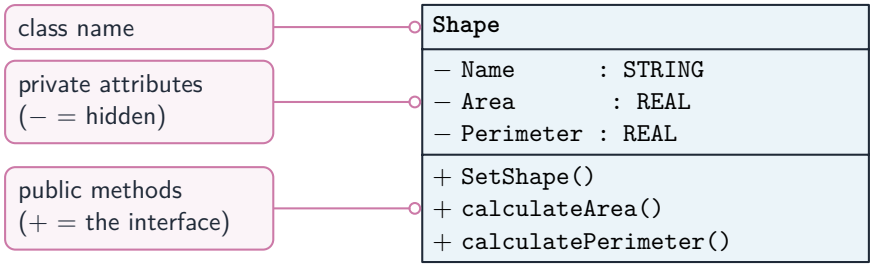
Math Class

The `Math` class provides static math methods: `Math.abs(x)`, `Math.pow(base, exp)`, `Math.sqrt(x)`, and `Math.random()` (a `double` in $[0, 1)$). To get a random integer from `0` to `n-1`: `(int)(Math.random() * n)`.

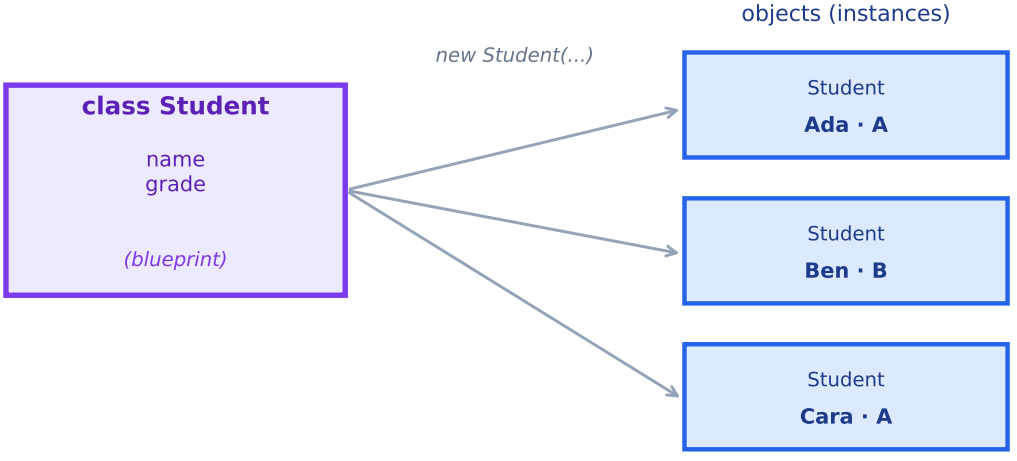
Objects: Instances of Classes

A **class** 类 is a blueprint; an **object** 对象 is a concrete **instance** 实例 built from it. A class bundles **data** (fields) with **behavior** (methods) – the heart of **object-oriented programming** 面向对象编程. `String`, `Scanner`, and `ArrayList` are all classes you instantiate.

Classes can be organised into a hierarchy. A **superclass** 父类 holds attributes and behaviors shared by several **subclasses** 子类 that `extend` it – an **inheritance relationship** 继承关系. Every class in Java is ultimately a subclass of the built-in `Object` class, which is why every object already has a `toString` method; writing a subclass method with the same signature as a superclass one is **method overriding** 方法重写. (Designing your own inheritance is beyond this course, but you are expected to recognise this vocabulary.)



A class diagram: private attributes and public methods



A class is a blueprint; each object is one instance built from it

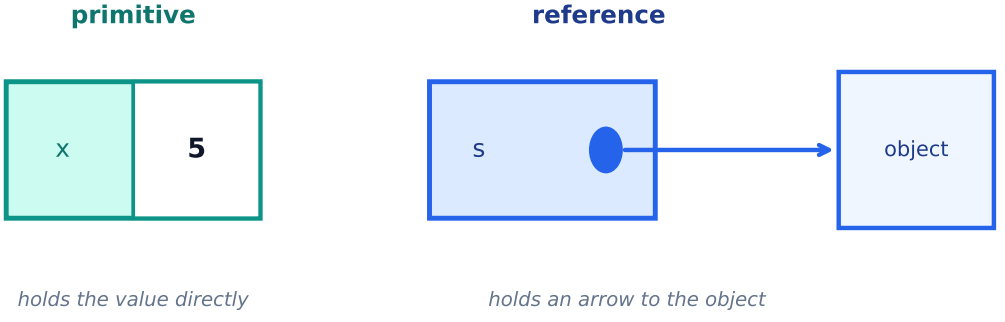
Object Creation and Storage (Instantiation)

Instantiation 实例化 creates an object with the `new` keyword, which calls a **constructor** 构造函数:

```
Scanner in = new Scanner(System.in);
String s = new String("hi");   // or just "hi"
```

The variable holds a **reference** 引用 (the object's address), not the object itself. Two references can point to the **same** object; comparing them with `==` compares addresses, not contents.

A reference can also point to nothing: the special value **null** 空值 means "not attached to any object". Calling a method on a null reference crashes at run time with a `NullPointerException`. Guard against it by testing with `==`/`!=` and checking null *first*, so `&&` short-circuits before the method runs: `if (s != null && s.length() > 0)`.



two references can alias the same object; assignment of a primitive copies the value

A primitive variable holds its value directly, a reference holds an arrow to the object

Calling Instance Methods

An **instance method** 实例方法 acts on a specific object, so you call it on the **object reference**: `object.method(args)`. Example: `in.nextInt()`, `word.length()`.

String Manipulation

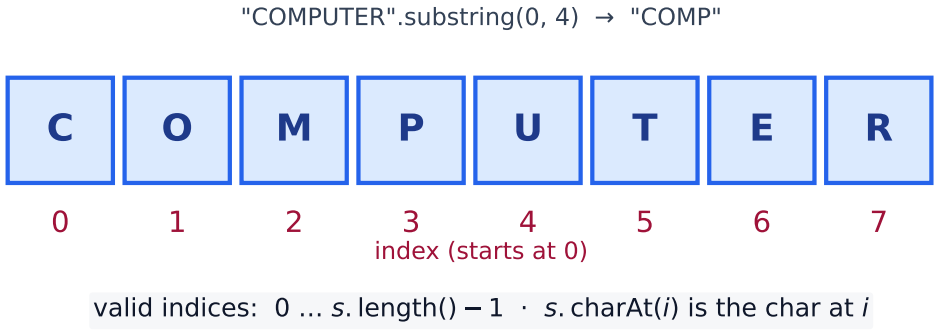
`String` objects are **immutable** 不可变 – methods return a **new** string rather than changing the original. Key methods (all indices start at `0`):

```
s.length();             // number of characters
s.substring(2, 5);      // chars at index 2,3,4 (5 excluded)
s.indexOf("ab");        // first position, or -1
s.equals(other);        // content comparison (never use == for Strings)
s.compareTo(other);     // <0, 0, >0 by dictionary order
```

Exam skill: `substring(a, b)` includes index `a` but **excludes** `b`, and String comparison must use `.equals`, not `==` – two of the most-tested String pitfalls.

Worked example. Let `String s = "COMPUTER";` (indices 0–7). Then `s.length()` is `8`; `s.substring(0, 4)` is `"COMP"` (indices `0,1,2,3` – index `4` excluded); `s.substring(4)` is `"UTER"` (from index `4` to the end); `s.indexOf("PU")` is `3`; and `s.indexOf("X")` is `-1` (not found). Counting the excluded endpoint of `substring` is the single most common slip.

Asking for an index outside `0` to `length()-1` (a bad `substring` or `charAt` argument, e.g. `s.substring(0, 20)` here) crashes with a `StringIndexOutOfBoundsException` – the String cousin of the array-index error.



String indices start at 0

Exam tips

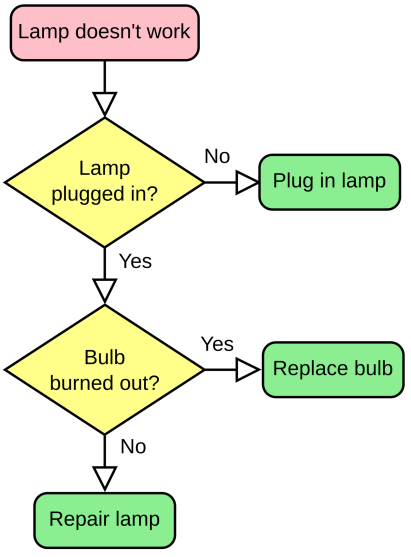
- Trace code by hand line by line, tracking each variable's value in a table — the exam rewards careful tracing over guessing.
- Know Java's primitive types and that integer division truncates (7/2 gives 3); use a cast or a double for real division.

- Distinguish **compile-time** errors (syntax, types) from **run-time** errors – know the named ones: `ArithmeticException` (`int` ÷ `0`), `NullPointerException` (method on a null reference), `StringIndexOutOfBoundsException` / `ArrayIndexOutOfBoundsExce` – and **logic** errors (wrong output).
- Follow operator precedence and initialise every variable before you use it.
- On the free-response, write **complete, compilable** Java — return the right type and match the method header exactly.

Selection and Iteration

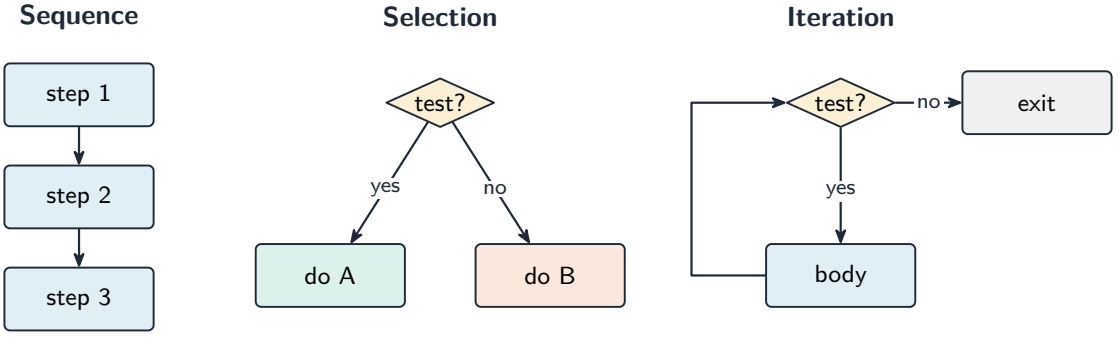
AP Computer Science A

Selection and Repetition in Algorithms



A flowchart with a decision diamond: selection chooses which path the algorithm takes

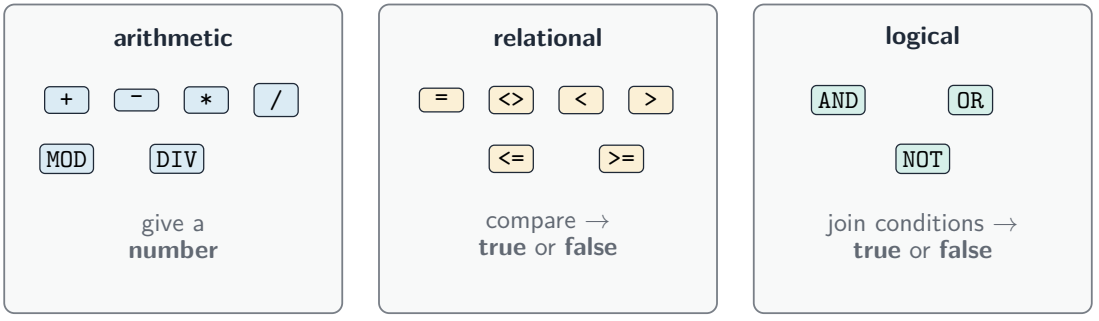
Algorithms are built from three **control structures** 控制结构: **sequence** (steps in order), **selection** 选择 (choosing a path), and **iteration** 迭代 (repeating steps). This topic covers selection and iteration – the tools that let a program make decisions and loop.



The three control structures: sequence, selection, and iteration

Boolean Expressions

A **boolean expression** 布尔表达式 evaluates to `true` or `false`, using **relational operators** 关系运算符: `==` (equal), `!=` (not equal), `<`, `>`, `<=`, `>=`. Note `==` compares primitive values but **object references** for objects, so use `.equals` for `String`s.



The three families of operators: arithmetic, relational, and logical

The if Statement

An **if statement** 条件语句 runs a block only when its condition is true; an optional `else` gives an alternative:

```
if (score >= 60) {
    System.out.println("Pass");
} else {
    System.out.println("Fail");
}
```

Traffic lights: selection chooses which branch runs, just as if statements choose code paths

Nested if Statements

Placing an `if` inside another, or chaining with `else if`, tests several cases in order. Only the **first** matching branch runs:

```
if (g >= 90) grade = 'A';
else if (g >= 80) grade = 'B';
else grade = 'C';
```

Compound Boolean Expressions

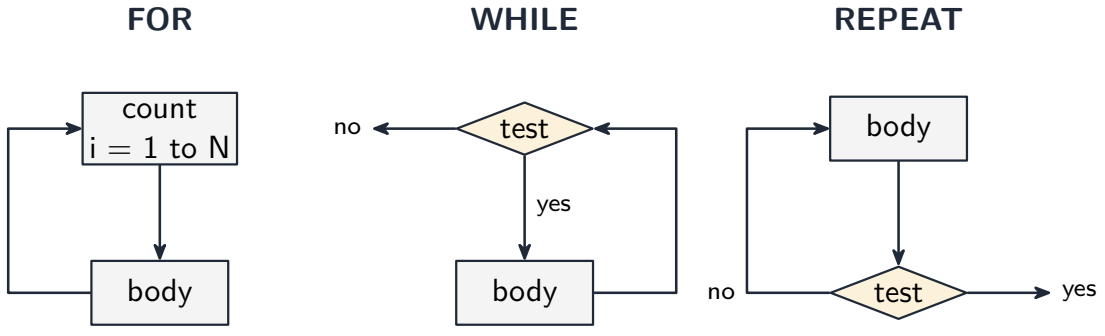
Logical operators 逻辑运算符 combine conditions: `&&` (**and** – both true), `||` (**or** – at least one true), `!` (**not** – reverse). Java uses **short-circuit evaluation** 短路求值: `&&` stops if the left side is false, and `||` stops if the left side is true – useful to guard against errors, e.g. `if (n != 0 && total / n > 5)`.

Comparing Boolean Expressions

De Morgan's laws 德摩根定律 rewrite negations: `!(a && b)` equals `!a || !b`, and `!(a || b)` equals `!a && !b`. Two boolean expressions are **equivalent** if they give the same result for every input – a truth table proves it. Simplifying conditions this way is a common exam task.

while Loops

A **while loop** 循环 repeats **while** its condition stays true, testing **before** each pass. You must change something inside so the loop eventually stops, or it becomes an **infinite loop** 无限循环:



The three loop types differ in where the condition is tested

```
int i = 0;
while (i < 5) {
    System.out.println(i);
    i++;
```

```
}
```

for Loops

A **for loop** packs initialization, condition, and update into one line – best when you know the count:

```
for (int i = 0; i < n; i++) {
    // runs n times, i = 0..n-1
}
```

A `for` and an equivalent `while` do the same work; be able to convert between them.

An assembly line: loops repeat a process for every item, like for and while

Building Complete Selection and Iteration Algorithms

Combine loops and conditions to solve real problems – count, sum, find a maximum, or test a property:

```
int max = arr[0];
for (int k = 1; k < arr.length; k++) {
    if (arr[k] > max) max = arr[k];
}
```

Two integer patterns the exam tests directly use `%` and `/`. To **read the digits of an integer** one at a time, repeatedly take `n % 10` (the last digit) and then `n = n / 10` (drop it). To test **divisibility**, `n % d == 0` means `n` is evenly divisible by `d`. Combine them with a counter to **find the frequency** with which some criterion is met.

Standard patterns like a running total, a counter, or a **flag** 标志 (a boolean that records whether something happened) recur throughout the course.

String Algorithms

Loop through a string by index to process each character:

```
for (int i = 0; i < s.length(); i++) {
    char c = s.charAt(i);
    // count vowels, reverse, check for a substring, ...
}
```

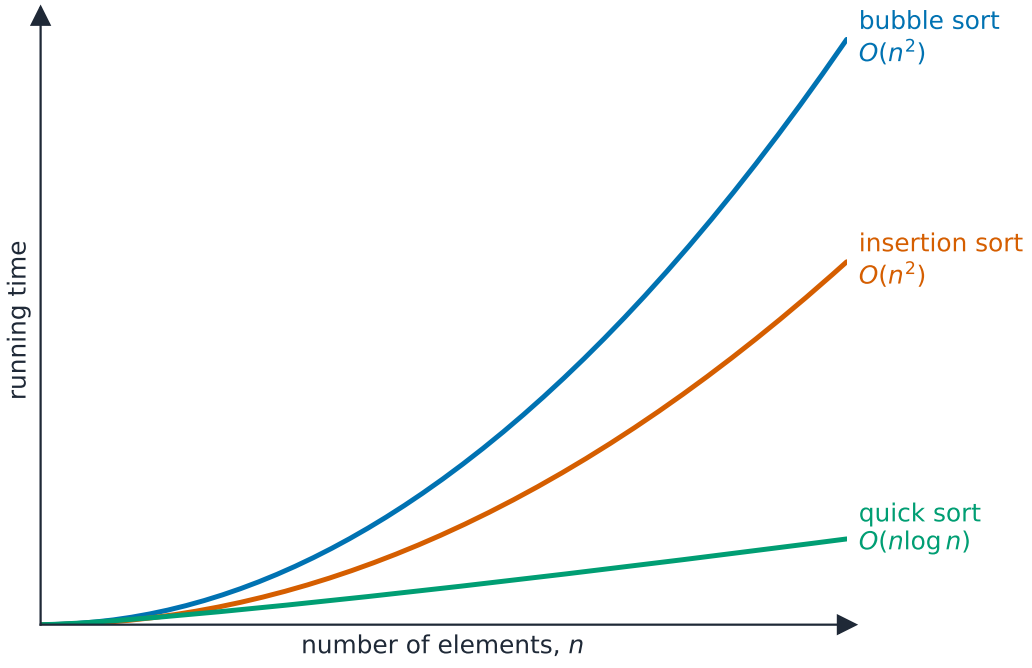
Typical tasks: count occurrences, build a reversed or filtered copy, or test whether one string contains another.

Nested Iteration

A **nested loop** 嵌套循环 puts one loop inside another; the inner loop completes fully for **each** pass of the outer. If the outer runs n times and the inner m times, the body runs $n \times m$ times – the basis for processing grids and comparing all pairs.

Informal Run-Time Analysis

Run-time analysis 运行时间分析 counts how many basic steps an algorithm takes as the input size n grows. Count the executions of the innermost statement: a single loop over n items is **linear** (n steps); two nested loops over n are **quadratic** (n^2). This informal counting lets you compare two algorithms' efficiency.



How the running time grows with the number of elements n

Exam skill: for a nested loop, be able to state how many times the inner statement runs in terms of the loop bounds – a frequent multiple-choice question.

Worked example. How many stars does this print?

```
for (int i = 0; i < 4; i++)
    for (int j = 0; j < i; j++)
        System.out.print("*");
```

The inner loop runs `i` times for each outer `i`: `0 + 1 + 2 + 3 = 6` stars. When the inner bound is the *outer* variable, the total is the triangular sum $0+1+\cdots+(n-1) = \dfrac{n(n-1)}{2}$ – here $\dfrac{4\times 3}{2} = 6$ – not the full $n^2 = 16$ of a rectangular nested loop.

Exam tips

- Get boundary conditions right: use `<` vs `<=` deliberately, and watch the first and last iteration of every loop (off-by-one is the classic bug).
- Build compound conditions with `&&`, `||`, `!` and remember **short-circuit** evaluation (put the null check first).
- Trace nested loops by counting how many times the **inner** body runs in total.
- Choose the right structure — `if/else if` for ranges, a loop for repetition — and avoid an infinite loop by updating the loop variable.
- Apply **De Morgan's laws** when you simplify or negate a boolean condition.

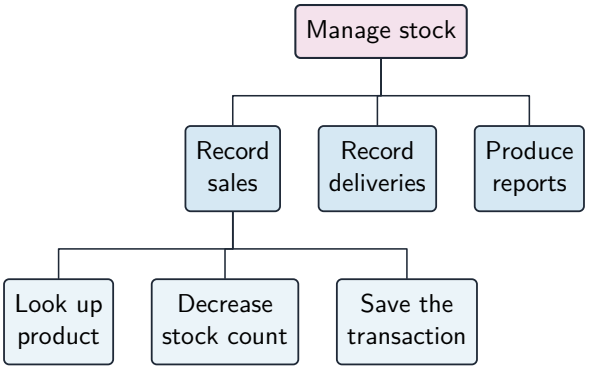
Class Creation

AP Computer Science A

Abstraction and Program Design

A jigsaw in progress: classes and methods are modular pieces of a larger program design

Abstraction 抽象 means hiding detail behind a simple interface – you use a `String` without knowing how it stores characters. Good design breaks a problem into classes, each responsible for one idea. This topic is about writing **your own** classes.



Decomposing a program into modules and sub-modules

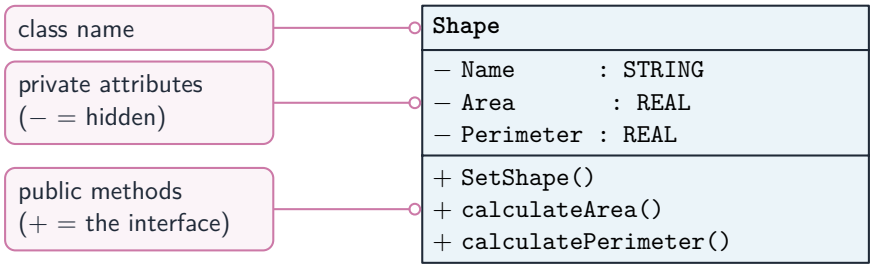
The Impact of Program Design

Design choices affect whether code is correct, readable, and reusable. **Encapsulation** 封装 – keeping data private and exposing it only through methods – protects an object's state from misuse and lets you change the inside without breaking users of the class. Thoughtful naming, single-purpose methods, and testing reduce bugs.

Design also carries responsibility beyond the code. **System reliability** 系统可靠性 - a program performing its tasks as expected, without failure - is something programmers should maximise through careful design and testing. Programs have real impacts on society, the economy, and culture that can be **both beneficial and harmful**. And creating programs raises **legal and intellectual-property** 知识产权 concerns: programmers often reuse code published as **open source** 开源 and free to use, but must respect its licence and give credit rather than copy others' work as their own.

The Anatomy of a Class

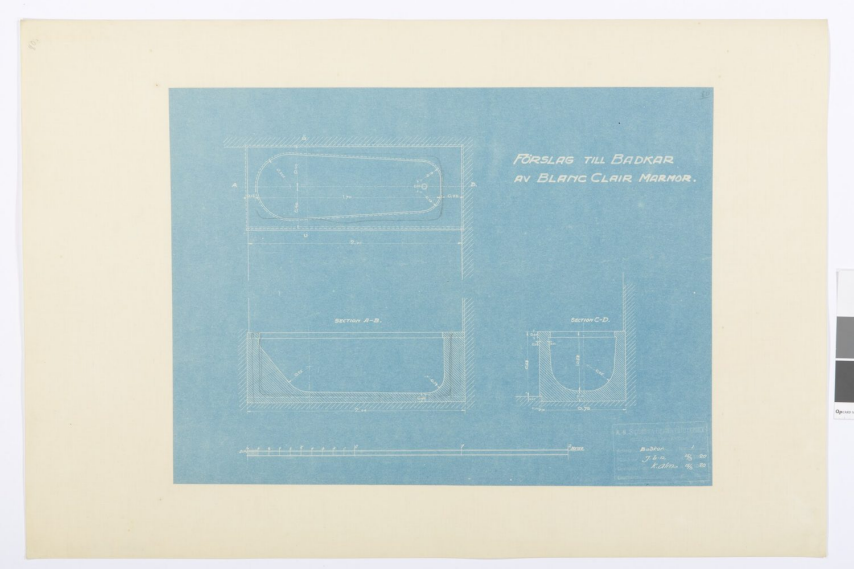
A class has three parts: **instance variables** 实例变量 (fields – the object's data), **constructors** (build objects), and **methods** (behavior). Fields are usually `private`; methods are usually `public`:



A class diagram: private attributes and public methods

```
public class Student {
    private String name;      // instance variable
    private int score;

    public Student(String n, int s) {   // constructor
        name = n;
        score = s;
    }
    public int getScore() { return score; }   // accessor
}
```



A blueprint: a class is a template that defines how objects of that type are built

Constructors

A **constructor** 构造函数 has the **same name as the class** and no return type. It runs when you write `new`, and its job is to initialize the fields. A class can have several constructors with different parameter lists (**overloading** 重载); a **no-argument** constructor sets defaults.

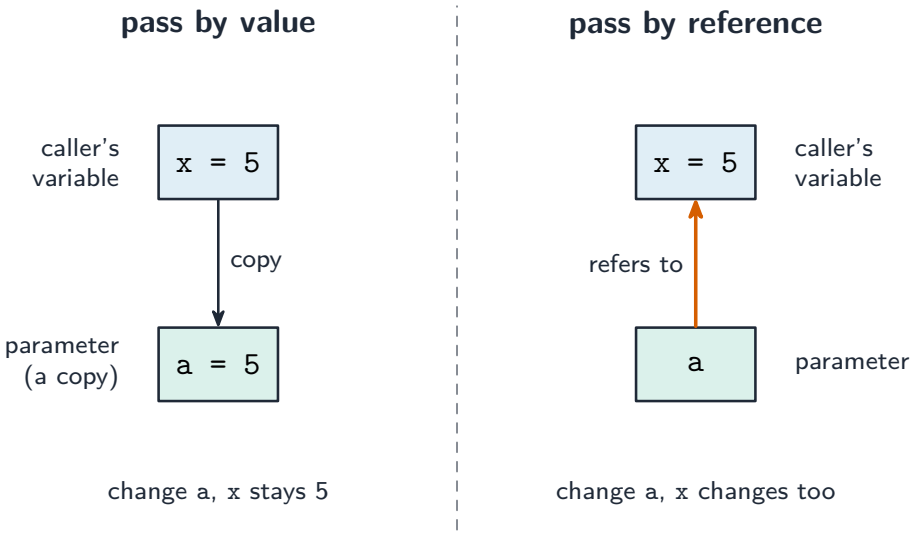
Methods: How to Write Them

A method has a signature, a return type, and a body. An **accessor (getter)** 访问器 returns information without changing the object; a **mutator (setter)** 修改器 changes a field. A method returning a value must have a `return` of the right type on every path; a `void` method returns nothing.

```
public void setScore(int s) { score = s; }   // mutator
public String toString() { return name + ": " + score; }
```

Passing and Returning References of an Object

When you pass an object to a method, Java copies the **reference**, so the method acts on the **same** object – changes to its fields are visible to the caller. (Primitives are copied by value, so changes to them are not.) A method can also **return** a reference to an object. Because a `String` is immutable, passing one is safe; passing a mutable object lets a method change it.



*Java always passes **by value** (left): the method gets a copy of the reference. True pass by reference (right) — which Java does not have — would let a method reassign the caller's own variable.*

Exam skill: know that mutating an object's fields inside a method affects the original, but **reassigning** the parameter (`param = new...`) does not affect the caller.

Worked example. Suppose `s` is a `Student` with score `50`, and we call `tweak(s)`:

```
public static void tweak(Student a) {
    a.setScore(100);          // (1) mutates the shared object
    a = new Student("Z", 0);  // (2) repoints the local copy only
    a.setScore(5);            // (3) changes only the new local object
}
```

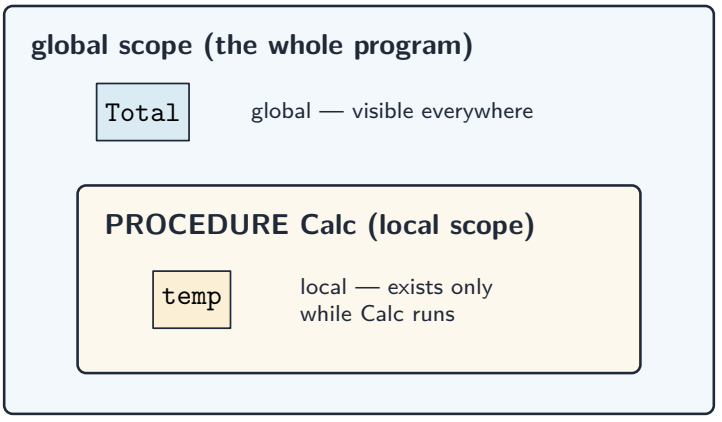
Line (1) changes the object `s` points to, so the caller now sees `100`. Line (2) makes the method's own copy of the reference point at a fresh object – the caller's `s` is untouched – and line (3) affects only that new object. After the call, `s.getScore()` is `100`: the mutation stuck, the reassignment did not.

Class Variables and Class Methods

A **static (class) variable** 类变量, marked `static`, is shared by **all** objects of the class – one copy total (e.g. a counter of how many objects exist). A **static method** belongs to the class and cannot use instance fields directly. Access them by class name: `Student.getCount()`.

Scope and Access

Scope 作用域 is where a name is visible. A **local variable** declared in a method exists only inside it; a **parameter** exists only in its method; an **instance variable** is visible throughout the object. Access modifiers control visibility across classes: `private` (this class only) versus `public` (anywhere). Local variables **shadow** fields of the same name – a source of bugs.



A global variable is visible everywhere; a local variable only inside its block

The this Keyword

`this` is a reference to the **current object**. Use it to tell a field apart from a parameter with the same name, or to call another method of the same object:

```
public Student(String name, int score) {
    this.name = name;      // this.name is the field; name is the
↪   parameter
    this.score = score;
}
```

Exam skill: when a constructor or setter's parameter has the same name as a field, you **must** write `this.field = param` – without `this`, the assignment does nothing useful.

Exam tips

- Design with methods and classes: encapsulate data as `private` fields and expose behaviour through public methods.

- Know the difference between an **object** and its **class**, and that objects are passed **by value** — the parameter gets a copy of the reference, so a method can change the object's state, but reassigning the parameter does not affect the caller (Java has no pass-by-reference).
- Traverse arrays and `ArrayList`s safely — size is `length` vs `.size()`, and removing during a loop shifts indices.
- **Trace** a **recursive** method to determine its result: find the base case first, then follow each recursive call to its return value (writing recursive code is outside the exam's scope).
- Recognise **inheritance** vocabulary — superclass, subclass, method overriding, and that every class is a subclass of `Object` (designing and implementing inheritance is outside the exam's scope).

Data Collections

AP Computer Science A

The Ethics of Collecting Data

Server racks in a data centre — large collections of data raise ethical questions about collection and use

Programs that gather data raise questions of **privacy** 隐私 and **consent** 同意. Collect only what is needed, protect it, and be honest about its use. Data can carry **bias** 偏见 if it does not represent everyone fairly, leading to unfair results – a responsibility that comes with storing information.

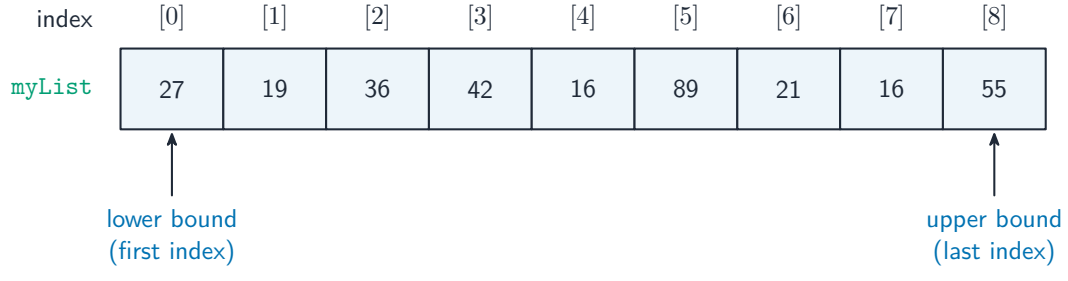
Why We Need Data Structures

A filing cabinet: collections store many values under one name so algorithms can process them

A single variable holds one value; real problems need to store **many** related values – a class roster, pixels, sensor readings. A **data structure** 数据结构 organizes a collection so we can store, find, and process items efficiently. The AP course uses three: the **array**, the **ArrayList**, and the **2D array**.

Making and Reading an Array

An **array** 数组 is a fixed-size, ordered collection of same-type values. Indices run from `0` to `length - 1`:



A one-dimensional array (a list) with its indices and bounds

```java
int[] nums = new int[5];        // five zeros
int[] vals = {3, 1, 4, 1, 5};   // initialized
int first = vals[0];            // 3
int n = vals.length;            // 5 (a field, not a method)
```

Accessing an index outside `0..length-1` throws an `ArrayIndexOutOfBoundsException`.

Visiting Every Element of an Array

Traverse 遍历 an array with a `for` loop (gives the index) or an **enhanced for / for-each** loop (gives each value, read-only):

```java
for (int i = 0; i < a.length; i++) { a[i] *= 2; }   // can modify
for (int v : a) { System.out.println(v); }          // read each value
```

Standard Array Algorithms

Master these patterns: compute a **sum** or **average**, find the **max/min**, **count** items meeting a condition, check for a **duplicate**, and **reverse** or **shift** elements. Each is a traversal with a running result:

```java
int sum = 0;
for (int v : a) sum += v;
double avg = (double) sum / a.length;
```

Reading Data from a Text File

`File` and `IOException` live in `java.io`, so a program that reads a file needs `import java.io.*;`. Opening a file can fail (it might not exist), and Java forces you to handle that – the simplest way is to add `throws IOException` to the method header. A `Scanner` then reads the file line by line, using `hasNext...` to test before reading:

```java
import java.io.*;
...
public static void readFile() throws IOException {
    Scanner f = new Scanner(new File("data.txt"));
    while (f.hasNextLine()) {
        String line = f.nextLine();
    }
}
```

Reading typed tokens with `nextInt()`, `nextDouble()`, or `nextBoolean()` throws an `InputMismatchException` if the next token is the wrong type – for example calling `nextInt()` when the next thing in the file is the word `cat`.

Wrapping a Number in an Object

An `ArrayList` stores **objects**, not primitives, so a primitive is **wrapped** in an object: `Integer` wraps `int`, `Double` wraps `double`. Java does this with **autoboxing** 自动装箱 (`int` to `Integer`) and **unboxing** (back again) automatically, so you can write `list.add(5)` and `int x = list.get(0)`.

The ArrayList Toolbox

An **ArrayList** 动态数组 grows and shrinks as you add or remove items. Declare it with the element type in `<>`:

```
ArrayList<String> names = new ArrayList<String>();
names.add("Amy");           // append
names.add(0, "Bob");        // insert at index
names.get(0);               // read
names.set(1, "Cara");       // replace
names.remove(0);            // delete, shifts the rest left
names.size();               // count (a method, unlike array.length)
```

Visiting Every Element of an ArrayList

Traverse with an index loop or a for-each loop, just like arrays (use `size()` and `get(i)`):

```
for (int i = 0; i < list.size(); i++) { ... list.get(i) ... }
for (String s : list) { ... }
```

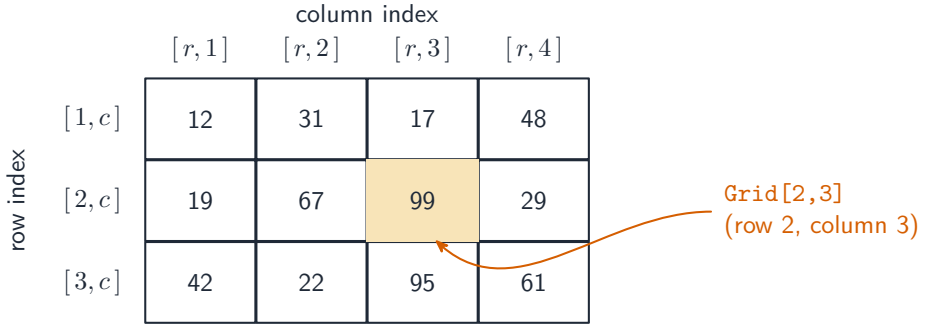
Exam skill: when **removing** items in an index loop, either loop **backwards** or do **not** increment `i` after a removal – otherwise removing shifts elements left and you skip one. And never add or remove elements while traversing an `ArrayList` with a **for-each** loop: changing its size mid-loop throws a `ConcurrentModificationException`, so use an index loop (backwards, as above) whenever you must remove.

Standard ArrayList Algorithms

The same algorithms as arrays – max/min, count, sum – plus **insertion** and **deletion** that arrays cannot do easily. A common task is to remove all elements matching a condition, handling the index-shift carefully.

Grids: Two-Dimensional Arrays

A **2D array** 二维数组 is a grid (rows and columns) – an array of arrays:



A two-dimensional array (a table) with row and column indices

```
int[][] grid = new int[3][4];   // 3 rows, 4 columns
grid[r][c] = 7;                 // row r, column c
int rows = grid.length;         // 3
int cols = grid[0].length;      // 4
```

Walking Through a Grid

Visit every cell with **nested loops** – the outer over rows, the inner over columns (**row-major order** 行主序):

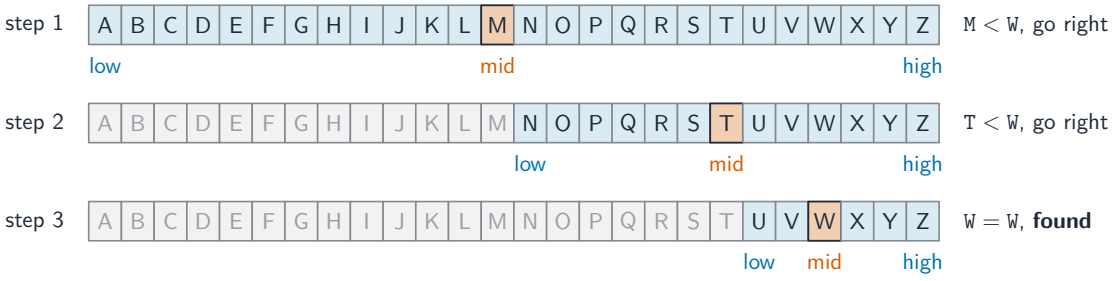
```
for (int r = 0; r < grid.length; r++)
    for (int c = 0; c < grid[0].length; c++)
        System.out.print(grid[r][c]);
```

Standard 2D Array Algorithms

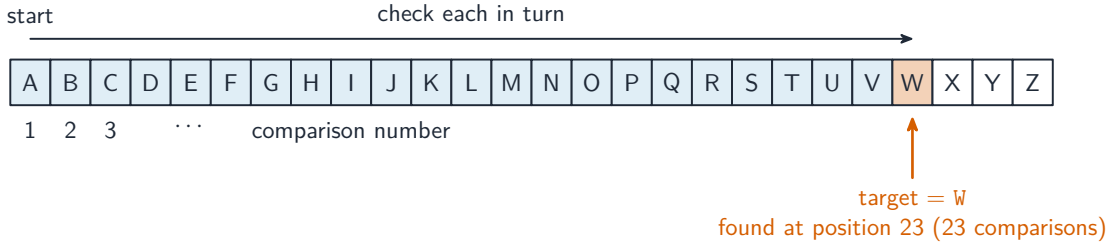
Typical grid tasks: sum a row or column, find the max in the grid, count matching cells, or sum a diagonal (where `r == c`). Each is a nested traversal with a running result.

Finding a Value: Linear and Binary Search

- **Linear search** 线性搜索 checks each element in turn – works on any list, taking up to n steps.
- **Binary search** 二分搜索 works only on a **sorted** list: check the middle, then discard the half that cannot contain the target, repeating. It takes about $\log_2 n$ steps – far faster on large data.



Binary search halves the range at each step



Linear search checks every element in turn until the target is found

```
int lo = 0, hi = a.length - 1;
while (lo <= hi) {
    int mid = (lo + hi) / 2;
    if (a[mid] == target) return mid;
    else if (a[mid] < target) lo = mid + 1;
    else hi = mid - 1;
}
```

Exam skill: binary search **requires sorted data**; know how many comparisons it makes and how `lo`, `hi`, `mid` update.

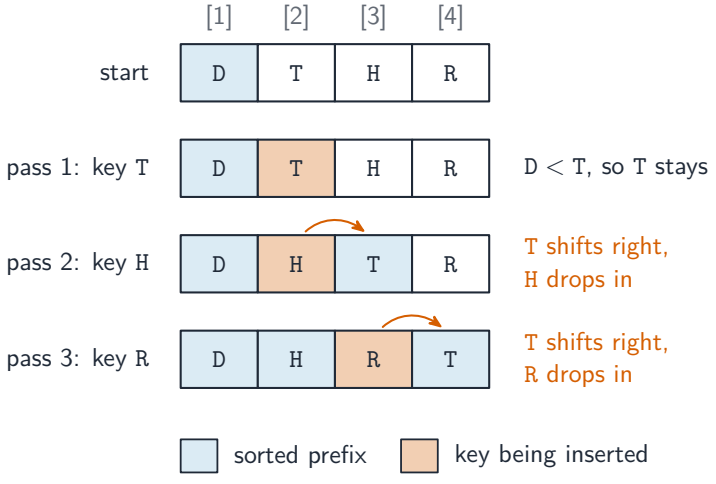
Worked example. Search for `target = 40` in the sorted array `{3, 9, 14, 23, 31, 42, 55}` (indices 0–6). Start `lo=0, hi=6`:

- `mid = (0+6)/2 = 3`, `a[3]=23 < 40`, so `lo = 4`;
- `mid = (4+6)/2 = 5`, `a[5]=42 > 40`, so `hi = 4`;
- `mid = (4+4)/2 = 4`, `a[4]=31 < 40`, so `lo = 5`;
- now `lo (5) > hi (4)`, so the loop ends – 40 is **not present**.

Each step halved the range, so even this miss took only three comparisons.

Putting Data in Order: Selection and Insertion Sort

- **Selection sort** 选择排序 repeatedly finds the smallest remaining element and swaps it into place.
- **Insertion sort** 插入排序 grows a sorted front, inserting each new element where it belongs.



An insertion sort, shifting each key into place pass by pass

Both are simple and take about n^2 steps on average – fine for small arrays. Be able to trace the array **after each pass**.

Methods That Call Themselves: Recursion

Recursion 递归 is a method that calls itself on a smaller input. It needs a **base case** 基本情况 that stops the calls, and a **recursive case** that moves toward the base:

```
public static int factorial(int n) {
    if (n <= 1) return 1;          // base case
    return n * factorial(n - 1);   // recursive case
}
```

Without a reachable base case, recursion never stops (a stack overflow).

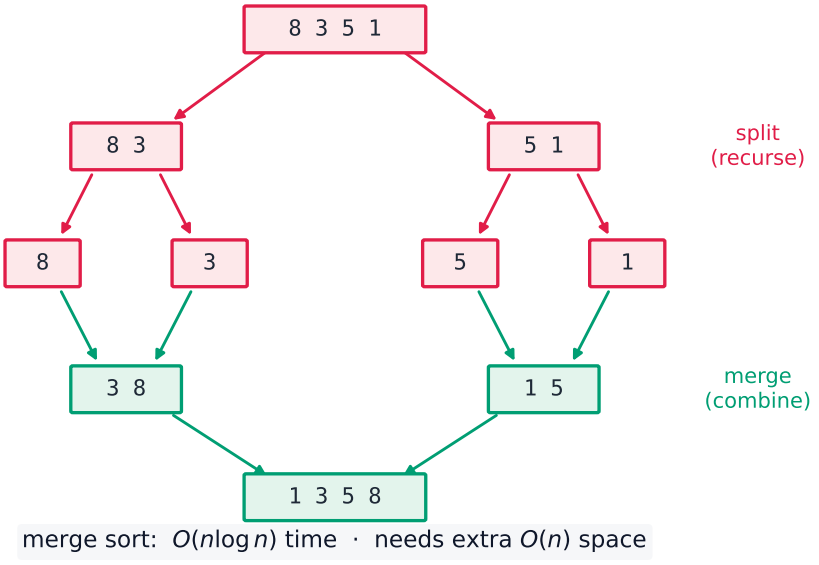
Recursion and iteration are interchangeable. Any recursive solution can be rewritten with a **loop** (an iterative approach), and any loop can be rewritten with recursion - they solve the same problems. The `factorial` above is identical in effect to an iterative version:

```
public static int factorial(int n) {
    int result = 1;
    for (int i = 2; i <= n; i++) result *= i;   // same answer, no
↪   self-call
    return result;
}
```

So the choice is about **clarity, not capability**: recursion reads naturally for problems with a self-similar structure (trees, merge sort), while iteration avoids the memory cost of stacking a call frame per step. The exam may ask you to convert one into the other.

Recursive Search and Merge Sort

Recursion powers efficient algorithms. **Binary search** can be written recursively (search the correct half). **Merge sort** 归并排序 splits the array in half, sorts each half recursively, then **merges** the two sorted halves – taking about $n \log_2 n$ steps, much faster than selection or insertion sort on large data.



Merge sort splits the array to single elements, then merges sorted halves back up

Worked example. Trace `factorial(4)`. Each call defers to a smaller one: `factorial(4) = 4 * factorial(3) = 4 * 3 * factorial(2) = 4 * 3 * 2 * factorial(1)`. `factorial(1)` hits the **base case** and returns `1`, so the calls unwind inward: `2 * 1 = 2`, then `3 * 2 = 6`, then `4 * 6 = 24`. Writing each call above its returned value is the reliable way to trace recursion.

Exam skill: trace a recursive method by writing out each call and its return value, and know that merge sort's efficiency ($n \log n$) beats the n^2 simple sorts.

Exam tips

- Weigh both benefits and harms of collecting data — this unit is tested through short written justification, not code.
- Protect **personally identifiable information (PII)** and explain privacy and security risks in context.
- Name real harms: data breaches, surveillance, and algorithmic **bias** from unrepresentative data.

- Respect intellectual property and licensing when you reuse code or data.
- Give a specific, reasoned answer — a vague "it could be bad" earns no marks.