Java Handout

English edition

exercises.lol exercises.com.cn

Contents

1 Java basics **3**
1.1 Classes, main & output . . . 3
1.2 Variables & primitive types . . . 4
1.3 Comments & style . . . 4

2 Operators & expressions **5**
2.1 Arithmetic & assignment . . . 5
2.2 Using objects: String, Math, wrappers . . . 6
2.3 Casting & type conversion . . . 6

3 Booleans & selection **8**
3.1 if / else . . . 8
3.2 Logical operators & comparisons . . . 9
3.3 switch . . . 9

4 Loops **10**
4.1 while loops . . . 10
4.2 for loops . . . 11
4.3 Accumulation . . . 11
4.4 Nested loops . . . 11

5 Strings **13**
5.1 String methods . . . 13
5.2 Building & traversing strings . . . 14

6 Arrays **16**
6.1 1-D arrays . . . 16
6.2 Array algorithms . . . 17
6.3 2-D arrays . . . 17

7 ArrayList **18**
7.1 ArrayList basics . . . 18
7.2 ArrayList algorithms & the remove bug . . . 19

8 Writing classes **20**
8.1 Fields, constructor & methods . . . 20
8.2 Encapsulation . . . 21

9 Inheritance & polymorphism **23**
9.1 Inheritance . . . 23
9.2 Polymorphism & toString . . . 24

10 Recursion **25**

11 Searching & sorting **28**
11.1 Linear & binary search 28
11.2 Selection & insertion sort 29

12 Files & the FRQ **31**
12.1 Text files with Scanner 31
12.2 The AP FRQ question types 32

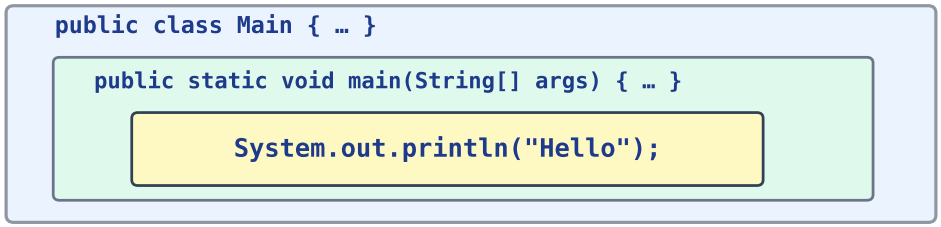
1 Java basics

1.1 ■ Classes, main & output

This site compiles Java using Java 8 language rules. These standalone examples put their entry method , `main`, inside a class . Helper classes and method-only answers do not need their own `main`. `System.out.println(...)` prints a line; `System.out.print(...)` prints with no new line.

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello, world!");
        System.out.println("I am learning Java.");
    }
}
```

- Java is compiled : the compiler checks the whole program, then it runs.
- Declarations, assignments, method calls and `return` statements end with a semicolon `;`. An `if`, `for` or `while` block uses braces; do not add a semicolon just because its closing brace appears.



Exam tip: every program lives in a class; execution starts at main; end each statement with ;

This standalone Java 8 example defines main inside a class

1.2 ■ Variables & primitive types

A variable must declare its type. Common primitive types : `int` (whole number), `double` (decimal), `boolean` (`true`/`false`), `char` (one UTF-16 code unit; some characters need two).

```
public class Main {
    public static void main(String[] args) {
        int age = 17;
        double price = 9.99;
        boolean passed = true;
        System.out.println(age + " " + price + " " + passed);
    }
}
```

1.3 ■ Comments & style

A comment is `//` (one line) or `/* ... */` (a block). Indent the code inside braces `{ }`. Class names start Capitalised; variables and methods use camelCase (lowercase first).

```
public class Main {
    public static void main(String[] args) {
        // greet the user
        String firstName = "Mei";
        System.out.println("Hi, " + firstName);
    }
}
```

Common mistakes

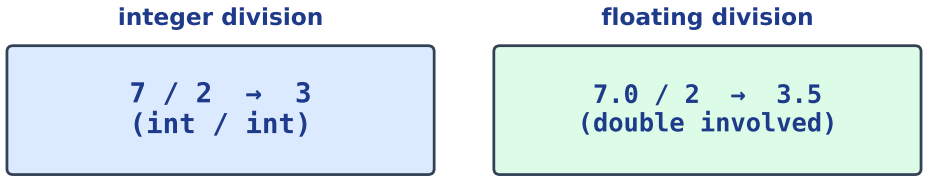
- Declarations, assignments, method calls and `return` statements end with `;`; control-flow blocks do not need an extra semicolon.
- Use the standard Java 8 entry point `public static void main(String[] args)`; the parameter name may differ, and `String... args` is also valid.
- `println` adds a new line; `print` does not.

2 Operators & expressions

2.1 ■ Arithmetic & assignment

Java arithmetic uses `+ - * /` and `%` (remainder). With two `int`s, `/` is integer division – it drops the decimal. `+=`, `-=`, and `++` are shortcuts.

```
public class Main {
    public static void main(String[] args) {
        int a = 7, b = 2;
        System.out.println(a / b);    // 3 (integer division)
        System.out.println(a % b);    // 1
        double x = 7.0 / 2;           // 3.5 (one side is double)
        System.out.println(x);
    }
}
```



Exam tip: int/int drops the decimal; make one side double for a real quotient; % is remainder

int/int truncates; involve a double for a real quotient

2.2 ■ Using objects: String, Math, wrappers

Some values are objects with methods. `String` has `.length()`, `.substring()`, `.toUpperCase()`. `Math` has `Math.max`, `Math.sqrt`, `Math.pow`. Wrapper classes (`Integer`, `Double`) wrap a primitive – e.g. `Integer.parseInt("42")`.

```
public class Main {
    public static void main(String[] args) {
        String s = "Hello";
        System.out.println(s.length());                  // 5
        System.out.println(s.toUpperCase());             // HELLO
        System.out.println(Math.max(3, 9));              // 9
        System.out.println(Integer.parseInt("42") + 1);  // 43
    }
}
```

`Math.random()` returns a random `double` from `0.0` up to (but not including) `1.0`. Scale it and cast to get whole numbers – this is the AP idiom:

```
public class Main {
    public static void main(String[] args) {
        // a random whole number from 1 to 6 (a dice roll)
        int roll = (int) (Math.random() * 6) + 1;
        System.out.println(roll >= 1 && roll <= 6);   // true
    }
}
```

- `Integer.MAX_VALUE` (2147483647) and `Integer.MIN_VALUE` are the limits of `int`; going past them wraps around (an overflow).
- An object variable that points at no object holds `null`; calling a method on it throws a NullPointerException.

2.3 ■ Casting & type conversion

A cast changes a value's type. `(int)` drops the decimal; `(double)` avoids integer division when you need an exact result.

```
public class Main {
    public static void main(String[] args) {
        double pi = 3.99;
        System.out.println((int) pi);            // 3
        int total = 7, n = 2;
        System.out.println((double) total / n);  // 3.5
    }
}
```

Common mistakes

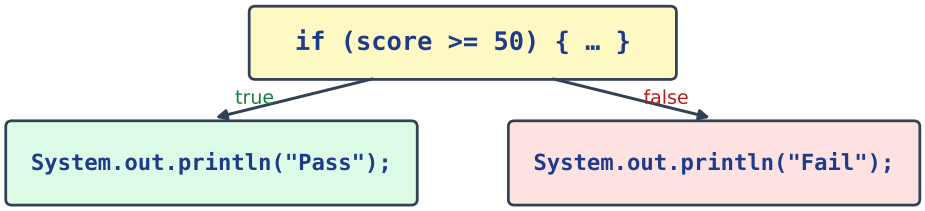
- Integer division: `5 / 2` is `2`, not `2.5`. Cast first: `(double) 5 / 2`.
- Compare Strings (and other objects) with `.equals()`, not `==`.
- `==` on two objects tests whether they are the SAME object, not whether they look equal.

3 Booleans & selection

3.1 ■ if / else

`if` runs a block when a condition is true; `else if` and `else` add more cases. The condition goes in `( )`, the block in `{ }`.

```
public class Main {
    public static void main(String[] args) {
        int score = 72;
        if (score >= 80) {
            System.out.println("A");
        } else if (score >= 60) {
            System.out.println("B");
        } else {
            System.out.println("fail");
        }
    }
}
```



Exam tip: always use braces; else if chains more cases; only one branch runs

if chooses the true branch; else the false branch

3.2 ■ Logical operators & comparisons

Compare with `==`, `!=`, `<`, `>`, `<=`, `>=` – a comparison gives a `boolean`. Combine with `&&` (and), `||` (or), `!` (not) – the logical operators . For `String`s, use `.equals(...)`, **not** `==`.

```
public class Main {
    public static void main(String[] args) {
        int age = 16;
        boolean member = true;
        System.out.println(age >= 18 && member);   // false
        String a = "hi";
        System.out.println(a.equals("hi"));        // true
    }
}
```

3.3 ■ switch

`switch` chooses among many fixed values. Use `break` to leave a case; without it, execution can continue into the next case. `default` is the fallback.

```
public class Main {
    public static void main(String[] args) {
        int day = 3;
        switch (day) {
            case 1: System.out.println("Mon"); break;
            case 3: System.out.println("Wed"); break;
            default: System.out.println("other");
        }
    }
}
```

Common mistakes

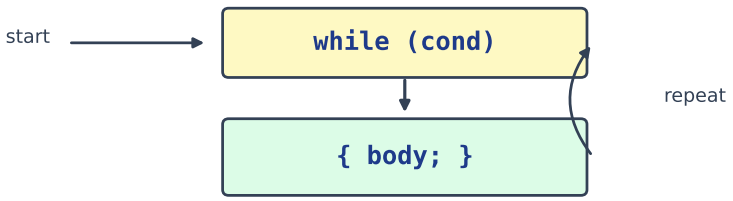
- A condition must be a `boolean`; `if (x = 5)` does not compile (use `==`).
- In a `switch`, `break`, `return` or `throw` can stop fall-through. Reaching the next case without one continues execution there.
- `&&` and `||` short-circuit. `&` and `|` operate bitwise on integers; with booleans they evaluate both operands.

4 Loops

4.1 ■ while loops

A `while` loop repeats while a condition is true. Change something inside, or it loops forever.

```
public class Main {
    public static void main(String[] args) {
        int n = 1;
        while (n <= 3) {
            System.out.println(n);
            n++;
        }
    }
}
```



Exam tip: condition is checked before each pass; body must eventually make it false

while checks the condition before each pass of the body

4.2 ■ for loops

A `for` loop packs the start, the condition, and the step into one line. Best when you know the count.

```
public class Main {
    public static void main(String[] args) {
        for (int i = 0; i < 5; i++) {
            System.out.print(i + " ");
        }
        System.out.println();   // 0 1 2 3 4
    }
}
```

4.3 ■ Accumulation

The accumulator pattern: start a variable before the loop, then update it each turn.

```
public class Main {
    public static void main(String[] args) {
        int total = 0;
        for (int i = 1; i <= 5; i++) {
            total += i;
        }
        System.out.println(total);   // 15
    }
}
```

4.4 ■ Nested loops

A loop inside a loop is a nested loop . The inner loop runs fully on each turn of the outer one.

```
public class Main {
    public static void main(String[] args) {
        for (int r = 0; r < 3; r++) {
            for (int c = 0; c < 3; c++) {
                System.out.print("*");
            }
            System.out.println();
        }
    }
}
```

Common mistakes

- `for (int i = 0; i < n; i++)` runs `n` times (`0` to `n - 1`); using `<=` runs one extra.

- Do not put a semicolon right after `for (...)` or `while (...)` – it makes an empty loop.
- Declare the counter in the `for` header so its scope ends with the loop.

5 Strings

5.1 ■ String methods

A `String` is text. Useful methods: `.length()`, `.charAt(i)`, `.substring(a, b)`, `.indexOf(x)`, `.toUpperCase()`, `.equals(...)`. Strings are immutable: their contents cannot be changed. Methods may return numbers, booleans or strings; a returned String may reuse an existing object.

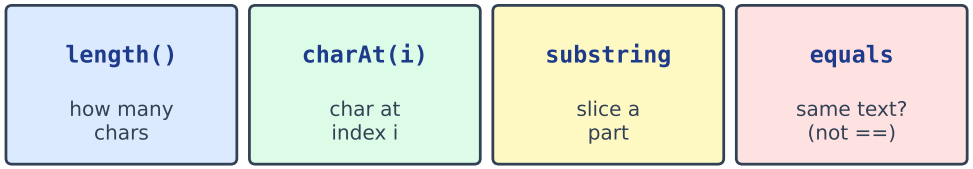
```
public class Main {
    public static void main(String[] args) {
        String s = "Python";
        System.out.println(s.length());          // 6
        System.out.println(s.charAt(0));         // P
        System.out.println(s.substring(0, 3));   // Pyt
        System.out.println(s.toUpperCase());     // PYTHON
    }
}
```

The methods you will use every day:

Method	Meaning	Example → result
`.length()`	how many characters	`"Hi".length()` → `2`
`.charAt(i)`	one character	`"Hi".charAt(0)` → `H`
`.substring(a, b)`	part, stops before `b`	`"Python".substring(0, 3)` → `Pyt`
`.substring(a)`	from `a` to the end	`"Python".substring(3)` → `hon`
`.indexOf(x)`	first position, `-1` if absent	`"banana".indexOf("na")` → `2`
`.equals(s)`	same text?	`"hi".equals("hi")` → `true`
`.compareTo(s)`	order: negative / 0 / positive	`"apple".compareTo("banana")` → negative

`compareTo` compares UTF-16 code units lexicographically, with case significant (for example, `"Z"` comes before `"a"`) – the AP exam uses it for sorting questions:

```
public class Main {
    public static void main(String[] args) {
        String a = "apple", b = "banana";
        System.out.println(a.compareTo(b) < 0);    // true (apple comes
↪   first)
        System.out.println(a.compareTo("apple"));  // 0 (equal)
    }
}
```



Exam tip: Strings are objects — use equals for text, not ==; indices start at 0

Key String methods: length, charAt, substring, equals

5.2 ■ Building & traversing strings

Join strings with `+` (concatenation). Visit each character with a loop and `.charAt(i)`.

```
public class Main {
    public static void main(String[] args) {
        String word = "banana";
        int count = 0;
        for (int i = 0; i < word.length(); i++) {
            if (word.charAt(i) == 'a') count++;
        }
        System.out.println(count);   // 3
    }
}
```

When you build a long String in a loop, `StringBuilder` is much faster: append the pieces, then call `.toString()` once.

```
public class Main {
    public static void main(String[] args) {
        StringBuilder sb = new StringBuilder();
        for (int i = 1; i <= 5; i++) {
            sb.append(i).append(" ");
        }
        System.out.println(sb.toString().trim());   // 1 2 3 4 5
    }
}
```

Common mistakes

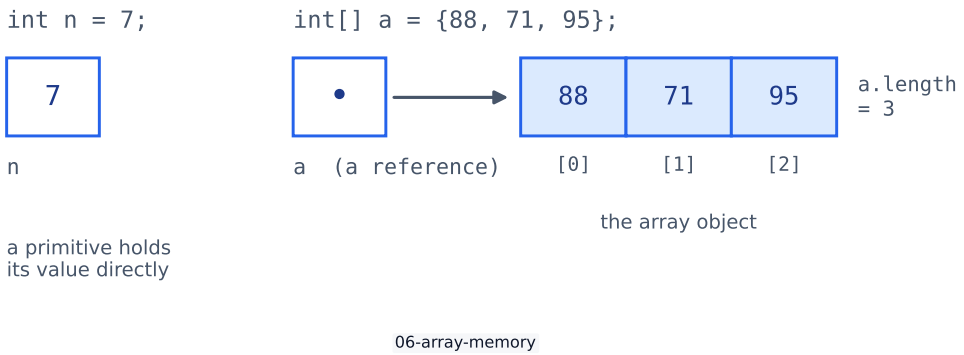
- Strings are immutable: `s.toUpperCase()` returns the upper-case result without changing `s`, so store the result.
- Read a character with `s.charAt(i)`; the length is `s.length()` (a method, with `()`).
- Building a String with `+=` in a big loop is slow; use a `StringBuilder`.

6 Arrays

6.1 ■ 1-D arrays

An array holds a fixed number of values of one type. Index from `0`, and get the size with `.length`.

```
public class Main {
    public static void main(String[] args) {
        int[] scores = {88, 71, 95};
        System.out.println(scores[0]);       // 88
        System.out.println(scores.length);   // 3
        scores[1] = 100;
        System.out.println(scores[1]);       // 100
    }
}
```



06-array-memory

A primitive holds its value; an array variable holds a reference to the array object

6.2 ■ Array algorithms

Walk the array with a loop to find a max, a total, a count, or to search . A for-each loop (`for (int x : a)`) reads each value in turn.

```
public class Main {
    public static void main(String[] args) {
        int[] a = {3, 9, 2, 7};
        int max = a[0], total = 0;
        for (int x : a) {
            if (x > max) max = x;
            total += x;
        }
        System.out.println(max + " " + total);   // 9 21
    }
}
```

6.3 ■ 2-D arrays

A 2-D array is a grid of rows and columns: `grid[row][col]`.

```
public class Main {
    public static void main(String[] args) {
        int[][] grid = {{1, 2, 3}, {4, 5, 6}};
        System.out.println(grid[1][2]);   // 6
        for (int[] row : grid) {
            for (int v : row) System.out.print(v + " ");
        }
        System.out.println();             // 1 2 3 4 5 6
    }
}
```

Common mistakes

- An array's size is `a.length` (no brackets, no `()`), and it is fixed when created.
- Valid indexes are `0` to `a.length - 1`; `a[a.length]` throws ArrayIndexOutOfBoundsException.
- A new `int[5]` is filled with zeros, not left empty.

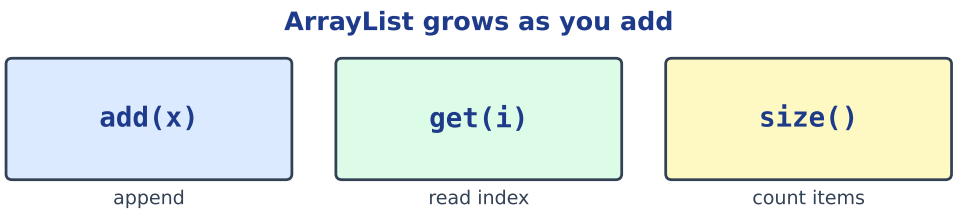
7 ArrayList

7.1 ■ ArrayList basics

An `ArrayList` is a resizable list – it grows and shrinks as you add or remove items. It stores objects, so use a wrapper type like `Integer` (not `int`). The `<Integer>` part is a generic type. Key methods: `.add(x)`, `.get(i)`, `.set(i, x)`, `.size()`, `.remove(i)`.

```
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        ArrayList<Integer> nums = new ArrayList<Integer>();
        nums.add(10);
        nums.add(20);
        nums.add(30);
        System.out.println(nums.size());   // 3
        System.out.println(nums.get(1));   // 20
        nums.set(0, 99);
        System.out.println(nums);          // [99, 20, 30]
    }
}
```



Exam tip: ArrayList<Type> needs a type; indices start at 0 like arrays

ArrayList: add, get, size on a resizable list

7.2 ■ ArrayList algorithms & the remove bug

`.remove(i)` shifts every later element one place left. If you remove while counting `i` **up**, you skip the next element. Fix: loop **backwards**, or don't increment `i` after a remove.

```
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        ArrayList<Integer> nums = new ArrayList<Integer>();
        for (int n : new int[]{4, 7, 4, 9, 4}) nums.add(n);
        // Remove every 4 - loop backwards so removals don't skip items.
        for (int i = nums.size() - 1; i >= 0; i--) {
            if (nums.get(i) == 4) nums.remove(i);
        }
        System.out.println(nums);   // [7, 9]
    }
}
```

Common mistakes

- `ArrayList` uses `.size()`, `.get(i)` and `.add(...)` – not the `[]` you use on arrays.
- Removing items while looping forward by index skips the next item (the remove bug). Loop backwards, or use an iterator.
- Store objects, not primitives: use `ArrayList<Integer>`, and Java auto-boxes `int` values.

8 Writing classes

8.1 ■ Fields, constructor & methods

A class is a blueprint for objects. Its fields store data, its constructor sets up a new object, and its methods are the actions. `this.name` means "this object's `name`". Create an object with `new`.

```
public class Main {
    public static void main(String[] args) {
        Dog d = new Dog("Rex", 3);
        System.out.println(d.describe());   // Rex is 3 years old
        d.haveBirthday();
        System.out.println(d.describe());   // Rex is 4 years old
    }
}

class Dog {
    private String name;
    private int age;

    public Dog(String name, int age) {   // constructor
        this.name = name;
        this.age = age;
    }

    public String describe() {
        return name + " is " + age + " years old";
    }

    public void haveBirthday() {
        age++;
    }
}
```

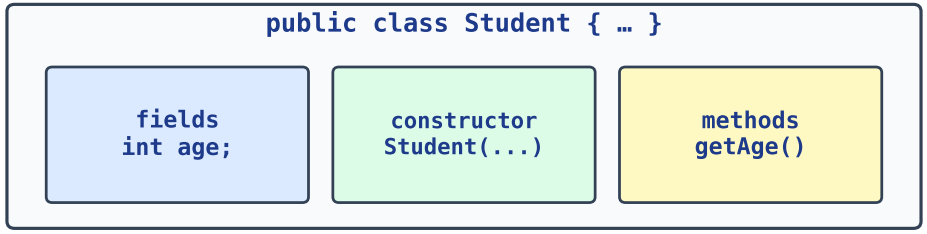
A `static` member belongs to the class itself, not to any one object. Call a static method on the class name – like `Math.max` – and a static field is shared by every object:

```
public class Main {
    public static void main(String[] args) {
        System.out.println(Counter.made());   // 0
        new Counter();
        new Counter();
        System.out.println(Counter.made());   // 2
    }
}

class Counter {
    private static int count = 0;      // shared by ALL Counter objects

    public Counter() { count++; }

    public static int made() {         // called on the class:
↪   Counter.made()
        return count;
    }
}
```



Exam tip: constructor name matches the class; fields store state; methods do work

Fields store state; constructor builds; methods act

8.2 ■ Encapsulation

Encapsulation controls access to object state. Mark fields `private` to block direct access from unrelated classes; an accessor (getter) reads state and a mutator changes it. A setter is one kind of mutator. `private` does not validate values: constructors and mutating methods need their own checks or stated preconditions. This example assumes a nonnegative `start` and additions that fit in `int`; its deposit

method rejects nonpositive amounts.

```
public class Main {
    public static void main(String[] args) {
        Account a = new Account(100);
        a.deposit(50);
        a.deposit(-999);                        // rejected by the guard
        System.out.println(a.getBalance());     // 150
    }
}

class Account {
    private int balance;                        // hidden from outside

    public Account(int start) {
        balance = start;
    }

    public void deposit(int amount) {
        if (amount > 0) balance += amount;      // this guard checks the
↪   deposit amount
    }

    public int getBalance() {                   // accessor (getter)
        return balance;
    }
}
```

Common mistakes

- A constructor has the class name and no return type (not even `void`).
- Use `this.field` to tell a field apart from a parameter with the same name.
- Make fields `private` and reach them through getter/setter methods (encapsulation).

9 Inheritance & polymorphism

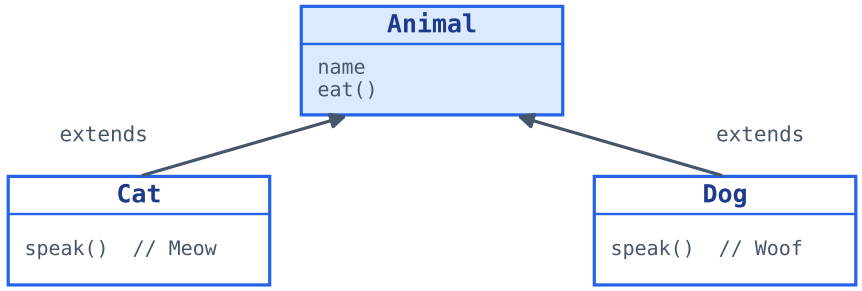
9.1 ■ Inheritance

Inheritance lets a subclass reuse a superclass. Write `class Cat extends Animal` and `Cat` inherits eligible fields and methods from `Animal`. Private members are not inherited; constructors are not inherited either. Call the parent constructor with `super(...)`.

```
public class Main {
    public static void main(String[] args) {
        Cat c = new Cat("Milo");
        c.eat();      // Milo is eating  (inherited from Animal)
        c.speak();    // Meow            (Cat's own method)
    }
}

class Animal {
    protected String name;
    public Animal(String name) { this.name = name; }
    public void eat() { System.out.println(name + " is eating"); }
}

class Cat extends Animal {
    public Cat(String name) { super(name); }     // call Animal's
↪   constructor
    public void speak() { System.out.println("Meow"); }
}
```



Cat and Dog inherit name and eat(), and add their own speak()

09-inheritance

Cat and Dog extend Animal: they inherit its members and add their own

9.2 ■ Polymorphism & toString

A subclass can override a method to replace the parent's version. Polymorphism means a `Shape` variable can hold any subtype, and Java picks the right `toString` at run time. `System.out.println(obj)` automatically calls `obj.toString()`.

```
public class Main {
    public static void main(String[] args) {
        Shape[] shapes = { new Circle(2), new Square(3) };
        for (Shape s : shapes) {
            System.out.println(s);          // each calls its own
↪   toString
        }
    }
}

class Shape {
    public String toString() { return "a shape"; }
}

class Circle extends Shape {
    private int r;
    public Circle(int r) { this.r = r; }
    public String toString() { return "Circle r=" + r; }     // override
}

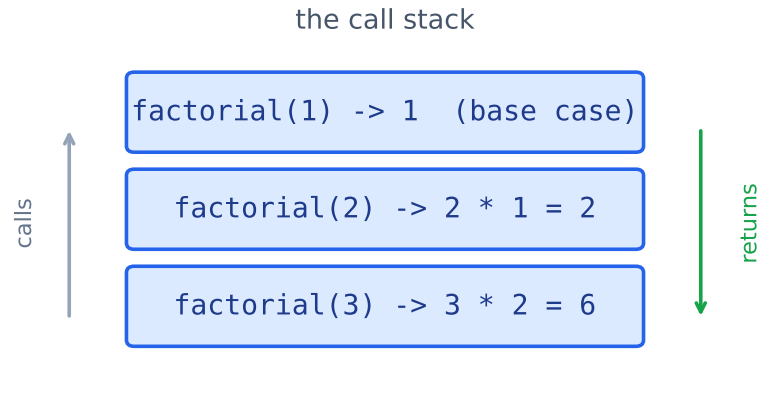
class Square extends Shape {
    private int side;
    public Square(int side) { this.side = side; }
    public String toString() { return "Square side=" + side; }   //
↪   override
}
```

Common mistakes

- An overriding method must match the signature exactly; add `@Override` so the compiler catches slips.
- `super(...)` must be the first line of a subclass constructor.
- A subclass object IS-A superclass object, but not the other way round.

10 Recursion

Recursion is a method that calls itself. Every recursion needs a base case (when to stop) and a recursive call that moves toward it. Without a base case it never stops and crashes with a *stack overflow*.



The call stack for factorial(3): each call waits, then returns in reverse order

```
public class Main {
    public static void main(String[] args) {
        System.out.println(factorial(5));   // 120
    }

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

Trace it: `factorial(5)` waits for `factorial(4)`, which waits for `factorial(3)`… down to `factorial(1)` returning `1`. Then the answers multiply back up: `1 → 2 → 6 → 24 → 120`.

Recursion also works on Strings – peel off one character each call:

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

    static String reverse(String s) {
        if (s.length() <= 1) return s;               // base case
        return reverse(s.substring(1)) + s.charAt(0);
    }
}
```

Merge sort is the recursive sort on the AP exam: split the array in half, sort each half recursively, then merge the two sorted halves. It runs in O(n log n) – far faster than the $O(n^2)$ sorts on big arrays.

```
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        int[] a = {5, 2, 9, 1, 7, 3};
        mergeSort(a, 0, a.length - 1);
        System.out.println(Arrays.toString(a));   // [1, 2, 3, 5, 7, 9]
    }

    static void mergeSort(int[] a, int lo, int hi) {
        if (lo >= hi) return;                 // base case: one element
        int mid = (lo + hi) / 2;
        mergeSort(a, lo, mid);                // sort the left half
        mergeSort(a, mid + 1, hi);            // sort the right half
        merge(a, lo, mid, hi);                // merge the two halves
    }

    static void merge(int[] a, int lo, int mid, int hi) {
        int[] tmp = new int[hi - lo + 1];
        int i = lo, j = mid + 1, k = 0;
        while (i <= mid && j <= hi) {
            if (a[i] <= a[j]) tmp[k++] = a[i++];
            else              tmp[k++] = a[j++];
        }
        while (i <= mid) tmp[k++] = a[i++];
        while (j <= hi)  tmp[k++] = a[j++];
        for (k = 0; k < tmp.length; k++) a[lo + k] = tmp[k];
    }
}
```


Common mistakes

- Recursion needs a base case, or it throws StackOverflowError.
- Each recursive call must move CLOSER to the base case.
- Trace a small example by hand to check the recursion returns the right value.
- In merge sort, the merge step does the real work; the recursion only splits the array.

11 Searching & sorting

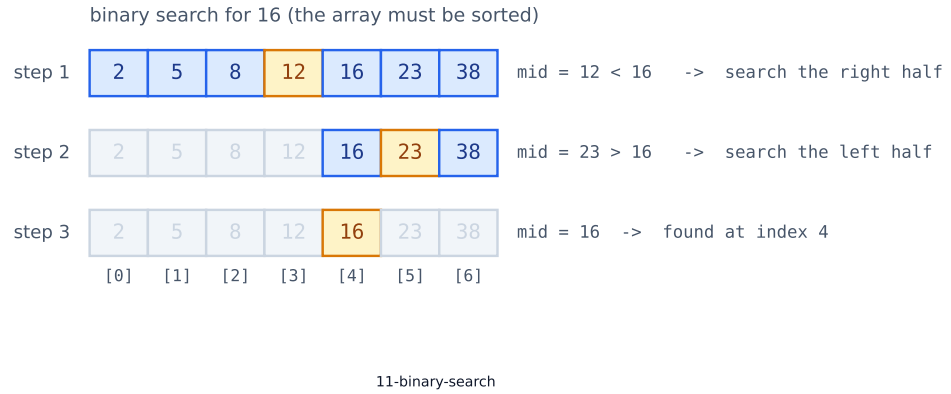
11.1 ■ Linear & binary search

Linear search checks every element – works on any array. Binary search is much faster but needs a **sorted** array: it looks at the middle, then throws away half each step. Both return the index, or `-1` if not found.

```
public class Main {
    public static void main(String[] args) {
        int[] a = {2, 5, 8, 12, 16, 23};      // sorted, so binary search
↪   works
        System.out.println(linear(a, 12));    // 3
        System.out.println(binary(a, 12));    // 3
        System.out.println(binary(a, 9));     // -1 (not found)
    }

    static int linear(int[] a, int target) {
        for (int i = 0; i < a.length; i++)
            if (a[i] == target) return i;
        return -1;
    }

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



11-binary-search

Binary search halves the range each step, so a sorted array is searched in O(log n)

11.2 ■ Selection & insertion sort

Selection sort repeatedly finds the smallest remaining value and swaps it to the front. Insertion sort takes each value and slides it back into its place among the already-sorted values.

```
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        int[] a = {5, 2, 9, 1, 7};
        for (int i = 0; i < a.length - 1; i++) {
            int min = i;
            for (int j = i + 1; j < a.length; j++)
                if (a[j] < a[min]) min = j;
            int t = a[min]; a[min] = a[i]; a[i] = t;   // swap into place
        }
        System.out.println(Arrays.toString(a));   // [1, 2, 5, 7, 9]
    }
}
```

```java
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        int[] a = {5, 2, 9, 1, 7};
        for (int i = 1; i < a.length; i++) {
            int key = a[i], j = i - 1;
            while (j >= 0 && a[j] > key) {    // shift bigger values
↪   right
                a[j + 1] = a[j];
                j--;
            }
            a[j + 1] = key;                   // drop key into the gap
        }
        System.out.println(Arrays.toString(a));   // [1, 2, 5, 7, 9]
    }
}
```

How the speeds compare:

Algorithm	Time
linear search	O(n)
binary search	O(log n), sorted arrays only
selection / insertion sort	$O(n^2)$
merge sort (topic 10)	O(n log n)

Common mistakes

- Binary search only works on a **sorted** array.
- Linear search is O(n); binary search is O(log n) but needs the sort first.
- Selection and insertion sort are $O(n^2)$ – clear to learn, slow on big data.

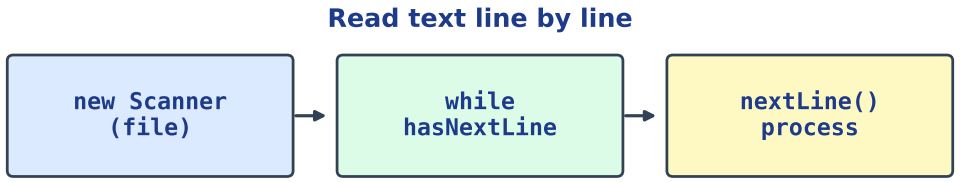
12 Files & the FRQ

12.1 ■ Text files with Scanner

A `Scanner` reads text one line at a time. For a real file you write `new Scanner(new File("scores.txt"))`; here we wrap a `String` so the example runs anywhere. Use `.split(" ")` to split a line into parts and `Integer.parseInt(...)` to parse a number from text.

```java
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        // Real file:  Scanner in = new Scanner(new File("scores.txt"));
        String data = "Alice 80\nBob 95\nCara 72";
        Scanner in = new Scanner(data);
        int total = 0, count = 0;
        while (in.hasNextLine()) {
            String line = in.nextLine();
            String[] parts = line.split(" ");      // break the line on
↪   the space
            total += Integer.parseInt(parts[1]);
            count++;
        }
        System.out.println("average = " + (total / count));   // average
↪   = 82
    }
}
```



Exam tip: hasNextLine guards the loop; split + parseInt turn a line into data

Scanner reads lines while hasNextLine is true

12.2 ■ The AP FRQ question types

The AP CS A exam has four free-response questions, each a fixed shape:

- **Q1 – Methods & control:** write methods to a given spec; loops, `if`, `String`/`Math`.
- **Q2 – Class design:** write a full class (fields, constructor, methods) from a description.
- **Q3 – Array / ArrayList:** process a 1-D array or `ArrayList` (search, count, build a new list).
- **Q4 – 2-D array:** traverse a grid by row and column.

The skill is always the same: read the spec, write the method exactly as described, return the right type.

```
public class Main {
    public static void main(String[] args) {
        // Q1 style: implement a method to a spec, then it is tested.
        System.out.println(countEven(new int[]{4, 7, 10, 3, 6}));   // 3
    }

    /** Returns how many values in arr are even. */
    public static int countEven(int[] arr) {
        int count = 0;
        for (int x : arr)
            if (x % 2 == 0) count++;
        return count;
    }
}
```

Common mistakes

- `nextInt()` leaves the newline behind, so a following `nextLine()` reads an empty line – read it away first.
- Check `hasNext()` before reading, to avoid running off the end of the file.
- In the FRQ, read the method header carefully: match the return type and parameters exactly.