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

1. Using Objects and Methods 2

2. Selection and Iteration 15

3. Class Creation 29

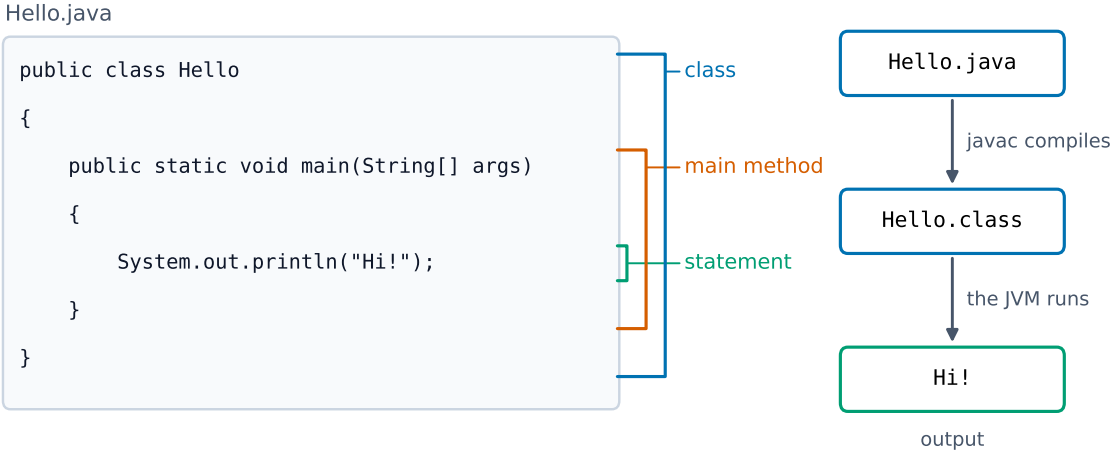
4. Data Collections 41

Using Objects and Methods

AP Computer Science A

Introduction to Algorithms, Programming, and Compilers

Why it matters. On 4 June 1996 the first Ariane 5 rocket broke apart about 37 seconds after launch. Its guidance software converted a 64-bit floating-point number (an internal value related to horizontal velocity) into a 16-bit signed integer. Ariane 5 was faster than the older rocket the code was written for, the value was too large for 16 bits, and the conversion failed - the navigation computers shut down and the rocket veered off course. One line about data types and ranges cost a rocket and its satellites. This unit covers exactly those ideas: types, casting, ranges and what happens when a value does not fit.



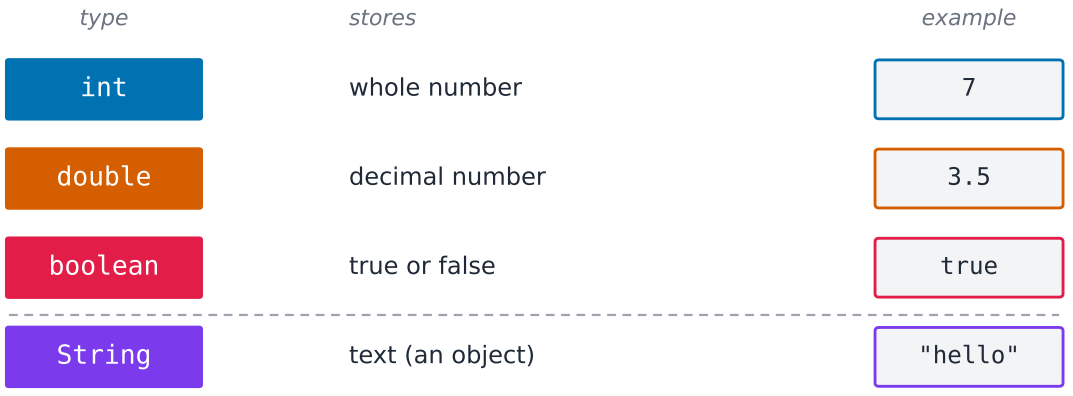
A Java program: a class holds the main method, which holds the statements; javac compiles it to bytecode and the JVM runs it

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.

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:

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



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

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"`.

code

```
System.out.print("Hi ");
System.out.println("there");
System.out.println(3 + 4 + "!");
System.out.println("!" + 3 + 4);
```

console output

```
Hi there
7!
!34
```

print stays on the line; println ends it. + adds numbers left to right until a String appears, then joins text.

print stays on the line, println ends it; + adds numbers until a String appears

Assignment Statements and Input

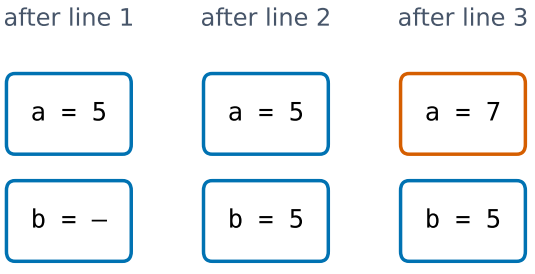
An **assignment** 赋值 `x = expr;` evaluates the right side and stores it in the left variable.

A program's **input** can come in many forms: touch (tactile), sound (audio), images (visual) or text. The `Scanner` class is one way to read text typed on the keyboard:

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

code

```
int a = 5;
int b = a;
a = a + 2;
```



b copied a's value at line 2, so changing a later does not change b

Assignment copies a value into a variable's box; later changes to one variable do not affect the other

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.

`(int) 3.99`	→ `3`	drops the decimal part
`(int) -3.99`	→ `-3`	truncates toward zero
`(int) (3.99 + 0.5)`	→ `4`	round a positive double
`(double) 7 / 2`	→ `3.5`	cast first, then divide
`7 / 2`	→ `3`	int ÷ int stays int
`Integer.MAX_VALUE + 1`	→ `-2147483648`	overflow wraps around

Casting truncates toward zero; int arithmetic overflows past Integer.MAX_VALUE

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.

statement	value after	
`int x = 10;`	`x = 10`	
`x += 5;`	`x = 15`	same as x = x + 5
`x -= 3;`	`x = 12`	x = x - 3
`x *= 2;`	`x = 24`	x = x * 2
`x /= 5;`	`x = 4`	int division: 24 / 5 is 4
`x %= 3;`	`x = 1`	remainder of 4 ÷ 3
`x++;`	`x = 2`	add one

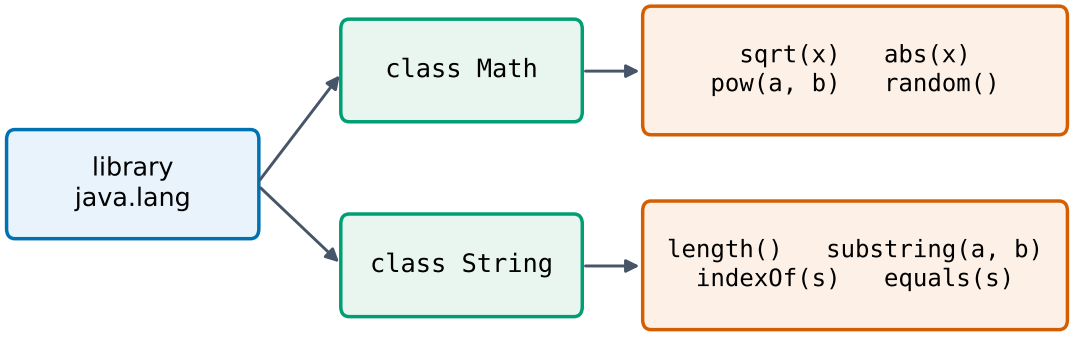
Compound assignment operators traced on one variable

Common mistakes.

- `x =+ 3;` is not `x += 3;`: it reads as `x = +3`, so `x` becomes 3.
- The integer-division rule still applies: with `int x = 25;`, `x /= 4;` gives 6, not `6.25`.
- `x++;` changes `x` itself; `x + 1` on its own changes nothing and does not compile as a statement.

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.



you call methods from their documentation without seeing their code

A library groups classes; each class documents the methods you can call

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.

```
/**
 * Returns the average of a and b.
 * Precondition: a and b are not negative.
 * @param a the first mark
 * @param b the second mark
 * @return the mean of a and b
 */
public double mean(int a, int b)
```

what it does
precondition
@param: each parameter
@return: the result

A Javadoc comment: purpose, precondition, one `@param` per parameter, `@return`

A **precondition** 前置条件 is what must be true before the method is called (the caller's job); a **postcondition** 后置条件 is what the method guarantees afterwards. If a precondition is broken, the method is not required to work: `mean(-3, 5)` breaks the precondition that `a` and `b` are not negative, so its result is not promised.

On the free-response. The exam's directions say that, unless a question says otherwise, methods are called only when their preconditions are satisfied. So when a method's comment says "Precondition: `arr.length > 0`", you may read `arr[0]` straight away: there is no need to test the length or the arguments first. A precondition also tells you which cases you do **not** have to handle, which can make the code shorter.

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.

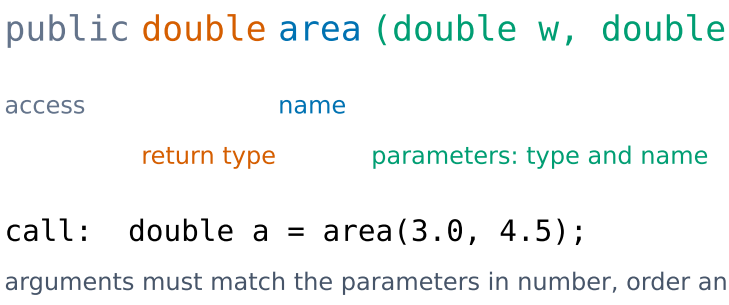
> **Worked example.** `String` has `substring(int beginIndex, int endIndex)` and `indexOf(String str)`. For `String w = "network";`:

- `w.substring(2, 5)` matches: two `int` arguments, returns `"two"`;
- `w.substring("2", "5")` does not compile: `String` arguments where `int` is required;
- `w.indexOf("work")` matches: one `String` argument, returns `3`;

exercises.lol exercises.com.cn

- `int n = w.substring(3);` does not compile: the method **returns** a `String`, which cannot be stored in an `int`.

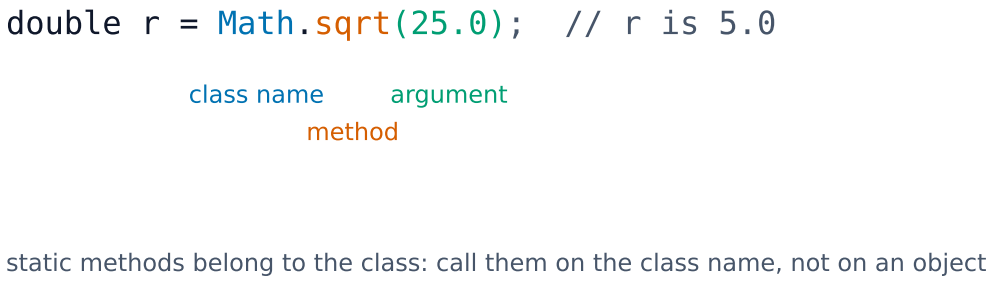
Several methods with the same name but different parameter lists, such as the two `substring` methods, are **overloaded** 重载.



The parts of a method signature and a call that matches it

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.



A class method is called on the class name: Math.sqrt(25.0)

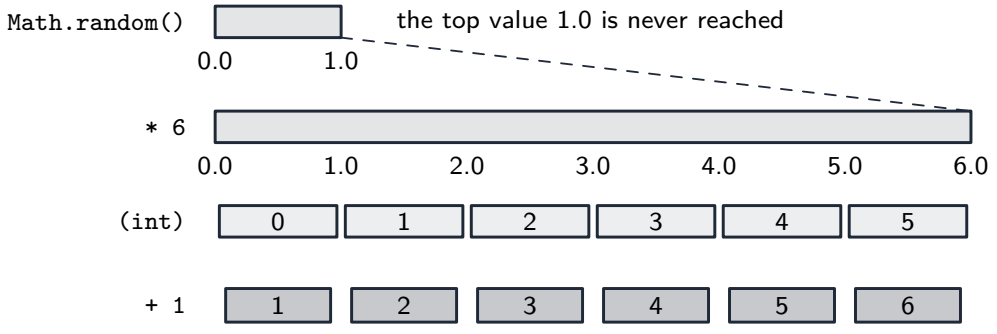
A `void` class method is a statement on its own; a non-`void` one returns a value that should be used:

```
double r = Math.sqrt(25.0);      // 5.0, stored
int big = Math.max(3, 8);        // 8
Math.sqrt(25.0);                 // compiles, but the result is lost
```

A class method that is called from inside its own class can be called by its name alone: `average(a, b)` instead of `Stats.average(a, b)`.

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)`.



From `Math.random()` to a dice roll

For a whole number from `low` to `high` inclusive: `(int) (Math.random() * (high - low + 1)) + low`. The bracket `high - low + 1` counts the values; the cast truncates; `+ low` shifts the start.

Want	Expression
a dice roll, 1 to 6	`(int) (Math.random() * 6) + 1`
10 to 20	`(int) (Math.random() * 11) + 10`
an even number 0 to 10	`(int) (Math.random() * 6) * 2`

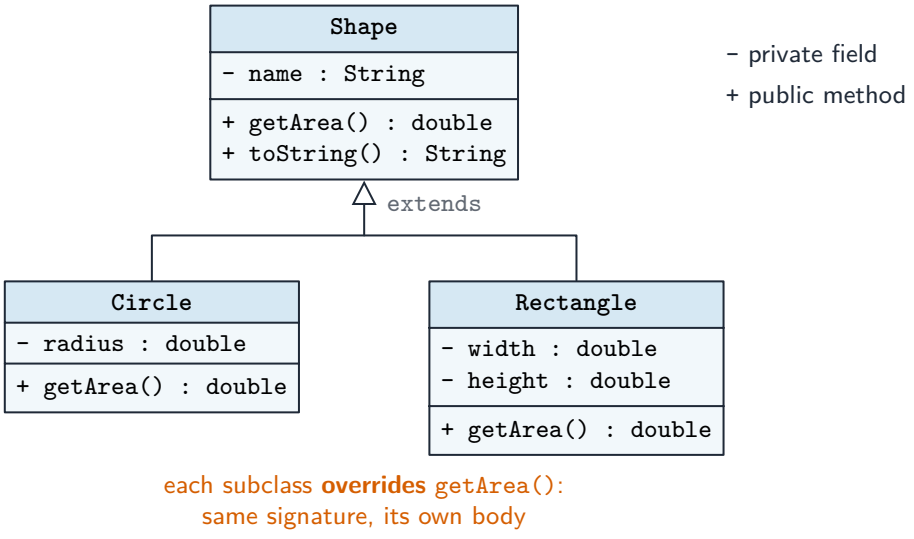
`Math.pow` and `Math.sqrt` always return a `double`, so `int p = Math.pow(2, 3);` does not compile; `(int) Math.pow(2, 3)` gives `8`. Casting **before** multiplying is a classic error: `(int) Math.random() * 6` is always `0`.

Objects: Instances of Classes

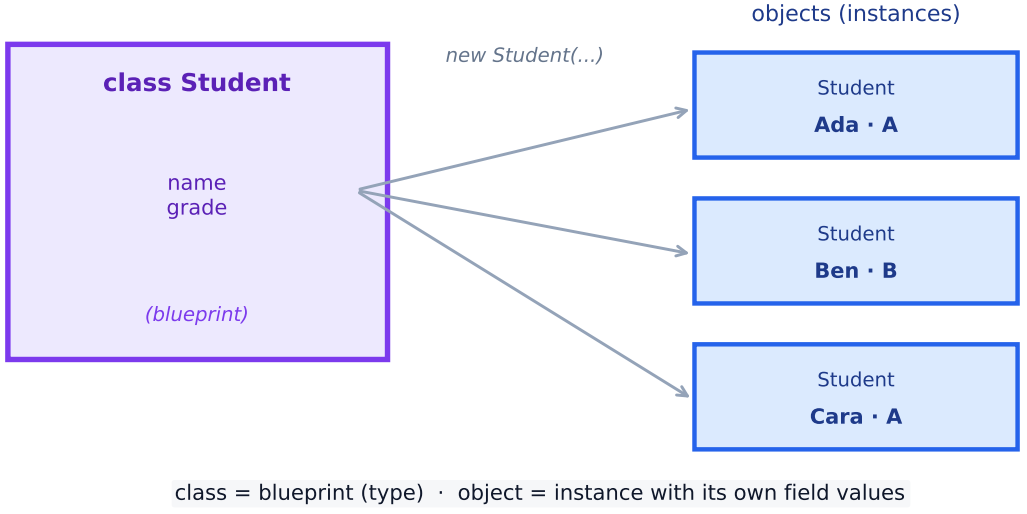
A **class** 类 is a blueprint; an **object** 对象 is a concrete **instance** 实例 built from it. A class bundles **data** (fields) with **behavior** (methods) – the heart of **object-oriented programming** 面向对象编程. `String`, `Scanner`, and `ArrayList` are all classes you instantiate.

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 other than `Object` 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: Circle and Rectangle extend Shape, and each overrides getArea()



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

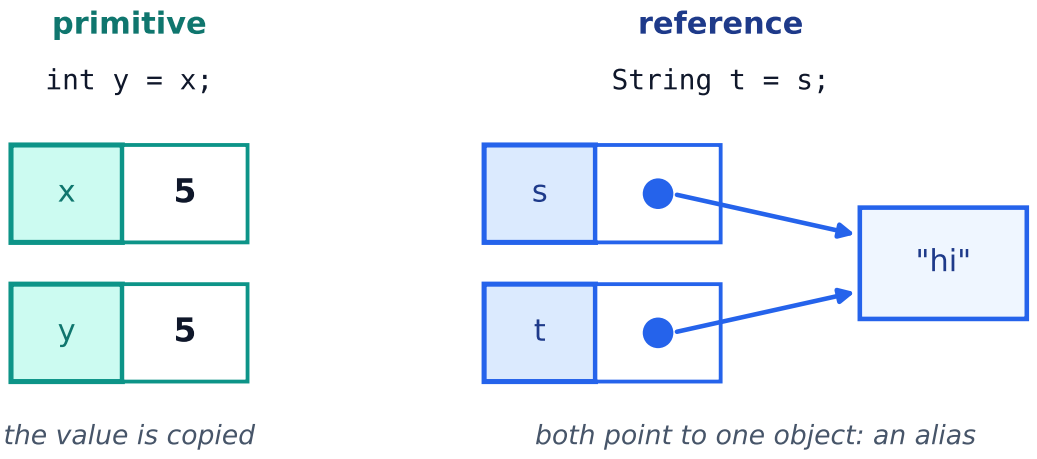
Object Creation and Storage (Instantiation)

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.

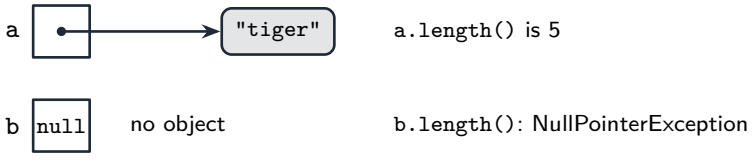
A reference can also point to nothing: the special value **null** 空值 means "not attached to any object". Calling a method on a null reference crashes at run time with a `NullPointerException`. Guard against it by testing with `==`/`!=` and checking null *first*, so `&&` short-circuits before the method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



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()`.



An instance method needs an object: on `null` it throws a `NullPointerException`

Worked example. A `BankAccount` class has `deposit(double amt)`, `withdraw(double amt)` and `getBalance()`.

```
BankAccount a = new BankAccount(100.0);
BankAccount b = a;                  // b refers to the SAME object
b.withdraw(30.0);
a.deposit(5.0);
System.out.println(a.getBalance() + " " + b.getBalance());
```

There is only one object, so both calls change it: $100 - 30 + 5 = 75$, and the output is `75.0 75.0`. Each instance method acts on the object **before the dot**; with two separate `new` objects the balances would differ.

String Manipulation

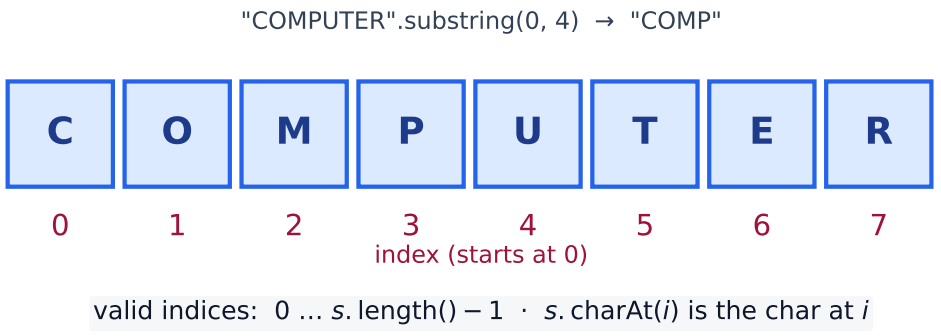
`String` objects are **immutable** 不可变 – methods return a **new** string rather than changing the original. Key methods (all indices start at `0`):

```
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.

For `charAt`, valid indices run from `0` to `length()-1`. For `substring(begin, end)`, the bounds must satisfy `0 <= begin <= end <= length()`: the excluded endpoint may equal the length, and `substring(length())` is an empty string. Invalid bounds, e.g. `s.substring(0, 20)` here, throw 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` / `ArrayIndexOutOfBoundsException` – 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.

Common mistakes (whole unit).

- Expecting `7 / 2` to be `3.5`. Two `int`s give integer division: `3`. Cast **before** dividing: `(double) 7 / 2` is `3.5`, but `(double) (7 / 2)` is `3.0`.
- Forgetting that `substring(a, b)` stops **before** index `b`.
- Calling a `String` method and ignoring the result: `s.toUpperCase();` does not change `s`, because strings are immutable. Write `s = s.toUpperCase();`.
- Calling a method on a variable that holds `null`, which throws a `NullPointerException`.
- Assuming `int` can hold any whole number. Past `Integer.MAX_VALUE` it overflows to a negative value.

Practice.

1. What does `System.out.println(17 / 5 + 17 % 5);` print?
2. Give the values of `(int) 7.9` and `(int) (7.9 + 0.5)`. What does the second expression do for positive numbers?

3. After `int x = Integer.MAX_VALUE; x++;`, what is `x`?
4. Let `String s = "PROGRAM";`. Evaluate `s.substring(2, 5)`, `s.indexOf("R")` and `s.length()`.
5. After `double d = 5 / 2;`, what is `d`, and why?
6. Write one statement that stores in `c` the hypotenuse of a right triangle with `double` legs `a` and `b`.
7. Write an expression that gives a random whole number from 1 to 6 inclusive.

Answers.

1. `17 / 5` is `3` and `17 % 5` is `2`, so it prints `5`.
2. `7` and `8`. Adding `0.5` before casting rounds a positive number to the nearest integer.
3. `Integer.MIN_VALUE` (`-2147483648`): the value overflows and wraps around.
4. `"OGR"`, `1`, `7`.
5. `2.0`: `5 / 2` is integer division (`2`), and only then is it widened to a `double`.
6. `double c = Math.sqrt(a * a + b * b);`
7. `(int) (Math.random() * 6) + 1`

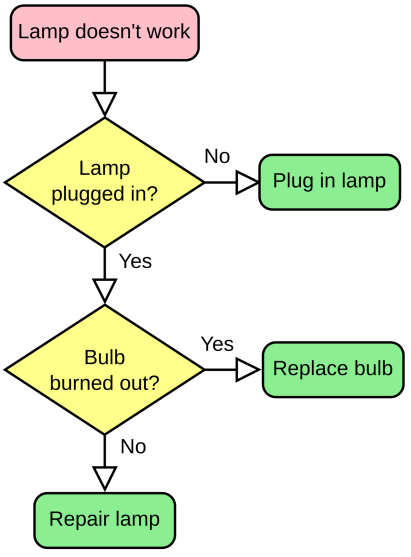
exercises.lol exercises.com.cn

Selection and Iteration

AP Computer Science A

Selection and Repetition in Algorithms

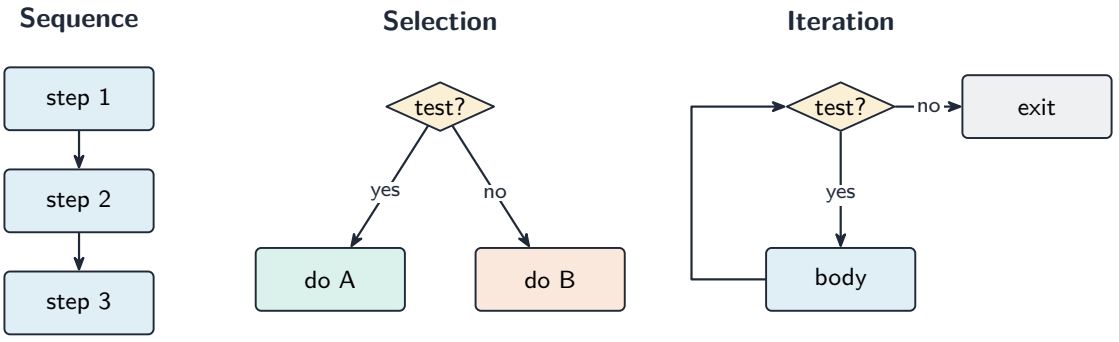
Why it matters. On 31 December 2008, thousands of Microsoft Zune 30 GB music players froze at the same moment. Their clock code used a `while` loop that subtracted 365 or 366 days a year from a day count. On day 366 of a leap year, neither branch of the `if` inside the loop changed the count, so the loop never ended. The players worked again the next day, when the count moved on. One boundary condition in one loop - exactly the kind of selection and iteration logic this unit teaches you to trace.



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

Algorithms 算法 are built from three **control structures** 控制结构: **sequence** 顺序 (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.

exercises.lol exercises.com.cn



The three control structures: sequence, selection, and iteration

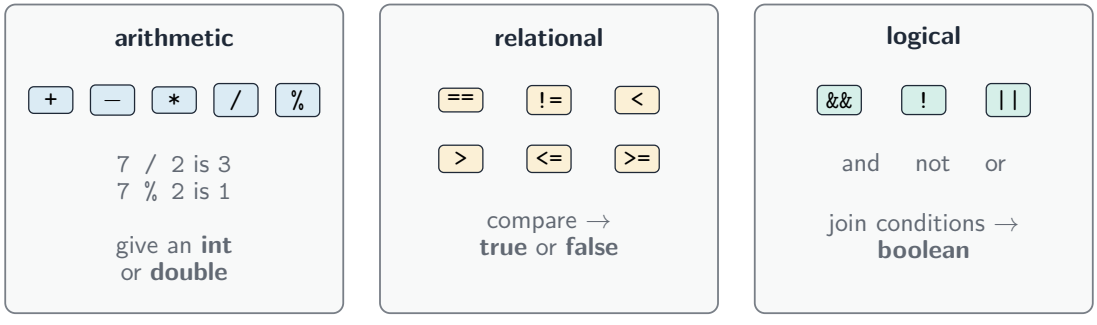
Boolean Expressions

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** 基本类型值 but **object references** 对象引用 for objects, so use `.equals` for `String`s.

Worked example. With `int a = 7, b = 3;` and `String s = "cat";`:

Expression	Value	Why
`a % b == 1`	`true`	`7 % 3` is `1`
`a / b >= 2.5`	`false`	`7 / 3` is the `int` `2`
`a != b * 2`	`true`	`b * 2` is 6; arithmetic runs before comparing
`s.equals("cat")`	`true`	same characters
`s.length() > b`	`false`	`3 > 3` is false

A common mistake is `=` for `==`: `if (a = 7)` does not compile, because `=` assigns and gives an `int`, not a `boolean`.



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

The if Statement

An **if statement** 条件语句 runs a **block** 代码块 only when its **condition** 条件 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

Common mistakes.

- A semicolon after the condition ends the `if`: in `if (x > 0); { count++; }` the block runs every time.
- Without braces only **one** statement belongs to the `if`. Below, `"Done"` prints whatever `t` is, despite the indentation:

```
if (t > 30)
    System.out.println("Hot");
    System.out.println("Done");
```


exercises.lol exercises.com.cn

Nested if Statements

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

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

Worked example. The order of the tests matters. This version never gives an `'A'`:

```
if (g >= 80) grade = 'B';
else if (g >= 90) grade = 'A';   // never reached for g >= 90
else grade = 'C';
```

For `g = 95` the first test is already true, so `'B'` is chosen and the rest is skipped. With ranges, test the **narrowest** case first (highest mark first), as in the correct version above. If the chain used separate `if` statements without `else`, every true test would run and the last one would win.

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** 防止 errors, e.g. `if (n != 0 && total / n > 5)`.

a	b	a && b	a || b	!a
true	true	true	true	false
true	false	false	true	false
false	true	false	true	true
false	false	false	false	true

short-circuit:

false && … stops: result is false

true || … stops: result is true

Truth table for &&, || and !, with short-circuit evaluation

Worked example. Evaluate `!(x > y) || x < 5 && y == 6` for `x = 4`, `y = 6`. The order is `!`, then `&&`, then `||`:

1. `x > y` is `4 > 6`, false, so `!(x > y)` is **true**.

exercises.lol exercises.com.cn

2. `x < 5 && y == 6` is `true && true`, **true**.
3. `true || true` is **true**. In fact, step 2 is never run: `||` already has `true` on its left and short-circuits.

Short-circuiting changes what runs. In `if (s != null && s.length() > 0)` the null check must come first; the other order throws a `NullPointerException` when `s` is `null`.

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** 等价 if they give the same result for every input – a **truth table** 真值表 proves it. Simplifying conditions this way is a common exam task.

> **Worked example.** Write `!(age >= 13 && age <= 19)` without the `!`.

1. De Morgan: `!(age >= 13) || !(age <= 19)` (the `&&` becomes `||`).
2. Negate each comparison: `!(age >= 13)` is `age < 13`; `!(age <= 19)` is `age > 19`.
3. Result: `age < 13 || age > 19`, every age outside the teens.

Check one value of each kind: `age = 15` gives `false` in both; `age = 25` gives `true` in both. A common error is to negate `>=` as `<=`: the opposite of `>=` is `<`.

`a`	`b`	`!(a && b)`	`!a || !b`
T	T	F	F
T	F	T	T
F	T	T	T
F	F	T	T

Comparing object references

For objects, `==` and `!=` compare **references** 引用, not contents: `a == b` is `true` only when `a` and `b` refer to the **same object**. Two variables that refer to the same object are **aliases** 别名.

```
String s1 = new String("cat");
String s2 = new String("cat");
String s3 = s1;
System.out.println(s1 == s2);       // false: two different objects
System.out.println(s1 == s3);       // true: s3 is an alias of s1
System.out.println(s1.equals(s2));  // true: same characters
```

A reference that does not refer to any object holds **`null`** 空值. Test for it with `==` or `!=` **before** calling a method, because calling a method on `null` throws a `NullPointerException`:

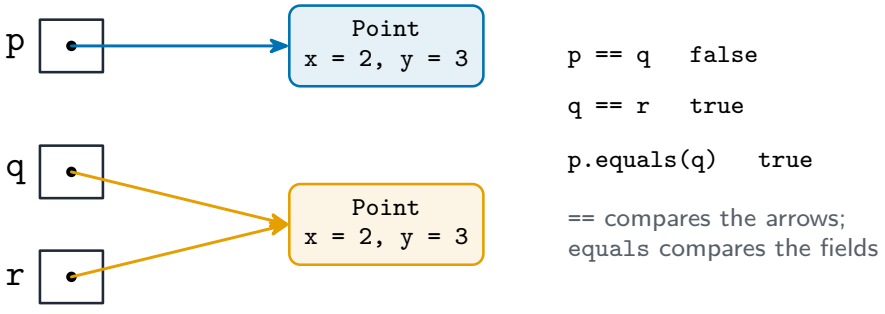
```
if (name != null && name.equals("Ana"))   // safe if name is null
```

Many classes define their own **`equals`** method, which decides when two objects count as equal, usually by comparing their **attributes** (instance variables). `String`'s `equals` compares the characters. A class that does not define one uses the `Object` version, which works like `==`. (You will not be asked to write an `equals` method.)

Worked example. The `Point` class has an `equals` method that compares `x` and `y`. Evaluate `p == q`, `q == r` and `p.equals(q)` after this code:

```
Point p = new Point(2, 3);
Point q = new Point(2, 3);
Point r = q;
```

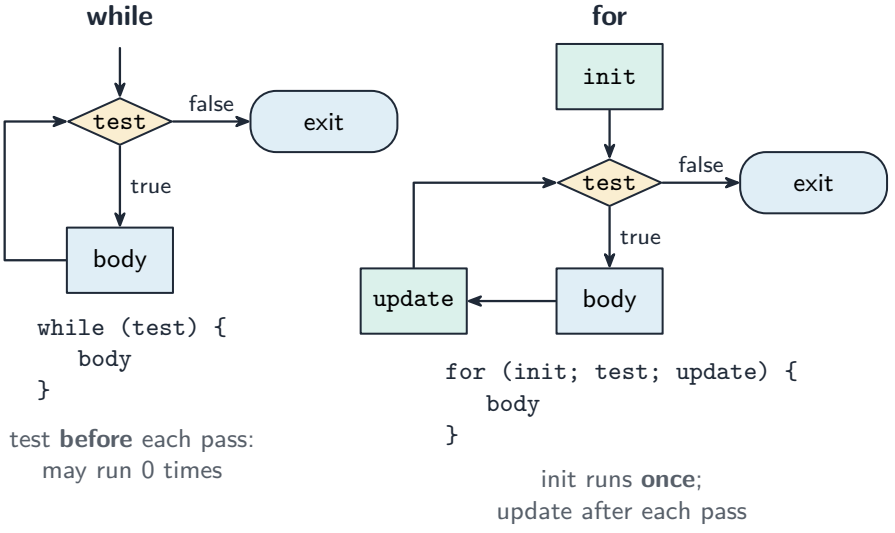
`p` and `q` were made by two separate `new` calls, so they are two objects: `p == q` is **`false`**. `r = q` copies the reference, so `r` and `q` are aliases: `q == r` is **`true`**. `equals` compares the attributes, which are both $(2, 3)$: `p.equals(q)` is **`true`**. If `q.setX(5)` is then called, `r.getX()` also returns 5, because `r` is the same object.



Two objects with equal fields, and one object with two names.

while Loops

A **while loop** 当型循环 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:



A while loop tests before each pass; a for loop is the same loop with an init that runs once and an update after each pass

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

Worked example. Trace a loop that adds the digits of `n = 472`:

```
int n = 472, sum = 0;
while (n > 0) {
    sum += n % 10;   // last digit
    n /= 10;         // drop it
}
```

Pass	`n % 10`	`sum`	`n` after
1	2	2	47
2	7	9	4
3	4	13	0

After pass 3 the test `n > 0` is false, so the loop stops with `sum = 13`. If `n` started at `0`, the body would run **zero** times: a `while` loop tests before the first pass.

for Loops

A **for loop** packs **initialization** 初始化, condition, and **update** 更新 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.

Worked example. How many times does this run, and what does it print?

```
for (int k = 10; k > 0; k -= 3)
    System.out.print(k + " ");
```

`k` takes the values 10, 7, 4, 1; the next value, -2, fails `k > 0`. So it runs **4** times and prints `10 7 4 1`. The equivalent `while` moves the three parts apart:

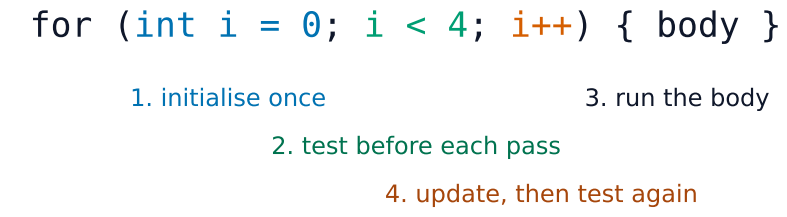
```
int k = 10;
while (k > 0) {
    System.out.print(k + " ");
    k -= 3;
}
```

A loop `for (int i = a; i < b; i++)` runs $b - a$ times; with `<=` it runs $b - a + 1$ times. A variable declared in the `for` header, such as `k`, cannot be used after the loop.



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

exercises.lol exercises.com.cn



The three parts of a for loop and the order they run

Building Complete Selection and Iteration Algorithms

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

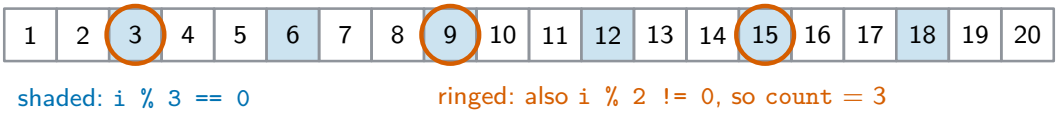
```
int max = arr[0];
for (int k = 1; k < arr.length; k++) {
    if (arr[k] > max) max = arr[k];
}
```

Two integer patterns the exam tests directly use `%` and `/`. For a nonnegative **integer** 整数 `n`, read its **digits** 数字位 one at a time by taking `n % 10` (the last digit) and then `n = n / 10` (drop it). For nonzero `d`, `n % d == 0` tests **divisibility** 整除性: `n` is evenly divisible by `d`. Combine them with a **counter** 计数器 to find the **frequency** 频率 with which some **criterion** 条件标准 is met.

Worked example. Count the numbers from 1 to 20 that are divisible by 3 but not by 2.

```
int count = 0;
for (int i = 1; i <= 20; i++) {
    if (i % 3 == 0 && i % 2 != 0) {
        count++;
    }
}
```

The multiples of 3 are 3, 6, 9, 12, 15, 18; the odd ones are 3, 9, 15, so `count` is **3**. The counter starts at `0` **before** the loop; resetting it inside the loop is a common error.



Both conditions must hold: shaded and odd.

Standard patterns like a **running total** 累加和, 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** 索引 to process each **character** 字符:

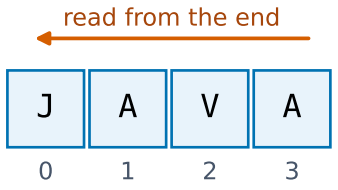
```
for (int i = 0; i < s.length(); i++) {
    char c = s.charAt(i);
    // count vowels, reverse, check for a substring, ...
}
```

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

Worked example. Count how many times `"ana"` occurs in `"banana"`, overlaps included.

```
String s = "banana", t = "ana";
int count = 0;
for (int i = 0; i <= s.length() - t.length(); i++) {
    if (s.substring(i, i + t.length()).equals(t)) {
        count++;
    }
}
```

`i` runs from 0 to $6 - 3 = 3$, checking `"ban"`, `"ana"`, `"nan"`, `"ana"`: `count` is **2**. With `i < s.length()` instead, `substring(4, 7)` is past the end and throws a `StringIndexOutOfBoundsException`.




```
String rev = "";
for (int i = s.length() - 1;
         i >= 0; i--) {
    rev += s.substring(i, i + 1);
}
```

```
i = 3:  rev = "A"
i = 1:  rev = "AVA"
```

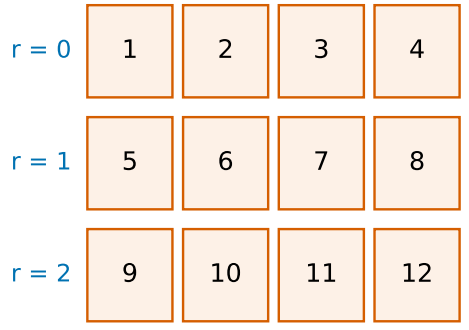
```
i = 2:  rev = "AV"
i = 0:  rev = "AVAJ"
```

Reversing a string by walking its indexes backwards

Nested Iteration

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

```
for (int r = 0; r < 3; r++) {
    for (int c = 0; c < 4; c++) {
        System.out.print("*");
    }
    System.out.println();
}
```



3 × 4 = 12 stars, printed in this order

The inner loop runs completely for each pass of the outer loop

Worked example. What does this print?

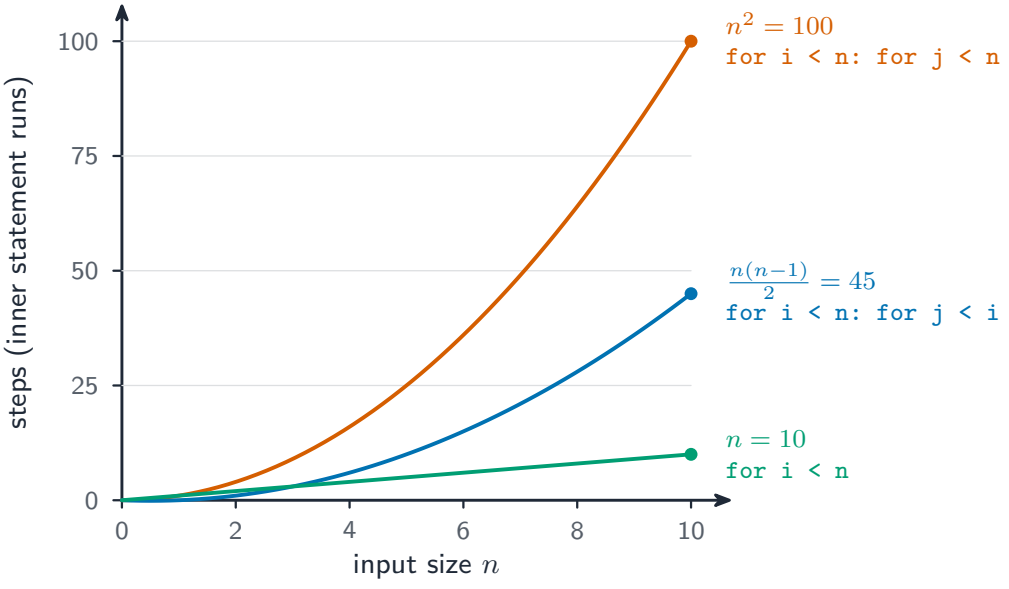
```
for (int r = 1; r <= 3; r++) {
    for (int c = 1; c <= r; c++) {
        System.out.print(c);
    }
    System.out.println();
}
```

r	c takes	Line printed
1	1	1
2	1, 2	12
3	1, 2, 3	123

The `println` is inside the outer loop but outside the inner one, so it ends each row.

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: with constant work per iteration, visiting each of n items once is **linear** 线性 (n steps); two full nested loops of n iterations each are **quadratic** 平方级 (n^2). This informal counting lets you compare two algorithms' **efficiency** 效率.



How the step count grows with n for one loop, a triangular nested loop and a full nested loop

Exam skill: for a nested loop, be able to state how many times the inner statement

exercises.lol exercises.com.cn

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** 边界条件 right: use `<` vs `<=` deliberately, and watch the first and last iteration of every loop (**off-by-one** 差一错误 is the classic **bug** 缺陷).
- 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** 循环变量.
- Apply **De Morgan's laws** when you simplify or negate a boolean condition.

Common mistakes (whole unit).

- Writing `if (x = 5)` instead of `if (x == 5)`. In Java this does not compile for an `int`, but the habit causes errors elsewhere.
- Putting a semicolon after the condition: `if (x > 0);` ends the `if` there, so the next block always runs.
- Looping with `i <= s.length()`, which reaches an index one past the end and throws a `StringIndexOutOfBoundsException`.
- Comparing strings with `==` instead of `.equals(...)`. `==` asks whether two references point to the **same object**; `equals` asks whether the contents match.
- Calling a method on a reference before checking it is not `null`. Put the `!= null` test first in an `&&`.
- Forgetting to update the loop variable in a `while` loop, making an infinite loop.

Practice.

1. Why is `if (s != null && s.length() > 0)` safe, but `if (s.length() > 0 && s != null)` not?
2. Use De Morgan's laws to rewrite `!(x < 3 || y >= 5)` without `!`.
3. How many times does `for (int i = 3; i <= 15; i += 4)` run?
4. What is `n` after this code runs? `int n = 1; while (n < 100) { n *= 3; }`

5. Write a code segment that counts the vowels (a, e, i, o, u) in a lowercase `String word`.
6. How many times does the inner body run? `for (int i = 0; i < 4; i++) { for (int j = 0; j < i; j++) { ... } }`

Answers.

1. `&&` short-circuits: if `s` is `null` the second test is never evaluated. In the second version `s.length()` runs first and throws a `NullPointerException`.
2. `x >= 3 && y < 5`.
3. Four times (`i` = 3, 7, 11, 15).
4. `243` (1, 3, 9, 27, 81, 243; the loop stops when `n` is no longer below 100).
5. One answer:

```
int count = 0;
for (int i = 0; i < word.length(); i++) {
    String c = word.substring(i, i + 1);
    if ("aeiou".indexOf(c) >= 0) {
        count++;
    }
}
```

6. $0 + 1 + 2 + 3 = 6$ times.

exercises.lol exercises.com.cn

Class Creation

AP Computer Science A

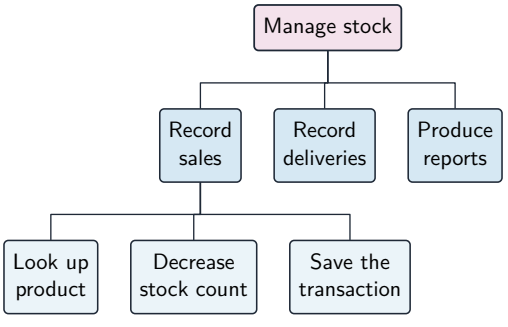
Abstraction and Program Design



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

In the 1980s the Therac-25 radiation therapy machine gave some patients massive overdoses, and some of them died. Part of the cause was its software: data and code interacted in ways that nobody had tested, and the earlier hardware safety locks had been removed. Careful class design is not only about tidy code. A class that controls its own data, and cannot be left in a wrong state, is much harder to break.

Abstraction 抽象 means hiding detail behind a simple **interface** 接口 – 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** 可读的, and **reusable** 可复用的. **Encapsulation** 封装 – keeping data **private** 私有的 and exposing it only through **methods** 方法 – protects an object's **state** 状态 from misuse and lets you change the inside without breaking users of the class. Thoughtful naming, single-purpose methods, and **testing** 测试 reduce **bugs** 缺陷.

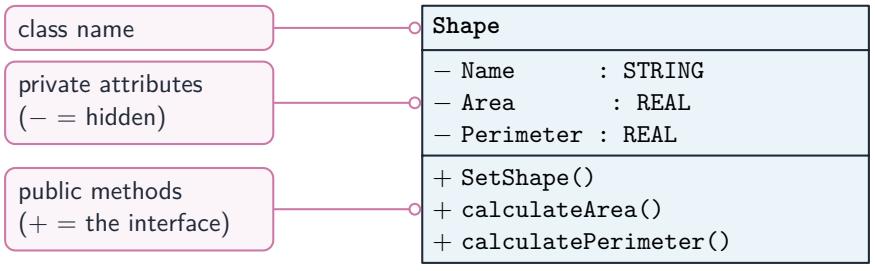
Design also carries responsibility beyond the code. **System reliability** 系统可靠性 - a program performing its tasks as expected, without failure - is something programmers should maximise through careful design and testing. Programs have real impacts on society, the economy, and culture that can be both **beneficial** 有益的 and **harmful** 有害的. And creating programs raises **legal** 法律 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** 许可证 and **give credit** 注明出处 rather than copy others' work as their own.

The Anatomy of a Class

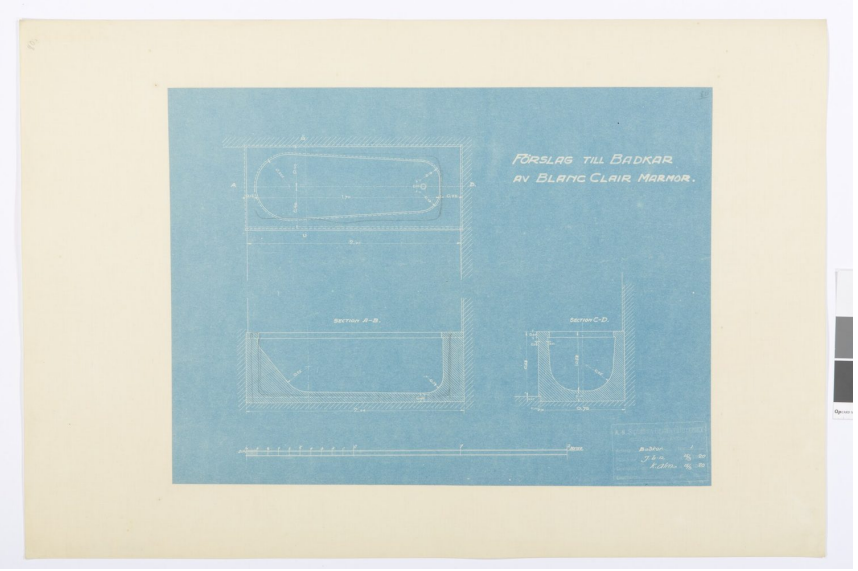
A class has three parts: **instance variables** 实例变量 (**fields** 字段 – the object's data), **constructors** (build objects), and **methods** (**behavior** 行为). Fields are usually `private`; methods are usually `public`:

```
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 class diagram: private attributes and public methods



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

Common mistake. Repeating the type inside the constructor declares a new **local** variable instead of setting the field:

```
public Student(String n, int s) {
    String name = n;   // local variable: the field name stays null
    score = s;         // correct: sets the field
}
```

The code compiles, but every `Student` built this way has `name` equal to `null`.

Constructors

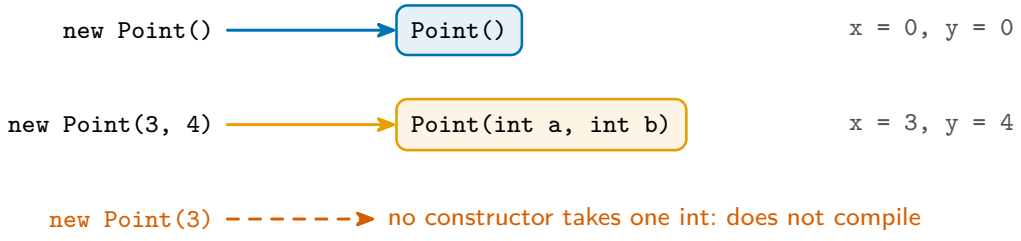
A **constructor** 构造函数 has the **same name as the class** and no **return type** 返回类型. It runs when you write `new`, and its job is to **initialize** 初始化 the fields. A class can have several constructors with different **parameter lists** 形参列表 (**overloading** 重载); a **no-argument** 无参 constructor accepts no arguments and can choose suitable **defaults** 默认值. Before constructor assignments, instance fields have Java's type-dependent initial values, such as `0` for `int` and `null` for references.

Worked example. A `Point` class with two constructors:

```
public class Point {
    private int x, y;

    public Point() {              // no-argument: the origin
        x = 0;
        y = 0;
    }
    public Point(int a, int b) { // two int parameters
        x = a;
        y = b;
    }
}
```

Call	Constructor used	Fields
`new Point()`	no-argument	`x = 0, y = 0`
`new Point(3, 4)`	`(int, int)`	`x = 3, y = 4`
`new Point(3)`	none matches	does not compile



Java picks the constructor whose parameter list matches the arguments.

A field that a constructor does not set keeps its default value: `0` for `int`, `0.0` for `double`, `false` for `boolean` and `null` for an object. Writing `public void Point()` makes a method, not a constructor: a constructor has **no** return type, not even `void`.

Methods: How to Write Them

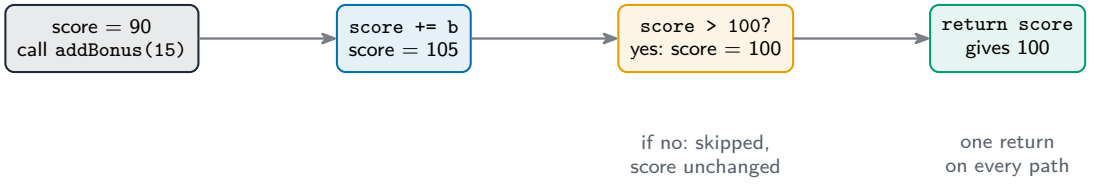
A method has a **signature** 方法签名, a return type, and a **body** 方法体. An **accessor (getter)** 访问器 returns information without changing the object; a **mutator (setter)** 修改器 changes a field. A method returning a value must have a `return` of the right type on every path; a `void` method returns nothing.

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

Worked example. Write a mutator `addBonus(int b)` that adds `b` to the score but never lets it go above 100, and returns the new score.

```
public int addBonus(int b) {
    score += b;
    if (score > 100) {
        score = 100;
    }
    return score;
}
```

1. The return type is `int`, because the method gives back the new score.
2. The parameter `b` is used; the field `score` is changed, so this is a mutator.
3. For a score of 90, `addBonus(15)` sets the score to 100 and returns `100`.

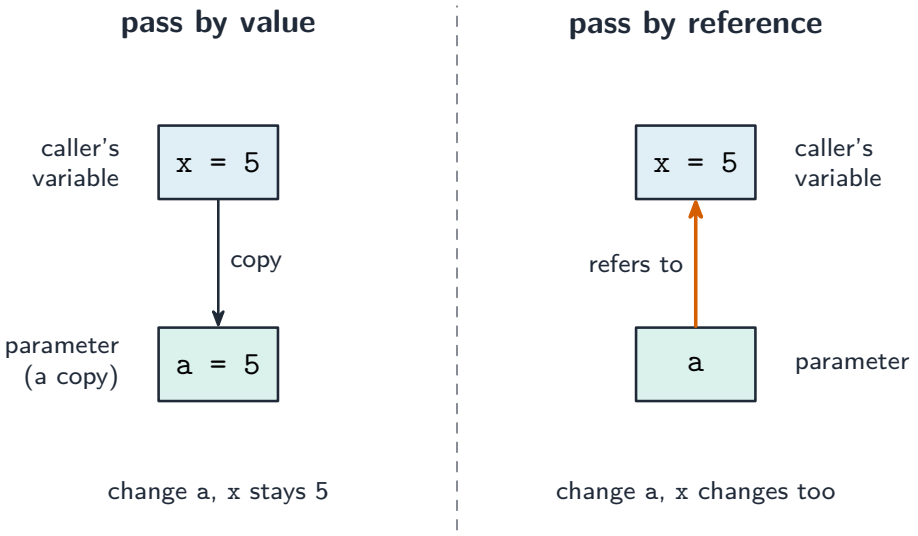


The cap runs only when needed; the single return covers both paths.

A common mistake is a `return` inside the `if` only: `if (score > 100) { return 100; }` with nothing after it does not compile, because the other path returns nothing. Another is printing the answer instead of returning it: a method asked to **return** a value must not just `System.out.println` it.

Passing and Returning References of an Object

When you pass an object to a method, Java copies the **reference** 引用, so the method acts on the **same** object – changes to its fields are visible to the **caller** 调用者. (**Primitives** 基本类型 are **copied by value** 按值复制, so changes to them are not.) A method can also **return** a reference to an object. Because a `String` is **immutable** 不可变, passing one is safe; passing a **mutable** 可变 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** 静态方法 belongs to the class and cannot use instance fields directly. Access them by class name: `Student.getCount()`.

Worked example. Give each new `Ticket` a number, counting up from 1:

```
public class Ticket {
    private static int count = 0;   // one copy, shared
    private int number;             // one per object

    public Ticket() {
        count++;
        number = count;
    }
    public int getNumber() { return number; }
    public static int getCount() { return count; }
}
```

After `Ticket a = new Ticket(); Ticket b = new Ticket(); Ticket c = new Ticket();`, `a.getNumber()` is 1, `c.getNumber()` is 3 and `Ticket.getCount()` is 3. If `count` were not `static`, each object would have its own `count`, every ticket would be number 1, and `getCount` could not be `static`.

Scope and Access

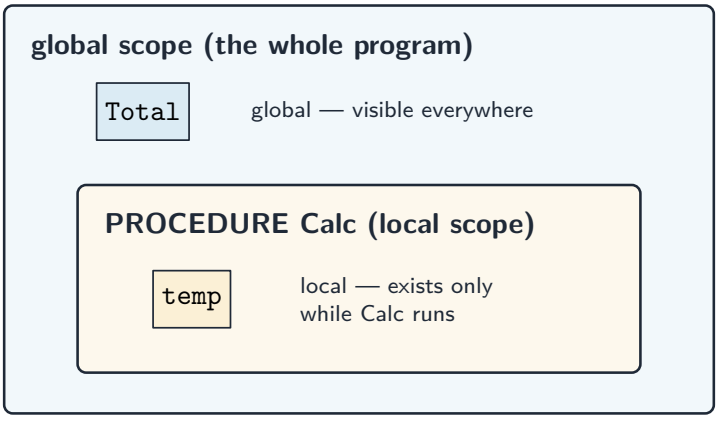
Scope 作用域 is where a name is visible. A **local variable** 局部变量 declared in a method exists only inside it; a **parameter** 形参 exists only in its method; an **instance variable** is visible throughout the object. **Access modifiers** 访问修饰符 control **visibility** 可见性 across classes: `private` (this class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

Worked example. Find the bug:

```
public void setScore(int score) {
    score = score;
}
```

The parameter `score` shadows the field, so both sides mean the parameter: the method copies the parameter onto itself and the field never changes. The fix is `this.score = score;` (see the next section) or a different parameter name such as `newScore`.

Name	Declared in	Visible in
instance variable	the class, outside methods	instance methods directly; static methods need an object reference
parameter	the method header	that method only
local variable	a method or block	from its line to the end of its block
loop variable in `for (int i …)`	the `for` header	that loop only



General scope illustration: a local variable is visible only inside its block. Java uses class fields rather than standalone global variables; field access also depends on access modifiers.

The this Keyword

`this` is a reference to the **current object** 当前对象. 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.

Writing a complete class

Free-response question 2 usually asks you to write a whole class, given a table of example calls. The method is always the same. Here is a new example in that style.

A `PunchCard` records stamps at a coffee shop. Each time a customer has collected the needed number of stamps, the stamps reset to 0 and the customer earns one free drink.

Statement	Result
`PunchCard c = new PunchCard(3);`	3 stamps are needed for each free drink
`c.redeem();`	returns `false`: no free drink yet
`c.stamp(); c.stamp();`	no return value
`c.status();`	returns `"2/3, free: 0"`
`c.stamp();`	the third stamp: stamps reset, one free drink earned
`c.status();`	returns `"0/3, free: 1"`
`c.redeem();`	returns `true`, and uses the free drink
`c.redeem();`	returns `false`

Step 1: find the state. Ask what the object must remember between calls. Here it needs three things: how many stamps are needed, how many it has, and how many free drinks are waiting. These become `private` instance variables. A value used only inside one method stays a local variable.

Step 2: write the headers from the table. The call `c.status()` is used for its value, so it returns a `String`; `c.stamp();` gives nothing back, so it is `void`.

Step 3: fill in the bodies, one row at a time.

```
public class PunchCard
{
    private int needed;
    private int stamps;
    private int freeDrinks;

    public PunchCard(int n)
    {
        needed = n;
        stamps = 0;
        freeDrinks = 0;
    }
```


exercises.lol exercises.com.cn

```
    public void stamp()
    {
        stamps++;
        if (stamps == needed)
        {
            stamps = 0;
            freeDrinks++;
        }
    }

    public boolean redeem()
    {
        if (freeDrinks > 0)
        {
            freeDrinks--;
            return true;
        }
        return false;
    }

    public String status()
    {
        return stamps + "/" + needed + ", free: " + freeDrinks;
    }
}
```

Step 4: trace the table. Follow each row with the three variables. After the three stamps they hold 3, 0 and 1, so `status()` gives `"0/3, free: 1"`, which matches.

Common mistakes.

- Using `System.out.println` in `status()`. The table says **returns**, so write `return`.
- Writing `int stamps = 0;` inside the constructor. This makes a new local variable, and the instance variable is not set by that line.
- Making instance variables `public`, or `static`. Each card must keep its own count.
- Keeping the method's parameter `n` and not storing it. It is gone when the constructor ends, so `stamp()` cannot use it.
- Writing `redeem()` so that it returns `true` without reducing `freeDrinks`. The method has a **side effect** that the table shows in its last row.

Practice

Answer each before reading the answers below.

1. Write a class `Book` with private instance variables `title` (a `String`) and `pages` (an `int`), a constructor that sets both, and an accessor `getPages`.

2. A class has `private static int count = 0;` and its constructor contains `count++;`. After three objects are created, what is `count`, and why is it `static`?
3. A constructor has a parameter `name` and contains `name = name;`. What goes wrong, and how do you fix it?
4. A method receives a `Book` reference `b` and runs `b = new Book("New", 10);`. Does the caller's object change? What if it instead called a mutator, `b.setPages(10);`?
5. Why are instance variables usually declared `private`?
6. Can a static method use an instance variable directly? Explain.

Answers

1. One answer:

```
public class Book {
    private String title;
    private int pages;

    public Book(String title, int pages) {
        this.title = title;
        this.pages = pages;
    }

    public int getPages() {
        return pages;
    }
}
```

2. 3. A static variable belongs to the class, so all objects share one counter.
3. The parameter hides the instance variable, so it assigns the parameter to itself and the instance variable is never set. Write `this.name = name;`.
4. No: reassigning the parameter only changes the local copy of the reference. A mutator does change the caller's object, because both references point to the same object.
5. Encapsulation: other classes can change the data only through methods, which can check that values are valid.
6. No. A static method belongs to the class, not to an object, so there is no `this` object whose instance variables it could use.

Exam tips

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

exercises.lol exercises.com.cn

- 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** 基本情况 first, then follow each **recursive call** 递归调用 to its return value (writing recursive code is outside the exam's scope).
- Recognise **inheritance** 继承 vocabulary – **superclass** 父类, **subclass** 子类, **method overriding** 方法重写, and that every Java class other than `Object` is a direct or indirect subclass of `Object` (designing and implementing inheritance is outside the exam's scope).

exercises.lol exercises.com.cn

Data Collections

AP Computer Science A

The Ethics of Collecting Data

Why it matters. In 2006 Netflix released 100 million film ratings from about 480,000 customers for a public prize competition, with names replaced by numbers. Two researchers at the University of Texas, Arvind Narayanan and Vitaly Shmatikov, showed in 2008 that many of those "anonymous" customers could be identified by matching a few of their ratings and dates with public reviews on IMDb. A planned second competition was cancelled after a privacy lawsuit. Data structures make it easy to collect and search huge amounts of data - this unit teaches both how to do it and why to think before you do.



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.

Protecting privacy. Personal data stored by a program can be leaked in a data breach, sold, or combined with other data to identify a person, as in the Netflix case. When you design a program:

- collect only the data the program really needs;
- ask for the user's consent, and say how the data will be used;
- store passwords and personal details securely, and delete data that is no longer needed.

exercises.lol exercises.com.cn

Data quality and algorithmic bias. Algorithmic bias is a **systemic**, repeated error in a program that gives unfair results to one group of users. It often comes from the data set, not from a coding mistake:

- **how the data was collected** – an online survey only reaches people who are online; data from one city may not describe a whole country. Check the collection method before you draw conclusions from the data.
- **incomplete data** – missing values (for example a blank age) can make averages wrong or crash a program that expects a number.
- **inaccurate data** – typing errors or old records give wrong results, however correct the code is.

> **Worked example.** A company trains a program to sort job applications, using the records of people it hired over the last ten years. Most of those people were men. Explain how this could cause algorithmic bias and suggest one way to reduce it. The program learns that features common in past hires predict success, so it scores applications from women lower – a repeated, unfair outcome for one group, even though no line of code mentions gender. To reduce it: check the data set for groups that are under-represented before using it, test the program's results separately for each group, and remove features that stand in for the group (such as membership of a women's club).

Why We Need Data Structures



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

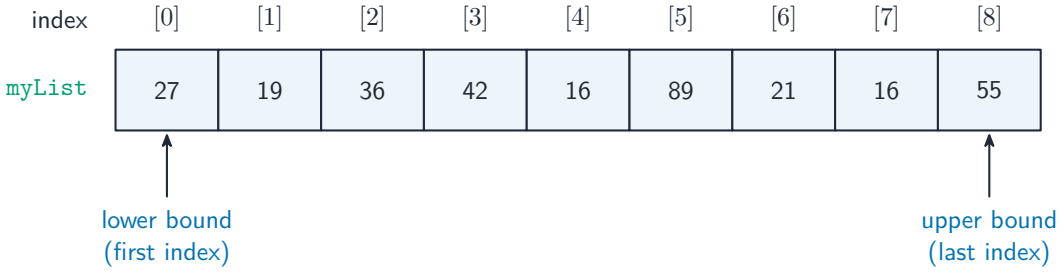
A single **variable** 变量 holds one value; real problems need to store **many** related values – a class roster, pixels, sensor readings. A **data structure** 数据结构 organizes

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** 索引 run from `0` to `length - 1`:

```
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)
```



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

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

Worked example. For `int[] vals = {3, 1, 4, 1, 5};`:

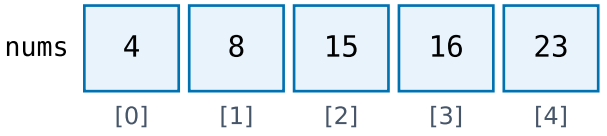
Expression	Value
`vals[2]`	4
`vals[vals.length - 1]`	5, the last element
`vals[1] + vals[3]`	2
`vals[vals.length]`	`vals[5]`: throws the exception

`new int[5]` fills the array with `0`; `new double[5]` with `0.0`, `new boolean[5]` with `false` and `new String[5]` with `null`. An array's size cannot change once it is made.

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):

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




```
for (int i = 0; i < nums.length; i++)
    sum += nums[i];
```

indexed: knows each position i;
can change elements

```
for (int n : nums)
    sum += n;
```

enhanced: a copy of each value;
cannot change the array

Traversing an array with an indexed for loop or an enhanced for loop

Common mistake. An enhanced `for` loop gives a **copy** of each value, so this changes nothing in the array:

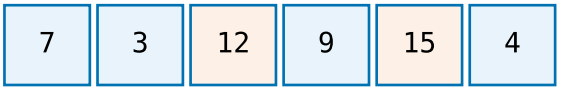
```
int[] a = {1, 2, 3};
for (int v : a) {
    v *= 2;        // doubles the copy v only
}
// a is still {1, 2, 3}
```

To change the elements, use the index loop: `a[i] *= 2;`. Use an index loop also when the position matters, such as comparing `a[i]` with `a[i + 1]`; then stop at `i < a.length - 1`.

Standard Array Algorithms

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

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




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

start max at arr[0], not 0, so an all-negative array still works

```
start: max = 7
i=1: 3 > 7? no
i=2: 12 > 7? yes, max = 12
i=3: 9 > 12? no
i=4: 15 > 12? yes, max = 15
i=5: 4 > 15? no
```

Finding the maximum of an array, traced pass by pass

Worked example. Reverse an array in place by swapping from both ends.

```
for (int i = 0; i < a.length / 2; i++) {
    int temp = a[i];
    a[i] = a[a.length - 1 - i];
    a[a.length - 1 - i] = temp;
}
```

i	swaps	a after
start		{3, 1, 4, 1, 5}
0	a[0] and a[4]	{5, 1, 4, 1, 3}
1	a[1] and a[3]	{5, 1, 4, 1, 3}

`a.length / 2` is 2, so the middle element stays put. Looping to `a.length` would swap every pair **twice** and put the array back as it was. A swap needs `temp`: `a[i] = a[j]; a[j] = a[i];` loses the old `a[i]`.

Worked example: is there a duplicate? Write a method that returns `true` if any value appears more than once in an `int` array. Compare every element with every element **after** it, using a nested loop; return `true` as soon as a pair matches, and `false` only after every pair has been checked.

```
public static boolean hasDuplicate(int[] a) {
    for (int i = 0; i < a.length - 1; i++) {
        for (int j = i + 1; j < a.length; j++) {
            if (a[i] == a[j]) {
                return true;
            }
        }
    }
    return false;
}
```

For `{3, 1, 4, 1, 5}` the pairs checked are (3,1), (3,4), (3,1), (3,5), then (1,4), (1,1): a match, so the method returns `true`. Starting `j` at `i + 1` matters: starting at `0` would compare `a[0]` with itself and always return `true`. Putting `return false;` inside the loop, in an `else`, is a common error: the method would stop after the first pair.

Reading Data from a Text File

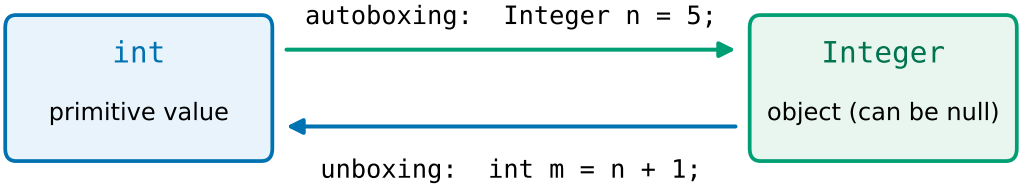
`File` and `IOException` live in `java.io`, so a program that reads a file needs `import java.io.*;` as well as `import java.util.Scanner;` for the `Scanner`. Opening a file can fail (it might not exist), and Java forces you to handle that – the simplest way is to add `throws IOException` to the **method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

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

Reading typed **tokens** 标记 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** 对象, not **primitives** 基本类型, 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** 自动拆箱 (back again) automatically, so you can write `list.add(5)` and `int x = list.get(0)`.



ArrayList<Integer> needs the wrapper; Java converts automatically

Autoboxing and unboxing between int and Integer

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)
```

Worked example. The list after each call above:

Call	names	Returns
`add("Amy")`	`[Amy]`	`true`
`add(0, "Bob")`	`[Bob, Amy]`	nothing
`get(0)`	`[Bob, Amy]`	`"Bob"`
`set(1, "Cara")`	`[Bob, Cara]`	`"Amy"`, the old value
`remove(0)`	`[Cara]`	`"Bob"`, the removed value
`size()`	`[Cara]`	`1`

Indexes must be from `0` to `size() - 1`, except that `add(index, x)` also accepts `size()`, adding at the end; any other index throws an `IndexOutOfBoundsException`.

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** 算法 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.

> **Worked example.** Remove every even number from `nums`, which holds `[2, 4, 5, 6]`.

A forward loop with `i++` after every pass goes wrong:

i	nums.get(i)	action	nums after
0	2	remove	[4, 5, 6]
1	5	keep	[4, 5, 6]
2	6	remove	[4, 5]

The `4` slid into index 0 after the first removal and was never checked. Looping backwards avoids this, because removing shifts only elements already checked:

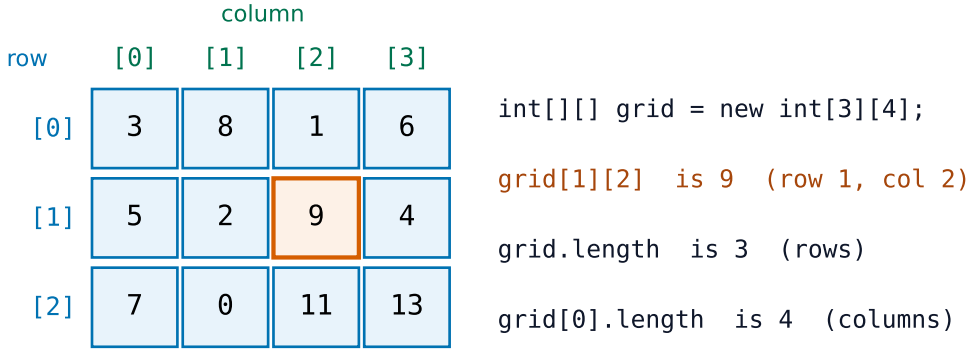
```
for (int i = nums.size() - 1; i >= 0; i--) {
    if (nums.get(i) % 2 == 0) {
        nums.remove(i);
    }
}
```

This leaves `[5]`.

Grids: Two-Dimensional Arrays

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

```
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
```



A Java 2D array: grid[row][col], with both indexes starting at 0

Walking Through a Grid

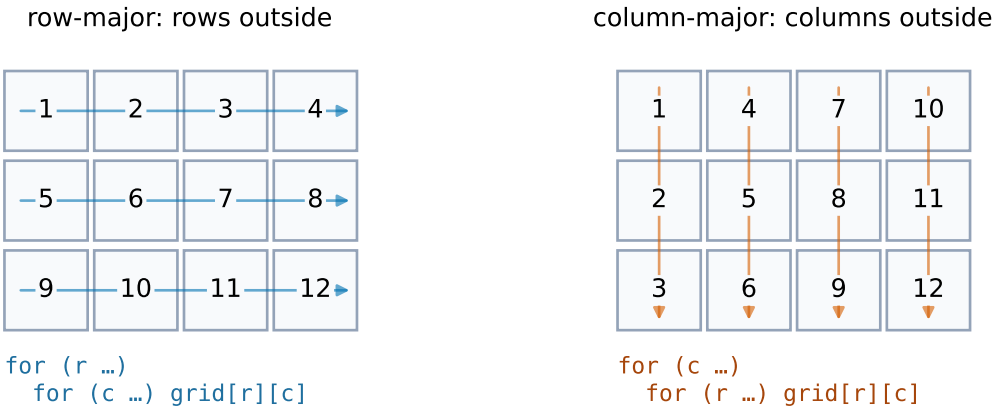
Visit every cell with **nested loops** 嵌套循环 – the outer over rows, the inner over columns (**row-major order** 行主序):

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

To go **column by column** (column-major order), swap the loops: the outer over `c < grid[0].length`, the inner over `r < grid.length`, still writing `grid[r][c]`. A common mistake is to use `grid.length` as the column limit: in a 3×4 grid that skips column 3, and in a 4×3 grid it throws an `ArrayIndexOutOfBoundsException`.

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.



Row-major and column-major traversal of a 2D array

Worked example. Find the sum of each column of

```
int[][] m = { {1, 2, 3},
              {4, 5, 6} };
```

```
for (int c = 0; c < m[0].length; c++) {
    int colSum = 0;                    // reset for each column
    for (int r = 0; r < m.length; r++) {
        colSum += m[r][c];
    }
    System.out.print(colSum + " ");
}
```

The outer loop picks a column, the inner loop goes down it: $1+4=5$, $2+5=7$, $3+6=9$, so the output is `5 7 9`. `colSum` is set to `0` inside the outer loop; declared before both loops, it would print the running totals `5 12 21`. The main diagonal of a square grid needs only one loop: `sum += g[i][i];`.

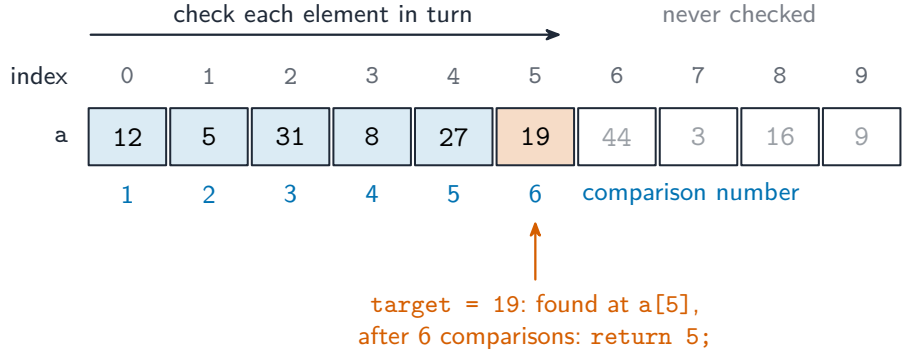
Finding a Value: Linear and Binary Search

exercises.lol exercises.com.cn

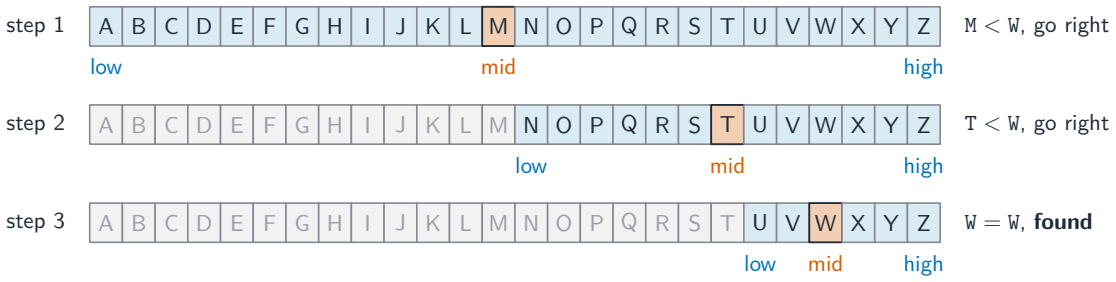


A card catalogue: because the cards are sorted, you can halve the search again and again, as binary search does

- **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** 已排序 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.



Linear search checks each element in turn: the target at index 5 takes 6 comparisons



Binary search halves the range at each step

```
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.

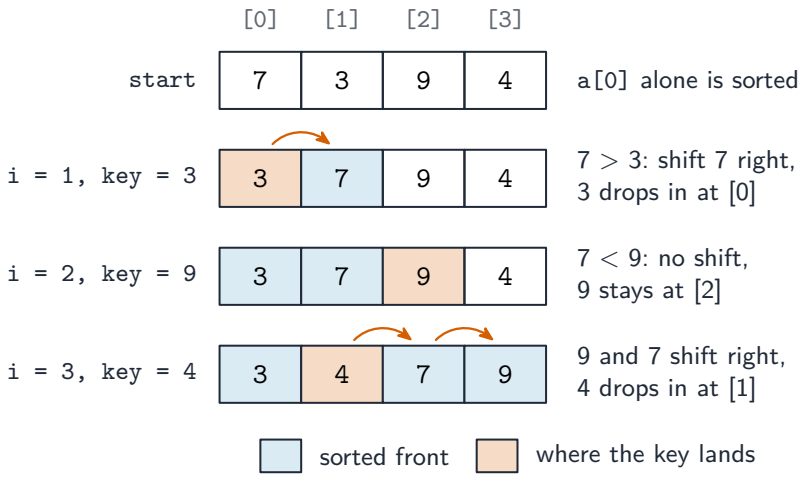
> **Worked example.** Search for `target = 40` in the sorted array `{3, 9, 14, 23, 31, 42, 55}` (indices 0–6). Start `lo=0, hi=6`:

- `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.



Insertion sort: each pass takes key = a[i], shifts larger elements of the sorted front right, and drops the key into the gap

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** 递归情形 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
}
```

Deep recursion without a reachable base case can cause a **stack overflow** 栈溢出.

Recursion and iteration are interchangeable. Any recursive solution can be rewritten with a **loop** (an **iterative** 迭代的 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:

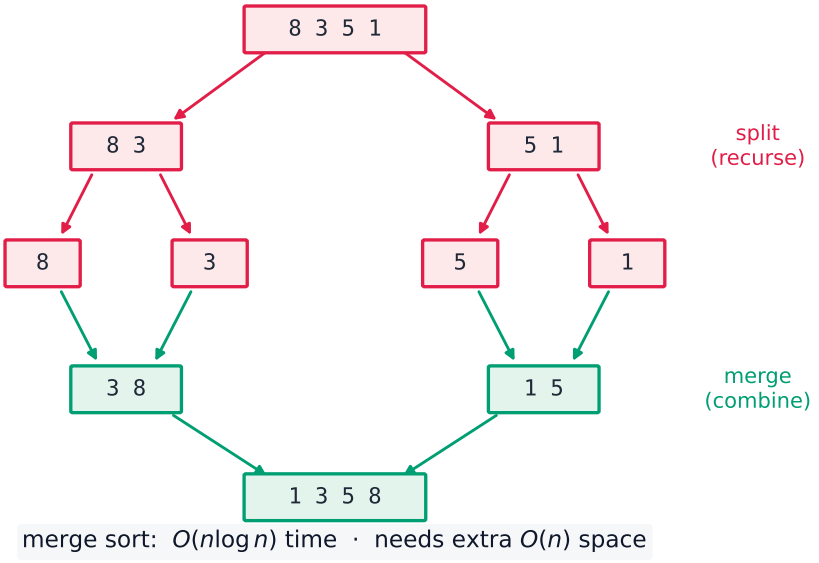
```
public static int factorial(int n) {
    int result = 1;
    for (int i = 2; i <= n; i++) result *= i;   // same answer, no
↪   self-call
    return result;
}
```

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 for multiple-choice questions. The free-response section requires Java code, including ArrayList and 2D-array algorithms.
- Protect **personally identifiable information (PII)** 个人身份信息 and explain privacy and security risks in context.

- Name real harms: data breaches, surveillance, and algorithmic **bias** from unrepresentative data.
- For a data-quality question, say how the data was **collected**, and whether it is **incomplete** or **inaccurate**, and link that to the wrong or unfair result.
- To check for duplicates, compare each element with every later one (`j = i + 1`); return `false` only after both loops finish.
- 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.

Common mistakes (whole unit).

- Mixing up the three "sizes": `arr.length` (array, no brackets), `str.length()` (String), `list.size()` (ArrayList).
- Removing from an `ArrayList` while looping forwards with an index, which skips the element after each removal. Loop backwards, or only increase the index when nothing is removed.
- Trying to change array elements through an enhanced `for` loop variable.
- Running binary search on unsorted data.
- Writing a recursive method with no base case, or one that the recursive calls never reach.

Practice.

1. For `int[] a = {5, 2, 8, 1};`, what is `a[a.length - 1]`, and what happens with `a[4]`?
2. `list` holds `[10, 20, 30]`. After `list.add(1, 15);` and then `list.remove(0);`, what does `list` hold?
3. What is the greatest number of elements binary search examines in a sorted array of 16 elements?
4. Write a method `public static int max(int[] arr)` that returns the largest value in a non-empty array.
5. After `int[][] g = new int[3][4];`, what are `g.length` and `g[0].length`?
6. `public static int mystery(int n) { if (n <= 1) return 1; return n + mystery(n - 2); }` What is `mystery(7)`?
7. A school app collects students' locations to track attendance. Give one benefit, one risk, and one way to reduce the risk.

Answers.

1. `1`; `a[4]` throws an `ArrayIndexOutOfBoundsException` (valid indices are 0 to 3).
2. `[15, 20, 30]`.
3. 5 ($16 \to 8 \to 4 \to 2 \to 1$: at most $\log_2 16 + 1$ checks).

exercises.lol exercises.com.cn

4. One answer:

```
public static int max(int[] arr) {
    int best = arr[0];
    for (int v : arr) {
        if (v > best) {
            best = v;
        }
    }
    return best;
}
```

5. `3` (rows) and `4` (columns).
6. $7 + 5 + 3 + 1 = 16$.
7. Example: benefit - accurate, automatic attendance; risk - a data breach could reveal where students are; reduction - collect location only on school grounds during school hours, and delete it after attendance is recorded.

exercises.lol exercises.com.cn