Java Data Structures
The Very Simple Book

Keywords, operators, loops, arrays, collections
and 100 worked programs - every one dry run by hand

Shammi Bharti

Table of Contents

PART 1 - THE GROUND FLOOR

CHAPTER 1

How To Use This Book

Read this page first, or the rest will be slower

This book has one goal: you should be able to look at any small Java program and say out loud, line by line, what the computer is doing and why. Not "it sorts the array". Line by line. Value by value. That skill is called **dry running**, and it is the single skill that separates people who can code from people who can only copy code.

Everything here is built on four repeating blocks. When you see them, you know what to expect.

Block	What it means	What you must do
Explanation	Plain English. No jargon without a definition.	Read once. Do not memorise.
Code	A complete, runnable Java program or fragment.	Type it. Do not copy-paste.
Dry Run	A table showing every variable at every step.	Cover the table. Predict. Then check.
Rule Box	A boxed rule you will use forever.	Write it in your own notebook.

The "five year old" promise, and its honest limit

You asked for explanations a five year old could follow. Here is the honest version of that promise, because a false promise wastes your time.

A five year old can understand the **idea** of every concept in this book. A box that holds a number. A line of people waiting. A family tree. Those pictures are real and they are used everywhere in this book.

A five year old cannot understand Java **syntax**, because syntax is arbitrary rules invented by humans, like spelling. So the plan is: the idea is always explained with a picture from daily life, and the syntax is explained as a separate, small, mechanical thing. Idea first. Spelling second. Never mixed.

> **THE CORE PROMISE**
> Every concept is introduced in three steps: (1) a picture from real life, (2) the smallest possible Java code, (3) a dry run table showing the values changing.
> If any of the three is missing, the explanation is incomplete.

How to actually study this

1) Read the explanation once, at normal speed. Do not stop to memorise.
2) Look at the code. Before reading the dry run, predict the output on paper.
3) Read the dry run. Find the exact step where your prediction went wrong. That step is your real lesson.
4) Close the book. Type the program from memory. It will not compile. Fix it. That fixing is the learning.
5) Change one thing in the program on purpose and predict what breaks.

Step 5 is the one people skip and it is the one that matters most. A program you have broken on purpose is a program you understand.

What this book covers, in order

Part	Content	Why it is here
1	Machine model, types, keywords, operators	You cannot dry run without knowing what each symbol does
2	Control flow, loops, methods, recursion	The shapes all logic is made from
3	Arrays and strings	The first real data structures
4	Objects, references, generics	Required before linked structures
5	Complexity, brute force, optimisation	How to judge a solution

Part	Content	Why it is here
6	Hand built data structures	Linked list, stack, queue, hash table, tree, heap, graph, trie
7	Java Collections Framework	The ready made versions
8	100 worked programs	Brute force, better version, dry run for each
9	Mistakes, gotchas, cheat sheets	The things that cost hours

CHAPTER 2

What Actually Happens When Java Runs

The picture first

Imagine you write a letter in English. Your friend in another country speaks only Japanese. You give your letter to a translator. The translator makes a version in a third, neutral language that every translator in the world understands. Then each country has a local reader who turns that neutral language into their own.

That is Java.

Real life	Java	File
Your letter in English	Your source code	`Main.java`
Neutral middle language	Bytecode	`Main.class`
Translator	The compiler, `javac`	-
Local reader in each country	The JVM (Java Virtual Machine)	`java`

This is why Java is called *write once, run anywhere*. The bytecode is the same on Windows, Linux and Mac. Only the JVM is different on each machine.

The smallest complete program

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello");
    }
}
```

Every single word above is doing a job. Here is each one.

Word	Job, in plain English
`public`	Anyone is allowed to use this. Nothing is hidden.
`class`	"I am declaring a new kind of thing." All Java code lives inside a class.
`Main`	The name of the thing. The file must be `Main.java`. Same name. Always.
`static`	"You do not need to build an object first. Just run it."
`void`	"This gives nothing back when it finishes."
`main`	The magic name. The JVM looks for exactly this name to start.
`String[] args`	A box of words typed on the command line. Usually empty. Still required.
`System.out.println`	System = the computer. out = the screen. println = print then move to a new line.
`;`	End of one instruction. Like a full stop in a sentence.
`{ }`	A bag holding instructions that belong together.

> **RULE**
> The JVM starts at `public static void main(String[] args)`. Nothing else. If you change even one word of that signature, the program compiles but refuses to run.

The two rooms of memory: Stack and Heap

This is the most important picture in the whole book. If you understand it, linked lists and trees become easy later. If you skip it, they stay confusing forever.

The computer gives your program two rooms to keep things in.

Room 1: the Stack

The Stack is a **tower of plates**. When a method starts, a new plate is put on top. All the local variables of that method live on that plate. When the method finishes, the whole plate is thrown away instantly, with everything on it.

- Fast. Very fast.
- Small.
- Automatically cleaned. You do nothing.
- Only the top plate is active at any moment.

Room 2: the Heap

The Heap is a **giant warehouse**. Objects live here. Objects are big, they live long, and many parts of the program may need the same object.

- Big.
- Slower to use than the stack.
- Cleaned by the Garbage Collector, which throws away anything nobody is pointing at.
- Everything made with the word `new` lives here.

How the two rooms connect

A variable on the stack does not hold an object. It holds the **address** of the object in the warehouse. Like a paper slip in your pocket with a locker number on it. The slip is small and in your pocket (stack). The stuff is big and in the locker (heap).

```
int a = 5;                // 5 is written directly on the stack plate
int[] nums = new int[3];  // 'nums' on the stack holds an ADDRESS
                          // the actual 3 boxes live in the heap
```

Stack (plate for main)	Heap (warehouse)
a = 5	-
nums = @1000	@1000 -> [0, 0, 0]

> **THE RULE THAT EXPLAINS 100 FUTURE BUGS**
> `int`, `char`, `boolean`, `double` and the other primitives store the **value itself**.
> Everything else - arrays, String, objects, ArrayList - stores an **address**.
> Copying a primitive copies the value. Copying an object variable copies only the address, so both names now point at the same thing in the warehouse.

Proving the rule with code

```
public class Proof {
    public static void main(String[] args) {
        int x = 10;
        int y = x;      // copies the VALUE 10
        y = 99;
        System.out.println(x);   // 10  -&gt; x untouched

        int[] p = {1, 2, 3};
        int[] q = p;    // copies the ADDRESS, not the boxes
        q[0] = 99;
        System.out.println(p[0]); // 99  -&gt; same boxes!
    }
}
```

Dry run

Step	Stack	Heap	Printed
1	x=10	-	-
2	x=10, y=10	-	-
3	x=10, y=99	-	-
4	-	-	10

Step	Stack	Heap	Printed
5	x,y, p=@2000	@2000 -> [1,2,3]	-
6	q=@2000	@2000 -> [1,2,3]	-
7	q=@2000	@2000 -> [99,2,3]	-
8	-	-	99

At step 6 nothing was copied except a locker number. That is why step 7 changed what `p` sees. This one idea will explain linked lists, tree nodes, and every "why did my list change by itself" bug you will ever have.

CHAPTER 3

Variables, Types And Literals

What a variable really is

A variable is a **labelled box**. Three parts: a type (what shape of thing fits in the box), a name (the label), a value (what is inside right now).

```
int age = 25;
//  ^    ^     ^
// type name  value
```

Java is **statically typed**: you must say the shape of the box before you use it, and the shape can never change. `int age` can never later hold the word "hello". This feels annoying for one week and then saves you for the rest of your life, because the compiler catches mistakes before the program ever runs.

The 8 primitive types - the complete list

These 8 are the only types that store a value directly. There are no others. Ever.

Type	Size	Range	Default	Example literal
`byte`	1 byte, 8 bits	-128 to 127	0	`byte b = 100;`
`short`	2 bytes	-32,768 to 32,767	0	`short s = 1000;`
`int`	4 bytes	about -2.1 billion to 2.1 billion	0	`int i = 42;`
`long`	8 bytes	about -9.2 quintillion to 9.2 quintillion	0L	`long l = 42L;`
`float`	4 bytes	about 7 decimal digits of precision	0.0f	`float f = 3.14f;`
`double`	8 bytes	about 15 decimal digits of precision	0.0	`double d = 3.14;`
`char`	2 bytes	0 to 65,535, one Unicode letter	'\u0000'	`char c = 'A';`
`boolean`	1 bit of meaning	`true` or `false` only	false	`boolean ok = true;`

THE TWO SUFFIXES PEOPLE FORGET

`long big = 10000000000;` does NOT compile. The number is read as an `int` first and it overflows. Write `10000000000L`.

`float f = 3.14;` does NOT compile. `3.14` is a `double`. Write `3.14f`.

Why an int has that exact range

Not magic. `int` has 32 bits. One bit is used for the sign (plus or minus). That leaves 31 bits for the number. 2 to the power 31 = 2,147,483,648. Counting from zero, the top is 2,147,483,647 and the bottom is -2,147,483,648.

```
int max = Integer.MAX_VALUE;   // 2147483647
System.out.println(max + 1);   // -2147483648  &lt;- it wrapped around
```

This is **overflow**. The number does not "get bigger". It rolls over like a car odometer hitting all nines. This is a real source of bugs in interview problems: `(low + high) / 2` can overflow when both are near the maximum. The safe version is `low + (high - low) / 2`.

Literals - the ways of writing a raw value

Kind	Example	Meaning
Decimal	`42`	Normal base 10
Binary	`0b1010`	10 in base 2
Octal	`052`	42 in base 8. A leading zero is dangerous, avoid it.
Hexadecimal	`0x2A`	42 in base 16

Kind	Example	Meaning
Underscored	`1_000_000`	1000000. Underscores are only for human eyes.
Char	`'A'`	Single quotes. Exactly one character.
String	`"A"`	Double quotes. Zero or more characters. Not a primitive.
Escape	`'\n'`	Newline. Also `\t` tab, `\\` backslash, `\'` quote.

Type casting - moving a value into a differently shaped box

Widening: small into big. Automatic. Safe.

```
int i = 100;
long l = i;      // fine, no loss
double d = l;    // fine
```

The order is: `byte -> short -> int -> long -> float -> double`, and `char -> int`.

Narrowing: big into small. Manual. Can lose data.

```
double d = 9.99;
int i = (int) d;      // 9   &lt;- the .99 is CHOPPED OFF, not rounded
int big = 300;
byte b = (byte) big;  // 44  &lt;- 300 does not fit in a byte, bits are cut
```

CASTING TRUTH

`(int)` does not round. It **truncates**: it deletes everything after the decimal point. `(int) 9.99` is 9. `(int) -9.99` is -9. If you want rounding, use `Math.round()`.

Variable scope - where a name is alive

Scope means: the region of code where a name exists. The rule is one sentence: **a variable lives from the line where it is declared until the closing brace of the block it was declared in.**

```
public class Scope {
    static int classLevel = 1;        // alive everywhere in the class

    public static void main(String[] args) {
        int methodLevel = 2;          // alive in main only

        if (true) {
            int blockLevel = 3;       // alive inside these braces only
            System.out.println(classLevel + methodLevel + blockLevel); // 6
        }
        // System.out.println(blockLevel); // ERROR: it is dead here

        for (int k = 0; k &lt; 3; k++) { }
        // System.out.println(k);         // ERROR: k died with the loop
    }
}
```

Scope name	Declared where	Lives how long	Default value
Local	Inside a method or block	Until the closing brace	NONE, must be set before use
Instance field	Inside class, no `static`	As long as the object lives	Yes, 0 / false / null
Static field	Inside class, with `static`	Whole program run	Yes, 0 / false / null
Parameter	In the method's brackets	For that one call	Given by the caller

THE MOST COMMON BEGINNER COMPILE ERROR

`variable might not have been initialized`.
Local variables get NO default. `int x; System.out.println(x);` fails to compile. Fields DO get a default. This asymmetry is deliberate: local mistakes are always bugs, so Java refuses to let them pass.

final - the box that can be filled only once

```
final int LIMIT = 10;
// LIMIT = 20;   // ERROR

final int[] list = {1, 2, 3};
list[0] = 99;      // ALLOWED! the ADDRESS is final, not the contents
// list = new int[5];  // ERROR, that would change the address
```

`final` freezes the **slip of paper in your pocket**, not the contents of the locker. Remember the stack and heap picture. This trips up almost everyone once.

CHAPTER 1

Every Java Keyword, One By One

What a keyword is

A keyword is a word that Java has taken for itself. You cannot use it as a name for your variable, method or class. There are 50 of them, plus 3 literals that behave like keywords (`true`, `false`, `null`), plus a handful of newer *contextual* keywords that are only special in certain places.

You do not need to memorise this chapter. You need to be able to come back to it. Read it once so you know what exists, then use it as a dictionary.

The full list at a glance

Group	Keywords
Primitive types	`byte` `short` `int` `long` `float` `double` `char` `boolean`
Declaring things	`class` `interface` `enum` `extends` `implements` `package` `import` `record`
Access and modifiers	`public` `private` `protected` `static` `final` `abstract` `synchronized` `native` `transient` `volatile` `strictfp` `sealed` `permits` `non-sealed`
Control flow	`if` `else` `switch` `case` `default` `for` `while` `do` `break` `continue` `return` `yield`
Errors	`try` `catch` `finally` `throw` `throws` `assert`
Objects	`new` `this` `super` `instanceof` `void`
Literals	`true` `false` `null`
Unused / reserved	`goto` `const`
Modules (Java 9+)	`module` `requires` `exports` `opens` `uses` `provides` `to` `with` `open`
Other	`var` (contextual, Java 10+)

Group 1 - The primitive types

Already covered fully in the previous chapter. Short version:

Keyword	One line
`byte`	Tiny whole number, -128 to 127. Used for raw file and network data.
`short`	Small whole number. Rarely used in practice.
`int`	The default whole number. Use this unless you have a reason not to.
`long`	Big whole number. Timestamps, IDs, large counters. Needs `L` suffix.
`float`	Small decimal. Rarely used. Not precise enough for money.
`double`	The default decimal. Still not precise enough for money.
`char`	One letter. Actually a number underneath: `'A'` is 65.
`boolean`	`true` or `false`. Nothing else. No 0 and 1 like in C.

> **MONEY WARNING**
> Never use `double` for money. `0.1 + 0.2` gives `0.30000000000000004`. Decimal fractions cannot be stored exactly in binary, the same way 1/3 cannot be written exactly in decimal. Use `BigDecimal`, or store paise/cents as `long`.

Group 2 - Declaring things

class

Declares a new kind of thing. A blueprint. A `class` is not an object, the same way a house plan is not a house.

```
class Dog {
    String name;
    void bark() { System.out.println(name + " says woof"); }
}
```

interface

A **contract**, a list of things a class promises to be able to do. No bodies (before Java 8). A class can implement many interfaces but extend only one class.

```
interface Flyable {
    void fly();          // no body: "whoever implements me MUST have this"
}
class Bird implements Flyable {
    public void fly() { System.out.println("flap"); }
}
```

Why it matters for data structures: `List`, `Set`, `Map`, `Queue` are all interfaces. `ArrayList` and `LinkedList` are classes that fulfil the `List` contract. That is why you can swap one for the other without changing the rest of your code.

enum

A fixed set of named values. Use when there are only a few valid options.

```
enum Day { MON, TUE, WED }
Day today = Day.MON;
if (today == Day.MON) System.out.println("Monday");
```

Better than `int day = 1;` because the compiler stops you from writing `Day.PIZZA`.

extends

"Is a kind of". Inheritance. The child gets everything the parent has.

```
class Animal { void eat() { System.out.println("eating"); } }
class Dog extends Animal { }     // Dog automatically has eat()
```

implements

"Promises to do". Used with interfaces. A class can implement many:

```
class X implements Comparable<X>, Runnable { }
```

package

The folder / namespace a class belongs in. Must be the very first line of the file.

```
package com.shammi.ds;
```

import

Brings a name from another package into view so you can write `ArrayList` instead of `java.util.ArrayList`.

```
import java.util.ArrayList;    // one class
import java.util.*;            // everything in the package
import static java.lang.Math.max;  // now you can write max(2,3) directly
```

`java.lang` is imported automatically. That is why `String`, `System` and `Math` need no import.

record (Java 16+)

A short way to declare a class that only carries data. Java writes the constructor, getters, `equals`, `hashCode` and `toString` for you.

```
record Point(int x, int y) { }
Point p = new Point(3, 4);
System.out.println(p.x());       // 3
System.out.println(p);           // Point[x=3, y=4]
```

Group 3 - Access modifiers and other modifiers

The four access levels

Modifier	Same class	Same package	Subclass elsewhere	Anywhere
`private`	Yes	No	No	No
(nothing)	Yes	Yes	No	No
`protected`	Yes	Yes	Yes	No
`public`	Yes	Yes	Yes	Yes

The "nothing" row is called **package-private** or **default access**. It is what you get when you write no modifier at all.

Rule of thumb used by professionals: **make everything** `private` **first, and only open it up when something outside genuinely needs it.** This is called encapsulation. In a linked list, the `Node` class and `head` pointer should be private, because if outside code can reach in and change `head`, your list can be broken from anywhere and you will never find the bug.

static

The single most misunderstood keyword. It means: **belongs to the class itself, not to any one object.** One copy, shared by everybody.

```
class Counter {
    static int total = 0;    // ONE copy shared by all Counters
    int mine = 0;            // each Counter gets its OWN

    void hit() { total++; mine++; }
}

Counter a = new Counter();
Counter b = new Counter();
a.hit(); a.hit(); b.hit();
System.out.println(Counter.total);  // 3  - shared
System.out.println(a.mine);         // 2  - a's own
System.out.println(b.mine);         // 1  - b's own
```

STATIC RULE
A `static` method cannot use `this`, and cannot directly touch non-static fields, because there is no object involved. That is why `main` is static: the JVM must run it before any object exists.

final

Three different meanings depending on where you put it.

Used on	Meaning
Variable	Can be assigned only once. `final int X = 5;`
Method	Cannot be overridden by a subclass.
Class	Cannot be extended at all. `String` is final. That is why nobody can make a broken String.

abstract

`abstract` on a method means "no body here, children must supply it". `abstract` on a class means "this is incomplete, you cannot make an object of it directly".

```
abstract class Shape {
    abstract double area();          // no body
    void describe() { System.out.println("Area is " + area()); }
}
class Square extends Shape {
    double side;
    Square(double s) { side = s; }
    double area() { return side * side; }
}
// new Shape();   // ERROR
new Square(3).describe();    // Area is 9.0
```

synchronized, volatile, transient, native, strictfp

These five are for special situations. You will not need them for data structures, but you should know they exist.

Keyword	One line
`synchronized`	Only one thread at a time may enter. Used to prevent two threads corrupting shared data.
`volatile`	Always read this variable from main memory, never from a CPU cache copy.
`transient`	Skip this field when saving the object to a file (serialization).
`native`	The body of this method is written in C or C++, not Java.
`strictfp`	Force identical floating point results on every platform. Obsolete since Java 17.

sealed, permits, non-sealed (Java 17+)

`sealed` means: only the classes I name in `permits` are allowed to extend me. It closes the hierarchy so the compiler knows every possible case.

```
sealed interface Shape permits Circle, Square { }
final class Circle implements Shape { }
final class Square implements Shape { }
```

Group 4 - Control flow keywords

Full treatment is in the control flow chapter. Here is the dictionary version.

Keyword	Meaning in one line
`if`	Run the block only when the condition is `true`.
`else`	Run this instead when the `if` was false.
`switch`	Jump straight to the matching `case`. Faster to read than a long if-chain.
`case`	One labelled option inside a switch.
`default`	The "none of the above" option inside a switch.
`for`	Repeat a fixed number of times. Counter built in.
`while`	Repeat while a condition stays true. Check first, then run.
`do`	Run once first, then check. Always runs at least once.
`break`	Leave the current loop or switch immediately.
`continue`	Skip the rest of this round and start the next round.
`return`	Leave the method now, optionally handing back a value.
`yield`	Give back a value from a `switch` expression block (Java 14+).

Group 5 - Error handling keywords

```
try {
    int x = 10 / 0;              // this will explode
} catch (ArithmeticException e) {
    System.out.println("Cannot divide by zero");
} finally {
    System.out.println("This runs no matter what");
}
```

Keyword	Meaning
`try`	"Watch this block for problems."
`catch`	"If that specific problem happened, do this instead of crashing."
`finally`	"Run this whether it worked or failed." Used for closing files.
`throw`	Create a problem on purpose, right now. `throw new IllegalStateException("empty");`
`throws`	A warning on a method: "calling me may cause this problem, be ready."
`assert`	Check an assumption during testing. Disabled by default at runtime.

For data structures you will use `throw` a lot: popping from an empty stack should throw, not return a wrong answer.

Group 6 - Object keywords

new

Builds an object in the heap and gives back its address. Every `new` allocates memory.

this

"The object I am currently inside." Used mainly to fix name clashes.

```
class Dog {
    String name;
    Dog(String name) {
        this.name = name;   // this.name = the field, name = the parameter
    }
}
```

super

"My parent." Used to call the parent's constructor or a parent's overridden method.

```
class Animal { Animal(String s) { System.out.println("Animal " + s); } }
class Dog extends Animal {
    Dog() { super("built"); System.out.println("Dog built"); }
}
```

instanceof

Asks "is this object of that type?" Returns `true` or `false`.

```
Object o = "hello";
if (o instanceof String s) {     // Java 16+ : also assigns s for you
    System.out.println(s.length());   // 5
}
```

void

"This method returns nothing."

Group 7 - The literals

Literal	Meaning
`true`	The boolean yes.
`false`	The boolean no.
`null`	"This reference points at nothing." An empty pocket slip. Touching a member on it causes `NullPointerException`.

> **NULLPOINTEREXCEPTION IN ONE SENTENCE**
> `null` means the slip in your pocket has no locker number written on it. When you try to open that locker, the program crashes. Every `NullPointerException` you ever see is this and only this.

Group 8 - The two dead keywords

`goto` and `const` are reserved but do nothing. Java's designers kept them reserved so that no one uses them as variable names and so that helpful error messages can be shown. Java deliberately has no `goto` because it makes code impossible to follow.

var - the contextual keyword (Java 10+)

`var` is not really a keyword, it is a hint to the compiler: "work the type out yourself from the right hand side."

```
var name = "Shammi";              // compiler decides: String
var list = new ArrayList&lt;String&gt;(); // compiler decides: ArrayList&lt;String&gt;
// var x;         // ERROR: nothing to infer from
// var y = null;  // ERROR: null has no type
```

The variable is still strongly typed. `var` only saves typing, it does not make Java dynamic like Python.

CHAPTER 1

Every Operator, With Dry Runs

What an operator is

An operator is a symbol that takes one or two values and produces a new value. That is the whole idea. + takes 2 and 3 and produces 5.

Vocabulary you will need, defined once:

Word	Meaning
Operand	The value an operator works on. In `2 + 3`, both 2 and 3 are operands.
Unary	Takes one operand. `-x`, `!flag`, `x++`
Binary	Takes two operands. `a + b`, `a > b`
Ternary	Takes three operands. Java has exactly one: `cond ? a : b`
Precedence	Which operator goes first when several appear together.
Associativity	Direction of evaluation when precedence ties. Usually left to right.

1. Arithmetic operators

Operator	Name	Example	Result
+	Add	`7 + 2`	9
-	Subtract	`7 - 2`	5
*	Multiply	`7 * 2`	14
/	Divide	`7 / 2`	**3** not 3.5
%	Modulus, remainder	`7 % 2`	1

INTEGER DIVISION IS THE NUMBER ONE BEGINNER TRAP
When both sides are whole numbers, `/` throws away the fraction. `7 / 2` is 3. `1 / 2` is 0.
To get 3.5 you must make at least one side a decimal: `7 / 2.0` or `(double) 7 / 2`.

Understanding % properly

`%` gives what is left over after dividing. Picture 7 sweets shared between 2 children: each gets 3, and 1 sweet is left in your hand. That 1 is `7 % 2`.

`%` is used constantly in data structures:

Use	Code	Why it works
Even or odd	`n % 2 == 0`	Even numbers leave nothing over when split into 2
Last digit	`n % 10`	1234 % 10 = 4
Wrap around a circle	`(i + 1) % size`	At the last index it jumps back to 0
Hash bucket	`hash % capacity`	Squashes any big number into the table size

Dry run: last digit and remove last digit

```
int n = 1234;
while (n &gt; 0) {
    int digit = n % 10;   // grab last digit
    System.out.print(digit + " ");
    n = n / 10;           // chop last digit off
}
// prints: 4 3 2 1
```

Round	n at start	n % 10	printed	n / 10 (new n)
1	1234	4	4	123
2	123	3	3	12
3	12	2	2	1
4	1	1	1	0
5	0	loop condition 0 > 0 is false	-	stop

This pair - `% 10` to read a digit and `/ 10` to remove it - is the engine behind reversing numbers, palindromes, digit sums, Armstrong numbers, and about fifteen of the programs later in this book. Learn it now and those become free.

Negative modulus

In Java, `%` keeps the sign of the **left** operand.

```
System.out.println(-7 % 3);   // -1   (not 2)
System.out.println(7 % -3);   //  1
```

To get a always-positive remainder: `((a % n) + n) % n`.

2. Unary operators

Operator	Name	Example
`+`	Unary plus. Does nothing.	+5
`-`	Negation	`-x`
`++`	Increment by 1	x++ or ++x
`--`	Decrement by 1	`x--` or `--x`
`!`	Logical NOT	`!found`
`~`	Bitwise NOT	~5 is -6
`(type)`	Cast	`(int) 9.9`

The famous ++ confusion, solved forever

There is only one rule. Read the operator position:

- `x++` is **post**-increment. "Give the old value, then add one."
- `++x` is **pre**-increment. "Add one first, then give the new value."

Both leave `x` one bigger. The difference is only what the **expression** hands back.

```
int a = 5;
int b = a++;      // b gets 5 (old), a becomes 6
System.out.println(a + " " + b);   // 6 5

int c = 5;
int d = ++c;      // c becomes 6 first, d gets 6
System.out.println(c + " " + d);   // 6 6
```

Statement	Value of x before	Value handed to expression	Value of x after
`b = x++`	5	5	6
`b = ++x`	5	6	6

PRACTICAL ADVICE

On its own line, `i++` and `++i` are identical. Use `i++` because everyone reads it faster.
Inside a bigger expression, they differ. Best practice: do not put ++ inside a bigger expression at all. `arr[i++] = arr[j++] + k--;` is legal and unreadable. Split it into lines.

Trap: the self assignment that does nothing

```
int i = 5;
i = i++;
System.out.println(i);    // 5, NOT 6
```

Step by step: Java first takes the old value 5 to be assigned later. Then `i` is bumped to 6. Then the saved 5 is written back over it. The increment is wiped out. Never write this.

3. Relational (comparison) operators

They always produce a `boolean`.

Operator	Meaning	a=5, b=3 **result**
==	Equal to	a == b is false
!=	Not equal to	a != b is true
>	Greater than	a > b is true
<	Less than	a < b is false
>=	Greater than or equal	a >= 5 is true
<=	Less than or equal	b <= 3 is true

== WITH OBJECTS IS THE MOST EXPENSIVE MISTAKE IN JAVA

For primitives, == compares values. Correct.
For objects, == compares **addresses** - "are these the exact same object in the warehouse?"
To compare contents you must use `.equals()`.

```
String a = new String("hi");
String b = new String("hi");
System.out.println(a == b);        // false  &lt;- different lockers
System.out.println(a.equals(b));   // true   &lt;- same contents

String c = "hi";
String d = "hi";
System.out.println(c == d);        // true  &lt;- literals share one locker (String pool)
```

That last case is why the bug hides: == sometimes appears to work with strings, so people believe it does. It only works because Java reuses identical literals from a shared pool. Read from a file or a Scanner and it instantly breaks. **Always use `.equals()` for objects.**

4. Logical operators

Operator	Name	Truth
&&	AND	true only when both sides are true
||	OR	true when at least one side is true
!	NOT	flips true to false
&	Non-short-circuit AND	same result, but always evaluates both sides
|	Non-short-circuit OR	same result, always evaluates both sides
^	XOR	true when the two sides DIFFER

Truth tables

A	B	A && B	A || B	A ^ B
true	true	true	true	false
true	false	false	true	true
false	true	false	true	true
false	false	false	false	false

Short circuiting - and why it saves you from crashes

&& stops the moment it finds a false, because the answer cannot change. || stops the moment it finds a true.

```
String s = null;
if (s != null &amp;amp;&amp;amp; s.length() &amp;gt; 0) {   // SAFE
    System.out.println("has text");
}
```

`s != null` is false, so Java never runs `s.length()` and never crashes. Swap the order and it crashes immediately. Swap `&&` to `&` and it also crashes, because `&` insists on evaluating both sides.

> **THE GUARD PATTERN**
> Always put the cheap safety check on the left of `&&`.
> `if (i < arr.length && arr[i] == target)` is safe.
> `if (arr[i] == target && i < arr.length)` crashes with ArrayIndexOutOfBounds.

5. Bitwise operators

These work on the individual 1s and 0s of a number. You will not need them daily, but they show up in hashing, in flags, and in clever interview tricks.

Operator	Name	a=5 (0101), b=3 (0011)	Result
`&`	AND, 1 only if both are 1	`5 & 3`	1 (0001)
`|`	OR, 1 if either is 1	`5 | 3`	7 (0111)
`^`	XOR, 1 only if they differ	`5 ^ 3`	6 (0110)
`~`	NOT, flip every bit	`~5`	-6
`<<`	Left shift	`5 << 1`	10
`>>`	Right shift, keep sign	`5 >> 1`	2
`>>>`	Right shift, fill with zeros	`-5 >>> 28`	15

Bit by bit dry run of 5 & 3

Bit position	8	4	2	1
5	0	1	0	1
3	0	0	1	1
AND (both 1?)	0	0	0	1

Result is 1.

Shifts are multiply and divide by 2

Expression	Binary before	Binary after	Value
`5 << 1`	0101	1010	10 = 5 x 2
`5 << 3`	0101	101000	40 = 5 x 8
`20 >> 2`	10100	101	5 = 20 / 4

Shifting left by `n` multiplies by 2 to the power n. Shifting right divides. `HashMap` uses this internally: `capacity << 1` doubles the table.

Three XOR tricks worth memorising

```
// 1. Any number XOR itself is 0
System.out.println(7 ^ 7);      // 0

// 2. Any number XOR 0 is itself
System.out.println(7 ^ 0);      // 7

// 3. Therefore: in an array where every number appears twice
//    except one, XOR everything and the loner survives.
int[] arr = {4, 1, 2, 1, 2};
int x = 0;
for (int v : arr) x ^= v;
System.out.println(x);          // 4
```

Step	v	x before	x after (x ^ v)
1	4	0	4
2	1	4	5
3	2	5	7
4	1	7	6
5	2	6	4

The pairs cancel themselves out no matter what order they appear in. This turns an O(n) space problem into O(1) space.

Checking and setting a single bit

Goal	Code
Is bit `i` set?	`(n >> i) & 1) == 1`
Set bit `i` to 1	`n | (1 << i)`
Clear bit `i` to 0	`n & ~(1 << i)`
Flip bit `i`	`n ^ (1 << i)`
Is n a power of 2?	`n > 0 && (n & (n - 1)) == 0`
Count set bits	`Integer.bitCount(n)`

6. Assignment operators

Operator	Long form	Meaning
`=`	-	Put the right value into the left box
`+=`	`a = a + b`	Add and store
`-=`	`a = a - b`	Subtract and store
`*=`	`a = a * b`	Multiply and store
`/=`	`a = a / b`	Divide and store
`%=`	`a = a % b`	Remainder and store
`&= |= ^=`	`a = a & b` etc	Bitwise and store
`<<= >>= >>>=`	`a = a << b` etc	Shift and store

HIDDEN CAST INSIDE COMPOUND ASSIGNMENT

`byte b = 10; b += 300;` compiles fine. `b = b + 300;` does NOT.
The compound form silently inserts a cast: it is really `b = (byte)(b + 300)`. Convenient, and occasionally the source of a silent wrong answer.

7. The ternary operator

The only operator in Java with three parts. It is a compact `if-else` that produces a value.

```
int a = 5, b = 9;
int max = (a &gt; b) ? a : b;    // read as: if a&gt;b then a else b
System.out.println(max);      // 9
```

Structure: `condition ? valueIfTrue : valueIfFalse`

Use it when you are choosing a **value**. Do not use it when you are choosing an **action**, and never nest it more than one level, because nested ternaries are unreadable.

Good	Bad
`String s = n == 1 ? "item" : "items";`	`x>0?a():x<0?b():c();`
`int abs = n < 0 ? -n : n;`	Anything three levels deep

8. instanceof

Already met. Returns `true` if the object is of that type or a subtype.

```
Object o = new ArrayList&lt;String&gt;();
System.out.println(o instanceof List);        // true
System.out.println(o instanceof String);      // false
System.out.println(null instanceof String);   // false, never crashes
```

9. Precedence - who goes first

When several operators appear together, Java uses this order. Highest at the top.

Level	Operators	Direction
1	`() [] . x++ x--`	Left to right
2	`++x --x +x -x ! ~ (cast) new`	Right to left
3	`* / %`	Left to right
4	`+ -`	Left to right
5	`<< >> >>>`	Left to right
6	`< <= > >= instanceof`	Left to right
7	`== !=`	Left to right
8	`&`	Left to right
9	`^`	Left to right
10	`|`	Left to right
11	`&&`	Left to right
12	`||`	Left to right
13	`? :`	Right to left
14	`= += -=` and friends	Right to left

Worked example

```
int r = 2 + 3 * 4 &gt; 10 &amp;&amp; 5 % 2 == 1 ? 100 : 200;
```

Step	Applying	Expression becomes
1	`*` before `+`	`2 + 12 > 10 && 5 % 2 == 1 ? 100 : 200`
2	`%` at level 3	`2 + 12 > 10 && 1 == 1 ? 100 : 200`
3	`+` at level 4	`14 > 10 && 1 == 1 ? 100 : 200`
4	`>` at level 6	`true && 1 == 1 ? 100 : 200`
5	`==` at level 7	`true && true ? 100 : 200`
6	`&&` at level 11	`true ? 100 : 200`
7	`? :`	`100`

PROFESSIONAL ADVICE

Nobody memorises this table. Professionals add brackets. `((2 + (3*4)) > 10)` costs two seconds to type and saves an hour of debugging. Write for the reader, not for the compiler.

10. Operators that Java deliberately does not have

Missing	Why
Pointer arithmetic `*p`, `&x`	Java hides memory addresses on purpose so you cannot corrupt memory.
Operator overloading	You cannot teach `+` to add two of your own objects. Only `String` gets special `+` treatment, built into the language.
`goto`	Reserved but unusable. It makes code unfollowable.

Missing	Why
`sizeof`	Sizes are fixed by the specification, so it is not needed.

PART 2 - THE SHAPES OF LOGIC

CHAPTER 1

Decisions: if, else, switch

The picture

Your program is a person walking down a road. A decision is a fork in the road. A condition is the signboard. Control flow is nothing more than: which path, and how many times.

There are only **three shapes** in all of programming:

1) Sequence - do this, then that.
2) Selection - do this OR that.
3) Repetition - do this again and again.

Every program ever written, including the operating system you are reading this on, is built from those three. Nothing else exists.

if

```
int marks = 75;
if (marks &gt;= 40) {
    System.out.println("Pass");
}
```

The thing in the brackets must be a `boolean`. In C you can write `if (5)`. In Java that is a compile error, and this saves you from the classic `if (x = 5)` bug where a single = silently assigns instead of comparing.

if - else

```
if (marks &gt;= 40) {
    System.out.println("Pass");
} else {
    System.out.println("Fail");
}
```

Exactly one of the two blocks runs. Never both. Never neither.

if - else if - else ladder

```
int m = 75;
if (m &gt;= 90)      System.out.println("A");
else if (m &gt;= 75) System.out.println("B");
else if (m &gt;= 60) System.out.println("C");
else              System.out.println("D");
```

LADDER RULE

Java checks top to bottom and stops at the FIRST true condition. So the order matters enormously.
If you write `if (m >= 60)` first, then 95 marks prints "C", because 95 is also >= 60 and Java stops there.
Always order a ladder from the most specific to the most general.

Dry run of the ladder with m = 75

Step	Condition tested	Result	Action
1	`75 >= 90`	false	move on
2	`75 >= 75`	true	print "B", **skip everything else**
3	-	-	ladder finished

Nested if

An `if` inside an `if`. Legal, but each level of nesting makes code harder to read.

```
if (loggedIn) {
    if (isAdmin) {
        System.out.println("Admin panel");
    }
}
```

Better, flatter version:

```
if (loggedIn &amp;amp;&amp;amp; isAdmin) {
    System.out.println("Admin panel");
}
```

The guard clause - a technique used by every professional

Instead of wrapping the good case in ever-deeper `if`s, kick out the bad cases early and keep the main logic flat.

```
// BAD: arrow shaped code
void process(String s) {
    if (s != null) {
        if (!s.isEmpty()) {
            if (s.length() &lt; 100) {
                System.out.println(s.toUpperCase());
            }
        }
    }
}

// GOOD: guard clauses
void process(String s) {
    if (s == null) return;
    if (s.isEmpty()) return;
    if (s.length() &gt;= 100) return;
    System.out.println(s.toUpperCase());   // the real work, at zero indent
}
```

The dangling else problem

```
if (a &gt; 0)
    if (b &gt; 0)
        System.out.println("both positive");
else
    System.out.println("???");
```

The indentation lies. `else` always binds to the **nearest unmatched if**, so that `else` belongs to `if (b > 0)`, not `if (a > 0)`. With `a = -1` nothing at all is printed.

> **RULE WITH NO EXCEPTIONS**
> Always use braces `{ }`, even for one line. This bug class disappears completely.

switch - the classic form

Use when you are comparing one variable against several fixed values.

```
int day = 3;
switch (day) {
    case 1: System.out.println("Mon"); break;
    case 2: System.out.println("Tue"); break;
    case 3: System.out.println("Wed"); break;
    default: System.out.println("Unknown");
}
```

Fall through - the behaviour everyone forgets

Without `break`, execution **falls into the next case and keeps going**. It does not stop at the matching case.

```
int x = 2;
switch (x) {
    case 1: System.out.println("one");
    case 2: System.out.println("two");
    case 3: System.out.println("three");
    default: System.out.println("other");
}
// prints: two, three, other
```

Step	Where	Break present?	Result
1	jump to `case 2`	-	print "two"
2	fall to `case 3`	no	print "three"
3	fall to `default`	no	print "other"
4	end of switch	-	stop

Sometimes fall through is exactly what you want:

```
switch (day) {
    case 6:
    case 7:
        System.out.println("Weekend");
        break;
    default:
        System.out.println("Weekday");
}
```

What can a switch check?

`byte`, `short`, `char`, `int`, their wrapper classes, `String` (since Java 7), and `enum`. It cannot check `long`, `float`, `double` or `boolean`.

switch - the modern arrow form (Java 14+)

```
int day = 3;
String name = switch (day) {
    case 1, 2, 3, 4, 5 -&gt; "Weekday";
    case 6, 7 -&gt; "Weekend";
    default -&gt; "Invalid";
};
System.out.println(name);   // Weekday
```

Old switch	New switch
Statement only	Can be an expression that produces a value
Needs `break`	No fall through, no `break` needed
One value per case	Comma separated list allowed
`default` optional	`default` required when used as an expression

When a case needs several lines and must produce a value, use a block with `yield`:

```
int size = switch (n) {
    case 1 -&gt; 10;
    default -&gt; {
        int t = n * n;
        yield t + 1;      // 'yield' hands the value back
    }
};
```

CHAPTER 2

Loops: Doing It Again

The picture

A loop is a *washing machine cycle*. Three questions answer every loop:

1) Where do I start?
2) When do I stop?
3) How do I move to the next round?

If you can answer those three for any loop, you can dry run it.

The while loop

Check first, then run. May run zero times.

```
int i = 1;              // 1. START
while (i &lt;= 5) {        // 2. STOP when this is false
    System.out.print(i + " ");
    i++;                // 3. MOVE
}
// 1 2 3 4 5
```

Dry run

Round	i at check	`i <= 5`	Printed	i after `i++`
1	1	true	1	2
2	2	true	2	3
3	3	true	3	4
4	4	true	4	5
5	5	true	5	6
6	6	false	-	loop ends

Notice round 6. The condition is checked **one extra time** and fails. The variable ends at 6, not 5. Missing this is the cause of a large share of off-by-one bugs.

> **INFINITE LOOP**
> Forget the `i++` and the condition never becomes false. The program hangs forever. Whenever you write `while`, immediately write the line that changes the condition, before writing anything else.

The do-while loop

Run first, then check. Always runs **at least once**.

```
int i = 10;
do {
    System.out.println("ran once");
} while (i &lt; 5);
// prints once even though 10 &lt; 5 is false
```

Use it for menus and input validation, where you must ask at least once before you can test the answer.

Loop	Check happens	Minimum runs	Typical use
`while`	Before the body	0	"keep going while there is data"
`do-while`	After the body	1	menus, retry until valid input
`for`	Before the body	0	counting a known number of times

The for loop

All three parts in one line. This is the most used loop in data structures because array indexes are counted.

```
for (int i = 0; i &lt; 5; i++) {
    System.out.print(i + " ");
}
// 0 1 2 3 4
```

Part	Code	When it runs
Initialisation	`int i = 0`	Once, at the very start
Condition	`i < 5`	Before every round, including the first
Body	`{ ... }`	When the condition is true
Update	`i++`	After every round, before the next check

Exact order of execution

Order	Action
1	init
2	check condition
3	if false, exit
4	run body
5	run update
6	go to step 2

Why loops start at 0

Because array indexes start at 0. An array of size 5 has indexes 0, 1, 2, 3, 4. So `for (int i = 0; i < n; i++)` visits every element exactly once. Using <= instead of < is the single most common runtime crash in Java: `ArrayIndexOutOfBoundsException`.

> **THE TWO LOOP TEMPLATES YOU WILL USE 90% OF THE TIME**
> Forward: `for (int i = 0; i < n; i++)`
> Backward: `for (int i = n - 1; i >= 0; i--)`
> Memorise both exactly. Do not improvise them.

Variations of for

```
for (int i = 10; i &gt; 0; i--) { }        // countdown
for (int i = 0; i &lt; 100; i += 5) { }    // step of 5
for (int i = 1; i &lt; 100; i *= 2) { }    // 1,2,4,8,16... logarithmic
for (int i = 0, j = 9; i &lt; j; i++, j--) { }   // two counters, two pointers
for (;;) { }                              // infinite, needs a break inside
```

That last two-counter form is exactly the **two pointer technique** used for reversing arrays and checking palindromes. You already know the syntax for it.

The for-each loop (enhanced for)

Reads every element without any index. Cleaner and impossible to go out of bounds.

```
int[] arr = {10, 20, 30};
for (int value : arr) {
    System.out.print(value + " ");
}
// 10 20 30
```

Read the colon as the word "in": *for each value in arr*.

Situation	Use
You only need the values	for-each
You need the index too	classic for
You need to modify array elements	classic for (for-each gives a copy of a primitive)
You need to go backwards	classic for
You need to remove from a collection while looping	Iterator, not for-each

Why for-each cannot modify primitives

```
int[] a = {1, 2, 3};
for (int x : a) { x = 99; }
System.out.println(a[0]);   // still 1
```

`x` is a fresh copy of the value on the stack. Changing the copy does nothing to the array. Go back to the stack and heap picture in Chapter 2 - this is the same rule.

break and continue

Keyword	Effect
`break`	Leave the loop completely, right now.
`continue`	Abandon the rest of THIS round, jump to the next round.

```
for (int i = 1; i &lt;= 5; i++) {
    if (i == 3) break;
    System.out.print(i + " ");
}
// 1 2

for (int i = 1; i &lt;= 5; i++) {
    if (i == 3) continue;
    System.out.print(i + " ");
}
// 1 2 4 5
```

CONTINUE INSIDE A WHILE LOOP: THE CLASSIC INFINITE LOOP

`while (i < 5) { if (i == 3) continue; i++; }` hangs forever, because `continue` skips the `i++`.
In a `for` loop this is safe, because the update part still runs. In `while`, you must increment before the `continue`.

Nested loops

A loop inside a loop. The inner loop finishes completely for every single round of the outer loop.

```
for (int i = 1; i &lt;= 3; i++) {
    for (int j = 1; j &lt;= 2; j++) {
        System.out.println(i + "," + j);
    }
}
```

Order	i	j	Printed
1	1	1	1,1
2	1	2	1,2
3	2	1	2,1
4	2	2	2,2
5	3	1	3,1
6	3	2	3,2

Total rounds = outer x inner = 3 x 2 = 6. This multiplication is exactly why nested loops cost O(n squared) time. The chapter on complexity makes that precise.

Labelled break and continue

Normally `break` only escapes the innermost loop. A label lets you escape several at once.

```
outer:
for (int i = 0; i &lt; 3; i++) {
    for (int j = 0; j &lt; 3; j++) {
        if (i * j == 2) break outer;    // leaves BOTH loops
        System.out.println(i + " " + j);
    }
}
```

i	j	i*j	Action
0	0,1,2	0	print each
1	0	0	print
1	1	1	print
1	2	2	break outer, program continues after both loops

Use labels rarely. Usually a well named method with a `return` is clearer.

Pattern printing - the best nested loop practice

Patterns exist in every syllabus for one honest reason: they force you to think about the relationship between the row number and the column count. That relationship is the skill.

Right triangle

```
int n = 4;
for (int i = 1; i &lt;= n; i++) {
    for (int j = 1; j &lt;= i; j++) {   // inner limit depends on i
        System.out.print("* ");
    }
    System.out.println();
}
```

Row i	Inner runs `j <= i`	Output
1	1 time	`*`
2	2 times	`* *`
3	3 times	`* * *`
4	4 times	`* * * *`

Inverted triangle

```
for (int i = n; i &gt;= 1; i--) {
    for (int j = 1; j &lt;= i; j++) System.out.print("* ");
    System.out.println();
}
```

Pyramid

The key insight: each row needs **spaces first**, then stars. Spaces shrink as stars grow.

```
int n = 4;
for (int i = 1; i &lt;= n; i++) {
    for (int s = 1; s &lt;= n - i; s++) System.out.print(" ");
    for (int j = 1; j &lt;= 2 * i - 1; j++) System.out.print("*");
    System.out.println();
}
```

Row i	Spaces `n - i`	Stars `2i - 1`	Output
1	3	1	`*`
2	2	3	`***`
3	1	5	`*****`

Row `i`	Spaces `n - i`	Stars `2i - 1`	Output
4	0	7	`*******`

Number pyramid and Pascal's triangle

```
// 1
// 1 2
// 1 2 3
for (int i = 1; i &lt;= n; i++) {
    for (int j = 1; j &lt;= i; j++) System.out.print(j + " ");
    System.out.println();
}
```

```
// Pascal's triangle: each value = previous * (i - j) / j
for (int i = 0; i &lt; n; i++) {
    int val = 1;
    for (int j = 0; j &lt;= i; j++) {
        System.out.print(val + " ");
        val = val * (i - j) / (j + 1);
    }
    System.out.println();
}
```

HOW TO SOLVE ANY PATTERN QUESTION

Write the desired output on paper. Number the rows. For each row write two numbers: how many spaces, how many symbols. Find the formula linking those to the row number. The code writes itself after that. The thinking is on paper, not in the editor.

CHAPTER 1

Methods And The Call Stack

The picture

A method is a **recipe with a name**. You write the steps once. After that you just say the name and the steps happen. If the recipe needs ingredients, you hand them over. If it produces a dish, it hands it back.

```
int add(int a, int b) {
    return a + b;
}
```

Part	Name	Meaning
`int`	Return type	The shape of the thing handed back. `void` means nothing is handed back.
`add`	Name	What you call it. Use a verb.
`(int a, int b)`	Parameters	Ingredients the recipe needs.
`{ ... }`	Body	The steps.
`return a + b;`	Return statement	Hand back the answer and stop immediately.

Parameters versus arguments

- **Parameter** is the name in the method definition. It is the empty labelled bowl.
- **Argument** is the actual value you pass in when calling. It is what you put in the bowl.

```
int add(int a, int b) { ... }   // a and b are parameters
add(3, 5);                       // 3 and 5 are arguments
```

Java is always pass by value. Always.

This confuses almost everyone, so here is the precise statement:

> **THE RULE**
> Java **always** copies the value in the variable and hands over the copy.
> For a primitive, the value IS the number, so the method gets a copy of the number.
> For an object, the value is the ADDRESS, so the method gets a copy of the address. Both now point at the same object in the heap.

```
static void changeNumber(int n) { n = 99; }
static void changeArray(int[] a) { a[0] = 99; }
static void replaceArray(int[] a) { a = new int[]{7,7,7}; }

public static void main(String[] args) {
    int x = 1;
    changeNumber(x);
    System.out.println(x);        // 1   - copy changed, original safe

    int[] arr = {1, 2, 3};
    changeArray(arr);
    System.out.println(arr[0]);   // 99  - same heap object touched

    replaceArray(arr);
    System.out.println(arr[0]);   // 99  - NOT 7. only the local copy of
                                  //       the address was repointed
}
```

Call	What was copied	Original changed?
`changeNumber(x)`	the number 1	No
`changeArray(arr)`	the address @2000	Yes, contents at @2000 changed

Call	What was copied	Original changed?
`replaceArray(arr)`	the address @2000	No, the local slip was rewritten, the caller's slip was not

That third case is the exact reason a method cannot swap two objects for its caller. If you understand this table you understand Java's memory semantics completely.

Method overloading

Same name, different parameter list. Java picks the right one by looking at the arguments.

```
int add(int a, int b)          { return a + b; }
double add(double a, double b) { return a + b; }
int add(int a, int b, int c)   { return a + b + c; }

add(2, 3);        // calls version 1
add(2.5, 3.5);    // calls version 2
add(1, 2, 3);     // calls version 3
```

OVERLOADING RULE
The methods must differ in the NUMBER or the TYPE or the ORDER of parameters.
A different **return type alone is not enough**. `int f()` and `double f()` will not compile together, because Java cannot tell from the call site which one you meant.

Varargs - an unknown number of arguments

```
static int sum(int... nums) {     // nums behaves like an int[]
    int total = 0;
    for (int n : nums) total += n;
    return total;
}
sum();            // 0
sum(1, 2);        // 3
sum(1, 2, 3, 4);  // 10
```

Only one varargs parameter is allowed and it must be last.

The call stack, step by step

Every method call puts a new **plate** (a stack frame) on the tower. The frame holds that call's parameters, its local variables, and a note of where to come back to.

```
public class Stack1 {
    static int square(int n) { return n * n; }
    static int sumOfSquares(int a, int b) {
        int x = square(a);
        int y = square(b);
        return x + y;
    }
    public static void main(String[] args) {
        System.out.println(sumOfSquares(2, 3));
    }
}
```

Time	Stack from bottom to top	Note
1	main	starts
2	main, sumOfSquares(2,3)	called
3	main, sumOfSquares, square(2)	called
4	main, sumOfSquares	square returned 4, its plate destroyed, x = 4
5	main, sumOfSquares, square(3)	called
6	main, sumOfSquares	returned 9, y = 9
7	main	sumOfSquares returned 13, plate destroyed
8	-	prints 13

STACKOVERFLOWERROR

The tower of plates has a fixed height, usually around 10,000 to 20,000 frames.
If methods keep calling without ever returning, the tower topples: `StackOverflowError`.
This is almost always caused by recursion with a missing or wrong base case.

CHAPTER 2

Recursion

The picture

Recursion is a method that calls itself. The everyday picture is a **line of people in a dark corridor**, each asking the person in front "how far to the door?" and nobody knowing, until the person at the very front says "I am at the door, zero steps". Then the answer travels back down the line, each person adding one.

Two things must exist, always:

1) **Base case** - the person at the door. The condition where the method stops calling itself and returns a direct answer.
2) **Recursive case** - the person asking the next person. The method calls itself with a **smaller** problem.

If the base case is missing, or if the problem does not shrink, you get `StackOverflowError`. That is the only way recursion fails.

Factorial

```
static int fact(int n) {
    if (n &lt;= 1) return 1;         // BASE CASE
    return n * fact(n - 1);       // RECURSIVE CASE, n shrinks
}
```

Dry run of fact(4) - the going down phase

Depth	Call	Base case?	Returns
1	fact(4)	no	4 * fact(3), waiting
2	fact(3)	no	3 * fact(2), waiting
3	fact(2)	no	2 * fact(1), waiting
4	fact(1)	YES	1

The coming back up phase

Depth	Waiting expression	Value received	Result returned
4	-	-	1
3	2 * ?	1	2
2	3 * ?	2	6
1	4 * ?	6	24

The answer 24 is only assembled on the way **back up**. Nothing is computed on the way down except the promise to multiply later. Understanding these two phases is what makes tree traversal make sense later.

Fibonacci - and why the naive version is a disaster

```
static int fib(int n) {
    if (n &lt;= 1) return n;
    return fib(n - 1) + fib(n - 2);
}
```

The call tree for fib(5)

```
                fib(5)
             /           \
         fib(4)           fib(3)
        /     \          /     \
    fib(3)   fib(2)   fib(2)   fib(1)
    /   \    /   \    /   \
 fib(2) fib(1) fib(1) fib(0) fib(1) fib(0)
  /  \
 fib(1) fib(0)
```

Count `fib(2)`: it is computed **three separate times**. For `fib(40)` the same subproblems are recomputed billions of times. The cost is roughly O(2^n): doubling with each step.

n	Approximate calls	Time on a normal laptop
10	177	instant
30	2.7 million	~0.02 s
40	331 million	~2 s
50	40 billion	~4 minutes

The fix: memoisation - remember what you already computed

```
static long[] memo = new long[100];
static long fib(int n) {
    if (n &lt;= 1) return n;
    if (memo[n] != 0) return memo[n];       // already known, reuse
    memo[n] = fib(n - 1) + fib(n - 2);
    return memo[n];
}
```

Now each `fib(k)` is computed once. Cost drops from O(2^n) to O(n). This single idea - store the answer to a subproblem so you never solve it twice - is the entire foundation of **dynamic programming**.

The even better fix: no recursion at all

```
static long fib(int n) {
    if (n &lt;= 1) return n;
    long a = 0, b = 1;
    for (int i = 2; i &lt;= n; i++) {
        long c = a + b;
        a = b;
        b = c;
    }
    return b;
}
```

i	a before	b before	c = a+b	a after	b after
2	0	1	1	1	1
3	1	1	2	1	2
4	1	2	3	2	3
5	2	3	5	3	5

O(n) time, O(1) space, no stack risk. This is the version to write in an interview.

THE HONEST VERDICT ON RECURSION

Recursion is not faster. It is usually slower and uses more memory, because every call costs a stack frame.
Recursion is used when the DATA is recursive: trees, graphs, nested folders, JSON. For those, recursive code is dramatically shorter and clearer.
For simple counting, use a loop.

Recursion versus iteration side by side

Aspect	Recursion	Iteration
Memory	O(depth) stack frames	O(1)

Aspect	Recursion	Iteration
Speed	Slower, call overhead	Faster
Readability for trees	Excellent	Painful, needs an explicit stack
Readability for counting	Poor	Excellent
Risk	StackOverflowError	Infinite loop

Classic recursive patterns you will reuse

Sum of an array

```
static int sum(int[] a, int i) {
    if (i == a.length) return 0;          // base: past the end
    return a[i] + sum(a, i + 1);          // this one plus the rest
}
```

Reverse a string

```
static String rev(String s) {
    if (s.length() &lt;= 1) return s;
    return rev(s.substring(1)) + s.charAt(0);
}
```

Binary search

```
static int bs(int[] a, int lo, int hi, int key) {
    if (lo &gt; hi) return -1;                      // base: not found
    int mid = lo + (hi - lo) / 2;
    if (a[mid] == key) return mid;
    if (a[mid] &lt; key) return bs(a, mid + 1, hi, key);
    return bs(a, lo, mid - 1, key);
}
```

Tower of Hanoi - the shape of "solve two smaller versions"

```
static void hanoi(int n, char from, char to, char via) {
    if (n == 0) return;
    hanoi(n - 1, from, via, to);                    // move top n-1 aside
    System.out.println("Move disk " + n + " " + from + "-&gt;" + to);
    hanoi(n - 1, via, to, from);                    // bring them back on
}
```

For n disks it prints 2^n - 1 moves, and that is provably the minimum. Three disks gives 7 moves.

How to write any recursive method - a fixed procedure

1) Ask: what is the smallest input where I already know the answer without thinking? That is the base case. Write it first.
2) Ask: if someone magically solved the problem for a slightly smaller input, how would I use their answer to solve mine? That is the recursive case.
3) Check that every path reaches the base case. If the input does not shrink on every call, it never will.

You do **not** need to trace the whole tree in your head. That is the mistake people make. Trust step 2. This is called the *recursive leap of faith* and it is how professionals actually write recursion.

PART 3 - ARRAYS AND STRINGS

CHAPTER 1

Arrays

The picture

An array is a **row of identical lockers, numbered from 0, built side by side in one block**. Three consequences follow from that picture, and every array behaviour comes from them:

1) All lockers are the same size, so the computer can jump straight to locker 7 by arithmetic: start address + 7 x size. That is why reading `arr[7]` takes the same time as `arr[0]`. **O(1) access.**
2) The lockers are built as one block, so the size must be decided at build time and **can never grow**.
3) Inserting in the middle means physically shifting everything after it. **O(n) insert.**

> **THE CORE TRADE OF THE ARRAY**
> Instant read by index. Fixed size. Slow insert and delete in the middle.
> Every other data structure in this book exists because one of those three is unacceptable for some job.

Declaring and creating

```
int[] a;                    // declaration only. 'a' is null. No lockers yet.
a = new int[5];             // now 5 lockers exist in the heap, all 0
int[] b = new int[5];       // both steps in one line
int[] c = {10, 20, 30};     // literal form, size inferred as 3
int[] d = new int[]{1, 2};  // needed when passing inline to a method
String[] names = new String[3];   // 3 slots, all null
```

Type	Default value in every slot
`int`, `short`, `byte`, `long`	0
`float`, `double`	0.0
`char`	'\u0000' (invisible)
`boolean`	false
Any object type	`null`

Notice arrays DO get default values, unlike local variables. The reason: `new` clears the memory it hands out.

Reading, writing, length

```
int[] a = {10, 20, 30};
System.out.println(a[0]);      // 10   first
System.out.println(a[2]);      // 30   last
System.out.println(a.length);  // 3    NOTE: no brackets, it is a field
System.out.println(a[a.length - 1]);  // 30   the standard 'last element' idiom
// a[3];   // ArrayIndexOutOfBoundsException at RUNTIME
```

> **LENGTH VERSUS LENGTH() VERSUS SIZE()**
> Array: `arr.length` - a field, no brackets.
> String: `s.length()` - a method, with brackets.
> Collection: `list.size()` - a different method again.
> Java is inconsistent here. There is no logic to learn, only the three forms to memorise.

Traversing an array - the four ways

```
int[] a = {5, 3, 8};

// 1. classic for - you get the index
for (int i = 0; i &lt; a.length; i++) System.out.print(a[i] + " ");

// 2. for-each - cleanest when you only need values
for (int v : a) System.out.print(v + " ");

// 3. backwards
for (int i = a.length - 1; i &gt;= 0; i--) System.out.print(a[i] + " ");

// 4. print the whole thing at once
System.out.println(Arrays.toString(a));   // [5, 3, 8]
```

WHY SYSTEM.OUT.PRINTLN(ARR) PRINTS GARBAGE

It prints something like `[I@6d06d69c`. That is not a bug. `[I` means "array of int" and the rest is the memory address in hexadecimal. Arrays do not override `toString()`. Always use `Arrays.toString(arr)`, or `Arrays.deepToString(arr)` for 2D.

Core array algorithms, each with a dry run

1. Find the largest element

```
int[] a = {3, 9, 2, 7};
int max = a[0];                       // assume the first is biggest
for (int i = 1; i &lt; a.length; i++) {
    if (a[i] &gt; max) max = a[i];
}
System.out.println(max);              // 9
```

i	a[i]	max before	a[i] > max?	max after
-	-	3 (from a[0])	-	3
1	9	3	yes	9
2	2	9	no	9
3	7	9	no	9

WHY START MAX AT A[0] AND NOT AT 0

If the array is all negative, starting at 0 gives the wrong answer 0. Starting at `a[0]` is always correct. If you must start with a constant, use `Integer.MIN_VALUE`.

2. Sum and average

```
int sum = 0;
for (int v : a) sum += v;
double avg = (double) sum / a.length;   // cast, or integer division ruins it
```

3. Linear search

```
static int find(int[] a, int key) {
    for (int i = 0; i &lt; a.length; i++) {
        if (a[i] == key) return i;      // return the INDEX
    }
    return -1;                          // the standard 'not found' signal
}
```

O(n). Works on unsorted data. Returning -1 rather than throwing is the Java convention (`indexOf` does the same).

4. Reverse an array in place - the two pointer technique

```
int[] a = {1, 2, 3, 4, 5};
int i = 0, j = a.length - 1;
while (i &lt; j) {
    int t = a[i]; a[i] = a[j]; a[j] = t;   // swap
    i++; j--;
}
```

Round	i	j	Array after swap
start	0	4	1 2 3 4 5
1	0	4	5 2 3 4 1
2	1	3	5 4 3 2 1
3	2	2	i < j false, stop

Notice the middle element never needs moving. The loop condition `i < j` handles both odd and even lengths automatically. O(n) time, O(1) space.

THE SWAP IDIOM

`int t = a[i]; a[i] = a[j]; a[j] = t;`

The temporary variable is required. Writing `a[i] = a[j]; a[j] = a[i];` destroys the first value. Say it out loud: *"save, overwrite, restore."*

5. Second largest - the classic trap

Wrong approach: sort, then take `a[n-2]`. That is O(n log n) and it breaks with duplicates.

```
int first = Integer.MIN_VALUE, second = Integer.MIN_VALUE;
for (int v : a) {
    if (v &gt; first) { second = first; first = v; }
    else if (v &gt; second &amp;&amp; v != first) { second = v; }
}
```

v	first before	second before	Branch	first after	second after
3	MIN	MIN	v > first	3	MIN
9	3	MIN	v > first	9	3
2	9	3	neither	9	3
7	9	3	second branch	9	7

O(n), one pass, handles duplicates. This pattern - keep the best two seen so far - generalises to top-k with a heap.

6. Count occurrences and find duplicates

```
// brute force: O(n^2)
for (int i = 0; i &lt; a.length; i++)
    for (int j = i + 1; j &lt; a.length; j++)
        if (a[i] == a[j]) System.out.println("dup " + a[i]);

// better: O(n) using a HashSet
Set&lt;Integer&gt; seen = new HashSet&lt;&gt;();
for (int v : a)
    if (!seen.add(v)) System.out.println("dup " + v);
```

`seen.add(v)` returns `false` if the value was already there. That single return value replaces the whole inner loop. This is the **hashing trade**: spend O(n) memory to remove a factor of n from the time.

7. Move all zeros to the end

```
int j = 0;                      // position for the next non-zero
for (int i = 0; i &lt; a.length; i++) {
    if (a[i] != 0) { int t = a[i]; a[i] = a[j]; a[j] = t; j++; }
}
```

i	a[i]	j before	Action	Array	j after
0	0	0	skip	0 1 0 3	0
1	1	0	swap i,j	1 0 0 3	1

i	a[i]	j before	Action	Array	j after
2	0	1	skip	1 0 0 3	1
3	3	1	swap i,j	1 3 0 0	2

Two pointers again, but this time one is a **write pointer** and one is a **read pointer**. That shape solves a whole family of problems: remove duplicates from a sorted array, remove a value, partition around a pivot.

8. Rotate an array by k - the reversal trick

Brute force: rotate one step, k times. O(n x k). The clever way is three reversals, O(n) time, O(1) space.

```
static void rotate(int[] a, int k) {
    int n = a.length;
    k = k % n;
    reverse(a, 0, n - 1);       // whole array
    reverse(a, 0, k - 1);       // first k
    reverse(a, k, n - 1);       // rest
}
```

Stage	Array (n=5, k=2)
Start	1 2 3 4 5
Reverse all	5 4 3 2 1
Reverse first 2	4 5 3 2 1
Reverse rest	4 5 1 2 3

2D arrays

A 2D array is an **array of arrays**. Picture a chess board, or a table with rows and columns.

```
int[][] grid = new int[3][4];       // 3 rows, 4 columns, all 0
int[][] m = {
    {1, 2, 3},
    {4, 5, 6}
};
System.out.println(m[1][2]);        // 6   -&gt; row 1, column 2
System.out.println(m.length);       // 2   -&gt; number of ROWS
System.out.println(m[0].length);    // 3   -&gt; length of row 0
```

MEMORY TRUTH

Java has no true 2D array. `m` is a 1D array holding 2 addresses. Each address points to a separate 1D array in the heap. This is why rows can have different lengths, and why `m[0].length` is asked per row.

Traversing

```
for (int i = 0; i &lt; m.length; i++) {           // row
    for (int j = 0; j &lt; m[i].length; j++) {    // column
        System.out.print(m[i][j] + " ");
    }
    System.out.println();
}
```

Round	i	j	m[i][j]
1	0	0	1
2	0	1	2
3	0	2	3
4	1	0	4
5	1	1	5
6	1	2	6

Jagged arrays

```
int[][] jag = new int[3][];      // 3 rows, lengths not yet decided
jag[0] = new int[2];
jag[1] = new int[5];
jag[2] = new int[1];
```

Matrix operations

```
// Transpose: swap rows and columns
for (int i = 0; i &lt; n; i++)
    for (int j = i + 1; j &lt; n; j++) {        // j starts at i+1, not 0!
        int t = m[i][j]; m[i][j] = m[j][i]; m[j][i] = t;
    }
```

If `j` started at 0, every pair would be swapped twice and you would get the original matrix back. Starting at `i + 1` visits each pair exactly once. That off-by-one is the whole trick.

```
// Rotate a square matrix 90 degrees clockwise = transpose, then reverse each row
transpose(m);
for (int[] row : m) reverse(row);
```

```
// Sum of the main diagonal
int sum = 0;
for (int i = 0; i &lt; n; i++) sum += m[i][i];
// Anti diagonal
for (int i = 0; i &lt; n; i++) sum += m[i][n - 1 - i];
```

The Arrays utility class

`java.util.Arrays` gives you tested implementations. Use them in real code; write your own only to learn.

Method	What it does	Cost
`Arrays.toString(a)`	Printable string	O(n)
`Arrays.deepToString(m)`	Printable string for 2D	O(n)
`Arrays.sort(a)`	Sorts ascending in place	O(n log n)
`Arrays.sort(a, from, to)`	Sorts a slice	O(k log k)
`Arrays.binarySearch(a, key)`	Index, or negative if absent. **Array must be sorted.**	O(log n)
`Arrays.fill(a, v)`	Put v in every slot	O(n)
`Arrays.copyOf(a, newLen)`	New array, padded or truncated	O(n)
`Arrays.copyOfRange(a, f, t)`	New array from a slice	O(k)
`Arrays.equals(a, b)`	Element by element comparison	O(n)
`Arrays.asList(a)`	Fixed size List view	O(1)
`Arrays.stream(a).sum()`	Sum via streams	O(n)

TWO TRAPS IN THIS TABLE

1. `Arrays.sort` on primitives uses dual-pivot quicksort, which is O(n^2) in the worst case. On objects it uses TimSort, which is guaranteed O(n log n) and stable.
2. `Arrays.asList(intArray)` gives a `List<int[]>` of size 1, not a list of numbers. Autoboxing does not apply to arrays. Use `Arrays.stream(a).boxed().toList()`.

Array copying - four ways, and which one is right

```
int[] a = {1, 2, 3};

int[] wrong = a;                          // NOT a copy. Same lockers.
int[] c1 = Arrays.copyOf(a, a.length);    // real copy, clearest
int[] c2 = a.clone();                     // real copy, shortest
int[] c3 = new int[3];
System.arraycopy(a, 0, c3, 0, 3);         // real copy, fastest, most verbose
```

SHALLOW VERSUS DEEP COPY

All four make a SHALLOW copy. For `int[]` that is a full copy, because ints are values.
For `Object[]`, the addresses are copied, so both arrays point at the same objects. Changing an object through one array changes what the other sees. For a deep copy you must copy each object yourself.

CHAPTER 1

Strings

The picture

A String is a **sealed envelope of letters**. Once sealed, the letters inside can never be changed. If you want different letters, you must make a brand new envelope. The old one is thrown away.

That single fact - **immutability** - explains every surprising String behaviour in Java.

```
String s = "Hello";
s.concat(" World");
System.out.println(s);        // "Hello"   &lt;- unchanged!

s = s.concat(" World");       // must reassign
System.out.println(s);        // "Hello World"
```

THE RULE
No String method ever modifies the String. Every one of them returns a NEW String.
`toUpperCase()`, `trim()`, `replace()`, `substring()` - all of them. If you do not assign the result, nothing happens.

Why immutable? Three real reasons

1) **Safety in the String pool.** Identical literals are shared. If one could be changed, changing "yes" in one place would change it everywhere in the program.
2) **Thread safety for free.** Something that cannot change cannot be corrupted by two threads.
3) **Safe as a HashMap key.** A key's hash code must never change. If a key could change, the map could no longer find its own entry.

The String pool

```
String a = "hi";
String b = "hi";
String c = new String("hi");

System.out.println(a == b);         // true  - both point to the pooled "hi"
System.out.println(a == c);         // false - 'new' forces a separate heap object
System.out.println(a.equals(c));    // true  - contents match
System.out.println(a == c.intern()); // true - intern() returns the pooled copy
```

Expression	Where it lives	Shared?
`"hi"` literal	String pool inside the heap	Yes, reused
`new String("hi")`	Normal heap	No, fresh object every time

PRACTICAL RULE
Never write `new String("...")`. It creates a needless object with no benefit.
Never compare Strings with ==. Use `.equals()`, or `.equalsIgnoreCase()` when case does not matter.

The essential String methods

Method	Meaning	Example on `s = "Hello"`
`length()`	Number of characters	5
`charAt(i)`	Character at index i	`s.charAt(1)` is 'e'
`indexOf(x)`	First index of x, or -1	`s.indexOf('l')` is 2
`lastIndexOf(x)`	Last index of x	`s.lastIndexOf('l')` is 3

Method	Meaning	Example on `s = "Hello"`
`substring(a)`	From a to the end	`s.substring(2)` is "llo"
`substring(a,b)`	From a up to but NOT including b	`s.substring(1,3)` is "el"
`contains(x)`	Is x inside?	`s.contains("ell")` is true
`equals(x)`	Same content?	-
`equalsIgnoreCase(x)`	Same ignoring case	-
`compareTo(x)`	Negative, 0, or positive by dictionary order	-
`toUpperCase()`	New upper case String	"HELLO"
`toLowerCase()`	New lower case String	"hello"
`trim()`	Remove spaces at both ends	-
`strip()`	Like trim but Unicode aware (Java 11+)	-
`replace(a,b)`	Swap every a for b	`s.replace('l','L')` is "HeLLo"
`split(regex)`	Break into an array	`"a,b".split(",")` is {"a","b"}
`toCharArray()`	Convert to `char[]`	-
`isEmpty()`	length == 0	-
`isBlank()`	Empty or only whitespace (Java 11+)	-
`startsWith` / `endsWith`	Prefix / suffix test	-
`repeat(n)`	Repeat n times (Java 11+)	`"ab".repeat(2)` is "abab"
`join(sep, parts)`	Glue with a separator	`String.join("-","a","b")` is "a-b"
`String.valueOf(x)`	Any value to String	-
`chars()`	IntStream of characters	-

SUBSTRING(A, B) - REMEMBER THE BOUNDARY
The start is INCLUDED, the end is EXCLUDED. `"Hello".substring(1, 3)` gives "el", not "ell".
The length of the result is always `b - a`. That subtraction is a fast sanity check.

String concatenation and why it is slow in a loop

```
// BAD - O(n^2)
String s = "";
for (int i = 0; i &lt; 10000; i++) s += i;

// GOOD - O(n)
StringBuilder sb = new StringBuilder();
for (int i = 0; i &lt; 10000; i++) sb.append(i);
String s = sb.toString();
```

Why the first is quadratic: because Strings are immutable, `s += i` must build a **whole new String** by copying every existing character. On round 5000 it copies about 5000 characters. Total copies are roughly 1 + 2 + 3 + ... + n, which is n(n+1)/2, which is O(n squared).

Loop size	String +=	StringBuilder
1,000	~2 ms	~0.1 ms
10,000	~150 ms	~0.3 ms
100,000	~30 seconds	~2 ms

Those numbers are approximate and machine dependent, but the shape is exact: one grows with the square, the other grows linearly.

StringBuilder - the mutable String

Picture a **whiteboard** instead of a sealed envelope. You can add, erase and rewrite without making a new board.

Method	Meaning
`append(x)`	Add to the end. Accepts any type.
`insert(i, x)`	Insert at index i
`delete(a, b)`	Remove a range
`deleteCharAt(i)`	Remove one character
`reverse()`	Reverse in place
`replace(a, b, str)`	Replace a range
`setCharAt(i, c)`	Change one character
`charAt(i)`, `length()`	Same as String
`toString()`	Freeze it into a String

```
StringBuilder sb = new StringBuilder("Hello");
sb.append(" World");     // Hello World
sb.insert(0, "&gt;&gt; ");     // &gt;&gt; Hello World
sb.reverse();            // dlroW olleH &gt;&gt;
System.out.println(sb.toString());
```

Class	Mutable?	Thread safe?	Speed	Use when
`String`	No	Yes, by nature	Slow to build	The value will not change
`StringBuilder`	Yes	No	Fastest	Building text in one thread. Default choice.
`StringBuffer`	Yes	Yes, synchronized	Slower	Multiple threads share it. Rare.

Characters are numbers

`char` is a 16-bit number. `'A'` is 65, `'a'` is 97, `'0'` is 48. Arithmetic works on them directly, and this unlocks a lot of clean code.

```
char c = 'A';
System.out.println((int) c);        // 65
System.out.println((char)(c + 1));  // B
System.out.println('5' - '0');      // 5   &lt;- character digit to real number
System.out.println((char)('a' + 2));// c
```

Trick	Code	Why it works
Digit char to int	`c - '0'`	'7' is 55, '0' is 48, difference is 7
Letter to index 0..25	`c - 'a'`	'c' is 99, 'a' is 97, gives 2
Index to letter	`(char)('a' + i)`	Reverse of the above
Upper to lower	`(char)(c + 32)`	Gap between 'A'(65) and 'a'(97) is 32
Toggle case	`(char)(c ^ 32)`	XOR flips exactly that one bit

The `c - 'a'` trick is the basis of the 26-slot frequency array used in anagram, first-unique-character and Trie problems.

Character utility methods

Method	Returns true when
`Character.isLetter(c)`	a to z or A to Z, any alphabet
`Character.isDigit(c)`	0 to 9
`Character.isLetterOrDigit(c)`	either of the above
`Character.isUpperCase(c)`	capital
`Character.isLowerCase(c)`	small
`Character.isWhitespace(c)`	space, tab, newline
`Character.toUpperCase(c)`	returns the capital version
`Character.getNumericValue(c)`	'7' becomes 7

Classic String algorithms with dry runs

1. Reverse a string

```
// Easiest, use in real code
String r = new StringBuilder(s).reverse().toString();

// By hand, two pointers on a char array
char[] c = s.toCharArray();
int i = 0, j = c.length - 1;
while (i &lt; j) {
    char t = c[i]; c[i] = c[j]; c[j] = t;
    i++; j--;
}
String r = new String(c);
```

2. Palindrome check

A palindrome reads the same forwards and backwards: "madam", "racecar".

```
static boolean isPal(String s) {
    int i = 0, j = s.length() - 1;
    while (i &lt; j) {
        if (s.charAt(i) != s.charAt(j)) return false;
        i++; j--;
    }
    return true;
}
```

Round	i	j	s.charAt(i)	s.charAt(j)	Match?
1	0	4	m	m	yes
2	1	3	a	a	yes
3	2	2	i < j is false, stop	-	return true

Better than reversing and comparing: it uses O(1) extra space, and it exits early on the first mismatch.

3. Count vowels and consonants

```
int v = 0, c = 0;
for (char ch : s.toLowerCase().toCharArray()) {
    if (!Character.isLetter(ch)) continue;
    if ("aeiou".indexOf(ch) &gt;= 0) v++;
    else c++;
}
```

`"aeiou".indexOf(ch) >= 0` replaces a five-way `||` chain. Shorter and easier to read.

4. Character frequency with a 26-slot array

```
int[] freq = new int[26];
for (char ch : s.toCharArray()) freq[ch - 'a']++;

for (int i = 0; i &lt; 26; i++)
    if (freq[i] &gt; 0)
        System.out.println((char)('a' + i) + " : " + freq[i]);
```

Character	`ch - 'a'`	Slot bumped
'b'	98 - 97	1
'a'	97 - 97	0
'n'	110 - 97	13

This is a **hash table with a perfect hash function**. It is O(1) lookup with zero collisions, and it is far faster than a `HashMap` when the alphabet is known and small. Use a `HashMap<Character,Integer>` only when the character set is unknown.

5. Anagram check

Two strings are anagrams if they use exactly the same letters: "listen" and "silent".

```
// Approach A: sort both. O(n log n)
static boolean anagram(String a, String b) {
    char[] x = a.toCharArray(), y = b.toCharArray();
    Arrays.sort(x); Arrays.sort(y);
    return Arrays.equals(x, y);
}

// Approach B: count. O(n), better
static boolean anagram2(String a, String b) {
    if (a.length() != b.length()) return false;
    int[] f = new int[26];
    for (int i = 0; i &lt; a.length(); i++) {
        f[a.charAt(i) - 'a']++;
        f[b.charAt(i) - 'a']--;
    }
    for (int v : f) if (v != 0) return false;
    return true;
}
```

Approach B walks both strings at once, adding for one and subtracting for the other. If they are anagrams every counter cancels back to zero. One pass, one small array.

6. First non-repeating character

```
static char firstUnique(String s) {
    int[] f = new int[26];
    for (char c : s.toCharArray()) f[c - 'a']++;
    for (char c : s.toCharArray()) if (f[c - 'a'] == 1) return c;
    return '_';
}
```

Two passes. The first counts everything, the second finds the first count of 1. You cannot do it in one pass, because "is this unique" cannot be answered before the whole string has been seen. Recognising *when two passes are unavoidable* is a real skill.

7. Remove duplicates from a string

```
StringBuilder sb = new StringBuilder();
Set&lt;Character&gt; seen = new HashSet&lt;&gt;();
for (char c : s.toCharArray())
    if (seen.add(c)) sb.append(c);
```

8. Count words in a sentence

```
String s = "  the quick  brown fox ";
String[] words = s.trim().split("\\s+");   // \s+ = one or more whitespace
System.out.println(words.length);          // 4
```

Splitting on a plain `" "` gives empty strings when there are double spaces. `\\s+` collapses runs of whitespace. The double backslash is needed because `\s` must survive both the Java String and the regex engine.

9. Reverse the words in a sentence

```
String[] w = s.trim().split("\\s+");
StringBuilder sb = new StringBuilder();
for (int i = w.length - 1; i &gt;= 0; i--) {
    sb.append(w[i]);
    if (i &gt; 0) sb.append(" ");
}
```

10. Longest word

```
String longest = "";
for (String w : s.split("\\s+"))
    if (w.length() &gt; longest.length()) longest = w;
```

Same "keep the best so far" shape as finding the maximum in an array. Once you see that shape, dozens of problems become the same problem.

String and text formatting

```
System.out.printf("Name: %s Age: %d Score: %.2f%n", "Asha", 25, 91.567);
// Name: Asha Age: 25 Score: 91.57

String s = String.format("%05d", 42);       // 00042
String t = "Total: %d items".formatted(7);  // Java 15+
```

Placeholder	Means
%d	whole number
%s	String or anything
%f	decimal, `%.2f` for 2 places
%c	character
%b	boolean
%n	newline, portable version of `\n`
%%	a literal percent sign

Text blocks (Java 15+)

```
String json = """
        {
          "name": "Asha",
          "age": 25
        }
        """;
```

Multi-line text without escaping quotes. Leading whitespace is stripped based on the least indented line.

PART 4 - OBJECTS, THE GLUE

CHAPTER 1

Just Enough Objects For Data Structures

Why this chapter exists

You cannot build a linked list, a tree or a graph without objects, because those structures are made of **nodes that point to other nodes**. This chapter covers exactly what is needed for that and nothing more.

Class and object

A class is a **blueprint**. An object is a **house built from it**. One blueprint, many houses. Changing one house does not change the others.

```
class Student {
    String name;              // field  (state)
    int marks;

    void show() {             // method (behaviour)
        System.out.println(name + " scored " + marks);
    }
}

Student a = new Student();    // build house 1
a.name = "Asha"; a.marks = 91;
Student b = new Student();    // build house 2
b.name = "Ravi"; b.marks = 78;
a.show();                     // Asha scored 91
```

Stack	Heap
a = @3000	@3000 -> Student{name="Asha", marks=91}
b = @3100	@3100 -> Student{name="Ravi", marks=78}

Constructors

A constructor is the **setup routine that runs at the moment of birth**. Same name as the class, no return type.

```
class Student {
    String name;
    int marks;

    Student() {                          // no-arg constructor
        this("Unknown", 0);              // calls the other one
    }
    Student(String name, int marks) {    // parameterised
        this.name = name;
        this.marks = marks;
    }
}
```

TWO CONSTRUCTOR RULES PEOPLE GET WRONG

1. If you write NO constructor, Java secretly gives you an empty one. The moment you write any constructor, that free gift disappears. So `new Student()` stops compiling once you add `Student(String, int)`.
2. `this(...)` must be the very first statement, and `super(...)` too. You cannot use both in one constructor.

The reference, drawn properly

This picture is the one that makes linked lists click.

```
class Node {
    int data;
    Node next;        // a Node holding the ADDRESS of another Node
}

Node a = new Node();
a.data = 10;
Node b = new Node();
b.data = 20;
a.next = b;           // a now points at b
```

Object	data	next
@4000 (a)	10	@4100
@4100 (b)	20	null

Read it as a chain: `a -> b -> null`. That is a linked list of two nodes. You have already built one.

> **WHY A NODE CAN CONTAIN A NODE**
> It does not contain one. It contains the **address** of one. If it truly contained one, the object would be infinitely large. This is precisely why Java's reference model makes linked structures possible at all.

Inheritance and polymorphism, briefly

```
class Animal { void speak() { System.out.println("..."); } }
class Dog extends Animal { void speak() { System.out.println("Woof"); } }
class Cat extends Animal { void speak() { System.out.println("Meow"); } }

Animal[] zoo = { new Dog(), new Cat() };
for (Animal x : zoo) x.speak();        // Woof, Meow
```

The variable type is `Animal`, but the actual object decides which `speak()` runs. That is **polymorphism**: one interface, many behaviours, decided at runtime.

Term	Meaning	Decided when
Overloading	Same name, different parameters	Compile time
Overriding	Child replaces parent's method, same signature	Runtime

> **WHY THIS MATTERS IN COLLECTIONS**
> `List<String> list = new ArrayList<>();` works because `ArrayList` is a kind of `List`.
> Declare the variable as the interface, create it as the class. Then swapping to `LinkedList` changes exactly one word in your whole program.

equals and hashCode - the contract you must not break

Every object inherits `equals()` and `hashCode()` from `Object`. The inherited versions compare **addresses**, which is almost never what you want.

```
class Point {
    int x, y;
    Point(int x, int y) { this.x = x; this.y = y; }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Point)) return false;
        Point p = (Point) o;
        return x == p.x &amp;&amp; y == p.y;
    }
    @Override
    public int hashCode() { return Objects.hash(x, y); }
}
```

THE CONTRACT, IN THREE LINES

1. If `a.equals(b)` is true, then `a.hashCode()` MUST equal `b.hashCode()`.
2. Equal hash codes do NOT require equal objects. Collisions are allowed.
3. Break rule 1 and `HashMap` and `HashSet` silently misbehave: you put an object in and can never find it again.

Why breaking it fails, mechanically

A `HashSet` finds the bucket using `hashCode()`, then compares inside the bucket using `equals()`. If you override `equals` but not `hashCode`, two equal objects land in **different buckets** and are never compared. The set happily stores both.

```
Set&lt;Point&gt; s = new HashSet&lt;&gt;();
s.add(new Point(1, 1));
s.add(new Point(1, 1));
System.out.println(s.size());   // 1 with both overridden, 2 with only equals
```

Comparable and Comparator - how sorting knows the order

Java cannot guess whether `Student` A comes before `Student` B. You must tell it. There are two ways.

Comparable - the object's own natural order

```
class Student implements Comparable&lt;Student&gt; {
    String name; int marks;
    public int compareTo(Student other) {
        return this.marks - other.marks;    // ascending by marks
    }
}
Arrays.sort(students);       // now this works
```

`compareTo` **returns**	**Meaning**
negative	this comes BEFORE other
zero	order does not matter, they tie
positive	this comes AFTER other

THE SUBTRACTION TRICK AND ITS BUG

`a - b` is short and usually fine, but it OVERFLOWS with large numbers. `Integer.MIN_VALUE - 1` wraps to positive and your sort silently produces garbage. The safe form is `Integer.compare(a, b)`. Use that.

Comparator - an external, swappable order

```
Comparator&lt;Student&gt; byName = (p, q) -&gt; p.name.compareTo(q.name);
Arrays.sort(students, byName);

// modern chained version
students.sort(Comparator.comparingInt((Student s) -&gt; s.marks)
                        .reversed()
                        .thenComparing(s -&gt; s.name));
```

	Comparable	**Comparator**
Where the logic lives	Inside the class	Outside, separate object
How many orderings	One	As many as you like
Method	`compareTo(other)`	`compare(a, b)`
Use when	There is one obvious natural order	You need to sort several ways

Generics - the type in angle brackets

Generics let you write a container once and use it with any type, while keeping full compile-time checking.

```
// Without generics: unsafe
List raw = new ArrayList();
raw.add("hello");
Integer n = (Integer) raw.get(0);   // compiles, CRASHES at runtime

// With generics: safe
List&lt;String&gt; list = new ArrayList&lt;&gt;();
list.add("hello");
// list.add(42);         // compile error, caught immediately
String s = list.get(0);  // no cast needed
```

Writing your own generic class

This is exactly how you will write your linked list and stack.

```
class Box&lt;T&gt; {                      // T is a placeholder for a real type
    private T item;
    void put(T item) { this.item = item; }
    T get() { return item; }
}

Box&lt;String&gt; b1 = new Box&lt;&gt;();
b1.put("hi");
String s = b1.get();       // no cast

Box&lt;Integer&gt; b2 = new Box&lt;&gt;();
b2.put(42);
```

Letter	Conventional meaning
T	Type
E	Element, used in collections
K	Key
V	Value
N	Number
?	Wildcard, "some unknown type"

GENERICS ONLY ACCEPT OBJECTS

`List<int>` does not compile. You must write `List<Integer>`. Java's generics are erased at compile time and primitives have no object identity, so the wrapper classes exist to fill the gap.

Autoboxing and its hidden costs

Java automatically converts between `int` and `Integer`. Convenient, and occasionally expensive or wrong.

Primitive	Wrapper class
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`
`char`	`Character`
`boolean`	`Boolean`

```
List&lt;Integer&gt; list = new ArrayList&lt;&gt;();
list.add(5);                 // autoboxing: int 5 -&gt; Integer.valueOf(5)
int x = list.get(0);         // unboxing: Integer -&gt; int
```

Trap 1: the Integer cache

```
Integer a = 127, b = 127;
System.out.println(a == b);     // true

Integer c = 128, d = 128;
System.out.println(c == d);     // false !
System.out.println(c.equals(d));// true
```

Java caches `Integer` objects from -128 to 127 and reuses them. Above 127 each autobox creates a new object, so `==` compares different addresses. **Always use `.equals()` for wrapper comparison.**

Trap 2: unboxing null

```
Map&lt;String, Integer&gt; m = new HashMap&lt;&gt;();
int count = m.get("missing");   // NullPointerException!
```

`get` returns `null`, then Java tries to unbox `null` into an `int`. Use `m.getOrDefault("missing", 0)` instead.

Trap 3: performance

```
Long sum = 0L;                    // WRONG: object
for (long i = 0; i &lt; 1_000_000; i++) sum += i;
// creates a million Long objects
```

Use `long sum = 0L;`. In tight loops this can be 5 to 10 times slower.

Interfaces you will meet in collections

Interface	Its one job
`Iterable<T>`	Can be walked with a for-each loop. Requires `iterator()`.
`Iterator<T>`	The walker itself. `hasNext()`, `next()`, `remove()`.
`Comparable<T>`	I know my own natural order.
`Comparator<T>`	I can order two of those.
`Collection<E>`	Common ground for List, Set, Queue.
`List<E>`	Ordered, indexed, duplicates allowed.
`Set<E>`	No duplicates.
`Map<K,V>`	Key to value lookup. Note: NOT a Collection.
`Queue<E>`	Add at one end, remove at the other.

Making your own class work with for-each

```
class Bag implements Iterable&lt;Integer&gt; {
    private int[] items = {1, 2, 3};
    public Iterator&lt;Integer&gt; iterator() {
        return new Iterator&lt;Integer&gt;() {
            int i = 0;
            public boolean hasNext() { return i &lt; items.length; }
            public Integer next() { return items[i++]; }
        };
    }
}

for (int x : new Bag()) System.out.print(x + " ");   // 1 2 3
```

The for-each loop is not magic. The compiler rewrites it into `hasNext()` and `next()` calls. Once you have seen this, `ConcurrentModificationException` will make sense: it happens because the iterator notices the collection changed underneath it.

PART 5 - THINKING ABOUT COST

CHAPTER 1

Complexity: How To Judge A Solution

The picture

Two people can both find your house. One asks every person on the street. One reads the house number. Both work. On a street of 10 houses the difference is nothing. On a street of 10 million houses one of them is still walking.

Complexity is not about seconds. Seconds depend on your laptop. Complexity is about **how the work grows when the input grows**. That is machine independent, and it is the thing that actually decides whether your program finishes.

Big-O, defined without mathematics

Big-O answers one question: *if the input doubles, what happens to the work?*

Notation	Name	If input doubles...	Example
O(1)	Constant	nothing changes	`arr[5]`, `map.get(k)`, push on a stack
O(log n)	Logarithmic	one extra step	binary search, balanced tree lookup
O(n)	Linear	work doubles	one loop over an array
O(n log n)	Linearithmic	slightly more than doubles	merge sort, `Arrays.sort`
O(n^2)	Quadratic	work x 4	two nested loops, bubble sort
O(n^3)	Cubic	work x 8	three nested loops, naive matrix multiply
O(2^n)	Exponential	work squares	naive fibonacci, all subsets
O(n!)	Factorial	catastrophic	all permutations, brute force travelling salesman

The numbers that make it real

Assume a computer does about 100 million simple steps per second.

n	O(log n)	O(n)	O(n log n)	O(n^2)	O(2^n)
10	3	10	33	100	1,024
100	7	100	664	10,000	10^30
1,000	10	1,000	9,966	1 million	-
100,000	17	100,000	1.7 million	10 billion	-
1,000,000	20	1 million	20 million	10^12	-

n	O(n^2) real time	O(n log n) real time
1,000	0.01 sec	instant
10,000	1 sec	instant
100,000	100 sec	0.02 sec
1,000,000	3 hours	0.2 sec

THE PRACTICAL RULE USED IN EVERY INTERVIEW

Look at the input size in the problem statement.
n up to 10 -> O(n!) or O(2^n) is fine.
n up to 1,000 -> O(n^2) is fine.
n up to 100,000 -> you need O(n log n) or better.
n up to 10,000,000 -> you need O(n) or O(log n).
The constraint tells you the required complexity before you have written a line.

How to count complexity from code

Four mechanical rules. No theory needed.

1) A simple statement is O(1).
2) A loop multiplies. `for i in n` around O(1) work is O(n).
3) Nested loops multiply together. n x n is O(n^2).
4) Sequential blocks add, then you keep only the biggest.

```
for (int i = 0; i &lt; n; i++) { }          // O(n)
for (int i = 0; i &lt; n; i++)              // O(n^2)
    for (int j = 0; j &lt; n; j++) { }
// total: O(n) + O(n^2) = O(n^2)
```

Drop constants and lower terms

`O(3n^2 + 500n + 9000)` is written `O(n^2)`. Why? Because as n grows huge, only the fastest growing term matters. At n = 1,000,000, the `n^2` term is a million times bigger than the `n` term. The constants are noise.

> **BUT CONSTANTS DO MATTER IN REAL LIFE**
> O(n) with a huge constant can be slower than O(n log n) with a small one, at realistic input sizes. Big-O tells you which one wins **eventually**. Always measure when performance actually matters.

The tricky one: why halving gives log n

```
for (int i = 1; i &lt; n; i = i * 2) { }
```

i goes 1, 2, 4, 8, 16... How many doublings to reach n? That is exactly the definition of log base 2 of n. For n = 1,000,000 that is 20 rounds. Twenty. That is why binary search feels like magic.

Two loops that look nested but are not O(n^2)

```
int j = 0;
for (int i = 0; i &lt; n; i++) {
    while (j &lt; n &amp;&amp; cond) j++;    // j NEVER resets
}
```

`j` only moves forward, at most n times total across the whole run. So this is O(n), not O(n^2). This is the **sliding window** pattern. Counting the total movement of a pointer, rather than the nesting depth, is the correct way to analyse it.

Space complexity

Same idea, applied to extra memory. The input itself does not count; only what you allocate.

Code	Space
A few variables	O(1)
A new array of size n	O(n)
A HashMap of every element	O(n)
Recursion of depth n	O(n) for the stack frames
A 2D DP table n by n	O(n^2)

> **THE UNIVERSAL TRADE**
> Almost every optimisation in this book is the same trade: spend memory to save time.
> Linear search is O(n) time, O(1) space. Add a HashMap and it becomes O(1) time, O(n) space.
> When you are asked to optimise, ask first: "what can I afford to remember?"

Best, average and worst case

Case	Meaning	Example: linear search
Best	Luckiest input	O(1), the item is first
Average	Typical input	O(n/2) which is O(n)
Worst	Nastiest input	O(n), the item is last or absent

Big-O usually means the **worst case**, because that is what you must survive. Some structures are stated as **amortised**: `ArrayList.add` is O(1) amortised. Any single add is usually O(1), but occasionally the array is full and everything must be copied, costing O(n). Spread across many adds, the average is O(1). Amortised means "averaged over a long run of operations", not "usually".

CHAPTER 2

From Brute Force To Optimised

What brute force actually means

Brute force means: **try every possibility, check each one, keep what works.** No cleverness. It is not a bad word. It is the correct first step.

> **THE PROFESSIONAL WORKFLOW**
> 1. Write the brute force. It is usually right and always simple.
> 2. Make it correct. Test edge cases.
> 3. Only then, find the wasted work and remove it.
> Skipping step 1 is how people freeze in interviews. A working O(n^2) beats a broken O(n) every single time.

The worked example: two sum

Given an array and a target, find two indexes whose values add to the target.

Level 1: brute force, O(n^2)

```
for (int i = 0; i &lt; n; i++)
    for (int j = i + 1; j &lt; n; j++)
        if (a[i] + a[j] == target) return new int[]{i, j};
```

Try every pair. For `a = {2, 7, 11, 15}` and target 9:

i	j	a[i]	a[j]	Sum	Match?
0	1	2	7	9	YES, return {0,1}

Correct. But for n = 100,000 that is 5 billion pairs. Too slow.

Finding the waste

Ask: **what am I recomputing?** For each `i`, the inner loop searches the whole rest of the array looking for the value `target - a[i]`. That is a *search*. Searching is the thing hash tables make instant.

Level 2: hashing, O(n)

```
Map&lt;Integer, Integer&gt; seen = new HashMap&lt;&gt;();
for (int i = 0; i &lt; n; i++) {
    int need = target - a[i];
    if (seen.containsKey(need)) return new int[]{seen.get(need), i};
    seen.put(a[i], i);
}
```

i	a[i]	need	Is need in map?	Map after
0	2	7	no	{2:0}
1	7	2	YES, return {0,1}	-

One pass. O(n) time, O(n) space. We paid memory and bought back a factor of n in time.

Level 3: if the array is sorted, two pointers, O(n) time and O(1) space

```
int i = 0, j = n - 1;
while (i &lt; j) {
    int sum = a[i] + a[j];
    if (sum == target) return new int[]{i, j};
    if (sum &lt; target) i++;     // need bigger, move left pointer right
    else j--;                 // need smaller, move right pointer left
}
```

i	j	a[i]	a[j]	Sum	Versus target 22	Move
0	4	2	15	17	too small	i++
1	4	7	15	22	match	return

THE LESSON OF THIS EXAMPLE

The three solutions are not three tricks. They are three different pieces of knowledge being exploited:
brute force exploits nothing, hashing exploits "I can remember what I have seen", two pointers exploits "the data is sorted".
Optimisation is always: find a property of the problem you are not using yet.

The six patterns that solve most problems

Pattern	Recognise it when	Turns
Hashing	You are searching repeatedly	O(n^2) into O(n)
Two pointers	Data is sorted, or you work from both ends	O(n^2) into O(n)
Sliding window	"Subarray" or "substring" of some size or property	O(n^2) into O(n)
Prefix sum	Many range-sum queries	O(n) per query into O(1)
Sorting first	"Group", "duplicate", "closest", "k-th"	often unlocks two pointers
Binary search	Data sorted, or the answer is monotonic	O(n) into O(log n)

Sliding window, explained

Picture a **window frame on a train**, sliding along the scenery. You do not re-look at everything each time. You add what enters on the right and drop what leaves on the left.

Problem: largest sum of any 3 consecutive numbers.

```
// Brute force O(n*k)
for (int i = 0; i + k &lt;= n; i++) {
    int s = 0;
    for (int j = i; j &lt; i + k; j++) s += a[j];
    max = Math.max(max, s);
}

// Sliding window O(n)
int sum = 0;
for (int i = 0; i &lt; k; i++) sum += a[i];    // first window
int max = sum;
for (int i = k; i &lt; n; i++) {
    sum += a[i] - a[i - k];                // add new, remove old
    max = Math.max(max, sum);
}
```

i	Entering a[i]	Leaving a[i-k]	Sum	Max
-	first window 1+2+3	-	6	6
3	4	1	6+4-1 = 9	9
4	5	2	9+5-2 = 12	12

One addition and one subtraction per step, instead of re-adding k numbers. The whole idea is: *reuse the previous answer instead of rebuilding it.*

Prefix sum, explained

Precompute running totals once. Then any range sum is one subtraction.

```
int[] pre = new int[n + 1];
for (int i = 0; i &lt; n; i++) pre[i + 1] = pre[i] + a[i];
// sum of a[l..r] inclusive:
int sum = pre[r + 1] - pre[l];
```

Index	0	1	2	3	4
a	2	4	1	6	-
pre	0	2	6	7	13

Sum of `a[1..2]` = `pre[3] - pre[1]` = 7 - 2 = 5, and indeed 4 + 1 = 5. Setup costs O(n) once, then every query is O(1). If there are a million queries, this is a million-fold improvement.

Binary search, the technique behind O(log n)

Precondition: the data must be **sorted**. Then each check throws away half of what remains.

```
static int bs(int[] a, int key) {
    int lo = 0, hi = a.length - 1;
    while (lo &lt;= hi) {
        int mid = lo + (hi - lo) / 2;      // overflow-safe midpoint
        if (a[mid] == key) return mid;
        if (a[mid] &lt; key) lo = mid + 1;    // answer is on the right
        else hi = mid - 1;                 // answer is on the left
    }
    return -1;
}
```

Dry run: find 7 in {1, 3, 5, 7, 9, 11}

Round	lo	hi	mid	a[mid]	Compare to 7	Action
1	0	5	2	5	too small	lo = 3
2	3	5	4	9	too big	hi = 3
3	3	3	3	7	equal	return 3

Six elements, three checks. A million elements would take twenty.

> **THE THREE BINARY SEARCH BUGS, AND THEIR FIXES**
> 1. `(lo + hi) / 2` overflows for large indexes. Use `lo + (hi - lo) / 2`.
> 2. `while (lo < hi)` versus `while (lo <= hi)` - with <= and `mid +/- 1` updates, the loop always shrinks and terminates. Learn one form and never improvise.
> 3. Running it on unsorted data gives a wrong answer with no error message. Always check the precondition.

Sorting algorithms, compared honestly

You will rarely write these in production - `Arrays.sort` is better than anything you will write. You learn them because they teach loop invariants, swapping, recursion and divide and conquer.

Algorithm	Best	Average	Worst	Space	Stable?	Idea in one line
Bubble	O(n)	O(n^2)	O(n^2)	O(1)	Yes	Repeatedly swap neighbours that are out of order
Selection	O(n^2)	O(n^2)	O(n^2)	O(1)	No	Find the smallest, put it at the front, repeat
Insertion	O(n)	O(n^2)	O(n^2)	O(1)	Yes	Like sorting cards in your hand
Merge	O(n log n)	O(n log n)	O(n log n)	O(n)	Yes	Split in half, sort each, merge
Quick	O(n log n)	O(n log n)	O(n^2)	O(log n)	No	Pick a pivot, partition, recurse
Heap	O(n log n)	O(n log n)	O(n log n)	O(1)	No	Build a heap, repeatedly take the max
Counting	O(n+k)	O(n+k)	O(n+k)	O(k)	Yes	Count each value, rebuild. Small integer ranges only.

Stable means two equal elements keep their original relative order. It matters when you sort by one field and then another.

Bubble sort with a full dry run

```
for (int i = 0; i &lt; n - 1; i++) {
    boolean swapped = false;
    for (int j = 0; j &lt; n - 1 - i; j++) {
        if (a[j] &gt; a[j + 1]) {
            int t = a[j]; a[j] = a[j+1]; a[j+1] = t;
            swapped = true;
        }
    }
    if (!swapped) break;      // already sorted, stop early
}
```

Array `{5, 1, 4, 2}`:

Pass	Comparisons	Array after pass	Note
1	5v1 swap, 5v4 swap, 5v2 swap	1 4 2 5	5 has bubbled to the end
2	1v4 no, 4v2 swap	1 2 4 5	4 is in place
3	1v2 no	1 2 4 5	no swaps, break

Two details worth noticing. `n - 1 - i` shrinks the inner loop because the last `i` elements are already correct. The `swapped` flag makes the best case O(n) on already-sorted data. Both are small changes that change the complexity class.

Selection sort

```
for (int i = 0; i &lt; n - 1; i++) {
    int min = i;
    for (int j = i + 1; j &lt; n; j++) if (a[j] &lt; a[min]) min = j;
    int t = a[i]; a[i] = a[min]; a[min] = t;
}
```

Always O(n^2), even on sorted data, because it always scans for the minimum. But it does at most n swaps, which matters when writes are expensive.

Insertion sort

```
for (int i = 1; i &lt; n; i++) {
    int key = a[i], j = i - 1;
    while (j &gt;= 0 &amp;&amp; a[j] &gt; key) { a[j + 1] = a[j]; j--; }
    a[j + 1] = key;
}
```

This is how you sort playing cards in your hand: take the next card, slide it left until it fits. Nearly sorted data makes it almost O(n), which is why real library sorts switch to insertion sort for small or nearly-sorted chunks.

Merge sort

```
static void sort(int[] a, int l, int r) {
    if (l &gt;= r) return;
    int m = l + (r - l) / 2;
    sort(a, l, m);
    sort(a, m + 1, r);
    merge(a, l, m, r);
}
```

Divide and conquer. Splitting takes log n levels, and each level does O(n) merging work, so O(n log n) total. Guaranteed, always, even in the worst case. Its cost is O(n) extra memory.

Quick sort

```
static void quick(int[] a, int lo, int hi) {
    if (lo &gt;= hi) return;
    int p = partition(a, lo, hi);
    quick(a, lo, p - 1);
    quick(a, p + 1, hi);
}
static int partition(int[] a, int lo, int hi) {
    int pivot = a[hi], i = lo - 1;
    for (int j = lo; j &lt; hi; j++)
        if (a[j] &lt; pivot) { i++; swap(a, i, j); }
    swap(a, i + 1, hi);
    return i + 1;
}
```

Faster than merge sort in practice because it sorts in place with good cache behaviour. Its weakness: if the pivot is always the smallest or largest element - which happens on already-sorted data with a naive pivot choice - it degrades to O(n^2). Real implementations pick the pivot randomly or use median-of-three to make that essentially impossible.

PART 6 - BUILDING THE STRUCTURES BY HAND

CHAPTER 1

Linked List

The picture

An array is a row of joined lockers. A linked list is a **treasure hunt**: each note tells you where the next note is. The notes can be scattered anywhere in the building.

	Array	**Linked list**
Memory	One solid block	Scattered, connected by addresses
Access `get(i)`	O(1), jump straight there	O(n), must walk from the start
Insert at front	O(n), shift everything	O(1), just repoint
Insert at end	O(1) if space, else O(n)	O(1) with a tail pointer
Delete from middle	O(n) shift	O(1) once you are there
Extra memory	None	One address per node
Cache friendliness	Excellent	Poor, jumps all over memory

THE HONEST VERDICT

Linked lists are taught everywhere and used rarely. In real Java code `ArrayList` beats `LinkedList` almost always, because modern CPUs love contiguous memory.

You learn linked lists because they teach POINTER MANIPULATION, which is the exact skill trees and graphs need. That is the real payoff.

The Node

```
class Node {
    int data;
    Node next;
    Node(int data) { this.data = data; this.next = null; }
}
```

That is the whole idea: a value, plus the address of the next node. The last node's `next` is `null`, which is how you know you have reached the end.

Singly linked list, complete

```
class LinkedList {
    private Node head;      // address of the first node, null if empty
    private int size;

    // 1. Insert at the FRONT - O(1)
    void addFirst(int data) {
        Node n = new Node(data);
        n.next = head;      // new node points to old first
        head = n;           // head now points to new node
        size++;
    }

    // 2. Insert at the END - O(n) without a tail pointer
    void addLast(int data) {
        Node n = new Node(data);
        if (head == null) { head = n; size++; return; }
        Node cur = head;
        while (cur.next != null) cur = cur.next;   // walk to the last node
        cur.next = n;
        size++;
    }

    // 3. Insert at a position
    void addAt(int index, int data) {
        if (index == 0) { addFirst(data); return; }
        Node cur = head;
        for (int i = 0; i &lt; index - 1; i++) cur = cur.next;  // stop BEFORE
        Node n = new Node(data);
        n.next = cur.next;
        cur.next = n;
        size++;
    }

    // 4. Delete the first - O(1)
    void removeFirst() {
        if (head == null) throw new RuntimeException("empty");
        head = head.next;    // old head becomes unreachable, GC collects it
        size--;
    }

    // 5. Delete by value
    void remove(int data) {
        if (head == null) return;
        if (head.data == data) { head = head.next; size--; return; }
        Node cur = head;
        while (cur.next != null &amp;&amp; cur.next.data != data) cur = cur.next;
        if (cur.next != null) { cur.next = cur.next.next; size--; }
    }

    // 6. Search
    int indexOf(int data) {
        Node cur = head;
        int i = 0;
        while (cur != null) {
            if (cur.data == data) return i;
            cur = cur.next; i++;
        }
        return -1;
    }

    // 7. Print
    void print() {
        for (Node cur = head; cur != null; cur = cur.next)
            System.out.print(cur.data + " -&gt; ");
        System.out.println("null");
    }
}
```

Dry run: addFirst on an empty list, then twice more

Action	head before	New node	n.next = head	head = n	List
addFirst(10)	null	@100{10}	@100.next = null	head=@100	10 -> null

Action	head before	New node	n.next = head	head = n	List
addFirst(20)	@100	@200{20}	@200.next = @100	head=@200	20 -> 10 -> null
addFirst(30)	@200	@300{30}	@300.next = @200	head=@300	30 -> 20 -> 10 -> null

THE ORDER OF THE TWO LINES IS EVERYTHING

`n.next = head; head = n;` is correct.

`head = n; n.next = head;` destroys the list: head is overwritten first, so the new node points at itself and every old node is lost forever.

When manipulating pointers, always attach the new node to the existing chain BEFORE moving the entry pointer.

Reversing a linked list - the most asked question in interviews

Three pointers. Walk forward, flipping each arrow backwards as you go.

```
Node reverse(Node head) {
    Node prev = null, cur = head;
    while (cur != null) {
        Node next = cur.next;   // 1. SAVE the rest of the list
        cur.next = prev;        // 2. FLIP this arrow backwards
        prev = cur;             // 3. move prev forward
        cur = next;             // 4. move cur forward
    }
    return prev;                // prev is the new head
}
```

Dry run on 1 -> 2 -> 3 -> null

Round	prev	cur	next saved	After flip	List so far
start	null	1	-	-	1->2->3
1	null	1	2	1->null	prev=1, cur=2
2	1	2	3	2->1	prev=2, cur=3
3	2	3	null	3->2	prev=3, cur=null
end	3	null	-	-	3->2->1->null

Step 1 (saving `next`) is the step everyone forgets, and without it the rest of the list is lost the instant you overwrite `cur.next`. Say the four steps aloud: **save, flip, advance, advance.**

Floyd's cycle detection - the tortoise and the hare

Two runners on a circular track. The fast one runs two steps for every one of the slow one. If the track is a loop, they must eventually meet. If there is a finish line, the fast one reaches it first.

```
boolean hasCycle(Node head) {
    Node slow = head, fast = head;
    while (fast != null &amp;&amp; fast.next != null) {
        slow = slow.next;
        fast = fast.next.next;
        if (slow == fast) return true;
    }
    return false;
}
```

O(n) time and O(1) space. The obvious alternative - put every node in a HashSet - also works but costs O(n) memory.

The same trick finds the middle

```
Node middle(Node head) {
    Node slow = head, fast = head;
    while (fast != null &amp;amp;&amp;amp; fast.next != null) {
        slow = slow.next;
        fast = fast.next.next;
    }
    return slow;      // when fast reaches the end, slow is at the middle
}
```

Round	slow at	fast at	List 1->2->3->4->5
1	2	3	-
2	3	5	-
3	fast.next is null, stop	-	slow = 3, the middle

Fast moves twice as far, so when fast has covered n, slow has covered n/2. Simple, and it needs only one pass.

Doubly linked list

Each node also knows the node **behind** it. This makes backwards traversal and O(1) deletion possible.

```
class DNode {
    int data;
    DNode prev, next;
    DNode(int d) { data = d; }
}

class DoublyLinkedList {
    DNode head, tail;

    void addLast(int d) {
        DNode n = new DNode(d);
        if (tail == null) { head = tail = n; return; }
        n.prev = tail;
        tail.next = n;
        tail = n;
    }

    void remove(DNode n) {                  // O(1), no walking needed!
        if (n.prev != null) n.prev.next = n.next; else head = n.next;
        if (n.next != null) n.next.prev = n.prev; else tail = n.prev;
    }
}
```

	Singly	Doubly
Memory per node	1 pointer	2 pointers
Backwards walk	Impossible	Easy
Delete a known node	O(n), need the previous one	O(1)
Used by	simple stacks and queues	Java's LinkedList, LRU caches, browser history

Circular linked list

The last node points back to the first instead of to `null`. Used for round-robin scheduling and circular buffers. The traversal condition changes from `cur != null` to `cur != head` after the first step, otherwise you loop forever.

CHAPTER 2

Stack

The picture

A **pile of plates**. You put a plate on top. You take a plate off the top. You cannot reach the one at the bottom without removing everything above it. That rule is called **LIFO**: Last In, First Out.

Operation	Meaning	Cost
`push(x)`	Put x on top	O(1)
`pop()`	Remove and return the top	O(1)
`peek()`	Look at the top without removing	O(1)
`isEmpty()`	Any plates?	O(1)
`size()`	How many	O(1)

Stack using an array

```
class ArrayStack {
    private int[] a;
    private int top = -1;      // -1 means empty

    ArrayStack(int cap) { a = new int[cap]; }

    void push(int x) {
        if (top == a.length - 1) throw new RuntimeException("Stack overflow");
        a[++top] = x;          // move up first, then write
    }
    int pop() {
        if (top == -1) throw new RuntimeException("Stack underflow");
        return a[top--];       // read first, then move down
    }
    int peek() {
        if (top == -1) throw new RuntimeException("empty");
        return a[top];
    }
    boolean isEmpty() { return top == -1; }
}
```

Dry run

Operation	top before	Array	top after	Returned
push(10)	-1	[10, _, _]	0	-
push(20)	0	[10,20, _]	1	-
push(30)	1	[10,20,30]	2	-
pop()	2	[10,20,30]	1	30
peek()	1	[10,20,30]	1	20
pop()	1	[10,20,30]	0	20

Note that `pop` does not erase the old value. It only moves `top`. The value at index 2 is still 30, but it is unreachable and will be overwritten by the next push. This is normal and correct.

> **++TOP VERSUS TOP++**
> `a[++top] = x` increments FIRST, so it writes into the new slot. Correct for push.
> `return a[top--]` reads FIRST, then decrements. Correct for pop.
> Get these backwards and your stack silently loses or duplicates elements. This is the single place where pre and post increment genuinely matter.

Stack using a linked list

No fixed capacity. Push and pop both happen at the head, which is the O(1) end.

```
class LinkedStack {
    private Node top;

    void push(int x) {
        Node n = new Node(x);
        n.next = top;
        top = n;
    }
    int pop() {
        if (top == null) throw new RuntimeException("empty");
        int v = top.data;
        top = top.next;
        return v;
    }
}
```

Notice this is exactly `addFirst` and `removeFirst` from the linked list. A stack **is** a linked list with the other operations hidden. Hiding them is the point: a restricted interface is easier to reason about and impossible to misuse.

What stacks are actually used for

Use	How the stack helps
Method calls	The JVM call stack IS a stack. That is where StackOverflowError comes from.
Undo / Ctrl-Z	Push each action; pop to undo
Browser back button	Push each page visited
Bracket matching	Push openers, pop and check on closers
Expression evaluation	Infix to postfix, then evaluate
DFS on a graph	The explicit stack version of recursion
Backtracking	Push a choice, pop to unchoose

Balanced brackets - the classic stack problem

```
static boolean isBalanced(String s) {
    Deque&lt;Character&gt; st = new ArrayDeque&lt;&gt;();
    for (char c : s.toCharArray()) {
        if (c == '(' || c == '[' || c == '{') st.push(c);
        else if (c == ')' || c == ']' || c == '}') {
            if (st.isEmpty()) return false;
            char open = st.pop();
            if (c == ')' &amp;&amp; open != '(') return false;
            if (c == ']' &amp;&amp; open != '[') return false;
            if (c == '}' &amp;&amp; open != '{') return false;
        }
    }
    return st.isEmpty();      // leftovers mean unclosed brackets
}
```

Dry run on `{[()]}`

Char	Action	Stack after
{	push	{
[	push	{ [
(	push	{ [(
)	pop gives (, matches	{ [
]	pop gives [, matches	{
}	pop gives { , matches	empty
end	stack empty	return true

The final `st.isEmpty()` check is essential. Without it, `"((("` returns true. The two failure modes are: a closer with nothing to match, and openers left over at the end. You must check both.

Postfix evaluation

Postfix (also called Reverse Polish) writes the operator after its operands: `2 3 +` means 2 + 3. No brackets are ever needed. Machines love it.

```
static int evalPostfix(String[] tokens) {
    Deque&lt;Integer&gt; st = new ArrayDeque&lt;&gt;();
    for (String t : tokens) {
        if (t.matches("-?\\d+")) st.push(Integer.parseInt(t));
        else {
            int b = st.pop(), a = st.pop();      // NOTE the order!
            switch (t) {
                case "+" -&gt; st.push(a + b);
                case "-" -&gt; st.push(a - b);
                case "*" -&gt; st.push(a * b);
                case "/" -&gt; st.push(a / b);
            }
        }
    }
    return st.pop();
}
```

Dry run on `2 3 4 * +`

Token	Action	Stack after
2	push	2
3	push	2 3
4	push	2 3 4
*	pop 4, pop 3, push 12	2 12
+	pop 12, pop 2, push 14	14

The order matters: the **second** value popped is the left operand. Get it backwards and subtraction and division give wrong answers while addition still looks fine, which makes it a nasty bug to spot.

CHAPTER 3

Queue

The picture

A **line at a ticket counter**. You join at the back. You are served from the front. First In, First Out: **FIFO**.

Operation	Common names	Meaning	Cost
Add at back	`enqueue`, `offer`, `add`	Join the line	O(1)
Remove from front	`dequeue`, `poll`, `remove`	Serve the first person	O(1)
Look at front	`peek`, `element`	Who is next?	O(1)

The naive array queue, and why it is broken

```
class BadQueue {
    int[] a = new int[5];
    int front = 0, rear = -1, size = 0;
    void enqueue(int x) { a[++rear] = x; size++; }
    int dequeue() { size--; return a[front++]; }
}
```

Enqueue 5 items, dequeue 5, and now `front` and `rear` are both at the end. The array is completely empty but `enqueue` throws "full". You have wasted the whole array. This is called **false overflow**.

Circular queue - the fix

Let the indexes **wrap around** using `%`. The array becomes a ring.

```
class CircularQueue {
    private int[] a;
    private int front = 0, rear = -1, size = 0;

    CircularQueue(int cap) { a = new int[cap]; }

    void enqueue(int x) {
        if (size == a.length) throw new RuntimeException("full");
        rear = (rear + 1) % a.length;      // wrap
        a[rear] = x;
        size++;
    }
    int dequeue() {
        if (size == 0) throw new RuntimeException("empty");
        int v = a[front];
        front = (front + 1) % a.length;    // wrap
        size--;
        return v;
    }
    int peek() { return a[front]; }
    boolean isEmpty() { return size == 0; }
    boolean isFull() { return size == a.length; }
}
```

Dry run with capacity 4

Operation	front	rear	size	Array
enqueue(1)	0	0	1	[1,_,_,_]
enqueue(2)	0	1	2	[1,2,_,_]
enqueue(3)	0	2	3	[1,2,3,_]
dequeue -> 1	1	2	2	[1,2,3,_]
dequeue -> 2	2	2	1	[1,2,3,_]

Operation	front	rear	size	Array
enqueue(4)	2	3	2	[1,2,3,4]
enqueue(5)	2	0	3	[5,2,3,4]

Look at the last row. `rear` wrapped from 3 back to 0 and reused the slot that used to hold 1. No memory wasted.

> **WHY KEEP A SEPARATE SIZE COUNTER**
> Without it, `front == rear + 1` is ambiguous: it could mean full or empty. Options are to keep a `size` counter, or to deliberately waste one slot. The counter is clearer, so use it.

Queue using a linked list

```
class LinkedQueue {
    private Node front, rear;

    void enqueue(int x) {
        Node n = new Node(x);
        if (rear == null) { front = rear = n; return; }
        rear.next = n;
        rear = n;
    }
    int dequeue() {
        if (front == null) throw new RuntimeException("empty");
        int v = front.data;
        front = front.next;
        if (front == null) rear = null;      // list became empty, reset rear
        return v;
    }
}
```

That `if (front == null) rear = null;` line is easy to forget and causes a subtle bug: `rear` keeps pointing at a deleted node, and the next enqueue attaches to a node nobody can reach.

Deque - the double ended queue

Add and remove from **both** ends. It can behave as a stack or as a queue, so in modern Java `ArrayDeque` replaces both `Stack` and `LinkedList` for these jobs.

What queues are used for

Use	Why FIFO fits
Printer job queue	First submitted, first printed
BFS on a graph	Explore all neighbours before going deeper
Task scheduling	Fairness, no starvation
Producer-consumer buffers	Decouple fast producer from slow consumer
Call centre lines	Whoever called first is served first

CHAPTER 1

Hash Table

The picture

A **library with numbered shelves and a rule for which shelf each book goes on**. The rule turns the book's title into a shelf number. To find a book you apply the same rule and walk straight to that shelf. You never search the whole library.

That rule is the **hash function**. It is the entire idea.

From array to hash table, step by step

An array gives O(1) access **by index**. A hash table gives O(1) access **by any key**, by turning the key into an index.

```
int index = Math.abs(key.hashCode()) % capacity;
```

Key	hashCode()	% 8	Bucket
"cat"	98262	6	6
"dog"	99644	4	4
"cow"	98644	4	4 <- collision!

Collisions and how to survive them

Two different keys can land in the same bucket. This is not a bug and it is not avoidable: there are infinitely many possible keys and finitely many buckets. What matters is how you handle it.

Strategy	How it works	Used by
Separate chaining	Each bucket holds a small list of entries	Java's HashMap
Open addressing	On collision, probe the next free slot	Python dicts, some C++ maps

Java uses separate chaining. Since Java 8, when a bucket's chain grows past 8 entries it converts to a **balanced tree**, so a worst-case bucket degrades to O(log n) rather than O(n). That change was made specifically to defend against hash-collision denial-of-service attacks.

A hash map, built by hand

```
class MyHashMap {
    static class Entry {
        String key; int value; Entry next;
        Entry(String k, int v) { key = k; value = v; }
    }

    private Entry[] buckets = new Entry[16];
    private int size = 0;

    private int index(String key) {
        return Math.abs(key.hashCode()) % buckets.length;
    }

    void put(String key, int value) {
        int i = index(key);
        for (Entry e = buckets[i]; e != null; e = e.next)
            if (e.key.equals(key)) { e.value = value; return; }   // update
        Entry n = new Entry(key, value);      // insert at head of chain
        n.next = buckets[i];
        buckets[i] = n;
        size++;
        if (size &gt; buckets.length * 0.75) resize();
    }

    Integer get(String key) {
        for (Entry e = buckets[index(key)]; e != null; e = e.next)
            if (e.key.equals(key)) return e.value;
        return null;
    }

    void remove(String key) {
        int i = index(key);
        Entry prev = null;
        for (Entry e = buckets[i]; e != null; prev = e, e = e.next) {
            if (e.key.equals(key)) {
                if (prev == null) buckets[i] = e.next;
                else prev.next = e.next;
                size--;
                return;
            }
        }
    }

    private void resize() {
        Entry[] old = buckets;
        buckets = new Entry[old.length * 2];
        size = 0;
        for (Entry head : old)
            for (Entry e = head; e != null; e = e.next) put(e.key, e.value);
    }
}
```

Load factor - why 0.75

Load factor = number of entries divided by number of buckets. It measures crowding.

Load factor	Effect
Low, e.g. 0.25	Few collisions, fast, but lots of empty buckets wasting memory
0.75	Java's default. The measured sweet spot.
High, e.g. 2.0	Long chains. Lookups drift towards O(n).

When the load factor is exceeded, the table **doubles** and every entry is rehashed into the new table. That single resize costs O(n), but it happens rarely enough that the average cost per insert stays O(1). This is amortised analysis in action.

> **WHY CAPACITY IS ALWAYS A POWER OF 2**
> Java replaces the slow `%` with the fast bitwise `hash & (capacity - 1)`. That trick only produces a correct spread when capacity is a power of 2. If you ask for `new HashMap<>(20)`, Java quietly gives you 32.

The cost table

Operation	Average	Worst case	Worst case cause
`put`	O(1)	O(log n) since Java 8	Everything collides into one bucket
`get`	O(1)	O(log n)	Same
`remove`	O(1)	O(log n)	Same
Iterate all	O(n + capacity)	Same	Must visit empty buckets too

THE REQUIREMENT NOBODY TELLS BEGINNERS

If you use your own class as a HashMap key, you MUST override both `equals` and `hashCode`, and the fields they use must be `final` or never changed.

Change a key's field after inserting it and the entry becomes permanently unreachable: it sits in the old bucket while lookups go to the new one. Nothing crashes. The data just disappears.

CHAPTER 2

Trees

The picture

A **family tree**, drawn upside down. One person at the top, children below, and no loops - you cannot be your own grandfather.

Term	Meaning
Node	One item in the tree
Root	The top node, the only one with no parent
Parent / Child	Direct connection, one level apart
Leaf	A node with no children
Edge	The link between a parent and a child
Height	Longest path from a node down to a leaf
Depth / Level	Distance from the root down to a node
Subtree	Any node plus everything below it

Binary tree

Each node has at most **two** children, called left and right.

```
class TreeNode {
    int data;
    TreeNode left, right;
    TreeNode(int d) { data = d; }
}
```

Type	Rule
Full	Every node has 0 or 2 children, never 1
Complete	All levels filled except possibly the last, which fills left to right
Perfect	All leaves at the same level, every internal node has 2 children
Balanced	Left and right heights differ by at most 1 at every node
Degenerate	Every node has one child. It is a linked list in disguise.

A perfect tree of height h holds 2^(h+1) - 1 nodes. That is why a balanced tree of a million nodes has height about 20: doubling capacity at each level means depth grows logarithmically. **That is the whole reason trees are fast.**

Binary Search Tree

Add one rule to a binary tree and it becomes searchable:

> **THE BST RULE**
> For every node: everything in the LEFT subtree is smaller, everything in the RIGHT subtree is larger.
> This must hold for every node, not just the root. That "every node" part is what people get wrong when validating a BST.

```
        50
       /  \
     30    70
    /  \   /  \
  20   40 60   80
```

Searching for 40: start at 50, 40 is smaller so go left to 30, 40 is larger so go right, found. Three comparisons instead of seven. Every comparison discards half the remaining tree.

BST operations

```
class BST {
    TreeNode root;

    // INSERT - recursive, returns the (possibly new) subtree root
    TreeNode insert(TreeNode node, int val) {
        if (node == null) return new TreeNode(val);       // found the spot
        if (val &lt; node.data)      node.left  = insert(node.left, val);
        else if (val &gt; node.data) node.right = insert(node.right, val);
        return node;                                      // duplicates ignored
    }

    // SEARCH
    boolean contains(TreeNode node, int val) {
        if (node == null) return false;
        if (node.data == val) return true;
        return val &lt; node.data ? contains(node.left, val)
                                  : contains(node.right, val);
    }

    // MINIMUM: walk left until you cannot
    TreeNode min(TreeNode node) {
        while (node.left != null) node = node.left;
        return node;
    }

    // DELETE - the hard one
    TreeNode delete(TreeNode node, int val) {
        if (node == null) return null;
        if (val &lt; node.data)      node.left  = delete(node.left, val);
        else if (val &gt; node.data) node.right = delete(node.right, val);
        else {
            // found it. three cases:
            if (node.left == null)  return node.right;   // 0 or 1 child
            if (node.right == null) return node.left;    // 1 child
            TreeNode succ = min(node.right);             // 2 children
            node.data = succ.data;                       // copy successor up
            node.right = delete(node.right, succ.data);  // delete successor
        }
        return node;
    }
}
```

Why deletion uses the in-order successor

If a node has two children, you cannot simply remove it - who takes its place? The replacement must be **larger than everything on the left and smaller than everything on the right**. Exactly two nodes qualify: the largest on the left (predecessor) or the smallest on the right (successor). Either works. Taking the successor is the convention.

Tree traversals - the four ways to visit every node

```
void inorder(TreeNode n)   { if (n==null) return; inorder(n.left);  visit(n); inorder(n.right); }
void preorder(TreeNode n)  { if (n==null) return; visit(n); preorder(n.left); preorder(n.right); }
void postorder(TreeNode n) { if (n==null) return; postorder(n.left); postorder(n.right); visit(n); }
```

The names describe **when the node itself is visited** relative to its children. Pre = before. In = between. Post = after. That is all you need to remember.

Traversal	Order on the example tree	Use
Inorder	20 30 40 50 60 70 80	**Sorted output** from a BST. The killer feature.
Preorder	50 30 20 40 70 60 80	Copying a tree, saving its structure
Postorder	20 40 30 60 80 70 50	Deleting a tree, evaluating expression trees
Level order	50 30 70 20 40 60 80	Printing by level, shortest path, BFS

INORDER ON A BST GIVES SORTED ORDER, ALWAYS
This is the single most useful fact about BSTs. It also gives you a free BST-validation method: do an inorder walk and check the values only ever increase.

Level order traversal needs a queue, not recursion

```
void levelOrder(TreeNode root) {
    if (root == null) return;
    Queue&lt;TreeNode&gt; q = new LinkedList&lt;&gt;();
    q.add(root);
    while (!q.isEmpty()) {
        TreeNode n = q.poll();
        System.out.print(n.data + " ");
        if (n.left != null)  q.add(n.left);
        if (n.right != null) q.add(n.right);
    }
}
```

Step	Polled	Printed	Queue after
1	50	50	30, 70
2	30	30	70, 20, 40
3	70	70	20, 40, 60, 80
4	20	20	40, 60, 80

Recursion naturally goes **deep** first, which is why it gives you the three depth-first traversals for free. Breadth requires an explicit queue. This is exactly the DFS versus BFS distinction you will meet again with graphs.

Useful recursive tree methods

```
int height(TreeNode n) {
    if (n == null) return -1;                  // -1 so a leaf has height 0
    return 1 + Math.max(height(n.left), height(n.right));
}
int count(TreeNode n) {
    return n == null ? 0 : 1 + count(n.left) + count(n.right);
}
int sum(TreeNode n) {
    return n == null ? 0 : n.data + sum(n.left) + sum(n.right);
}
boolean isBalanced(TreeNode n) {
    if (n == null) return true;
    return Math.abs(height(n.left) - height(n.right)) &lt;= 1
        &amp;&amp; isBalanced(n.left) &amp;&amp; isBalanced(n.right);
}
TreeNode invert(TreeNode n) {
    if (n == null) return null;
    TreeNode t = n.left; n.left = invert(n.right); n.right = invert(t);
    return n;
}
```

Notice the shape repeating: **base case for null, then combine the results of the two children.** Once you see that shape, most tree questions are the same question with a different combining step.

Why balancing matters

Insert 1, 2, 3, 4, 5 into a plain BST in that order:

```
1
 \
  2
   \
    3
     \
      4
```

Every node has only a right child. This is a linked list. Search is O(n), not O(log n). All the benefit is gone.

Self-balancing trees (AVL, Red-Black) automatically rotate nodes to keep the height near log n. Java's `TreeMap` and `TreeSet` are Red-Black trees, which is why they guarantee O(log n) rather than merely hoping for it.

CHAPTER 3

Heap And Priority Queue

The picture

A **hospital emergency room**. Not first come first served. The most urgent case goes first, always. A heap keeps the most urgent item instantly available without keeping everything fully sorted.

The heap property

- **Min-heap**: every parent is smaller than or equal to both children. The smallest value is at the root.
- **Max-heap**: every parent is larger than or equal to both children. The largest is at the root.

That is the only rule. Siblings have **no** required order. A heap is *partially* ordered, and that is exactly why it is cheaper to maintain than a fully sorted structure.

A heap is stored in a plain array

A heap is always a **complete** binary tree, which means it can be packed into an array with no gaps and no pointers at all.

For index i	Formula
Left child	`2*i + 1`
Right child	`2*i + 2`
Parent	`(i - 1) / 2`

```
Tree:        10
            /  \
          20    15
         /  \
       30    40

Array:  [10, 20, 15, 30, 40]
Index:    0   1   2   3   4
```

Check it: index 1 holds 20, its children should be at 3 and 4, which hold 30 and 40. Correct. Zero pointers, perfect cache locality, and this is why heaps are fast in practice as well as in theory.

Building a heap by hand

```
class MinHeap {
    private int[] a;
    private int size = 0;

    MinHeap(int cap) { a = new int[cap]; }

    void insert(int val) {
        a[size] = val;              // put at the end
        siftUp(size);               // bubble it up to its place
        size++;
    }

    private void siftUp(int i) {
        while (i &gt; 0) {
            int parent = (i - 1) / 2;
            if (a[i] &gt;= a[parent]) break;      // heap property satisfied
            swap(i, parent);
            i = parent;
        }
    }

    int extractMin() {
        if (size == 0) throw new RuntimeException("empty");
        int min = a[0];
        a[0] = a[--size];           // move the last item to the root
        siftDown(0);                // sink it to its place
        return min;
    }

    private void siftDown(int i) {
        while (true) {
            int l = 2*i + 1, r = 2*i + 2, smallest = i;
            if (l &lt; size &amp;&amp; a[l] &lt; a[smallest]) smallest = l;
            if (r &lt; size &amp;&amp; a[r] &lt; a[smallest]) smallest = r;
            if (smallest == i) break;
            swap(i, smallest);
            i = smallest;
        }
    }

    private void swap(int i, int j) { int t=a[i]; a[i]=a[j]; a[j]=t; }
}
```

Dry run: inserting 5 into [10, 20, 15, 30, 40]

Step	Array	i	parent	a[i] < a[parent]?	Action
1	[10,20,15,30,40,5]	5	2	5 < 15 yes	swap
2	[10,20,5,30,40,15]	2	0	5 < 10 yes	swap
3	[5,20,10,30,40,15]	0	-	i = 0	stop

Three swaps for six elements. The path from any leaf to the root is at most log n long, so insert is O(log n).

Dry run: extractMin from [5, 20, 10, 30, 40, 15]

Step	Array	Action
1	[5,20,10,30,40,15]	save min = 5
2	[15,20,10,30,40]	move last (15) to root, size 5
3	children of 0 are 20 and 10, smallest is 10 at index 2	swap
4	[10,20,15,30,40]	index 2 has no children, stop

Cost table

Operation	Cost	Why
`peek` min or max	O(1)	It is always at index 0
`insert`	O(log n)	At most one path up the tree

Operation	Cost	Why
`extract`	O(log n)	At most one path down
Build from an array	O(n)	Not O(n log n). Most nodes are near the bottom and sift down very little.
Search for an arbitrary value	O(n)	A heap is not ordered for searching

That last row matters: **a heap is not a search structure**. If you need "does it contain x", a heap is the wrong tool.

PriorityQueue - Java's ready-made heap

```
// Min-heap by default
PriorityQueue&lt;Integer&gt; min = new PriorityQueue&lt;&gt;();
min.add(30); min.add(10); min.add(20);
System.out.println(min.poll());   // 10

// Max-heap: reverse the comparator
PriorityQueue&lt;Integer&gt; max = new PriorityQueue&lt;&gt;(Comparator.reverseOrder());

// Custom order
PriorityQueue&lt;Student&gt; pq =
    new PriorityQueue&lt;&gt;((a, b) -&gt; b.marks - a.marks);
```

PRIORITYQUEUE GOTCHA

Iterating a PriorityQueue, or printing it, does NOT give sorted order. It shows you the raw array. Only repeated `poll()` gives you the sorted sequence.

The problem heaps are made for: top K

Find the 5 largest numbers in a stream of 10 million.

Approach	Cost	Memory
Sort everything, take last 5	O(n log n)	O(n), must hold all 10 million
Min-heap of size 5	O(n log k)	O(k) = 5 elements

```
PriorityQueue&lt;Integer&gt; heap = new PriorityQueue&lt;&gt;();   // MIN heap for top-K
for (int x : stream) {
    heap.add(x);
    if (heap.size() &gt; k) heap.poll();    // throw away the smallest
}
// the heap now holds the k largest
```

The counter-intuitive part: to find the **largest** K you use a **min**-heap. The root is the weakest survivor, so it is the cheapest one to evict. Once you see why, the reverse case (smallest K needs a max-heap) is obvious.

CHAPTER 4

Graphs

The picture

A **map of cities and roads**. Cities are vertices. Roads are edges. Unlike a tree, cycles are allowed and there is no root.

Term	Meaning
Vertex / Node	One point, e.g. a city
Edge	A connection between two vertices
Directed	One way street. A to B does not imply B to A.
Undirected	Two way. A to B implies B to A.
Weighted	Each edge has a cost: distance, time, price
Degree	Number of edges touching a vertex
Path	A sequence of connected vertices
Cycle	A path that returns to where it started
Connected	Every vertex is reachable from every other

Two ways to store a graph

Adjacency matrix

A 2D array where `m[i][j] = 1` means an edge exists from i to j.

```
int[][] m = new int[n][n];
m[0][1] = 1;   // edge 0 -&gt; 1
m[1][0] = 1;   // and back, for an undirected graph
```

Adjacency list

For each vertex, a list of its neighbours. This is what you should use almost always.

```
List&lt;List&lt;Integer&gt;&gt; g = new ArrayList&lt;&gt;();
for (int i = 0; i &lt; n; i++) g.add(new ArrayList&lt;&gt;());
void addEdge(int u, int v) {
    g.get(u).add(v);
    g.get(v).add(u);      // omit this line for a directed graph
}
```

	Adjacency matrix	Adjacency list
Space	O(V^2)	O(V + E)
Is there an edge u-v?	O(1)	O(degree of u)
List all neighbours of u	O(V)	O(degree of u)
Best for	Dense graphs, few vertices	Sparse graphs, which is almost all real graphs

A social network with 1 billion users and 300 friends each: a matrix needs 10^18 cells, a list needs 3 x 10^11. That is the whole argument.

BFS - breadth first search

Explore **level by level**, like ripples on a pond. Uses a **queue**.

```
void bfs(int start, List&lt;List&lt;Integer&gt;&gt; g, int n) {
    boolean[] visited = new boolean[n];
    Queue&lt;Integer&gt; q = new LinkedList&lt;&gt;();
    q.add(start);
    visited[start] = true;

    while (!q.isEmpty()) {
        int u = q.poll();
        System.out.print(u + " ");
        for (int v : g.get(u)) {
            if (!visited[v]) {
                visited[v] = true;     // mark when ADDING, not when polling
                q.add(v);
            }
        }
    }
}
```

THE ONE BFS BUG EVERYONE WRITES

Mark `visited` at the moment you ADD to the queue, not when you poll it.
If you mark on poll, a vertex with three neighbours pointing at it gets added three times before any of them is processed, and it is printed three times.

Dry run on 0-1, 0-2, 1-3, 2-3

Step	Polled	Neighbours	Newly added	Queue after	Visited
1	0	1, 2	1, 2	1, 2	0,1,2
2	1	0, 3	3	2, 3	0,1,2,3
3	2	0, 3	none	3	-
4	3	1, 2	none	empty	-

Output: 0 1 2 3.

BFS finds the shortest path in an unweighted graph. That is its defining property, and it follows directly from exploring in order of distance.

DFS - depth first search

Go as deep as possible, then back up. Uses **recursion** (which is really the call stack) or an explicit stack.

```
void dfs(int u, List&lt;List&lt;Integer&gt;&gt; g, boolean[] visited) {
    visited[u] = true;
    System.out.print(u + " ");
    for (int v : g.get(u))
        if (!visited[v]) dfs(v, g, visited);
}
```

	BFS	DFS
Data structure	Queue	Stack or recursion
Explores	Level by level	One branch to the end
Shortest path (unweighted)	Yes	No
Memory	O(width of the graph)	O(depth of the graph)
Good for	Shortest path, nearest neighbours	Cycle detection, topological sort, connected components, maze solving

Cycle detection

```
// Undirected: a cycle exists if we reach a visited node
// that is not the one we came from
boolean hasCycle(int u, int parent, List&lt;List&lt;Integer&gt;&gt; g, boolean[] vis) {
    vis[u] = true;
    for (int v : g.get(u)) {
        if (!vis[v]) { if (hasCycle(v, u, g, vis)) return true; }
        else if (v != parent) return true;
    }
    return false;
}
```

The `v != parent` check exists because in an undirected graph, the edge you just walked along would otherwise look like a cycle of length 2. For a **directed** graph the technique is different: you track nodes currently on the recursion stack, because only a back edge to an in-progress node is a real cycle.

Dijkstra's shortest path, in brief

For **weighted** graphs, BFS is not enough because a path with more edges can be cheaper. Dijkstra always expands the cheapest unvisited vertex next, using a priority queue.

```
int[] dist = new int[n];
Arrays.fill(dist, Integer.MAX_VALUE);
dist[src] = 0;
PriorityQueue&lt;int[]&gt; pq = new PriorityQueue&lt;&gt;((a,b) -&gt; a[1] - b[1]);
pq.add(new int[]{src, 0});

while (!pq.isEmpty()) {
    int[] cur = pq.poll();
    int u = cur[0];
    if (cur[1] &gt; dist[u]) continue;              // stale entry, skip
    for (int[] edge : graph.get(u)) {            // edge = {neighbour, weight}
        int v = edge[0], w = edge[1];
        if (dist[u] + w &lt; dist[v]) {             // found a cheaper route
            dist[v] = dist[u] + w;
            pq.add(new int[]{v, dist[v]});
        }
    }
}
```

O((V + E) log V). It requires all weights to be non-negative; with negative edges you need Bellman-Ford instead, because Dijkstra's "once finalised, always finalised" assumption breaks.

CHAPTER 5

Trie

The picture

A **dictionary where the words share their beginnings**. "car", "card" and "care" all walk down the same c-a-r path and only then split. That shared path is the whole idea.

```
        root
         |
         c
         |
         a
         |
         r*        &lt;- * marks the end of the word "car"
        /  \
       d*   e*     &lt;- "card" and "care"
```

```
class TrieNode {
    TrieNode[] children = new TrieNode[26];
    boolean isEnd;
}

class Trie {
    private TrieNode root = new TrieNode();

    void insert(String word) {
        TrieNode cur = root;
        for (char c : word.toCharArray()) {
            int i = c - 'a';
            if (cur.children[i] == null) cur.children[i] = new TrieNode();
            cur = cur.children[i];
        }
        cur.isEnd = true;
    }

    boolean search(String word) {
        TrieNode n = find(word);
        return n != null &amp;&amp; n.isEnd;
    }

    boolean startsWith(String prefix) {
        return find(prefix) != null;
    }

    private TrieNode find(String s) {
        TrieNode cur = root;
        for (char c : s.toCharArray()) {
            cur = cur.children[c - 'a'];
            if (cur == null) return null;
        }
        return cur;
    }
}
```

Operation	Cost	Note
insert	O(L), L = word length	Independent of how many words are stored
search	O(L)	A HashSet is also O(L) because hashing reads the whole string
startsWith	O(L)	**A HashSet cannot do this at all**

WHEN TO USE A TRIE INSTEAD OF A HASHSET

Only when you need PREFIX queries: autocomplete, spell check, IP routing tables, word games.
For plain "is this word present", a HashSet is simpler and uses far less memory. A Trie allocates 26 pointers per node whether or not they are used.

The decision table: which structure do I use?

I need...	Use	Why
Fast access by position	Array / ArrayList	O(1) index
Fast lookup by key	HashMap	O(1) average
Sorted keys, or range queries	TreeMap	O(log n), keeps order
No duplicates	HashSet	O(1) contains
No duplicates, sorted	TreeSet	O(log n), keeps order
Last in first out	ArrayDeque as a stack	O(1) both ends
First in first out	ArrayDeque as a queue	O(1) both ends
Always the most urgent item	PriorityQueue	O(log n) insert, O(1) peek
Many inserts and deletes at the ends	ArrayDeque or LinkedList	O(1)
Prefix search	Trie	Nothing else does prefixes
Relationships and connections	Graph	Models the real problem directly
Hierarchy	Tree	Models the real problem directly

PART 7 - THE JAVA COLLECTIONS FRAMEWORK

CHAPTER 1

Collections: The Ready-Made Structures

Why this exists

Everything you built by hand in Part 6 already ships with Java, written by experts, tested by millions of programs, and faster than what you or I would write. You built them to understand them. You use these to get work done.

The family tree

```
                Iterable
                    |
               Collection                    Map  (NOT a Collection)
             /      |      \                  |
         List      Set     Queue          HashMap
          |         |        |             LinkedHashMap
      ArrayList   HashSet  ArrayDeque      TreeMap
      LinkedList  LinkedHashSet  PriorityQueue   Hashtable
      Vector      TreeSet
```

MAP IS NOT A COLLECTION

`Map` sits outside the `Collection` hierarchy because it stores PAIRS, not single elements. It has no `add(E)` and it cannot be iterated directly. You iterate `map.keySet()`, `map.values()` or `map.entrySet()`.

The interfaces, one line each

Interface	Rule	Duplicates?	Ordered?
`List`	Positional, indexed	Yes	Yes, insertion order
`Set`	A mathematical set	No	Depends on the implementation
`Queue`	Add at one end, remove at the other	Yes	FIFO usually
`Deque`	Both ends usable	Yes	Yes
`Map`	Key to value	Keys no, values yes	Depends

CHAPTER 2

List: ArrayList And LinkedList

ArrayList - a growable array

Internally it holds a plain `Object[]`. When it fills up, it makes a new array 1.5 times larger and copies everything over.

```
List&lt;String&gt; list = new ArrayList&lt;&gt;();
list.add("a");                  // append
list.add(0, "z");               // insert at index 0, shifts everything right
list.get(0);                    // "z"
list.set(0, "y");               // replace
list.remove(0);                 // remove by index
list.remove("a");               // remove by object
list.size();
list.contains("a");
list.indexOf("a");
list.isEmpty();
list.clear();
list.addAll(otherList);
list.sort(Comparator.naturalOrder());
```

Operation	Cost	Why
`get(i)`, `set(i, v)`	O(1)	Direct array index
`add(v)` at the end	O(1) amortised	Usually free, occasionally a resize copy
`add(i, v)` in the middle	O(n)	Everything after i shifts right
`remove(i)`	O(n)	Everything after i shifts left
`contains(v)`	O(n)	Linear scan

REMOVE(INT) VERSUS REMOVE(OBJECT) - A REAL TRAP

`List<Integer> l; l.remove(1);` removes the element at INDEX 1.
`l.remove(Integer.valueOf(1));` removes the VALUE 1.
Java picks `remove(int)` because the argument is an `int`. This silently deletes the wrong element and it is one of the most common bugs in real Java code.

The growth mechanism

Adds so far	Internal capacity	What happened
0	0	Empty array, lazily allocated
1	10	First add allocates the default
11	15	Grew by 1.5x, copied 10 elements
16	22	Grew again

If you know the final size, say so: `new ArrayList<>(10000)`. This avoids every resize copy and is a genuine, measurable optimisation for large lists.

LinkedList - a doubly linked list

Operation	ArrayList	LinkedList
`get(i)`	O(1)	O(n), must walk
`add` at end	O(1) amortised	O(1)
`add` at front	O(n)	O(1)
`remove` from front	O(n)	O(1)
`remove` in the middle by index	O(n) shifting	O(n) walking
Memory per element	Just the reference	Reference plus 2 pointers, roughly 3x

Operation	ArrayList	LinkedList
Cache performance	Excellent, contiguous	Poor, scattered

THE HONEST RECOMMENDATION

Use `ArrayList`. Almost always.

The theoretical advantage of `LinkedList` at the front is real, but in practice CPU cache behaviour usually makes `ArrayList` win anyway, and when you truly need fast operations at both ends the right answer is `ArrayDeque`, not `LinkedList`.

Java's own documentation effectively says this. `LinkedList` is a structure you should understand and rarely choose.

Iterating a list, and the removal trap

```
// WRONG - throws ConcurrentModificationException
for (String s : list) {
    if (s.equals("x")) list.remove(s);
}

// RIGHT - use the iterator's own remove
Iterator&lt;String&gt; it = list.iterator();
while (it.hasNext()) {
    if (it.next().equals("x")) it.remove();
}

// ALSO RIGHT - Java 8+, cleanest
list.removeIf(s -&gt; s.equals("x"));
```

The for-each loop hides an `Iterator`. That iterator keeps a modification counter. When the list changes behind its back, the counter no longer matches and the iterator refuses to continue - **fail fast**, on purpose. Silently continuing with a corrupt position would be far worse.

Immutable and fixed-size lists

```
List&lt;String&gt; a = Arrays.asList("x", "y");      // FIXED SIZE, but set() works
// a.add("z");    // UnsupportedOperationException
a.set(0, "q");    // allowed

List&lt;String&gt; b = List.of("x", "y");            // FULLY IMMUTABLE (Java 9+)
// b.set(0, "q"); // UnsupportedOperationException

List&lt;String&gt; c = new ArrayList&lt;&gt;(List.of("x", "y"));   // real, modifiable copy
```

CHAPTER 3

Set: No Duplicates

The three implementations

	`HashSet`	`LinkedHashSet`	`TreeSet`
Order	None, unpredictable	Insertion order	Sorted
`add`/`contains`/`remove`	O(1)	O(1)	O(log n)
Backed by	HashMap	HashMap + linked list	Red-Black tree
Allows null	One	One	No, throws NPE
Use when	You just need uniqueness	Uniqueness plus predictable order	You need sorted or range queries

```
Set&lt;String&gt; s = new HashSet&lt;&gt;();
System.out.println(s.add("a"));   // true, it was new
System.out.println(s.add("a"));   // false, already present
s.contains("a");
s.remove("a");
```

That boolean return from `add` is genuinely useful: it detects duplicates in one operation, with no separate `contains` call.

Set operations

```
Set&lt;Integer&gt; a = new HashSet&lt;&gt;(List.of(1,2,3));
Set&lt;Integer&gt; b = new HashSet&lt;&gt;(List.of(2,3,4));

Set&lt;Integer&gt; union = new HashSet&lt;&gt;(a);
union.addAll(b);            // [1,2,3,4]

Set&lt;Integer&gt; inter = new HashSet&lt;&gt;(a);
inter.retainAll(b);         // [2,3]

Set&lt;Integer&gt; diff = new HashSet&lt;&gt;(a);
diff.removeAll(b);          // [1]
```

TreeSet extras

Because it is sorted, `TreeSet` can answer questions a `HashSet` cannot.

Method	Returns
`first()` / `last()`	Smallest / largest
`floor(x)`	Largest element <= x
`ceiling(x)`	Smallest element >= x
`lower(x)` / `higher(x)`	Strictly less / strictly greater
`headSet(x)` / `tailSet(x)`	Everything below / above x
`subSet(a, b)`	Range view
`descendingSet()`	Reverse view

`ceiling` and `floor` are the "find the nearest available slot" operations that appear constantly in scheduling and interval problems.

CHAPTER 4

Map: Key To Value

The picture

A **phone book**. Look up a name, get a number. Names are unique, numbers can repeat.

The implementations

	`HashMap`	`LinkedHashMap`	`TreeMap`	`Hashtable`
Order	None	Insertion or access order	Sorted by key	None
get / put	O(1)	O(1)	O(log n)	O(1)
Null key	One allowed	One allowed	Not allowed	Not allowed
Thread safe	No	No	No	Yes, but slow
Use when	Default choice	LRU cache, ordered output	Sorted keys, ranges	Never. Legacy.

Core methods

```
Map&lt;String, Integer&gt; m = new HashMap&lt;&gt;();
m.put("a", 1);
m.put("a", 2);                    // overwrites, returns the old value 1
m.get("a");                       // 2
m.get("zzz");                     // null, does NOT throw
m.getOrDefault("zzz", 0);         // 0    &lt;- use this constantly
m.containsKey("a");
m.containsValue(2);
m.remove("a");
m.size();
m.keySet();                       // Set of keys
m.values();                       // Collection of values
m.entrySet();                     // Set of key-value pairs
```

The four modern methods that shorten real code

```
// 1. Counting frequencies - the classic
map.put(k, map.getOrDefault(k, 0) + 1);
map.merge(k, 1, Integer::sum);          // same thing, cleaner

// 2. Grouping into lists
map.computeIfAbsent(k, x -&gt; new ArrayList&lt;&gt;()).add(value);

// 3. Only set if absent
map.putIfAbsent(k, v);

// 4. Update only if present
map.computeIfPresent(k, (key, val) -&gt; val * 2);
```

Line 2 replaces this four-line dance, and is the single most useful Map idiom in Java:

```
if (!map.containsKey(k)) map.put(k, new ArrayList&lt;&gt;());
map.get(k).add(value);
```

Iterating a map

```
// Best: entrySet, one lookup per pair
for (Map.Entry&lt;String, Integer&gt; e : m.entrySet())
    System.out.println(e.getKey() + " = " + e.getValue());

// Keys only
for (String k : m.keySet()) { }

// Slower: keySet then get() does a SECOND hash lookup per pair
for (String k : m.keySet()) System.out.println(k + " = " + m.get(k));

// Java 8
m.forEach((k, v) -&gt; System.out.println(k + " = " + v));
```

Frequency counting - the pattern behind 20 interview questions

```
Map&lt;Character, Integer&gt; freq = new HashMap&lt;&gt;();
for (char c : s.toCharArray())
    freq.merge(c, 1, Integer::sum);

// find the most frequent
char best = ' '; int bestCount = 0;
for (Map.Entry&lt;Character,Integer&gt; e : freq.entrySet())
    if (e.getValue() &gt; bestCount) { bestCount = e.getValue(); best = e.getKey(); }
```

Recognise this shape. First non-repeating character, anagram check, majority element, top-K frequent, group anagrams, and ransom note are all this pattern with a different second step.

Sorting a map by value

A `HashMap` cannot be sorted in place. You copy the entries out, sort them, and rebuild.

```
List&lt;Map.Entry&lt;String,Integer&gt;&gt; list = new ArrayList&lt;&gt;(m.entrySet());
list.sort(Map.Entry.comparingByValue());            // ascending
list.sort(Map.Entry.&lt;String,Integer&gt;comparingByValue().reversed());

LinkedHashMap&lt;String,Integer&gt; sorted = new LinkedHashMap&lt;&gt;();
for (Map.Entry&lt;String,Integer&gt; e : list)
    sorted.put(e.getKey(), e.getValue());
```

The result must go into a `LinkedHashMap`, not a `HashMap`, because a `HashMap` would immediately discard the order you just built.

CHAPTER 5

Queue, Deque And PriorityQueue

Queue methods come in two flavours

Job	Throws an exception	Returns a special value
Insert	`add(e)`	`offer(e)` returns false
Remove	`remove()`	`poll()` returns null
Examine	`element()`	`peek()` returns null

Use `offer` / `poll` / `peek` by default. Returning `null` on an empty queue is usually easier to handle than catching an exception, and it makes loop conditions natural.

ArrayDeque - use this for both stacks and queues

```
Deque&lt;Integer&gt; d = new ArrayDeque&lt;&gt;();

// As a STACK
d.push(1); d.push(2);
d.pop();          // 2

// As a QUEUE
d.offer(1); d.offer(2);
d.poll();         // 1

// Both ends
d.addFirst(0); d.addLast(9);
d.peekFirst(); d.peekLast();
d.pollFirst(); d.pollLast();
```

DO NOT USE THE OLD STACK CLASS
`java.util.Stack` extends `Vector`, so every operation is synchronised and slow, and it exposes `get(i)` which breaks the whole point of a stack.
`ArrayDeque` is faster and has a cleaner interface. Use it. The only thing `ArrayDeque` refuses is `null` elements.

PriorityQueue

```
PriorityQueue&lt;Integer&gt; pq = new PriorityQueue&lt;&gt;();              // min-heap
PriorityQueue&lt;Integer&gt; mx = new PriorityQueue&lt;&gt;(Collections.reverseOrder());
PriorityQueue&lt;int[]&gt; byCost = new PriorityQueue&lt;&gt;((a,b) -&gt; a[1] - b[1]);

pq.offer(5); pq.offer(1); pq.offer(3);
pq.peek();     // 1, O(1)
pq.poll();     // 1, O(log n)
```

Operation	Cost
`offer`	O(log n)
`poll`	O(log n)
`peek`	O(1)
`contains`	O(n)
`remove(Object)`	O(n)

CHAPTER 6

Collections Utility And Streams

The Collections class

Method	Does
`Collections.sort(list)`	Sort ascending
`Collections.sort(list, cmp)`	Sort with a comparator
`Collections.reverse(list)`	Reverse in place
`Collections.shuffle(list)`	Random order
`Collections.max(list)` / `min(list)`	Largest / smallest
`Collections.frequency(list, x)`	Count of x
`Collections.swap(list, i, j)`	Swap two positions
`Collections.nCopies(n, x)`	Immutable list of n copies
`Collections.unmodifiableList(l)`	Read-only view
`Collections.emptyList()`	Immutable empty list
`Collections.binarySearch(list, k)`	O(log n), list must be sorted

Streams - the short version

Streams describe **what** you want rather than **how** to loop. They are worth knowing, but they are not always clearer and they are not always faster.

```
List&lt;String&gt; names = List.of("Asha", "Ravi", "Ben", "Anu");

List&lt;String&gt; result = names.stream()
    .filter(n -&gt; n.startsWith("A"))       // keep some
    .map(String::toUpperCase)              // transform each
    .sorted()                              // order them
    .toList();                             // collect back
// [ANU, ASHA]

int sum = IntStream.rangeClosed(1, 10).sum();          // 55
long count = names.stream().filter(n -&gt; n.length() &gt; 3).count();
String joined = String.join(", ", names);

Map&lt;Integer, List&lt;String&gt;&gt; byLength = names.stream()
    .collect(Collectors.groupingBy(String::length));
```

Operation type	Examples	Note
Intermediate, lazy	`filter`, `map`, `sorted`, `distinct`, `limit`	Nothing runs yet
Terminal, triggers work	`collect`, `forEach`, `count`, `sum`, `reduce`, `toList`	Runs the whole pipeline once

WHEN NOT TO USE STREAMS

A stream is harder to debug than a loop: you cannot easily put a breakpoint in the middle or print an intermediate value.

For a simple loop over an array, the plain `for` loop is clearer and faster. Use streams when the pipeline genuinely reads better than the loop - filtering, grouping and joining are the strong cases.

The complete cost cheat sheet

Structure	Access	Search	Insert	Delete	Extra space
Array	O(1)	O(n)	O(n)	O(n)	O(1)
Sorted array	O(1)	O(log n)	O(n)	O(n)	O(1)

Structure	Access	Search	Insert	Delete	Extra space
ArrayList	O(1)	O(n)	O(1) amortised at end	O(n)	O(n)
LinkedList	O(n)	O(n)	O(1) at ends	O(1) at ends	O(n)
Stack	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	O(n)	O(n)	O(1)	O(1)	O(n)
HashMap / HashSet	-	O(1) avg	O(1) avg	O(1) avg	O(n)
TreeMap / TreeSet	-	O(log n)	O(log n)	O(log n)	O(n)
Heap / PriorityQueue	O(1) peek	O(n)	O(log n)	O(log n)	O(n)
Balanced BST	O(log n)	O(log n)	O(log n)	O(log n)	O(n)
Trie	O(L)	O(L)	O(L)	O(L)	O(alphabet x nodes)
Graph (adjacency list)	-	O(V+E)	O(1)	O(E)	O(V+E)

PART 8 - 100 WORKED PROGRAMS

CHAPTER 1

Programs 1 to 25: Numbers And Logic

How to read this part

Every program follows the same layout: the problem, the code, and where it teaches something, a dry run table and a note on the trap. Where a brute force and a better version both exist, both are shown, because the *difference* between them is the lesson.

Do not read these. Solve them first, then read. A solution you have read is worth almost nothing; a solution you have attempted is worth a great deal, even when your attempt failed.

1. Check even or odd

```
static String evenOdd(int n) {
    return (n % 2 == 0) ? "Even" : "Odd";
}
```

For negative numbers `-7 % 2` is `-1`, not `1`, so testing `n % 2 == 1` fails for negatives. Testing `== 0` is always safe. The fastest version is `(n & 1) == 0`, which just checks the last bit.

2. Largest of three numbers

```
static int max3(int a, int b, int c) {
    return Math.max(a, Math.max(b, c));
}
```

The manual version, worth writing once: `int m = a; if (b > m) m = b; if (c > m) m = c;` This "keep the best so far" shape scales to n numbers, while nested if-else does not.

3. Sum of digits

```
static int digitSum(int n) {
    n = Math.abs(n);
    int sum = 0;
    while (n &gt; 0) { sum += n % 10; n /= 10; }
    return sum;
}
```

Round	n	n % 10	sum	n / 10
1	1234	4	4	123
2	123	3	7	12
3	12	2	9	1
4	1	1	10	0

4. Reverse a number

```
static int reverse(int n) {
    int rev = 0;
    while (n != 0) {
        rev = rev * 10 + n % 10;
        n /= 10;
    }
    return rev;
}
```

Round	n	n % 10	rev before	rev * 10 + digit	n after
1	1234	4	0	4	123
2	123	3	4	43	12
3	12	2	43	432	1
4	1	1	432	4321	0

`rev * 10` pushes the existing digits one place left to make room. This is the same idea as writing a number down from right to left.

5. Palindrome number

```
static boolean isPalin(int n) {
    if (n &lt; 0) return false;
    return n == reverse(n);
}
```

6. Count digits

```
static int countDigits(int n) {
    if (n == 0) return 1;                 // edge case that breaks the loop
    n = Math.abs(n);
    int c = 0;
    while (n &gt; 0) { c++; n /= 10; }
    return c;
}
```

The `n == 0` guard is essential: without it, zero returns 0 digits instead of 1. Every "repeatedly divide" loop has this same edge case.

7. Factorial, both ways

```
static long factIter(int n) {
    long f = 1;
    for (int i = 2; i &lt;= n; i++) f *= i;
    return f;
}
static long factRec(int n) {
    return (n &lt;= 1) ? 1 : n * factRec(n - 1);
}
```

`long` overflows at 21 factorial. Beyond that you need `BigInteger`.

8. Fibonacci series

```
static void fib(int n) {
    long a = 0, b = 1;
    for (int i = 0; i &lt; n; i++) {
        System.out.print(a + " ");
        long c = a + b; a = b; b = c;
    }
}
```

9. Prime check - and why the square root works

```
static boolean isPrime(int n) {
    if (n &lt; 2) return false;
    if (n % 2 == 0) return n == 2;
    for (int i = 3; (long) i * i &lt;= n; i += 2)
        if (n % i == 0) return false;
    return true;
}
```

WHY STOP AT THE SQUARE ROOT

If n = a x b and both a and b were bigger than the square root of n, their product would exceed n. So at least one factor is always at or below the square root. Checking beyond it is guaranteed wasted work.
This takes the cost from O(n) to O(sqrt n). For n = 1,000,000 that is 1000 checks instead of a million.

10. Print all primes up to n - Sieve of Eratosthenes

```
static void sieve(int n) {
    boolean[] composite = new boolean[n + 1];
    for (int i = 2; (long) i * i &lt;= n; i++)
        if (!composite[i])
            for (int j = i * i; j &lt;= n; j += i)      // start at i*i, not 2*i
                composite[j] = true;
    for (int i = 2; i &lt;= n; i++)
        if (!composite[i]) System.out.print(i + " ");
}
```

i	Marks as composite	Array state (2..12), C = composite
2	4, 6, 8, 10, 12	2 3 C 5 C 7 C 9 C 11 C
3	9, 12	2 3 C 5 C 7 C C C 11 C

Instead of testing each number for primality, cross out the multiples of each prime. O(n log log n), which is close to linear. The inner loop starts at `i*i` because everything smaller was already crossed out by a smaller prime.

11. Armstrong number

A number equal to the sum of its digits each raised to the number of digits. 153 = 1 + 125 + 27.

```
static boolean isArmstrong(int n) {
    int d = countDigits(n), t = n, sum = 0;
    while (t &gt; 0) { sum += Math.pow(t % 10, d); t /= 10; }
    return sum == n;
}
```

12. Perfect number

Equal to the sum of its proper divisors. 6 = 1 + 2 + 3.

```
static boolean isPerfect(int n) {
    int sum = 1;
    for (int i = 2; i * i &lt;= n; i++)
        if (n % i == 0) {
            sum += i;
            if (i != n / i) sum += n / i;    // add the paired divisor
        }
    return n &gt; 1 &amp;&amp; sum == n;
}
```

Divisors come in pairs: if 2 divides 36, so does 18. Finding one gives you the other free, which is why we can stop at the square root here too. The `i != n / i` guard prevents counting 6 twice for 36.

13. GCD and LCM

```
static int gcd(int a, int b) {
    while (b != 0) { int t = b; b = a % b; a = t; }
    return a;
}
static int lcm(int a, int b) { return a / gcd(a, b) * b; }
```

Round	a	b	a % b	New a	New b
1	48	18	12	18	12
2	18	12	6	12	6
3	12	6	0	6	0
4	b is 0, return 6				

Euclid's algorithm, over 2000 years old and still the fastest known. Note `a / gcd * b` rather than `a * b / gcd`: dividing first avoids overflowing on the intermediate product.

14. Swap two numbers without a temporary

```
a = a + b;  b = a - b;  a = a - b;      // can overflow
a = a ^ b;  b = a ^ b;  a = a ^ b;      // safe from overflow
```

Both break if `a` and `b` are the **same variable**: XOR-ing a variable with itself zeroes it. In real code just use a temporary. This is a puzzle, not a technique.

15. Multiplication table

```
for (int i = 1; i &lt;= 10; i++)
    System.out.printf("%d x %d = %d%n", n, i, n * i);
```

16. Decimal to binary

```
static String toBinary(int n) {
    if (n == 0) return "0";
    StringBuilder sb = new StringBuilder();
    while (n &gt; 0) { sb.append(n % 2); n /= 2; }
    return sb.reverse().toString();
}
```

Round	n	n % 2	Collected	n / 2
1	10	0	0	5
2	5	1	01	2
3	2	0	010	1
4	1	1	0101	0

Remainders come out **backwards**, so the final reverse is required. Result: 1010. The built-in is `Integer.toBinaryString(n)`.

17. Binary to decimal

```
static int toDecimal(String b) {
    int val = 0;
    for (char c : b.toCharArray()) val = val * 2 + (c - '0');
    return val;
}
```

Same shape as reversing a number, but with base 2 instead of 10. Recognising that two problems share a shape is worth more than memorising both.

18. Power without Math.pow - fast exponentiation

```
static long power(long base, int exp) {
    long result = 1;
    while (exp &gt; 0) {
        if ((exp &amp; 1) == 1) result *= base;   // odd exponent: take one out
        base *= base;                          // square the base
        exp &gt;&gt;= 1;                            // halve the exponent
    }
    return result;
}
```

Round	exp	Odd?	result	base after squaring	exp after halving
1	5	yes	1 x 2 = 2	4	2
2	2	no	2	16	1
3	1	yes	2 x 16 = 32	256	0

2^5 = 32 in three rounds instead of five multiplications. For exp = 1,000,000 it is 20 rounds instead of a million. O(log n).

19. Leap year

```
static boolean isLeap(int y) {
    return (y % 4 == 0 &amp;&amp; y % 100 != 0) || (y % 400 == 0);
}
```

Read it aloud: divisible by 4, except centuries, unless divisible by 400. 1900 is not a leap year. 2000 is. Testing only `y % 4 == 0` is the classic bug.

20. Sum of natural numbers

```
int sum = 0;
for (int i = 1; i &lt;= n; i++) sum += i;    // O(n)

int sum = n * (n + 1) / 2;                // O(1)
```

The formula version is not a trick, it is a reminder: sometimes the best optimisation is mathematics, not code.

21. Count set bits

```
static int countBits(int n) {
    int c = 0;
    while (n != 0) { n = n &amp; (n - 1); c++; }   // clears the lowest set bit
    return c;
}
```

Round	n binary	n - 1	n & (n-1)	count
1	1100	1011	1000	1
2	1000	0111	0000	2

Loops once per set bit rather than once per bit. `Integer.bitCount(n)` is the built-in.

22. Check power of two

```
static boolean isPowerOfTwo(int n) { return n &gt; 0 &amp;&amp; (n &amp; (n - 1)) == 0; }
```

A power of two has exactly one bit set. Clearing that one bit leaves zero.

23. Number pattern printing

```
// 1
// 2 3
// 4 5 6
int c = 1;
for (int i = 1; i &lt;= n; i++) {
    for (int j = 1; j &lt;= i; j++) System.out.print(c++ + " ");
    System.out.println();
}
```

24. Simple calculator

```
static double calc(double a, char op, double b) {
    return switch (op) {
        case '+' -&gt; a + b;
        case '-' -&gt; a - b;
        case '*' -&gt; a * b;
        case '/' -&gt; {
            if (b == 0) throw new ArithmeticException("divide by zero");
            yield a / b;
        }
        default -&gt; throw new IllegalArgumentException("bad operator " + op);
    };
}
```

25. Grade calculator

```
static char grade(int m) {
    if (m &lt; 0 || m &gt; 100) throw new IllegalArgumentException("bad marks");
    if (m &gt;= 90) return 'A';
    if (m &gt;= 75) return 'B';
    if (m &gt;= 60) return 'C';
    if (m &gt;= 40) return 'D';
    return 'F';
}
```

Validating the input first is a habit worth building. A silently wrong answer is far more expensive than a loud failure.

CHAPTER 2

Programs 26 to 55: Arrays

26. Sum, average, max, min in one pass

```
int sum = 0, max = a[0], min = a[0];
for (int v : a) {
    sum += v;
    if (v &gt; max) max = v;
    if (v &lt; min) min = v;
}
double avg = (double) sum / a.length;
```

One loop, four answers. Beginners often write four separate loops. Same complexity class, but four times the work and four chances for an off-by-one.

27. Linear search

```
static int search(int[] a, int key) {
    for (int i = 0; i &lt; a.length; i++) if (a[i] == key) return i;
    return -1;
}
```

28. Binary search, iterative

```
static int bsearch(int[] a, int key) {
    int lo = 0, hi = a.length - 1;
    while (lo &lt;= hi) {
        int mid = lo + (hi - lo) / 2;
        if (a[mid] == key) return mid;
        if (a[mid] &lt; key) lo = mid + 1; else hi = mid - 1;
    }
    return -1;
}
```

29. Reverse an array

```
int i = 0, j = a.length - 1;
while (i &lt; j) { int t = a[i]; a[i++] = a[j]; a[j--] = t; }
```

30. Second largest element

```
int first = Integer.MIN_VALUE, second = Integer.MIN_VALUE;
for (int v : a) {
    if (v &gt; first) { second = first; first = v; }
    else if (v &gt; second &amp;&amp; v != first) second = v;
}
```

31. Count even, odd, positive, negative

```
int ev = 0, od = 0, pos = 0, neg = 0, zero = 0;
for (int v : a) {
    if (v % 2 == 0) ev++; else od++;
    if (v &gt; 0) pos++; else if (v &lt; 0) neg++; else zero++;
}
```

32. Find duplicates - three approaches

```
// A. Brute force, O(n^2), O(1) space
for (int i = 0; i &lt; n; i++)
    for (int j = i + 1; j &lt; n; j++)
        if (a[i] == a[j]) System.out.println(a[i]);

// B. Sort first, O(n log n), O(1) space
Arrays.sort(a);
for (int i = 1; i &lt; n; i++) if (a[i] == a[i-1]) System.out.println(a[i]);

// C. HashSet, O(n) time, O(n) space
Set&lt;Integer&gt; seen = new HashSet&lt;&gt;();
for (int v : a) if (!seen.add(v)) System.out.println(v);
```

THE THREE-WAY TRADE, STATED PLAINLY

A is slow but uses no memory and does not disturb the array.
B is fast and memory-free but destroys the original order.
C is fastest but costs O(n) memory.
There is no universally best answer. In an interview, saying this out loud is worth more than writing any one of them.

33. Remove duplicates from a sorted array in place

```
static int removeDup(int[] a) {
    if (a.length == 0) return 0;
    int k = 1;                                  // write pointer
    for (int i = 1; i &lt; a.length; i++)
        if (a[i] != a[k - 1]) a[k++] = a[i];
    return k;                                   // new logical length
}
```

i	a[i]	a[k-1]	Different?	Array	k
1	1	1	no	1 1 2 2 3	1
2	2	1	yes	1 2 2 2 3	2
3	2	2	no	1 2 2 2 3	2
4	3	2	yes	1 2 3 2 3	3

The first `k` elements are the answer. The rest is leftover garbage, which is normal for in-place algorithms.

34. Move zeros to the end

```
int j = 0;
for (int i = 0; i &lt; n; i++)
    if (a[i] != 0) { int t = a[i]; a[i] = a[j]; a[j++] = t; }
```

35. Rotate array by k

```
static void rotate(int[] a, int k) {
    int n = a.length; k %= n;
    rev(a, 0, n-1); rev(a, 0, k-1); rev(a, k, n-1);
}
```

The `k %= n` matters: rotating a 5-element array by 7 is the same as rotating by 2, and without the modulus the code goes out of bounds.

36. Merge two sorted arrays

```
static int[] merge(int[] a, int[] b) {
    int[] r = new int[a.length + b.length];
    int i = 0, j = 0, k = 0;
    while (i &lt; a.length &amp;&amp; j &lt; b.length)
        r[k++] = (a[i] &lt;= b[j]) ? a[i++] : b[j++];
    while (i &lt; a.length) r[k++] = a[i++];
    while (j &lt; b.length) r[k++] = b[j++];
    return r;
}
```

Step	i	j	a[i]	b[j]	Take	r so far
1	0	0	1	2	a[0]=1	1

Step	i	j	a[i]	b[j]	Take	r so far
2	1	0	3	2	b[0]=2	1 2
3	1	1	3	4	a[1]=3	1 2 3
4	2	1	a exhausted	-	drain b	1 2 3 4

The two drain loops are not optional. One of the two arrays always has leftovers, and forgetting them silently truncates the result. Using <= rather than < is what makes this merge **stable**.

37. Missing number from 1 to n

```
// Sum approach
int expected = n * (n + 1) / 2, actual = 0;
for (int v : a) actual += v;
return expected - actual;

// XOR approach: no overflow risk
int x = 0;
for (int i = 1; i &lt;= n; i++) x ^= i;
for (int v : a) x ^= v;
return x;
```

38. Find the element that appears once, all others twice

```
int x = 0;
for (int v : a) x ^= v;
return x;
```

O(n) time, O(1) space, and one line. The HashMap version needs O(n) memory to do the same job.

39. Maximum subarray sum - Kadane's algorithm

```
static int maxSubArray(int[] a) {
    int best = a[0], cur = a[0];
    for (int i = 1; i &lt; a.length; i++) {
        cur = Math.max(a[i], cur + a[i]);   // extend, or start fresh here
        best = Math.max(best, cur);
    }
    return best;
}
```

i	a[i]	cur + a[i]	cur = max(a[i], cur+a[i])	best
-	-2	-	-2	-2
1	1	-1	1 (start fresh)	1
2	-3	-2	-2	1
3	4	2	4 (start fresh)	4
4	-1	3	3	4
5	2	5	5	5
6	1	6	6	6

The entire insight is one question asked at every position: *is the sum I have been carrying helping me, or dragging me down?* If it is negative, drop it and start again. O(n) replaces the O(n^2) brute force over all subarrays.

40. Two sum

```
Map&lt;Integer,Integer&gt; seen = new HashMap&lt;&gt;();
for (int i = 0; i &lt; n; i++) {
    int need = target - a[i];
    if (seen.containsKey(need)) return new int[]{seen.get(need), i};
    seen.put(a[i], i);
}
```

41. Three sum

```
Arrays.sort(a);
for (int i = 0; i &lt; n - 2; i++) {
    if (i &gt; 0 &amp;&amp; a[i] == a[i-1]) continue;         // skip duplicate anchors
    int l = i + 1, r = n - 1;
    while (l &lt; r) {
        int s = a[i] + a[l] + a[r];
        if (s == 0) {
            result.add(List.of(a[i], a[l], a[r]));
            while (l &lt; r &amp;&amp; a[l] == a[l+1]) l++;    // skip duplicates
            while (l &lt; r &amp;&amp; a[r] == a[r-1]) r--;
            l++; r--;
        } else if (s &lt; 0) l++; else r--;
    }
}
```

Sorting first turns an O(n^3) brute force into O(n^2): fix one number, then two-pointer the rest. The duplicate-skipping lines are what most people forget, and they are the difference between passing and failing.

42. Sort 0s, 1s and 2s - Dutch national flag

```
int low = 0, mid = 0, high = n - 1;
while (mid &lt;= high) {
    if (a[mid] == 0)      { swap(a, low++, mid++); }
    else if (a[mid] == 1) { mid++; }
    else                  { swap(a, mid, high--); }   // do NOT advance mid
}
```

Step	low	mid	high	a[mid]	Action	Array
1	0	0	4	2	swap mid,high	1 0 1 2 **2**
2	0	0	3	1	mid++	1 0 1 2 2
3	0	1	3	0	swap low,mid	0 1 1 2 2
4	1	2	3	1	mid++	0 1 1 2 2
5	1	3	3	2	swap mid,high	0 1 1 2 2

One pass, no counting, no sorting. `mid` is deliberately not advanced in the 2-case because the value swapped in from the right has not been examined yet. That asymmetry is the whole subtlety of the algorithm.

43. Majority element - Boyer-Moore voting

```
int candidate = a[0], count = 0;
for (int v : a) {
    if (count == 0) candidate = v;
    count += (v == candidate) ? 1 : -1;
}
// verify with a second pass if the majority is not guaranteed
```

v	count before	candidate	count after
2	0	2	1
2	1	2	2
1	2	2	1
1	1	2	0
2	0	2 (reset)	1

Picture every element cancelling one element of a different value. If one value appears more than half the time, it cannot be fully cancelled. O(n) time, O(1) space, and it feels like magic until you see the cancellation picture.

44. Left rotate an array by one

```
int first = a[0];
for (int i = 0; i &lt; n - 1; i++) a[i] = a[i + 1];
a[n - 1] = first;
```

45. Leaders in an array

An element is a leader if it is greater than everything to its right.

```
int maxRight = a[n - 1];
System.out.print(maxRight + " ");
for (int i = n - 2; i &gt;= 0; i--)
    if (a[i] &gt; maxRight) { maxRight = a[i]; System.out.print(a[i] + " "); }
```

Scanning **right to left** turns an O(n^2) problem into O(n). When a problem talks about "everything to the right", try reversing the direction of the loop first.

46. Equilibrium index

The index where the sum on the left equals the sum on the right.

```
int total = 0;
for (int v : a) total += v;
int left = 0;
for (int i = 0; i &lt; n; i++) {
    total -= a[i];                 // total now holds the right side
    if (left == total) return i;
    left += a[i];
}
return -1;
```

47. Best time to buy and sell stock

```
int minPrice = Integer.MAX_VALUE, profit = 0;
for (int p : prices) {
    minPrice = Math.min(minPrice, p);
    profit = Math.max(profit, p - minPrice);
}
```

p	minPrice before	minPrice after	p - minPrice	profit
7	MAX	7	0	0
1	7	1	0	0
5	1	1	4	4
3	1	1	2	4
6	1	1	5	5

The trap is trying to find the maximum and minimum separately: the maximum might come **before** the minimum, which is not a valid trade. Tracking the minimum-so-far as you move forward automatically enforces the ordering.

48. Container with most water

```
int l = 0, r = n - 1, best = 0;
while (l &lt; r) {
    best = Math.max(best, Math.min(h[l], h[r]) * (r - l));
    if (h[l] &lt; h[r]) l++; else r--;      // always move the SHORTER wall
}
```

Always moving the shorter wall is provably safe: moving the taller one can only reduce the width without ever increasing the limiting height, so it can never improve the result.

49. Product of array except self

```
int[] res = new int[n];
res[0] = 1;
for (int i = 1; i &lt; n; i++) res[i] = res[i-1] * a[i-1];   // prefix products
int right = 1;
for (int i = n - 1; i &gt;= 0; i--) { res[i] *= right; right *= a[i]; }
```

Two passes, no division, O(1) extra space beyond the output. Division would be simpler but breaks the moment the array contains a zero.

50. Matrix operations

```
// Transpose in place
for (int i = 0; i &lt; n; i++)
    for (int j = i + 1; j &lt; n; j++) { int t=m[i][j]; m[i][j]=m[j][i]; m[j][i]=t; }

// Rotate 90 degrees clockwise: transpose, then reverse each row
// Spiral print
int top=0, bottom=n-1, left=0, right=n-1;
while (top &lt;= bottom &amp;&amp; left &lt;= right) {
    for (int j = left; j &lt;= right; j++) System.out.print(m[top][j] + " ");
    top++;
    for (int i = top; i &lt;= bottom; i++) System.out.print(m[i][right] + " ");
    right--;
    if (top &lt;= bottom) {
        for (int j = right; j &gt;= left; j--) System.out.print(m[bottom][j]+" ");
        bottom--;
    }
    if (left &lt;= right) {
        for (int i = bottom; i &gt;= top; i--) System.out.print(m[i][left] + " ");
        left++;
    }
}
```

The two extra `if` checks before the bottom row and left column exist to stop a single remaining row or column being printed twice. Non-square matrices are exactly where that bug shows up.

Programs 51 to 75: Strings, Sorting And Searching

51. Reverse a string, three ways

```
String r = new StringBuilder(s).reverse().toString();      // library

char[] c = s.toCharArray();                                 // two pointers
for (int i = 0, j = c.length-1; i &lt; j; i++, j--) {
    char t = c[i]; c[i] = c[j]; c[j] = t;
}

static String rec(String s) {                               // recursion
    return s.isEmpty() ? s : rec(s.substring(1)) + s.charAt(0);
}
```

The recursive version is elegant and wasteful: each `substring` allocates a new String, giving O(n^2) time and O(n) stack. Elegance is not the same as efficiency, and knowing the difference is part of the job.

52. Palindrome string, ignoring case and punctuation

```
static boolean isPal(String s) {
    int i = 0, j = s.length() - 1;
    while (i &lt; j) {
        while (i &lt; j &amp;&amp; !Character.isLetterOrDigit(s.charAt(i))) i++;
        while (i &lt; j &amp;&amp; !Character.isLetterOrDigit(s.charAt(j))) j--;
        if (Character.toLowerCase(s.charAt(i)) !=
            Character.toLowerCase(s.charAt(j))) return false;
        i++; j--;
    }
    return true;
}
```

The `i < j` guard inside the skip loops prevents running off the end on a string of pure punctuation. Inner loops always need their own bounds check.

53. Count vowels, consonants, digits, spaces

```
int v=0, c=0, d=0, sp=0;
for (char ch : s.toLowerCase().toCharArray()) {
    if (Character.isDigit(ch)) d++;
    else if (ch == ' ') sp++;
    else if ("aeiou".indexOf(ch) &gt;= 0) v++;
    else if (Character.isLetter(ch)) c++;
}
```

54. Character frequency

```
int[] f = new int[26];
for (char c : s.toCharArray()) if (Character.isLetter(c)) f[c - 'a']++;
```

55. First non-repeating character

```
int[] f = new int[26];
for (char c : s.toCharArray()) f[c - 'a']++;
for (char c : s.toCharArray()) if (f[c - 'a'] == 1) return c;
return '_';
```

56. Check anagram

```
if (a.length() != b.length()) return false;
int[] f = new int[26];
for (int i = 0; i &lt; a.length(); i++) { f[a.charAt(i)-'a']++; f[b.charAt(i)-'a']--; }
for (int x : f) if (x != 0) return false;
return true;
```

57. Group anagrams

```
Map&lt;String, List&lt;String&gt;&gt; groups = new HashMap&lt;&gt;();
for (String w : words) {
    char[] c = w.toCharArray();
    Arrays.sort(c);
    String key = new String(c);                  // canonical form
    groups.computeIfAbsent(key, k -&gt; new ArrayList&lt;&gt;()).add(w);
}
```

The idea is a **canonical key**: all anagrams sort to the same string, so the sorted form identifies the group. Finding a canonical form is a general technique - it appears again in deduplication and caching.

58. Remove duplicate characters

```
StringBuilder sb = new StringBuilder();
boolean[] seen = new boolean[256];
for (char c : s.toCharArray())
    if (!seen[c]) { seen[c] = true; sb.append(c); }
```

59. Longest substring without repeating characters - sliding window

```
static int longest(String s) {
    int[] last = new int[128];
    Arrays.fill(last, -1);
    int best = 0, start = 0;
    for (int i = 0; i &lt; s.length(); i++) {
        char c = s.charAt(i);
        if (last[c] &gt;= start) start = last[c] + 1;   // shrink the window
        last[c] = i;
        best = Math.max(best, i - start + 1);
    }
    return best;
}
```

i	c	last[c]	start before	start after	Window	best
0	a	-1	0	0	a	1
1	b	-1	0	0	ab	2
2	c	-1	0	0	abc	3
3	a	0	0	1	bca	3
4	b	1	1	2	cab	3

Both pointers only ever move forward, so despite the nested feel this is O(n), not O(n^2).

60. Count words and reverse word order

```
String[] w = s.trim().split("\\s+");
System.out.println(w.length);
Collections.reverse(Arrays.asList(w));
System.out.println(String.join(" ", w));
```

61. Capitalise the first letter of every word

```
StringBuilder sb = new StringBuilder();
boolean newWord = true;
for (char c : s.toCharArray()) {
    if (Character.isWhitespace(c)) { newWord = true; sb.append(c); }
    else { sb.append(newWord ? Character.toUpperCase(c) : c); newWord = false; }
}
```

A single boolean flag replaces splitting, looping and rejoining. When a problem is a single left-to-right pass, a flag is usually cleaner than restructuring the data.

62. String compression - run length encoding

```
StringBuilder sb = new StringBuilder();
for (int i = 0; i &lt; s.length(); ) {
    char c = s.charAt(i);
    int j = i;
    while (j &lt; s.length() &amp;&amp; s.charAt(j) == c) j++;
    sb.append(c).append(j - i);
    i = j;
}
// aaabbc -&gt; a3b2c1
```

Note the `for` loop with no update clause: `i` is advanced by the inner loop instead. This is a legitimate and common pattern when the step size varies.

63. Check if two strings are rotations

```
static boolean isRotation(String a, String b) {
    return a.length() == b.length() &amp;&amp; (a + a).contains(b);
}
```

Every rotation of a string appears somewhere inside the string doubled. One line replaces a nested loop. Look for these structural insights - they are what separates a good solution from a merely correct one.

64. Longest common prefix

```
String prefix = strs[0];
for (String s : strs) {
    while (!s.startsWith(prefix))
        prefix = prefix.substring(0, prefix.length() - 1);
    if (prefix.isEmpty()) return "";
}
```

65. Valid parentheses

```
Deque&lt;Character&gt; st = new ArrayDeque&lt;&gt;();
Map&lt;Character,Character&gt; pairs = Map.of(')','(', ']','[', '}','{');
for (char c : s.toCharArray()) {
    if (pairs.containsValue(c)) st.push(c);
    else if (pairs.containsKey(c)) {
        if (st.isEmpty() || st.pop() != pairs.get(c)) return false;
    }
}
return st.isEmpty();
```

66. Bubble sort

```
for (int i = 0; i &lt; n - 1; i++) {
    boolean swapped = false;
    for (int j = 0; j &lt; n - 1 - i; j++)
        if (a[j] &gt; a[j+1]) { int t=a[j]; a[j]=a[j+1]; a[j+1]=t; swapped=true; }
    if (!swapped) break;
}
```

67. Selection sort

```
for (int i = 0; i &lt; n - 1; i++) {
    int min = i;
    for (int j = i + 1; j &lt; n; j++) if (a[j] &lt; a[min]) min = j;
    int t = a[i]; a[i] = a[min]; a[min] = t;
}
```

68. Insertion sort

```
for (int i = 1; i &lt; n; i++) {
    int key = a[i], j = i - 1;
    while (j &gt;= 0 &amp;&amp; a[j] &gt; key) { a[j+1] = a[j]; j--; }
    a[j+1] = key;
}
```

i	key	Array before	Shifts	Array after
1	2	5 2 4 6	5 moves right	2 5 4 6
2	4	2 5 4 6	5 moves right	2 4 5 6
3	6	2 4 5 6	none	2 4 5 6

69. Merge sort

```
static void mergeSort(int[] a, int l, int r) {
    if (l &gt;= r) return;
    int m = l + (r - l) / 2;
    mergeSort(a, l, m);
    mergeSort(a, m + 1, r);
    merge(a, l, m, r);
}
static void merge(int[] a, int l, int m, int r) {
    int[] tmp = new int[r - l + 1];
    int i = l, j = m + 1, k = 0;
    while (i &lt;= m &amp;&amp; j &lt;= r) tmp[k++] = (a[i] &lt;= a[j]) ? a[i++] : a[j++];
    while (i &lt;= m) tmp[k++] = a[i++];
    while (j &lt;= r) tmp[k++] = a[j++];
    System.arraycopy(tmp, 0, a, l, tmp.length);
}
```

70. Quick sort

```
static void quickSort(int[] a, int lo, int hi) {
    if (lo &gt;= hi) return;
    int p = partition(a, lo, hi);
    quickSort(a, lo, p - 1);
    quickSort(a, p + 1, hi);
}
static int partition(int[] a, int lo, int hi) {
    int pivot = a[hi], i = lo - 1;
    for (int j = lo; j &lt; hi; j++)
        if (a[j] &lt; pivot) { i++; swap(a, i, j); }
    swap(a, i + 1, hi);
    return i + 1;
}
```

j	a[j]	pivot	a[j] < pivot?	i after	Array
0	3	4	yes	0	3 7 8 5 2 4
1	7	4	no	0	3 7 8 5 2 4
2	8	4	no	0	3 7 8 5 2 4
3	5	4	no	0	3 7 8 5 2 4
4	2	4	yes	1	3 2 8 5 7 4
end	place pivot at i+1 = 2	-	-	-	3 2 4 5 7 8

After partitioning, everything left of the pivot is smaller and everything right is larger. The pivot is now in its **final** position and never moves again. That is the invariant.

71. Counting sort

```
static void countingSort(int[] a, int max) {
    int[] count = new int[max + 1];
    for (int v : a) count[v]++;
    int k = 0;
    for (int v = 0; v &lt;= max; v++)
        while (count[v]-- &gt; 0) a[k++] = v;
}
```

O(n + k), which beats O(n log n) - but only when the value range k is small. Sorting a million values in the range 0 to 100 is ideal. Sorting a hundred values ranging up to a billion is a disaster.

72. Search in a rotated sorted array

```
int lo = 0, hi = n - 1;
while (lo &lt;= hi) {
    int mid = lo + (hi - lo) / 2;
    if (a[mid] == key) return mid;
    if (a[lo] &lt;= a[mid]) {                      // left half is sorted
        if (key &gt;= a[lo] &amp;&amp; key &lt; a[mid]) hi = mid - 1; else lo = mid + 1;
    } else {                                     // right half is sorted
        if (key &gt; a[mid] &amp;&amp; key &lt;= a[hi]) lo = mid + 1; else hi = mid - 1;
    }
}
```

Even in a rotated array, at least one half is always properly sorted. Identify which half that is, check whether the target lies inside it, and binary search survives. O(log n) preserved.

73. First and last occurrence of a value

```
static int firstOccurrence(int[] a, int key) {
    int lo = 0, hi = a.length - 1, ans = -1;
    while (lo &lt;= hi) {
        int mid = lo + (hi - lo) / 2;
        if (a[mid] == key) { ans = mid; hi = mid - 1; }   // keep going LEFT
        else if (a[mid] &lt; key) lo = mid + 1;
        else hi = mid - 1;
    }
    return ans;
}
```

Do not stop on a match. Record it and keep searching in the direction of the boundary you want. Switching `hi = mid - 1` to `lo = mid + 1` gives the last occurrence. This one modification unlocks a whole family of binary search problems.

74. Square root by binary search

```
static int mySqrt(int x) {
    int lo = 0, hi = x, ans = 0;
    while (lo &lt;= hi) {
        int mid = lo + (hi - lo) / 2;
        if ((long) mid * mid &lt;= x) { ans = mid; lo = mid + 1; }
        else hi = mid - 1;
    }
    return ans;
}
```

Binary search is not only for arrays. Any time the answer space is **monotonic** - true, true, true, false, false - you can binary search over the answers themselves.

75. Kth largest element

```
PriorityQueue&lt;Integer&gt; pq = new PriorityQueue&lt;&gt;();   // MIN heap of size k
for (int v : a) { pq.offer(v); if (pq.size() &gt; k) pq.poll(); }
return pq.peek();
```

O(n log k) time and O(k) space, which beats sorting when k is much smaller than n.

Programs 76 to 100: Structures And Patterns

76. Linked list: insert, delete, search, print

```
void addFirst(int d) { Node n = new Node(d); n.next = head; head = n; }
void addLast(int d) {
    Node n = new Node(d);
    if (head == null) { head = n; return; }
    Node c = head; while (c.next != null) c = c.next; c.next = n;
}
void delete(int d) {
    if (head == null) return;
    if (head.data == d) { head = head.next; return; }
    Node c = head;
    while (c.next != null &amp;&amp; c.next.data != d) c = c.next;
    if (c.next != null) c.next = c.next.next;
}
```

77. Reverse a linked list

```
Node prev = null, cur = head;
while (cur != null) {
    Node next = cur.next;
    cur.next = prev;
    prev = cur;
    cur = next;
}
head = prev;
```

78. Find the middle node

```
Node slow = head, fast = head;
while (fast != null &amp;&amp; fast.next != null) { slow = slow.next; fast = fast.next.next; }
return slow;
```

79. Detect and find the start of a cycle

```
Node slow = head, fast = head;
while (fast != null &amp;&amp; fast.next != null) {
    slow = slow.next; fast = fast.next.next;
    if (slow == fast) {                    // cycle confirmed
        slow = head;
        while (slow != fast) { slow = slow.next; fast = fast.next; }
        return slow;                       // start of the loop
    }
}
return null;
```

The second phase is a genuinely surprising result: after the meeting point, moving one pointer back to the head and advancing both one step at a time makes them meet exactly at the loop's entry. It falls out of the distance arithmetic, and it is worth deriving on paper once.

80. Merge two sorted linked lists

```
Node dummy = new Node(0), tail = dummy;
while (a != null &amp;&amp; b != null) {
    if (a.data &lt;= b.data) { tail.next = a; a = a.next; }
    else { tail.next = b; b = b.next; }
    tail = tail.next;
}
tail.next = (a != null) ? a : b;
return dummy.next;
```

THE DUMMY NODE TRICK

Creating a throwaway node before the real head removes every "is this the first element?" special case. You return `dummy.next` at the end.

This trick simplifies merging, removing, and partitioning linked lists. Learn it once and use it in every linked list problem.

81. Remove the nth node from the end

```
Node dummy = new Node(0); dummy.next = head;
Node fast = dummy, slow = dummy;
for (int i = 0; i &lt;= n; i++) fast = fast.next;   // create a gap of n
while (fast != null) { fast = fast.next; slow = slow.next; }
slow.next = slow.next.next;
return dummy.next;
```

Two pointers held a fixed distance apart. When the leader reaches the end, the follower is exactly n from the end. One pass, no length calculation.

82. Stack using an array

```
class Stack {
    int[] a; int top = -1;
    Stack(int c) { a = new int[c]; }
    void push(int x) { if (top == a.length-1) throw new RuntimeException("full"); a[++top] = x; }
    int pop() { if (top &lt; 0) throw new RuntimeException("empty"); return a[top--]; }
    int peek() { return a[top]; }
    boolean isEmpty() { return top == -1; }
}
```

83. Queue using two stacks

```
class QueueFrom2Stacks {
    Deque&lt;Integer&gt; in = new ArrayDeque&lt;&gt;(), out = new ArrayDeque&lt;&gt;();
    void enqueue(int x) { in.push(x); }
    int dequeue() {
        if (out.isEmpty())
            while (!in.isEmpty()) out.push(in.pop());   // reverse once
        return out.pop();
    }
}
```

Operation	in	out	Returned
enqueue 1,2,3	3,2,1	empty	-
dequeue	empty	1,2,3	1
dequeue	empty	2,3	2
enqueue 4	4	2,3	-
dequeue	4	3	3

Pouring one stack into another reverses it, turning LIFO into FIFO. Each element is moved at most twice, so the **amortised** cost is O(1) even though a single dequeue can be O(n).

84. Stack using two queues

```
void push(int x) {
    q2.add(x);
    while (!q1.isEmpty()) q2.add(q1.poll());
    Queue&lt;Integer&gt; t = q1; q1 = q2; q2 = t;    // swap
}
int pop() { return q1.poll(); }
```

85. Next greater element

```
int[] res = new int[n];
Deque&lt;Integer&gt; st = new ArrayDeque&lt;&gt;();          // holds INDEXES
for (int i = n - 1; i &gt;= 0; i--) {
    while (!st.isEmpty() &amp;&amp; a[st.peek()] &lt;= a[i]) st.pop();  // discard the useless
    res[i] = st.isEmpty() ? -1 : a[st.peek()];
    st.push(i);
}
```

i	a[i]	Stack before (values)	Popped	res[i]	Stack after
4	3	empty	-	-1	3
3	2	3	none	3	3, 2
2	4	3, 2	2 and 3	-1	4
1	1	4	none	4	4, 1
0	5	4, 1	1 and 4	-1	5

This is a **monotonic stack**: it holds a decreasing sequence, and anything smaller than the incoming element can never be an answer for anything further left, so it is discarded permanently. Every element is pushed and popped at most once, giving O(n) instead of O(n^2).

86. Min stack - retrieve the minimum in O(1)

```
Deque&lt;Integer&gt; data = new ArrayDeque&lt;&gt;(), mins = new ArrayDeque&lt;&gt;();
void push(int x) {
    data.push(x);
    mins.push(mins.isEmpty() ? x : Math.min(x, mins.peek()));
}
void pop() { data.pop(); mins.pop(); }
int getMin() { return mins.peek(); }
```

A second stack stores "the minimum at the time of this push". Both stacks always move together, so the top of `mins` is always correct. Extra memory buys constant-time queries - the same trade as always.

87. BST insert, search, delete

```
TreeNode insert(TreeNode n, int v) {
    if (n == null) return new TreeNode(v);
    if (v &lt; n.data) n.left = insert(n.left, v);
    else if (v &gt; n.data) n.right = insert(n.right, v);
    return n;
}
boolean search(TreeNode n, int v) {
    if (n == null) return false;
    if (n.data == v) return true;
    return v &lt; n.data ? search(n.left, v) : search(n.right, v);
}
```

88. Tree traversals, all four

```
void inorder(TreeNode n)   { if(n==null)return; inorder(n.left); print(n); inorder(n.right); }
void preorder(TreeNode n)  { if(n==null)return; print(n); preorder(n.left); preorder(n.right); }
void postorder(TreeNode n) { if(n==null)return; postorder(n.left); postorder(n.right); print(n); }
void levelOrder(TreeNode root) {
    Queue&lt;TreeNode&gt; q = new LinkedList&lt;&gt;(); q.add(root);
    while (!q.isEmpty()) {
        TreeNode n = q.poll(); print(n);
        if (n.left != null) q.add(n.left);
        if (n.right != null) q.add(n.right);
    }
}
```

89. Height, node count, leaf count

```
int height(TreeNode n) { return n==null ? -1 : 1 + Math.max(height(n.left), height(n.right)); }
int count(TreeNode n)  { return n==null ? 0 : 1 + count(n.left) + count(n.right); }
int leaves(TreeNode n) {
    if (n == null) return 0;
    if (n.left == null &amp;amp;&amp;amp; n.right == null) return 1;
    return leaves(n.left) + leaves(n.right);
}
```

90. Validate a BST

```
boolean isBST(TreeNode n, long min, long max) {
    if (n == null) return true;
    if (n.data &lt;= min || n.data &gt;= max) return false;
    return isBST(n.left, min, n.data) &amp;&amp; isBST(n.right, n.data, max);
}
// call: isBST(root, Long.MIN_VALUE, Long.MAX_VALUE)
```

The common wrong answer only compares each node with its immediate children. That passes on trees that are clearly not BSTs, because a node deep in the left subtree can still be larger than the root. Passing a **narrowing range** down the recursion is the correct approach.

91. Lowest common ancestor in a BST

```
TreeNode lca(TreeNode n, int p, int q) {
    while (n != null) {
        if (p &lt; n.data &amp;&amp; q &lt; n.data) n = n.left;
        else if (p &gt; n.data &amp;&amp; q &gt; n.data) n = n.right;
        else return n;                // the split point IS the answer
    }
    return null;
}
```

92. Mirror or invert a tree

```
TreeNode invert(TreeNode n) {
    if (n == null) return null;
    TreeNode t = n.left;
    n.left = invert(n.right);
    n.right = invert(t);
    return n;
}
```

93. Diameter of a tree

```
int best = 0;
int depth(TreeNode n) {
    if (n == null) return 0;
    int l = depth(n.left), r = depth(n.right);
    best = Math.max(best, l + r);         // path THROUGH this node
    return 1 + Math.max(l, r);            // depth reported upwards
}
```

One recursion computing two different things: the value returned upwards, and a running answer updated as a side effect. This pattern - "return one thing, track another" - solves a large share of hard tree problems.

94. Graph BFS and DFS

```
void bfs(int s) {
    boolean[] vis = new boolean[n];
    Queue&lt;Integer&gt; q = new LinkedList&lt;&gt;();
    q.add(s); vis[s] = true;
    while (!q.isEmpty()) {
        int u = q.poll(); print(u);
        for (int v : adj.get(u)) if (!vis[v]) { vis[v] = true; q.add(v); }
    }
}
void dfs(int u, boolean[] vis) {
    vis[u] = true; print(u);
    for (int v : adj.get(u)) if (!vis[v]) dfs(v, vis);
}
```

95. Count connected components

```
int components = 0;
boolean[] vis = new boolean[n];
for (int i = 0; i &lt; n; i++)
    if (!vis[i]) { dfs(i, vis); components++; }
```

Each fresh DFS from an unvisited vertex discovers exactly one component. The loop over all vertices is what handles a disconnected graph - a single DFS from vertex 0 would silently miss the rest.

96. Detect a cycle in an undirected graph

```
boolean hasCycle(int u, int parent, boolean[] vis) {
    vis[u] = true;
    for (int v : adj.get(u)) {
        if (!vis[v]) { if (hasCycle(v, u, vis)) return true; }
        else if (v != parent) return true;
    }
    return false;
}
```

97. Number of islands in a grid

```
int count = 0;
for (int i = 0; i &lt; rows; i++)
    for (int j = 0; j &lt; cols; j++)
        if (grid[i][j] == '1') { sink(grid, i, j); count++; }

void sink(char[][] g, int i, int j) {
    if (i &lt; 0 || j &lt; 0 || i &gt;= g.length || j &gt;= g[0].length || g[i][j] != '1') return;
    g[i][j] = '0';                          // mark visited by overwriting
    sink(g, i+1, j); sink(g, i-1, j); sink(g, i, j+1); sink(g, i, j-1);
}
```

A grid **is** a graph: each cell is a vertex, and its up/down/left/right neighbours are its edges. Recognising that saves you from inventing a special algorithm. The single bounds check at the top of the recursion replaces four checks at each call site.

98. Word frequency counter

```
Map&lt;String,Integer&gt; freq = new HashMap&lt;&gt;();
for (String w : text.toLowerCase().split("\\W+"))
    if (!w.isEmpty()) freq.merge(w, 1, Integer::sum);

freq.entrySet().stream()
    .sorted(Map.Entry.&lt;String,Integer&gt;comparingByValue().reversed())
    .limit(10)
    .forEach(e -&gt; System.out.println(e.getKey() + " : " + e.getValue()));
```

99. LRU cache - a HashMap plus a doubly linked list

```
class LRUCache extends LinkedHashMap&lt;Integer,Integer&gt; {
    private final int cap;
    LRUCache(int cap) {
        super(cap, 0.75f, true);       // true = ACCESS order, not insertion
        this.cap = cap;
    }
    protected boolean removeEldestEntry(Map.Entry&lt;Integer,Integer&gt; e) {
        return size() &gt; cap;
    }
}
```

The manual version pairs a `HashMap` (for O(1) lookup) with a doubly linked list (for O(1) move-to-front and O(1) eviction from the tail). Neither structure alone can do both. **Combining two structures to get the strengths of each is a real design technique**, and the LRU cache is its clearest example.

100. Trie: insert, search, prefix search

```
class Trie {
    TrieNode root = new TrieNode();
    void insert(String w) {
        TrieNode c = root;
        for (char ch : w.toCharArray()) {
            int i = ch - 'a';
            if (c.children[i] == null) c.children[i] = new TrieNode();
            c = c.children[i];
        }
        c.isEnd = true;
    }
    boolean search(String w) { TrieNode n = walk(w); return n != null &amp;&amp; n.isEnd; }
    boolean startsWith(String p) { return walk(p) != null; }
    private TrieNode walk(String s) {
        TrieNode c = root;
        for (char ch : s.toCharArray()) {
            c = c.children[ch - 'a'];
            if (c == null) return null;
        }
        return c;
    }
}
```

PART 9 - MISTAKES, METHOD AND REFERENCE

CHAPTER 1

The Mistakes That Cost Hours

Compile-time errors: the compiler is helping you

Message	Real meaning	Fix
`cannot find symbol`	A name is misspelled, out of scope, or not imported	Check spelling, scope, imports
`incompatible types`	Wrong shape of value for the box	Cast, or change the type
`variable might not have been initialized`	A local was used before being given a value	Assign at declaration
`missing return statement`	Some path through the method returns nothing	Add a final return
`unreachable statement`	Code after `return`, `break` or `throw`	Delete it
`non-static method cannot be referenced from a static context`	Called an instance method from `main`	Make it static, or create an object
`class X is public, should be in file X.java`	Filename does not match the public class	Rename the file

Runtime errors: the ten you will actually meet

Exception	Caused by	Prevention
`NullPointerException`	Using a member on a null reference	Check for null; use `Optional`; prefer `Objects.requireNonNull` early
`ArrayIndexOutOfBoundsException`	Index below 0 or at/above length	`i < arr.length`, never `<=`
`StringIndexOutOfBoundsException`	Same, on a String	Check `length()` first
`ArithmeticException`	Integer divide by zero	Check the divisor. Note floating point gives Infinity instead.
`NumberFormatException`	`Integer.parseInt("abc")`	Validate, or catch
`ClassCastException`	Casting to the wrong type	Use `instanceof` first, or generics
`ConcurrentModificationException`	Modifying a collection while for-each looping	Use `Iterator.remove()` or `removeIf`
`StackOverflowError`	Runaway recursion	Fix the base case
`OutOfMemoryError`	Allocated too much, or a leak	Reduce; check for structures that only grow
`UnsupportedOperationException`	Modifying an immutable list	Copy into a new `ArrayList`

The silent bugs - no error, wrong answer

These are far more dangerous than exceptions, because nothing tells you anything is wrong.

Bug	Symptom	Fix
== on Strings or wrappers	Comparison sometimes works, sometimes not	`.equals()`
Integer division	`5/2` gives 2	Cast one operand to `double`
`double` for money	0.1 + 0.2 is not 0.3	`BigDecimal` or integer paise
`list.remove(int)` versus `remove(Object)`	Wrong element vanishes	`remove(Integer.valueOf(x))`
Overflow	Big numbers become negative	Use `long`, or `Math.addExact`

Bug	Symptom	Fix
`equals` without `hashCode`	HashMap loses entries	Override both
Mutating a HashMap key	Entry becomes unreachable	Use immutable keys
`%` on negatives	`-7 % 3` is -1	`((a % n) + n) % n`
Shallow copy of an object array	Both arrays see the same objects	Copy each element
Missing `break` in a switch	Extra cases run	Add `break` or use arrow syntax
Comparator returning `a - b`	Overflow flips the order	`Integer.compare(a, b)`

Off-by-one errors: a checklist

Almost every loop bug is one of these five.

1) `<` versus `<=`. For array indexes it is `< length`, always.
2) Starting at 0 versus 1. Arrays start at 0. Human counting starts at 1.
3) The last valid index is `length - 1`, never `length`.
4) `substring(a, b)` excludes b. The result length is `b - a`.
5) After a `while (i < n)` loop, `i` equals `n`, not `n - 1`.

> **THE TWO-MINUTE CHECK THAT CATCHES MOST OF THEM**
> Run every loop mentally with n = 0 and n = 1.
> Empty and single-element inputs break more code than any other case, and they take two minutes to check.

CHAPTER 2

How To Dry Run Properly

The method

Dry running is not "reading the code carefully". It is a mechanical procedure with a written record. Do it on paper.

1) Draw a table. One column per variable, plus a column for output.
2) Write the initial value of every variable in row one.
3) Execute exactly one line. Write a new row. Change only what that line changed.
4) At every condition, write down the actual values being compared and the resulting true or false.
5) Never skip a line because it looks obvious. Bugs live in obvious lines.

A worked example, done properly

```
int[] a = {4, 2, 7, 1};
int max = a[0];
for (int i = 1; i &lt; a.length; i++) {
    if (a[i] &gt; max) {
        max = a[i];
    }
}
System.out.println(max);
```

Line executed	i	a[i]	Condition evaluated	max	Output
`max = a[0]`	-	-	-	4	-
`i = 1; 1 < 4`	1	2	true	4	-
`if (2 > 4)`	1	2	false	4	-
`i++; 2 < 4`	2	7	true	4	-
`if (7 > 4)`	2	7	true	7	-
`i++; 3 < 4`	3	1	true	7	-
`if (1 > 7)`	3	1	false	7	-
`i++; 4 < 4`	4	-	false, exit	7	-
`println`	-	-	-	7	7

Notice the final row where `i` is 4 and the condition fails. Writing that row explicitly is what trains you to stop making off-by-one errors.

Dry running recursion

Use an **indented call log**, not a table. Indentation shows depth.

```
fact(4)
  fact(3)
    fact(2)
      fact(1) -&gt; returns 1
    returns 2 * 1 = 2
  returns 3 * 2 = 6
returns 4 * 6 = 24
```

Dry running pointer structures

Draw boxes and arrows. Redraw the whole picture after **every** pointer assignment. Do not try to hold it in your head - that is precisely the thing human working memory is bad at, and it is why linked list bugs are so common.

When dry running fails you, print

```
System.out.println("i=" + i + " sum=" + sum + " arr=" + Arrays.toString(a));
```

Put one print at the top of the loop body showing every variable that matters. Run it. Compare the real output to your paper prediction. **The first row where they differ is the exact location of your bug.** This technique has no upper limit on program size, and professionals use it constantly.

CHAPTER 3

Cheat Sheets

Syntax quick reference

```
// Declarations
int x = 5;                      double d = 3.14;
char c = 'A';                   boolean b = true;
String s = "hi";                final int MAX = 100;
int[] a = new int[5];           int[] b = {1,2,3};
int[][] m = new int[3][4];      var v = "inferred";

// Control
if (c) { } else if (c2) { } else { }
switch (x) { case 1 -&gt; "a"; default -&gt; "b"; };
for (int i = 0; i &lt; n; i++) { }
for (int v : arr) { }
while (c) { }                   do { } while (c);
break;  continue;  return v;

// Methods
public static int f(int a, String b) { return a; }
static int sum(int... nums) { }

// Class
class A { int f; A(int f){this.f=f;} int get(){return f;} }
class B extends A implements C { }
interface C { void m(); }
record P(int x, int y) { }
enum E { A, B }

// Collections
List&lt;String&gt; l = new ArrayList&lt;&gt;();
Set&lt;Integer&gt; st = new HashSet&lt;&gt;();
Map&lt;String,Integer&gt; m = new HashMap&lt;&gt;();
Deque&lt;Integer&gt; d = new ArrayDeque&lt;&gt;();
PriorityQueue&lt;Integer&gt; pq = new PriorityQueue&lt;&gt;();

// Exceptions
try { } catch (Exception e) { } finally { }
throw new IllegalArgumentException("msg");
```

Which loop, decided in one table

Situation	Loop
Known count, need the index	`for (int i = 0; i < n; i++)`
Need only the values	`for (T v : coll)`
Unknown count, condition based	`while (cond)`
Must run at least once	`do { } while (cond)`
Walking backwards	`for (int i = n-1; i >= 0; i--)`
Two ends moving inwards	`while (i < j)`
Removing while iterating	`Iterator` or `removeIf`

Which structure, decided in one table

Need	Structure	Cost
Indexed access	`ArrayList`	O(1)
Key lookup	`HashMap`	O(1)
Sorted keys or ranges	`TreeMap`	O(log n)

Need	Structure	Cost
Uniqueness	`HashSet`	O(1)
LIFO	`ArrayDeque` as stack	O(1)
FIFO	`ArrayDeque` as queue	O(1)
Highest priority first	`PriorityQueue`	O(log n)
Prefix search	`Trie`	O(L)
Connections	Adjacency list	O(V+E)

Pattern recognition: the problem says X, so use Y

The problem mentions	Reach for
"find a pair / a duplicate / seen before"	HashMap or HashSet
"sorted array"	Two pointers, or binary search
"subarray" or "substring" with a size or property	Sliding window
"top K" or "K largest / smallest"	Heap of size K
"next greater / smaller element"	Monotonic stack
"shortest path, unweighted"	BFS
"shortest path, weighted"	Dijkstra
"all paths / all combinations / all permutations"	Backtracking
"count the ways / minimum cost / optimal"	Dynamic programming
"prefix"	Trie
"range sum queried many times"	Prefix sum
"parentheses / undo / expression"	Stack
"grid"	BFS or DFS over the grid as a graph
"in place, O(1) space"	Two pointers, or swapping

A 30-day study plan

Days	Focus	Target
1-3	Types, operators, control flow	Dry run 20 small programs on paper without running them
4-6	Loops, patterns, methods	Print 10 patterns without help
7-9	Arrays	Programs 26-50 from Part 8
10-12	Strings	Programs 51-65
13-15	Recursion	Factorial, Fibonacci, Hanoi, plus binary search recursively
16-18	Sorting and searching	Write all six sorts from memory
19-21	Linked list, stack, queue	Build all three from scratch, no reference
22-24	Trees	Build a BST, all four traversals
25-26	Heap and hashing	Build a MinHeap and a HashMap by hand
27-28	Graphs	BFS, DFS, components, islands
29-30	Collections and revision	Redo any 20 programs from memory

> **THE ONLY STUDY RULE THAT MATTERS**
> Write the code by hand before you look at the solution. Every time.
> Reading a solution and understanding it feels identical to being able to write it, and it is not. The gap between recognising and producing is the entire difficulty of learning to program, and the only way to cross it is to produce.

Final word

You now have three separate things: the vocabulary (keywords, operators, types), the shapes (loops, recursion, structures), and the method (dry running and complexity analysis).

The vocabulary you will forget and look up. That is fine and normal - professionals look things up constantly.

The shapes you will keep, because they recur everywhere.

The method is the part that compounds. Someone who can dry run any program on paper and estimate its complexity can learn any language and any framework. Someone who has memorised a hundred solutions and cannot trace them has learned a hundred things instead of one.

Aim for the method.