Java Recall Book (Modules 0–7)

Prepared for quick revision

August 25, 2025

Contents

1 Module 0: Introduction to Java Basics

1.1 Variables

A **variable** names a memory location and has a declared type. Java is *statically and strongly typed*: the compiler enforces types.

```
int count = 5;           // integer
double price = 19.99;    // floating-point
String user = "Alice";   // text (reference type)
count = count + 1;       // ok, same type
```

Why it matters: correct types catch errors early; names make code readable.

1.2 Data Types

Primitive: byte, short, int, long, float, double, char, boolean.
Reference: String, arrays, classes, interfaces, records, etc.

```
boolean ok = true;
char grade = 'A';
long big = 1_000_000_000L; // suffix L for long literal
float f = 3.14f;           // suffix f for float literal
```

Tip: use `int` for counters, `double` for real numbers, `boolean` for flags.

1.3 Printing & Comments

Use `System.out.print/println/printf`. Comments are for humans.

```
// Single-line comment
/* Multi-line
   comment */
System.out.println("Hi");          // newline
System.out.print("No newline");    // no newline
System.out.printf("x=%d y=%.2f\n", 3, 2.5);
```

Why: printing is your first debugger; comments explain intent.

Slide Summary (Basics)

- Variables: name + type; Java enforces types at compile-time.
- Primitive vs Reference types; literals and suffixes (`L`, `f`).
- Printing: `print`, `println`, `printf`; comments for clarity.

Concise Notes (Basics)

- Prefer meaningful names: `total`, `isEmpty`.
- Use `final` for constants: `final double PI = 3.14159;`.
- `String` is immutable; concatenation makes new objects.

1.4 Operators

Arithmetic: `+ - * / %`. **Relational:** `== != > < >= <=`. **Logical:** `&&` — !—. **Assignment:** `= += -= *= /= %=`.

```java
int a = 7, b = 3;
System.out.println(a / b);    // 2 (integer division)
System.out.println(a % b);    // 1 (remainder)
System.out.println(a / 3.0); // 2.333... (double)
boolean t = (a > b) && (b > 0);
```

Pitfall: integer division truncates; cast if you need `double`.

1.5 User Input

Use `Scanner` from `java.util` to read from standard input.

```java
import java.util.Scanner;
Scanner sc = new Scanner(System.in);
System.out.print("Age: ");
int age = sc.nextInt();
sc.nextLine(); // consume end of line
System.out.print("Name: ");
String name = sc.nextLine();
System.out.println(name + " is " + age);
```

Note: after reading numbers, call `nextLine()` to consume the newline.

1.6 Errors in Java

Compile-time: syntax/type errors caught by `javac`.
Runtime: exceptions while running.
Logic: program runs but output is wrong.

```java
int[] a = new int[3];
System.out.println(a[3]); // Runtime: ArrayIndexOutOfBoundsException
```

Fix mindset: read error messages; minimize code to reproduce; add prints/tests.

1.7 Branching

if/else chooses between paths; **switch** is clean for many discrete cases.

```java
int score = 85;
if (score >= 90) {
  System.out.println("A");
} else if (score >= 80) {
  System.out.println("B");
} else {
  System.out.println("C");
}

int day = 2;
switch (day) {
  case 1 -> System.out.println("Mon");
  case 2 -> System.out.println("Tue");
  default -> System.out.println("Other");
}
```

Tip: use enhanced switch (arrow syntax) in Java 14+.

Slide Summary (Control Flow)

- Use `if/else` for ranges; `switch` for distinct labels.
- Guard against invalid states early (*fail fast*).

Concise Notes (Control Flow)

- Comparing `String`: use `equals`, not `==`.
- Chain conditions from most specific to least.

1.8 While Loops

Loop while condition is true; ensure progress to avoid infinite loops.

```
int i = 0;
while (i < 3) {
  System.out.println(i);
  i++;
}
```

1.9 For Loops

Counted loops and enhanced-for for collections/arrays.

```
for (int i = 1; i <= 5; i++) {
  System.out.print(i + " ");
}
int[] arr = {2,4,6};
for (int x : arr) { System.out.println(x); }
```

Tip: use enhanced-for when you don't need the index.

1.10 Strings

`String` is immutable; methods return new strings.

```
String s = "Java";
System.out.println(s.length());        // 4
System.out.println(s.toUpperCase());   // "JAVA"
System.out.println(s.charAt(0));       // 'J'
System.out.println(s.substring(1,3)); // "av"
```

Pitfall: concatenating in loops is slow; use `StringBuilder`.

Slide Summary (Strings)

- Strings are objects; operations create new instances.
- Compare with `equals`, not `==` (which compares references).

Concise Notes (Strings)

- Useful: `contains`, `indexOf`, `trim`, `replace`.
- Build text: `StringBuilder sb = new StringBuilder(); sb.append("...");`

2 Module 1: Array

2.1 Array Basics

Arrays hold a fixed number of elements of the same type; indices start at 0.

```
int[] a = {10, 20, 30};
System.out.println(a[0]);      // 10
System.out.println(a.length); // 3
```

Create then fill:

```
int[] b = new int[3]; // {0,0,0}
for (int i = 0; i < b.length; i++) { b[i] = i*i; }
```

2.2 Iteration Patterns

```
int sum = 0;
for (int i = 0; i < a.length; i++) sum += a[i];
for (int x : a) sum += x; // enhanced-for
```

Common tasks: min/max, search, reverse, copy.

2.3 Pitfalls

- Out-of-bounds: valid indices are $0..length-1$.
- Arrays are reference types: assignment copies the reference, not elements.

```
int[] x = {1,2};
int[] y = x;    // y points to same array
y[0] = 99;      // x[0] now 99 as well
```

Slide Summary (Arrays)

- Fixed size; use `ArrayList` for dynamic size.
- Enhanced-for for read-only traversal; classic for when you need indices.

Concise Notes (Arrays)

- Copy elements: `Arrays.copyOf`, `System.arraycopy`.
- Multi-d arrays: `int[][] m = new int[r][c];` (jagged allowed).

3 Module 2: Intro to OOP

3.1 OOP Basics

Key ideas: **abstraction** (focus on essentials), **encapsulation** (hide details), **inheritance** (reuse via `extends`), **polymorphism** (many forms via overrides).

3.2 Instance Variables

State stored per object; declared inside class but outside methods.

```
class Point {
  int x; // default 0
  int y; // default 0
}
Point p = new Point();
p.x = 3; p.y = 4;
```

3.3 Instance Methods

Behavior that often reads/modifies instance variables.

```
class Counter {
  private int value; // encapsulated
  void inc() { value++; }
  int get() { return value; }
}
Counter c = new Counter();
c.inc(); System.out.println(c.get());
```

4 Module 3: Instance Method and Designing Class

4.1 Designing a Class

Choose fields to represent state; provide constructors; expose intent-revealing methods; override `toString()`.

```
class BankAccount {
  private String owner;
  private double balance;

  BankAccount(String owner, double initial) {
    this.owner = owner;
    this.balance = initial;
  }

  void deposit(double amount) { balance += amount; }
  boolean withdraw(double amount) {
    if (amount <= balance) { balance -= amount; return true; }
    return false;
  }

  public String toString() {
    return owner + " has $" + balance;
  }
}
```

Example use:

```
BankAccount a = new BankAccount("Alice", 100);
a.deposit(50);
if (!a.withdraw(200)) System.out.println("Insufficient");
System.out.println(a); // calls toString
```

4.2 Instance Methods: Tips

- Use verbs: `deposit`, `withdraw`, `move`.
- Keep methods small; one responsibility.
- Prefer returning results instead of printing inside methods.

5 Module 4: Method Overloading

5.1 Concept

Multiple methods with the *same name* but *different parameter lists* in the same class. Resolved at **compile-time** by the best match.

```
class MathUtil {
  static int add(int a, int b) { return a + b; }
  static double add(double a, double b) { return a + b; }
  static int add(int a, int b, int c) { return a + b + c; }
}
System.out.println(MathUtil.add(2,3));        // int
System.out.println(MathUtil.add(2.5,3.1));    // double
System.out.println(MathUtil.add(1,2,3));      // 3-args
```

Rules: parameter number/types/order must differ; return type alone is not enough.

5.2 Ambiguity, Varargs, Autoboxing

```
class Demo {
  static void show(int x) { }
  static void show(Integer x) { }
  static void show(int... xs) { }
}
Demo.show(5);       // picks int version
Demo.show(Integer.valueOf(5)); // Integer version
Demo.show();        // varargs (empty)
```

Tip: avoid overloads that are too similar; can confuse readers.

6 Module 5: Constructor & Pass by Reference

6.1 Constructors

Special methods to initialize new objects; no return type; name = class.

```
class Person {
  String name; int age;
  Person() { this("Unknown", 0); } // no-arg delegates
  Person(String name, int age) {
    this.name = name; this.age = age;
  }
}
Person p1 = new Person();
Person p2 = new Person("Bob", 30);
```

Overloading: provide multiple ways to create valid objects.

6.2 Pass by Value (References)

Java is pass-by-value. For objects, the *value passed* is the reference. Methods can mutate the object via the reference, but reassigning the parameter doesn't affect the caller's reference.

```
class Box { int n; }

void bump(Box b) { b.n++; }         // mutates caller's object
void reassign(Box b) { b = new Box(); b.n = 99; } // local only

Box x = new Box(); x.n = 1;
bump(x);           // x.n -> 2
reassign(x);       // x still refers to original; x.n stays 2
```

Mental model: the method gets a copy of the reference.

7 Module 6: Access Modifiers & Encapsulation

7.1 Access Modifiers

public (everywhere), **protected** (package + subclasses), ***default*** (package-private), **private** (within class).

```
package bank;
public class Account {
  private double balance;        // hidden
  public double getBalance() { return balance; }
  protected void audit() { /* for subclasses */ }
  void packageHelper() { /* package-private */ }
}
```

Tip: default to `private` fields; open what you must.

7.2 Encapsulation

Hide representation; expose operations. Use getters/setters to control invariants.

```
class Temperature {
  private double celsius;
  public double getCelsius() { return celsius; }
  public void setCelsius(double c) {
    if (c < -273.15) throw new IllegalArgumentException("below absolute
        zero");
    this.celsius = c;
  }
}
```

Benefit: you can change internals without breaking callers.

8 Module 7: Static Variable & Static Method

8.1 Static Variables

Belong to the *class*, shared by all instances. Useful for constants and shared counters.

```
class Id {
  private static int next = 1;        // shared counter
  private int id = next++;
  public int getId() { return id; }
}
Id a = new Id(); Id b = new Id();
System.out.println(a.getId()); // 1
System.out.println(b.getId()); // 2
```

Constants:

```
class MathConst { public static final double PI = 3.1415926535; }
```

8.2 Static Methods

Called on the class; do not use instance state. Great for utilities, factories, and pure functions.

```
class Util {
  static int clamp(int x, int lo, int hi) {
    if (x < lo) return lo; if (x > hi) return hi; return x;
  }
}
int c = Util.clamp(15, 0, 10); // 10
```

Notes:

- Static methods cannot access instance fields directly.
- Prefer instance methods when behavior depends on object state.

Final Quick Tips

- Read errors carefully; they usually tell you the file, line, and cause.
- Keep methods small and single-purpose; name them by intent.
- Write tests and small experiments (`main`) to confirm behavior.