Object-Oriented Programming

Tripos Revision Guide

CST Part IA — Paper 1

Complete coverage of Java, Encapsulation, Memory, Inheritance,
Polymorphism, Generics, Collections, Exceptions, Design Patterns and Streams

Topic Overview

Topic	Key Content
Intro to Java & the JVM	Bytecode, portability, jshell, primitives vs references
Objects, Classes & Encapsulation	Fields, constructors, static, SRP, cohesion, immutability, access modifiers
Memory Model	Stack vs heap, pass-by-value, pointers vs references, box-and-arrow
Inheritance & Polymorphism	extends/implements, overriding, abstract classes, interfaces, dynamic dispatch
OOP Principles	Open–Closed Principle, Liskov Substitution Principle
Object Lifecycle	Garbage collection, mark-and-sweep, cloning, copy constructors
Collections & Comparison	Lists, Sets, Maps, equals/hashCode, Comparable/Comparator
Generics	Type erasure, bounded types, wildcards, covariance
Coupling, Errors & Exceptions	Checked/unchecked exceptions, assertions, boxing
Design Patterns	Composite, Decorator, State, Strategy, Singleton, Observer
Lambdas & Streams	Functional interfaces, method references, pipelines

Tripos years covered: 2010–2025 • Full worked examples • Model answer templates

July 10, 2026

0 Contents

1 Motivations, Languages & Intro to Java

1.1 Why OOP?

As software grows, the dominant cost is not writing it but **maintaining** it. OOP is a set of tools for battling complexity by organising a program as a collection of interacting **objects**, each bundling state (data) with the behaviour (methods) that operates on it.

Maintainability Wishlist

Good code should be: **readable**, **modular** (changes stay local), **reusable**, **robust** (fails gracefully), and **testable**. OOP's core mechanisms — encapsulation, inheritance, polymorphism — are tools towards these goals, not goals in themselves.

TRAP: OOP dogma

Two warnings the notes stress: (1) real languages are rarely "pure" OOP — Java has primitives, static methods and lambdas that break the pure object model; (2) OOP is not *always* appropriate — for small scripts or heavily numerical/functional code, procedural or functional styles may be simpler. A Tripos question that asks you to critique OOP expects you to know its costs (boilerplate, indirection, over-engineering) as well as its benefits.

1.2 Imperative vs Functional Languages

- **Imperative** (e.g. Java, C): programs are sequences of statements that mutate state; execution order matters.
- **Functional** (e.g. OCaml, Haskell): programs are expressions built from pure functions; no (or minimal) mutable state; favours recursion over loops.

Java is fundamentally imperative and object-oriented but has absorbed functional features (lambdas, streams) since Java 8.

1.3 The Java Virtual Machine

Traditional vs Java Compilation Model

Traditional model: source → compiler → *platform-specific* machine code. Fast, but you must recompile per platform.
Java model: source → compiler (`javac`) → *platform-independent* bytecode (`.class`) → JVM interprets/JIT-compiles bytecode to native machine code at runtime. "Write once, run anywhere."

Pros: portability, memory safety (no raw pointers), automatic garbage collection, rich standard library. **Cons:** historically slower start-up, an extra runtime dependency (the JVM must be installed), some execution overhead versus native code (mitigated heavily by JIT compilation).

Aside: Python does this too

CPython compiles source to platform-independent bytecode (`.pyc`) which is then interpreted by the Python VM — the same two-stage idea as Java, just without an explicit separate compile step exposed to the user.

1.4 Practicalities: javac, java, jshell

Compiling and Running Java

```
javac HelloWorld.java // produces HelloWorld.class (bytecode)
java HelloWorld // JVM loads and runs the bytecode
jshell // interactive REPL - great for quick experiments
```

Java "Hello World"

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, world!");
```

```
    }
}
```

Note the `public class` name must match the filename (`HelloWorld.java`); `main` is the JVM's entry point and must be exactly `public static void main(String[] args)`.

2 Representing State: Variables, Types, Arrays

2.1 Primitive vs Reference Types

The Two Kinds of Java Type

Primitives (`int, long, short, byte, float, double, boolean, char`): fixed-size, stored directly (by value), never `null`, live on the stack (or inline in an object on the heap).
Reference types (classes, arrays, interfaces, `String`): a variable holds a *reference* (pointer-like handle) to an object that lives on the heap. Can be `null`. Assignment copies the reference, not the object.

2.2 Naming Conventions

Java uses `camelCase` for variables/methods, `PascalCase` for classes/interfaces, and `UPPER_SNAKE_CASE` for constants (`static final`). This matters for the Tripos: following convention is often explicitly marked, and mixing styles (e.g. Python's `snake_case`) inside Java code is a common trap in "spot the error" questions.

2.3 Type Conversion and Inference

Widening (implicit, safe): `int` → `long` → `float` → `double`. **Narrowing** (must be explicit, may lose precision): `(int) someDouble`.

`var` performs **local variable type inference** — the compiler infers the static type from the initialiser; it is still statically typed (unlike dynamic typing), just with less boilerplate:

```
var list = new ArrayList<String>(); // inferred as ArrayList<String>
```

2.4 Arrays

Arrays are fixed-size, homogeneous, reference-type containers indexed from 0. `int[] a = new int[5];` allocates a zero-initialised array on the heap; `a.length` gives its size (a field, not a method).

TRAP: Arrays are objects

Because arrays are reference types, `int[] b = a;` copies the *reference*, so `b` and `a` alias the same underlying array — mutating through `b` is visible through `a`. This is a very common Tripos "explain the output" trap (see §5 on pass-by-value).

2.5 Functions, Overloading

Java's "functions" are really **methods** (procedures attached to a class). **Overloading** lets multiple methods share a name if their **parameter signatures** differ (type or arity) — return type alone is *not* sufficient to distinguish overloads.

TRAP: Overload resolution

```
int max(int a, int b) { return a > b ? a : b; }
float max(int a, int b) { return a > b ? a : b; } // ILLEGAL: same erasure signature
```

This fails to compile: overload resolution only looks at parameter types, so this is treated as a duplicate method declaration.

3 Objects and Classes

3.1 Core Terminology

Object vs Class

A **class** is a blueprint: it declares the fields (state) and methods (behaviour) that its instances will have. An **object** is a concrete instance of a class, allocated on the heap at runtime via `new`. Loosely: class = type, object = value of that type.

3.2 Class Structure

```
public class BankAccount {
    private double balance; // field
    private final String owner; // final field: set once

    public BankAccount(String owner, double balance) { // constructor
        this.owner = owner;
        this.balance = balance;
    }

    public void deposit(double amount) { // instance method
        balance += amount;
    }
}
```

The constructor's job is to establish the class **invariant** — the set of conditions that must hold for every valid object of the class (e.g. `balance >= 0`).

3.3 Overloading Constructors

Multiple constructors may co-exist if their parameter lists differ, letting objects be built from different combinations of information. Use `this(...)` to delegate to another constructor and avoid duplicated initialisation logic:

```
public BankAccount(String owner) {
    this(owner, 0.0); // delegate to the two-arg constructor
}
```

3.4 Static Fields and Methods

Static vs Instance

`static` members belong to the **class itself**, not to any one instance — there is exactly one copy, shared by all objects, accessible without creating an instance (`ClassName.member`).
Typical uses: constants (`static final`), counters shared across instances, utility/helper methods (`Math.sqrt`), and factory methods.

TRAP: static cannot see instance state

A `static` method has no implicit `this`, so it cannot directly access instance (non-static) fields or call instance methods — only other static members. This is a favourite "why doesn't this compile" Tripos trap.

4 Class Design and Encapsulation

4.1 Rookie Error: The God/Monster Class

A class that tries to do everything (parse input, compute, store state, render output, log...) is low quality even if it "works". Good OOP design distributes responsibility across small, focused, cooperating classes.

4.2 UML Class Diagrams

A class box has three compartments: **name**, **fields** (with visibility +/-/# for public/private/protected), and **methods**. Relationships: a plain line is an **association**; a diamond-ended line is **aggregation/composition** ("has-a"); a hollow triangle arrow is **inheritance** ("is-a"); a dashed arrow is a **dependency**.

4.3 The has-a Association

Where inheritance models "is-a", **composition** models "has-a": a class holds a reference to another class as a field. Composition is generally preferred over inheritance for code reuse because it is more flexible (can be changed at runtime, avoids deep fragile hierarchies) — "favour composition over inheritance".

4.4 Single Responsibility Principle (SRP)

SRP

A class should have **one, and only one, reason to change**. If a class mixes unrelated responsibilities (e.g. parsing a file *and* rendering a UI *and* persisting to a database), a change to any one concern risks breaking the others.

Benefits: easier to understand, test in isolation, reuse, and modify without unintended side effects. A common architectural application of SRP is **Model-View-Controller (MVC)**: the Model owns state/logic, the View renders it, the Controller mediates user input — each has a single responsibility.

4.5 Cohesion

Cohesion measures how strongly the responsibilities *within* a single class belong together. High cohesion (good): all fields/methods work together towards one purpose. Low cohesion (bad): a class is a random grab-bag of unrelated functionality.

- **Good ways to get cohesion:** group by a single well-defined concept; keep methods operating on the same core state.
- **Less good ways:** grouping purely because things happen at the same *time* (temporal cohesion) or are merely called from the same place (logical cohesion) — these often indicate the class should be split.

SRP vs Cohesion

SRP is about *why* a class would change (external reasons); cohesion is about *how well its internals relate* to each other. They are closely linked — low cohesion classes usually violate SRP — but a Tripos question may ask you to distinguish them explicitly.

4.6 Encapsulation (OOP Principle 1)

Core Idea

Encapsulation bundles data with the methods that operate on it, and **hides** the internal representation from the outside world ("information hiding"), exposing only a controlled public interface. Callers depend on *what* an object can do, not *how* it does it.

Encapsulation Example

```
public class Temperature {
    private double celsius; // representation hidden

    public double getFahrenheit() { // controlled access
        return celsius * 9.0 / 5.0 + 32;
    }
    public void setCelsius(double c) {
        if (c < -273.15) throw new IllegalArgumentException("Below absolute zero");
        this.celsius = c;
    }
}
```

Callers never touch `celsius` directly, so the class can later change its internal representation (e.g. store Kelvin instead) without breaking client code, and can enforce invariants (no temperature below absolute zero) at the single point of mutation.

Why make fields private? (a) protects invariants — illegal states become unrepresentable; (b) decouples clients from the representation, allowing it to change freely; (c) centralises validation logic in one place instead of scattering checks across every caller.

4.7 Access Modifiers

Modifier	Visible from	Typical use
`private`	same class only	internal representation, helper methods
(default)	same package	package-internal collaboration
`protected`	same package + subclasses (any package)	extension points for subclasses
`public`	everywhere	the class's public API

TRAP: interface and default methods

Fields and methods in an `interface` are implicitly `public` (and constants are implicitly `public static final`) even with no modifier written — forgetting this is a common source of "what is the visibility here" Tripos marks.

4.8 Immutability

Creating an Immutable Class

An object is **immutable** if its observable state cannot change after construction. Recipe:

1. Make the class `final` (or all methods final) so it cannot be subclassed into a mutable variant.
2. Make all fields `private final`.
3. Provide no setters.
4. If a field is a mutable reference type (e.g. an array or `List`), **defensively copy** it in the constructor *and* in any getter that returns it — otherwise callers can mutate your "immutable" internal state through the returned reference.

```
public final class ImmutablePoint {
    private final int x, y;
    public ImmutablePoint(int x, int y) { this.x = x; this.y = y; }
    public int getX() { return x; }
    public int getY() { return y; }
    public ImmutablePoint translate(int dx, int dy) {
        return new ImmutablePoint(x + dx, y + dy); // return a NEW object
    }
}
```

Benefits of Immutability

Thread-safe with no synchronisation needed (no data race is possible if nothing changes); safe to share freely and cache/reuse; simpler to reason about (no "spooky action at a distance"). `String`, the boxed types (`Integer`, `Double`, ...), and much of `java.time` are immutable in the JDK for exactly these reasons.

5 Memory, Pointers and References

5.1 Stack vs Heap

Where Stuff Goes

Stack: one per thread; holds **stack frames**, one per active method call, each containing local variables (primitives, and references) and the return address. LIFO; automatically reclaimed when a method returns; fast, fixed-size (risk of `StackOverflowError` with deep/unbounded recursion).
Heap: shared across the program; holds all **objects** (created with `new`), which live until unreachable, at which point the garbage collector reclaims them. Slower to allocate from, size limited by JVM settings, not automatically bounded by call depth.

5.2 The Call Stack

Each method call pushes a new frame containing its parameters and locals; returning pops the frame. **Recursive** calls push one frame per call — deep recursion can overflow the stack, which **iteration** or **tail-call** style avoids (though standard Java does *not* guarantee tail-call optimisation, unlike languages such as OCaml/Scheme).

TRAP: Java does not optimise tail calls

Rewriting a recursive method to be tail-recursive does *not* prevent `StackOverflowError` in standard Java — the JVM specification does not mandate tail-call elimination. If asked to test this, write a counting tail-recursive method and observe it still overflows for a large enough input.

5.3 Pointers vs References

Box and Arrow Model

A **pointer** (as in C) is a raw memory address that can be dereferenced, incremented, or arithmetic'd on directly — it is essentially an integer with special meaning. A Java **reference** is a restricted, managed handle to a heap object: it cannot be arithmetic'd, cast to an integer, or point into the middle of an object; the JVM controls its lifetime via garbage collection. Conceptually, draw the variable as a box, and an arrow from the box to the object it references.

Java deliberately has *no* pointer arithmetic: this rules out an entire class of memory-safety bugs (buffer overruns, dangling pointers, use-after-free) at the cost of some low-level control.

5.4 Pass-by-Value (Always, in Java)

The Single Most Important Memory Rule

Java is **always pass-by-value**. For a primitive, the *value* is copied into the parameter. For an object, the *reference* (the handle/arrow) is copied — *not* the object itself. Both the caller's and callee's variables are now separate copies of the same reference, pointing at the same single heap object.

Consequence: what CAN and CANNOT happen

```
void reassign(int[] arr) { arr = new int[]{9, 9, 9}; } // no visible effect outside
void mutate(int[] arr) { arr[0] = 9; } // IS visible outside

int[] a = {1, 2, 3};
reassign(a); System.out.println(a[0]); // prints 1 - reassigning the local copy of the reference
mutate(a); System.out.println(a[0]); // prints 9 - mutating through the shared reference
```

Reassigning the parameter inside a method never affects the caller's variable (you only changed your local copy of the reference). **Mutating** the object the reference points to *is* visible to the caller, because both variables point at the same heap object.

TRAP: "Java is pass-by-reference for objects"

This is a common misconception and a favourite Tripos trick statement to evaluate as true/false — it is **false**. Java has no pass-by-reference mechanism at all (unlike, say, C++'s `&` parameters, where reassignment inside the callee *is* visible to the caller). What Java has is pass-by-value *of a reference*.

5.5 Values are Copied on Assignment

`int b = a;` for primitives copies the value — `a` and `b` are now independent. `Object b = a;` for references copies the reference — `a` and `b` are two names for the *same* object; mutating via one is visible via the other; but reassigning one (`b = new Object();`) does not affect the other.

5.6 Reference Equality vs Value Equality

`==` on references compares whether they point to the *same object* (identity), not whether the objects have equal content. `.equals()` should be overridden to compare content (see §10).

6 Inheritance

6.1 What is Inheritance?

Inheritance models an **"is-a"** relationship: a **subclass** (`extends` a **superclass**) inherits its fields and methods, and may add new members or **override** inherited methods. Every class in Java implicitly extends `Object` if no other superclass is named, giving every object `equals`, `hashCode`, `toString`, `getClass`, etc.

6.2 What Gets Inherited?

A subclass inherits all `public` and `protected` members (and package-private members if in the same package), but *not* `private` members and *not* constructors (though it can invoke them via `super(...)`).

6.3 Constructor Chaining: super()

The first line of a subclass constructor is (implicitly or explicitly) a call to a superclass constructor via `super(...)` — this ensures the superclass's invariants are established before the subclass adds its own state.

```
class Animal {
    protected String name;
    Animal(String name) { this.name = name; }
}
class Dog extends Animal {
    private String breed;
    Dog(String name, String breed) {
        super(name); // must be first statement
        this.breed = breed;
    }
}
```

6.4 Casting: Widening and Narrowing

Widening (upcasting) to a superclass type is always safe and often implicit: a `Dog` *is-a* `Animal`. **Narrowing (downcasting)** to a subclass type must be explicit and can fail at runtime with `ClassCastException` if the object is not actually an instance of that subclass — guard with `instanceof` first.

```
Animal a = new Dog("Rex", "Labrador"); // widening, implicit
if (a instanceof Dog d) { // pattern-matching instanceof
    System.out.println(d.getBreed()); // narrowing, explicit & safe here
}
```

6.5 Fields and Inheritance: Shadowing

TRAP: Fields do not override, they shadow

If a subclass declares a field with the *same name* as one in its superclass, the subclass field **shadows** (hides) rather than overrides it. Unlike method overriding, **field access is resolved statically**, by the compile-time (declared) type of the reference, not the runtime type of the object.

```
class A { int x = 1; }
class B extends A { int x = 2; }
A a = new B();
System.out.println(a.x); // prints 1 (A's x - static type is A)
System.out.println(((B) a).x); // prints 2 (B's x)
```

This is a classic Tripos "what does this print" trap, contrasted directly with method overriding below.

6.6 Methods and Inheritance: Overriding

A subclass **overrides** an inherited method by redeclaring it with an identical signature (and a covariant or identical return type). Unlike fields, **method calls are resolved dynamically** (see §7, dynamic dispatch) — by the object's *runtime* type, not the reference's declared type.

```
class A { void speak() { System.out.println("A"); } }
class B extends A { @Override void speak() { System.out.println("B"); } }
A a = new B();
a.speak(); // prints "B" - dynamic dispatch uses the RUNTIME type
```

`@Override` is optional but strongly recommended — the compiler checks the signature actually matches a superclass method, catching typos that would otherwise silently create an unrelated overload.

6.7 Abstract Classes

An **abstract class** (`abstract class Shape {...}`) cannot be instantiated directly; it may mix concrete (implemented) methods with **abstract methods** (declared, no body) that subclasses *must* implement to become concrete. Useful when subclasses share common state/implementation but each must supply some behaviour uniquely.

```java
abstract class Shape {
    abstract double area(); // no body - must be implemented
    void describe() { // shared concrete implementation
        System.out.println("Area: " + area());
    }
}
```

6.8 Interfaces

An **interface** declares a contract — a set of method signatures (all implicitly `public abstract` unless marked `default` or `static`) that any implementing class must provide. A class `implements` any number of interfaces, unlike single-class inheritance (`extends`). Since Java 8, interfaces may include `default` methods with a body, giving a limited, safe form of multiple behaviour inheritance.

Abstract Class or Interface?

	Abstract class	Interface
Multiple inheritance	No (single `extends`)	Yes (multiple `implements`)
Instance fields	Yes	No (only `static final` constants)
Constructors	Yes	No
Non-public members	Yes (`private`, `protected`)	No (implicitly public, plus `private` helpers since Java 9)
Use when	Subclasses share state/implementation, tight "is-a" family	Unrelated classes share a *capability* ("can-do")

6.9 When Inheritance Goes Wrong

The JDK Stack Anti-Pattern

`java.util.Stack` extends `Vector`, inheriting *all* of `Vector`'s public methods — including `insertElementAt`, which lets callers insert into the middle of what is supposed to be a strict LIFO stack, silently violating the stack invariant. The lesson: only use `extends` for a genuine, invariant-preserving "is-a" relationship; if you only want to *reuse* another class's implementation, prefer **composition** (a `Stack` that *has-a* `Vector` internally and exposes only `push`/`pop`).

7 Polymorphism and Multiple Inheritance

7.1 (Subtype) Polymorphism

Core Idea

Polymorphism ("many forms") lets code written against a supertype (class or interface) operate uniformly over any of its subtypes, with the *actual* behaviour determined by the object's true runtime type. This is what allows `List<Shape> shapes` to hold a mix of `Circle`, `Square`, etc., and `shapes.forEach(s -> s.draw())` to invoke each shape's own `draw`.

7.2 Static vs Dynamic Polymorphism

Static (compile-time)	Dynamic (run-time)
Method **overloading**: resolved by the compiler using the declared (static) types of the arguments.	Method **overriding**: resolved by the JVM at call time using the object's actual runtime type ("dynamic dispatch" / "virtual method invocation").

The Canonical Example

```java
class Animal { String sound() { return "..."; } }
class Cat extends Animal { String sound() { return "Meow"; } }
class Dog extends Animal { String sound() { return "Woof"; } }
```

```
Animal[] zoo = { new Cat(), new Dog() };
for (Animal a : zoo) {
    System.out.println(a.sound()); // "Meow" then "Woof" - dynamic dispatch
}
```

Even though the *declared* type of every array element is `Animal`, the JVM looks up `sound()` using each object's actual runtime class at every call — this indirection (via a per-class virtual method table) is the mechanism underlying dynamic dispatch.

7.3 Multiple Inheritance and the Diamond Problem

The Diamond Problem

If class `D` inherits from both `B` and `C`, which both inherit from `A` and both override a method `m()` differently, which version does `D` get? This ambiguity is the classic argument against unrestricted multiple *class* inheritance.

7.4 Java's Take on It

Java **disallows** multiple class inheritance (`extends` takes exactly one class) specifically to avoid the diamond problem for *state*. It **does** allow implementing multiple *interfaces*, including multiple `default` methods with the same signature — Java resolves this with explicit rules rather than silent ambiguity:

Resolution Rules for Method Clashes

1. A method declared in the **class itself** (or inherited from a superclass) always wins over any interface `default` method.
2. If two interfaces provide conflicting `default` methods and neither is more specific, the implementing class **must** override the method itself (and may call a specific one via `InterfaceName.super.method()`) — otherwise it is a **compile error**, not a silent runtime choice.
3. A more specific interface's default (one that extends another interface providing the same default) wins automatically.

So Java sidesteps the diamond problem by (a) banning it for state entirely and (b) forcing an explicit, compile-time resolution for behaviour clashes rather than an implicit runtime rule.

Is multiple inheritance just evil?

Not inherently — languages like C++ and Python allow it and it can be useful (e.g. mixins). The problem is specifically *unconstrained* multiple inheritance of *state*, which creates ambiguity and fragile hierarchies. Java's design captures most of the benefit (interface-based multiple inheritance of *behaviour* contracts) while avoiding the state-ambiguity cost.

8 Principles for Good OOP: OCP and LSP

8.1 The Open–Closed Principle (OCP)

OCP

Classes should be **open for extension** but **closed for modification**: you should be able to add new behaviour without editing (and risking breaking) existing, tested code.

OCP Example: Shape Area Calculation

Violates OCP — every new shape requires editing this method:

```
double area(Object shape) {
    if (shape instanceof Circle c) return Math.PI * c.r * c.r;
    if (shape instanceof Square s) return s.side * s.side;
    // must add another branch for every new shape type...
}
```

Satisfies OCP — adding a new shape means adding a new class, with zero changes to existing code:

```
abstract class Shape { abstract double area(); }
class Circle extends Shape { double r; double area() { return Math.PI * r * r; } }
class Square extends Shape { double side; double area() { return side * side; } }
```

Polymorphism is the primary mechanism OOP gives us for satisfying OCP.

8.2 Liskov Substitution Principle (LSP)

LSP

Objects of a subclass should be **substitutable** for objects of the superclass without altering the correctness of the program: if `S` is a subtype of `T`, any code that works with `T` must continue to work correctly when given an `S`. In other words, subclassing must not *strengthen preconditions* or *weaken postconditions/invariants* that callers rely on.

The Classic Violation: Square extends Rectangle

```
class Rectangle {
    protected int width, height;
    void setWidth(int w) { width = w; }
    void setHeight(int h) { height = h; }
    int area() { return width * height; }
}
class Square extends Rectangle {
    @Override void setWidth(int w) { width = height = w; } // breaks the invariant!
    @Override void setHeight(int h) { width = height = h; }
}
```

Client code that does `r.setWidth(5); r.setHeight(4); assert r.area() == 20;` holds for any `Rectangle` but **breaks** when given a `Square` (area becomes 16), even though `Square` *is-a* `Rectangle` geometrically. `Square` silently violates a behavioural invariant the `Rectangle` contract implies, so it is *not* a valid LSP subtype despite the mathematical "is-a" relationship.

JDK LSP Violation: Object.equals

Many JDK classes and naive user code violate LSP around `equals()`: if a subclass adds fields and includes them in `equals`, symmetry (`a.equals(b) == b.equals(a)`) can break when comparing a superclass instance to a subclass instance. The Tripos may ask you to *identify* such a violation and propose a fix (e.g. use `getClass()` checks consistently, or favour composition over inheritance for value-like classes).

OCP vs LSP – how they relate

OCP is about designing so future *extension* doesn't require editing existing code. LSP is the precondition that makes safe substitution — and hence OCP via polymorphism — actually work: if subclasses don't honestly satisfy their supertype's contract, polymorphic client code (relying on OCP-style extension) will silently misbehave.

9 Object Lifecycle, Garbage Collection and Copying

9.1 Creation and Initialisation

`new` allocates memory on the heap, zero-initialises fields, runs field initialisers in declaration order, then runs the constructor body (after any `super(...)` call). An object exists from that point until it becomes **unreachable** (no live reference chain from a GC root reaches it), at which point it becomes eligible for collection.

9.2 Mark and Sweep

Mark-and-Sweep Garbage Collection

1. **Mark:** starting from a set of **GC roots** (local variables on active stack frames, static fields, ...), traverse the object graph and mark every reachable object as "live".
2. **Sweep:** scan the heap; any object *not* marked live is unreachable garbage and its memory is reclaimed.
3. **Compaction** (optional): reachable objects are moved together to eliminate fragmentation, and all references to them are updated — this is why Java references are *not* raw addresses (they must survive the collector moving the object).

The collector isn't free: tracing the whole live object graph takes CPU time and can introduce "stop-the-world" pauses; JVMs mitigate this with generational collection (most objects die young, so a small "young generation" is collected frequently and cheaply, while a large "old generation" is collected rarely).

TRAP: Cyclic references are still collected

Unlike naive reference-counting garbage collection, mark-and-sweep correctly reclaims cyclic structures (e.g. two objects that reference each other but are reachable from nothing else) because reachability from GC roots, not reference count, determines liveness.

9.3 Destructors and finalize()

Java has no deterministic destructor (unlike C++'s RAII). `Object.finalize()` was intended as a cleanup hook run before collection, but it is **deprecated**: its timing is unpredictable, it can resurrect objects, and it can silently swallow exceptions. Prefer `try-with-resources` and the `AutoCloseable`/`Closeable` interfaces for deterministic cleanup (e.g. closing file handles).

9.4 Shallow vs Deep Copies

Shallow vs Deep Copy

A **shallow copy** duplicates an object's fields, but for any reference-type field, it copies the *reference* — so the original and the copy still share the same nested objects. A **deep copy** recursively copies every nested reachable object too, so the copy shares nothing mutable with the original.

9.5 Copy Constructors

The idiomatic, safe way to copy in Java — a constructor that takes another instance of the same class and copies its state (deeply, where needed):

```
class Team {
    private List<String> members;
    Team(Team other) { // copy constructor
        this.members = new ArrayList<>(other.members); // deep-ish copy of the list
    }
}
```

9.6 Java's clone() and Why It's Awkward

TRAP: clone() is a footgun

`Object.clone()` is `protected` and does a *shallow* field-for-field copy by default. To use it: implement the (empty, marker) `Cloneable` interface, override `clone()` as `public`, call `super.clone()`, and then **manually deep-copy every mutable reference field** yourself — otherwise the clone shares mutable internals with the original (exactly the "Car/Tyre" bug seen in Tripos 2010 Q7: cloning `tyres` by simple assignment leaves both cars sharing the same tyre array). Because of this fragility, many style guides recommend copy constructors or copy factory methods over `clone()` entirely.

Correct clone() Pattern

```
class Car implements Cloneable {
    private Tyre[] tyres = new Tyre[4];
```

```
    @Override
    public Car clone() {
        try {
            Car c = (Car) super.clone(); // shallow copy first
            c.tyres = new Tyre[tyres.length];
            for (int i = 0; i < tyres.length; i++)
                c.tyres[i] = tyres[i].clone(); // deep-copy each mutable field
            return c;
        } catch (CloneNotSupportedException e) { throw new AssertionError(e); }
    }
}
```

9.7 Cloning Arrays

`array.clone()` *is* well-behaved for arrays — it always produces a shallow copy of the array itself (a new array object with the same length), which is safe for arrays of primitives but still shares nested objects for arrays of references.

9.8 Covariant Return Types

Since Java 5, an overriding method may narrow (rather than exactly match) its return type, provided the narrowed type is a subtype of the original — this is what lets `Car.clone()` above return `Car` rather than the inherited `Object` signature, avoiding an explicit cast at every call site.

9.9 Marker Interfaces

`Cloneable` and `Serializable` are **marker interfaces**: they declare *no* methods at all. Their sole purpose is to tag a class so that other code (here, `Object.clone()`) can check `instanceof` at runtime and change behaviour (throwing `CloneNotSupportedException` if the marker is absent) — a lightweight, if somewhat outdated, form of runtime capability tagging, largely superseded by annotations in modern Java.

10 Collections and Comparisons

10.1 The Java Collections Framework

Interface	Common implementations	Characteristics
`List`	`ArrayList`, `LinkedList`	ordered, indexed, duplicates allowed
`Set`	`HashSet`, `TreeSet`, `LinkedHashSet`	no duplicates (by `equals`)
`Map`	`HashMap`, `TreeMap`, `LinkedHashMap`	key → value, unique keys
`Queue`/`Deque`	`ArrayDeque`, `LinkedList`	FIFO / double-ended access

10.2 ArrayList vs LinkedList

Complexity Trade-offs

ArrayList (backed by a resizable array): $O(1)$ amortised random access and append; $O(n)$ insert/remove in the middle (must shift elements); good cache locality.
LinkedList (doubly-linked nodes): $O(1)$ insert/remove *once you have a reference to the node* (e.g. via an iterator), but $O(n)$ random access (`get(i)` must walk from an end); poor cache locality; implements both `List` and `Deque`.
Rule of thumb: default to `ArrayList` unless you specifically need frequent insertion/removal at arbitrary known positions (e.g. mid-traversal via an iterator) rather than random-access reads.

10.3 HashMap/HashSet vs TreeMap/TreeSet

`HashMap`/`HashSet` use **hashing** (via `hashCode`) for average $O(1)$ lookup/insert but give no ordering guarantee. `TreeMap`/`TreeSet` use a red-black tree, giving $O(\log n)$ operations but keeping keys/elements in **sorted** order (natural ordering via `Comparable`, or a supplied `Comparator`).

10.4 Iteration and Iterators

The enhanced for-loop (`for (T x : collection)`) is external-iteration sugar over the `Iterable`/`Iterator` interfaces: `hasNext()` and `next()`. To remove elements safely *while* iterating, use `Iterator.remove()` — mutating the underlying collection directly during a for-each loop throws `ConcurrentModificationException`.

10.5 Comparing Objects: == vs equals()

TRAP: == on objects is identity, not value equality

For reference types, `==` checks whether two variables point at the *same* object; it does *not* compare content. Two distinct `Integer` objects with the same numeric value, or two distinct `String` objects with the same characters, can be `==`-unequal but `.equals()`-equal. Always use `.equals()` for value comparison of objects (primitives are the exception — `==` is correct and sufficient for them).

10.6 The equals()/hashCode() Contract

Overriding equals() and hashCode() Together

If you override `equals()`, you **must** also override `hashCode()` so that **equal objects have equal hash codes** (the reverse need not hold). Breaking this contract silently corrupts `HashMap`/`HashSet` behaviour — an object may become "lost" (looked up with an equal key/element that hashes differently and so is never found in the correct bucket).

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

10.7 Comparable<T> vs Comparator<T>

Comparable<T>	Comparator<T>
Implemented *by* the class itself: `compareTo(T other)`. Defines the class's single "natural" ordering.	A *separate* object implementing `compare(T a, T b)`. Lets you define any number of orderings *external* to the class (e.g. sort a `Person` list by name, or separately by age).

Both return negative/zero/positive to mean less-than/equal/greater-than. `Collections.sort(list)` uses `Comparable`; `Collections.sort(list, comparator)` or `list.sort(comparator)` uses a `Comparator` — useful when you cannot modify the class, or need multiple orderings.

10.8 Operator Overloading

Unlike C++, Java deliberately does **not** support user-defined operator overloading (with the sole built-in exception of `+` for `String` concatenation) — a design choice trading some expressiveness for readability and predictability (no surprise behaviour hidden behind familiar symbols).

11 Generics

11.1 Why Generics?

Before generics, collections stored `Object`, requiring casts on retrieval and deferring type errors to **runtime**. Generics let a class/method be parameterised by a type, so the compiler enforces type safety at **compile time** and casts become unnecessary.

```
List list = new ArrayList(); // raw type - pre-generics style
list.add("hello");
list.add(42); // compiles! - no type safety
String s = (String) list.get(1); // ClassCastException at RUNTIME

List<String> safeList = new ArrayList<>();
safeList.add("hello");
// safeList.add(42); // COMPILE ERROR - caught early
```

11.2 Declaring a Generic Class

```
class Box<T> {
    private T value;
    void set(T value) { this.value = value; }
    T get() { return value; }
}
Box<String> box = new Box<>(); // T is instantiated as String
```

11.3 Bounded Type Parameters

`<T extends Number>` restricts `T` to `Number` or its subtypes, giving access to `Number`'s methods (e.g. `doubleValue()`) inside the generic class/method — despite the keyword `extends`, this also works for bounding by an interface.

11.4 Java's Generics Implementation: Type Erasure

Option 1 (templates) vs Option 2 (type erasure)

Templates (C++ style): the compiler generates a separate specialised class per type argument used. Fast at runtime, but code bloat, and requires the generic's source at every instantiation site.
Type erasure (Java's choice): generic type information exists only at *compile time*, for static checking; the compiler then **erases** it, replacing `T` with `Object` (or its bound) and inserting casts automatically. At *runtime*, `List<String>` and `List<Integer>` are the *same* class, `List`, with no way to recover `T` via reflection. Chosen primarily for **backward compatibility** with pre-generics (Java 1.4 and earlier) bytecode and libraries.

Consequences of Type Erasure (frequent Tripos topic)

- **Cannot use primitive type arguments:** `List<int>` is illegal (`T` is erased to `Object`, and primitives aren't objects) — must use the boxed wrapper, `List<Integer>`, relying on autoboxing.
- **Creation is tricky:** you cannot write `new T()` or `new T[]` inside a generic class, because at runtime the compiler has no concrete type `T` to instantiate.
- **Overloading is limited:** `void m(List<String> l)` and `void m(List<Integer> l)` cannot coexist in the same class — after erasure both have the identical signature `m(List l)`, a compile error.
- **No runtime type check of the parameter:** `list instanceof List<String>` is illegal — only `list instanceof List` is checkable, since `String` is erased away.

11.5 Generics, Inheritance and Covariance

TRAP: Java arrays are covariant, generics are NOT

Arrays are covariant: if `Dog` is a subtype of `Animal`, then `Dog[]` is treated as a subtype of `Animal[]` — this is convenient but **unsound**, because it lets code compile that fails only at *runtime*:

```
Animal[] animals = new Dog[3]; // legal - covariant
animals[0] = new Cat(); // compiles, but throws ArrayStoreException at RUNTIME!
```

This works because Java arrays are **reified** — they remember their element type at runtime and check every store against it.
Generics are invariant (by default) and are NOT reified (they are erased): `List<Dog>` is *not* a subtype of `List<Animal>`, even though `Dog` is a subtype of `Animal` — this is intentionally stricter than arrays, and catches the equivalent bug at **compile time** instead:

```
List<Animal> animals = new ArrayList<Dog>(); // COMPILE ERROR - invariant
```

11.6 Wildcards

Wildcards restore some flexibility lost to invariance, without sacrificing type safety:

Form	Meaning	Use (PECS)
`List<? extends T>`	unknown subtype of `T`	**Producer** — safe to *read* `T` out, unsafe to add (except `null`)
`List<? super T>`	unknown supertype of `T`	**Consumer** — safe to *write* `T` in, reads only give `Object`
`List<?>`	completely unknown type	read-only, type-agnostic operations (e.g. `size()`)

PECS mnemonic

Producer **E**xtends, **C**onsumer **S**uper. If a parameterised collection only supplies values to you, use `? extends`; if you only push values into it, use `? super`.

12 Coupling, Errors and Exceptions

12.1 Coupling

Coupling measures how much one class depends on the internal details of another. **Bad (tight) coupling**: class `A` reaches directly into `B`'s internals, or depends on a concrete implementation class rather than an interface — a change to `B` risks breaking `A`. **Good (loose) coupling**: classes communicate through small, stable interfaces; concrete implementations can be swapped freely (a key enabler of both OCP and testability, e.g. via mock objects).

12.2 Boxing and Unboxing

Autoboxing / Auto-unboxing

Each primitive has a corresponding **wrapper class** (`int`↔`Integer`, `double`↔`Double`, ...). **Autoboxing** implicitly wraps a primitive into its wrapper (needed wherever an `Object`/generic type is required, e.g. inside collections); **auto-unboxing** implicitly extracts the primitive back out.

TRAP: NullPointerException on unboxing

Auto-unboxing a `null` wrapper throws `NullPointerException`:

```
Integer count = null;
int x = count; // auto-unboxing attempts count.intValue() -> NPE!
```

Also note: `Integer` caches and interns small values (−128 to 127), so `==` *happens* to work for cached values but **fails** for larger ones — another reason to always use `.equals()` for wrapper comparison, never `==`.

12.3 Return Codes vs Exceptions

Why Exceptions over Return Codes

The C-style pattern of returning a sentinel error code (or setting a global `errno`) has real problems: callers can silently **ignore** the check (nothing forces you to inspect the return value); error-handling code is interleaved with, and obscures, the normal-case logic; and a valid return value can be confused with an error sentinel. **Exceptions** decouple error *detection* from error *handling* (a `throw` can propagate up through many stack frames to whichever `catch` is equipped to deal with it), and (for checked exceptions) the compiler *forces* acknowledgement.

12.4 Throwing, Catching, finally, try-with-resources

```
try {
    riskyOperation();
} catch (IOException | SQLException e) { // union/multi-catch
    log(e);
} finally {
    cleanup(); // ALWAYS runs (even on return/throw)
}

try (FileReader fr = new FileReader("f.txt")) { // try-with-resources
    // fr is AutoCloseable - closed automatically, even on exception
}
```

12.5 Java's Exception Hierarchy

Throwable Hierarchy

`Throwable` splits into `Error` (serious, generally unrecoverable JVM-level problems, e.g. `OutOfMemoryError`, `StackOverflowError` — not meant to be caught) and `Exception`, which splits further into:

- **Checked exceptions** (subclasses of `Exception`, excluding `RuntimeException`): the compiler forces every

caller to either `catch` or declare (`throws`) them. Used for *recoverable*, expected failure conditions (e.g. `IOException`).
- **Unchecked exceptions** (`RuntimeException` and its subclasses, e.g. `NullPointerException`, `IllegalArgumentException`): no compiler enforcement. Used for *programming errors* that typically shouldn't be routinely caught, only fixed.

12.6 Creating Custom Exceptions

```
class InsufficientFundsException extends Exception { // checked
    InsufficientFundsException(String msg) { super(msg); }
}
```

Extend `Exception` for checked, or `RuntimeException` for unchecked; give a meaningful subclass name and constructors that forward a message (and optionally a cause) to `super`.

12.7 Guidelines for Exception Use

Five Rules (frequently examined)

1. **Never ignore (swallow) an exception silently** — an empty `catch` block hides bugs; at minimum log it.
2. **Don't catch `Exception` (or `Throwable`) broadly** — catch the most specific type you can actually handle, or you risk masking unrelated bugs (including `Error`s you shouldn't touch).
3. **Document exceptions at the API level** (Javadoc `@throws`) so callers know what to expect without reading your implementation.
4. **Avoid leaking implementation-specific exceptions** across an abstraction boundary — e.g. a repository interface backed by SQL shouldn't force callers to catch `SQLException`; wrap it in a domain-specific exception.
5. **Never use exceptions for ordinary control flow** — they are (relatively) expensive (stack trace capture) and make code harder to follow; reserve them for genuinely exceptional conditions.

12.8 Assertions

`assert condition : message;` checks an invariant that the programmer believes must always hold. **Disabled by default** at runtime (must be enabled with the `-ea` JVM flag), so **never** rely on assertions for logic that must run in production, or for validating external/user input (use exceptions for that) — assertions are a debugging/-documentation aid for conditions that indicate a bug in *your own* code if false, typically used for postconditions and internal preconditions, not for checking environmental facts (e.g. don't assert your compiler works correctly).

13 Design Patterns

What is a Design Pattern?

A **design pattern** is a general, reusable solution to a commonly-occurring software design problem — a named vocabulary for a proven structural idea, not literal code you copy in. Originally catalogued (23 patterns) by the "Gang of Four". The Tripos typically expects you to: name the abstract problem it solves, sketch the structure (roles/relationships), and connect it back to OCP or another principle.

13.1 Composite

Problem: treat a single object and a group (tree) of objects **uniformly** through one interface — e.g. representing a DVD box-set alongside individual films without duplicating pricing logic.

```
interface PriceableItem { double price(); }
class Film implements PriceableItem {
    double price() { return 9.99; }
}
class BoxSet implements PriceableItem { // composite: also a PriceableItem
    private List<PriceableItem> items = new ArrayList<>();
    double price() {
        double total = items.stream().mapToDouble(PriceableItem::price).sum();
        return total * 0.9; // 10% bundle discount
    }
}
```

Client code calls `.price()` identically whether it holds a `Film` or a `BoxSet` — satisfies OCP (new item types slot in without touching client code).

13.2 Decorator

Problem: add responsibilities to an *individual* object dynamically, at runtime, without subclassing every combination — e.g. gift-wrapping a book.

```
interface Book { double cost(); }
class PlainBook implements Book { double cost() { return 10.0; } }
class GiftWrapped implements Book { // decorator wraps another Book
    private final Book inner;
    GiftWrapped(Book inner) { this.inner = inner; }
    double cost() { return inner.cost() + 2.0; }
}
Book b = new GiftWrapped(new PlainBook()); // stack decorators as needed
```

Avoids a combinatorial explosion of subclasses (`GiftWrappedSignedBook`, ...) that plain inheritance would require.

13.3 State

Problem: let an object change its *behaviour* cleanly when its internal state changes, without a sprawling `if/switch` on a status field everywhere — e.g. an academic's behaviour changing across career rank.

```
interface AcademicState { String duties(); }
class Lecturer implements AcademicState { public String duties() { return "Teach & research"; } }
class Professor implements AcademicState { public String duties() { return "Lead & advise"; } }
class Academic {
    private AcademicState state = new Lecturer();
    void promote(AcademicState next) { state = next; } // swap behaviour at runtime
    String duties() { return state.duties(); }
}
```

13.4 Strategy

Problem: select between interchangeable *algorithms* at runtime — e.g. multiple change-making algorithms.

```
interface ChangeStrategy { List<Integer> makeChange(int amount); }
class GreedyChange implements ChangeStrategy { /* ... */ public List<Integer> makeChange(int a){return
    null;} }
class DPChange implements ChangeStrategy { /* ... */ public List<Integer> makeChange(int a){return null
    ;} }
class Till {
    private ChangeStrategy strategy;
    Till(ChangeStrategy s) { strategy = s; } // injected algorithm
    List<Integer> giveChange(int amount) { return strategy.makeChange(amount); }
}
```

Strategy vs State – how to tell them apart

Structurally near-identical (both delegate to an interchangeable interface implementation)! The distinction is *intent*: **Strategy** is chosen once, externally, by the client for a whole algorithm; **State** is switched internally by the object itself in response to its own lifecycle/behaviour changes. A Tripos question may ask you to justify which pattern best fits a scenario — argue from intent, not structure.

13.5 Singleton

Problem: ensure a class has **exactly one** instance, with a single global access point — e.g. a shared database-connection manager.

```
class DatabaseConnection {
    private static DatabaseConnection instance;
    private DatabaseConnection() { /* connect */ } // private constructor!
    static DatabaseConnection getInstance() {
        if (instance == null) instance = new DatabaseConnection();
        return instance;
    }
}
```

TRAP: Singleton criticisms

Singleton is the most *criticised* GoF pattern and the Tripos likes to ask you to evaluate it: it introduces **global mutable state** (hard to reason about, hidden dependency rather than an explicit constructor parameter); it makes **unit testing hard** (cannot easily substitute a mock); and the naive version above is **not thread-safe** (two threads could both see `instance == null` and construct two instances) — fixed via synchronization, an eagerly-initialised static field, or the "initialization-on-demand holder" idiom.

13.6 Observer

Problem: when an object's state changes, automatically notify an arbitrary number of interested dependents, without the subject knowing their concrete types — e.g. a phone app reacting to accelerometer events.

```
interface Observer { void onChange(double value); }
class Accelerometer {
    private List<Observer> observers = new ArrayList<>();
    void subscribe(Observer o) { observers.add(o); }
    void update(double reading) {
        for (Observer o : observers) o.onChange(reading); // notify all dependents
    }
}
```

This is the structural basis of GUI event listeners and reactive/pub-sub systems.

Pattern Cheat-Sheet

Pattern	One-line problem it solves
Composite	Treat single objects and groups of objects uniformly
Decorator	Add behaviour/state to one object dynamically
State	Let an object change behaviour as its internal state changes
Strategy	Swap an algorithm implementation at runtime, chosen externally
Singleton	Guarantee exactly one instance with global access
Observer	Notify many dependents automatically when subject state changes

13.7 Bonus: Optional<T>

`java.util.Optional<T>` is a single-value container that either holds a value or is empty — an explicit, type-checked alternative to using `null` to mean "no value". `null` has three problems it addresses: it is error-prone (easy to forget a null-check, causing `NullPointerException`), verbose to check defensively everywhere, and carries no semantic meaning (a `null String` could mean "absent", "not yet loaded", or "error"). `Optional` forces the caller to actively unwrap it (`isPresent()`/`get()`, or better, `map`/`orElse`), making the possibility of absence visible in the type itself.

14 Lambdas, Method References and Streams

14.1 The Motivating Problem

Passing *behaviour* as a parameter (e.g. "sort by this custom rule") in pre-Java-8 code required either the **Strategy pattern** with a named class per behaviour, or a verbose **anonymous class**. Both work but are heavyweight for a one-off, short piece of logic:

```
// Anonymous class - verbose for something this small
Collections.sort(list, new Comparator<String>() {
    public int compare(String a, String b) { return a.length() - b.length(); }
});
// Lambda - the same intent, far more concise
Collections.sort(list, (a, b) -> a.length() - b.length());
```

14.2 Functional Interfaces

What is a Lambda?

A **lambda expression** is an anonymous function — syntax sugar for an instance of a **functional interface**: an interface with exactly **one** abstract method (a "SAM" type, optionally marked `@FunctionalInterface`). The lambda's parameter list and body supply that single method's implementation; its target type is inferred from context.

Interface	Abstract method	Use
`Function<T,R>`	`R apply(T t)`	transform a value
`Predicate<T>`	`boolean test(T t)`	test/filter condition
`Consumer<T>`	`void accept(T t)`	side-effecting action
`Supplier<T>`	`T get()`	lazily produce a value

14.3 Method References

When a lambda's body is simply "call this existing method", a **method reference** is a shorter equivalent:

```
list.forEach(x -> System.out.println(x)); // lambda
list.forEach(System.out::println); // equivalent method reference
names.stream().map(s -> s.toUpperCase()); // lambda
names.stream().map(String::toUpperCase); // equivalent method reference
```

Forms: `Class::staticMethod`, `instance::instanceMethod`, `Class::instanceMethod` (first lambda arg becomes the receiver), `Class::new` (constructor reference).

14.4 Streams

External vs Internal Iteration

External iteration (a for-loop): the caller explicitly drives each step ("get next, process, repeat"), interleaving *what* to do with *how* to iterate. **Internal iteration** (streams): the caller declares *what* transformation/aggregation they want; the library drives iteration internally — enabling optimisations (e.g. laziness, potential parallelism) invisible to, and unmanaged by, the caller.

A `Stream<T>` represents a (lazy, single-use) pipeline of data. Operations are either **intermediate** (lazy, return a new stream: `map`, `filter`, `sorted`, `distinct`) or **terminal** (eager, trigger execution and produce a result: `collect`, `forEach`, `reduce`, `count`). Nothing runs until a terminal operation is invoked (laziness allows the pipeline to be optimised/fused as a whole).

```
List<String> longNamesUpper = names.stream()
    .filter(n -> n.length() > 4) // intermediate (lazy)
    .map(String::toUpperCase) // intermediate (lazy)
    .sorted() // intermediate (lazy)
    .collect(Collectors.toList()); // terminal - triggers the pipeline
```

Streams and Immutability

Stream operations never mutate the source collection; each intermediate step produces a conceptual new stream. This dovetails with the course's broader push towards immutability and pure functions (§4) — fewer side effects, easier to reason about and (potentially) to parallelise via `parallelStream()`.

15 Past Paper Pattern Library

15.1 Complete Question Map (2010–2025)

Year	Topic & Pattern	Key Content
2010	Interfaces vs abstract classes; cloning	Differences table; spot-the-bug in a `clone()` implementation (shallow array copy); rewrite correctly
2011	Encapsulation; immutability; Singleton	Encapsulate a public-field class, throw on invalid input; bit-packing into one field; immutable class; apply Singleton
2012	Immutability + Generics	Immutable `Complex` class; generalise with `Object` then with Generics; type-erasure gotcha with nested generics and `LinkedList<Complex<Object>>`
2013	Encapsulation; collections choice; Comparable	Why fields shouldn't be public; justify `ArrayList`/`LinkedList` choice; exception-throwing accessors; `Comparable` for ordering
2014	Access modifiers; Singleton; Observer	Access modifier table; lazy Singleton (why a shared `SingletonBase` superclass fails); natural ordering with `equals`; Observer pattern via a custom `TreeSet` subclass
2015	Access modifiers; return codes vs exceptions; Generics; Decorator	Risks of missing access modifiers; replace return-code errors with exceptions; genericise a queue; Decorator pattern for a logging wrapper around a bytecode-only class
2016	`Object` as container anti-pattern; overload/override/shadow; polymorphism	Why `ArrayList<Object>` is bad; defensive copying pitfalls; full "what does this print" trace of overloading/overriding/shadowing; costs/benefits of *optional* dynamic dispatch
2017	Immutability; caching (HashMap + linked list, O(1)); has-a vs extends; checked exceptions	Cache design combining `HashMap` and a linked list; `Basket extends LinkedList` vs `Basket has-a LinkedList`; checked-exception design comparison (6 variants)
2018	`equals()`/`hashCode()` contract; design patterns from a UML diagram; multiple inheritance	Line-by-line `equals()` walkthrough; refactor to State pattern for trim levels; multiple inheritance of type/implementation/state; hybrid engine via interfaces
2019	Immutability with linked structures; Generics and covariance; OOP benefits; OCP	Immutable linked-list `Element`/`FuncList`; why genericising breaks immutability; is banning covariance necessary here; UML diagram satisfying OCP
2020	Multiple inheritance via interfaces; primitives vs objects; boxing; generic substitutability of immutable/mutable hierarchies	`ButtonCanvas` multiple-interface design; auto-(un)boxing NPE example; pass-by-value guarantee for which `T`; `Mutable`/`ImmutableList` subtyping trade-offs (4 variants)
2021	Emulating static without inheritance; the four pillars in a working program; Decorator; OCP	Environment-object pattern to emulate static fields; identify encapsulation/abstraction/inheritance/polymorphism in a `Swapper Reader`; improve a broken Decorator; explain an OCP violation
2022	Immutability trade-offs; conditional immutability design; type erasure and wildcard casting	Pros/cons of immutable classes; refactor "optionally immutable" class (subclassing / builder); true/false on casting `Set<Integer>` to `Set<Number>`/`Set<? extends Number>`/`Set<?>`; diamond operator variants
2023	Array covariance vs generic invariance; cohesion & coupling; Observer vs Strategy; type erasure of nested generics; LSP & SRP violations	`ArrayStoreException` example; UML + code for Observer/Strategy notification system; erasure of `List<List<Integer>>` etc.; insurance `Policy` hierarchy demonstrating both an SRP and an LSP precondition violation
2024	Access modifiers deep dive; public static final fields; class vs abstract class vs interface; `ClassCastException` & iterator-safe removal; class-hierarchy redesign	Compare Java/Python/no-modifier-methods access models; `Student` hierarchy design choice; fix unsafe cast-and-remove loop with an iterator; redesign year-group promotion
2025	`Comparable` vs `Comparator` for a `PriorityQueue`; Observer-based auto-updating queue; four pillars supporting OCP; PECS wildcards	`WorkTask` priority ordering both ways; `AutoUpdatableQueue` design pattern; four OOP pillars → OCP link; wildcard method compiles/fails trace (`? extends`/`? super`)

Reading the Map

Across fifteen years, four themes dominate almost every single paper: (1) **encapsulation/access modifiers** (nearly every year), (2) **a design pattern applied to a concrete scenario** (Singleton, Observer, Decorator, State, Strategy all recur), (3) **immutability** (extremely frequent, often combined with Generics), and (4) **a "trace the code" question** on overloading/overriding/shadowing, casting, or generics/wildcards. Master these four and you can answer the great majority of any given year's two questions.

15.2 Topic Frequency Table

Topic	Frequency	Notes
Encapsulation / access modifiers	∼13/16 years	Near-universal; often the opening sub-part
Immutability	∼10/16 years	Frequently paired with Generics or a design pattern
Design pattern applied to a scenario	∼12/16 years	Singleton, Observer, Decorator, State, Strategy all seen
Generics: erasure, wildcards, covariance	∼10/16 years	Increasingly common in recent years (2019–2025)
Overloading/overriding/shadowing trace	∼6/16 years	"What does this print" style
Exceptions (checked vs unchecked, design)	∼5/16 years	Often as a full standalone question
OCP / LSP / SRP	∼6/16 years	Increasingly common in recent years
equals()/hashCode() contract	∼3/16 years	Always paired with a full code walkthrough
Collections choice & justification	∼5/16 years	ArrayList vs LinkedList, HashMap-backed caches

15.3 Model Answer Template 1: "Design an Immutable Class"

Template: Immutability

1. State the definition: an object whose observable state cannot change after construction.
2. Give the recipe: `final` class (or all-final methods), `private final` fields, no setters, defensive copies of mutable fields in both constructor *and* getters.
3. Write the code, explicitly marking each of the four recipe steps with a comment.
4. State the benefits relevant to the scenario: thread-safety without synchronisation, safe sharing/caching, simpler reasoning.
5. If asked for a disadvantage: extra allocation on every "mutation" (a new object each time), which can be costly for large objects modified frequently — mention this trade-off explicitly, it is often worth a mark.

15.4 Model Answer Template 2: "Apply a Design Pattern to a Scenario"

Template: Applying a Design Pattern

1. Name the abstract problem the scenario matches (e.g. "we need to notify multiple, decoupled dependents of a change" → Observer).
2. Name the pattern and give a one-sentence definition of its intent.
3. Sketch the roles (e.g. Subject/Observer, Context/Strategy, Component/Decorator) either as a short UML-style diagram (boxes + arrows, with a key if the exact UML notation isn't required) or as an interface skeleton.
4. Write the code implementing the pattern specifically for the given scenario's classes and method names (don't just paste a generic template — Tripos markers reward scenario-specific implementation).
5. Justify the design choice: what problem would arise *without* the pattern, and how does this design tie back to OCP or another principle?

15.5 Model Answer Template 3: "Trace This Code" (Overloading/Overriding/Shadowing/Casting)

Template: Code Tracing Questions

For *each* numbered line:

1. **Does it compile?** Check this using only the *declared (static) type* of each reference — overload resolution and field access are both resolved at compile time using declared types only.
2. **If it compiles, what prints?** Now switch to the *runtime* type of the receiver object to resolve any **overridden method** call (dynamic dispatch) — but keep using the *declared* type for any **field** access or **overloaded** method resolution.
3. **If it doesn't compile, say exactly why** (e.g. "no applicable overload", "incompatible types", "method is static and cannot be called this way").

Golden rule to state explicitly in your answer: *fields and overloads resolve statically (declared type); overridden methods resolve dynamically (runtime type).* Examiners award marks for stating this principle, not just the final printed output.

15.6 Model Answer Template 4: "Identify and Fix a Principle Violation" (LSP / SRP / OCP)

Template: Principle Violation

1. State the principle precisely (e.g. LSP: subtypes must be substitutable for their supertype without altering correctness).
2. Point to the specific violation in the given code — quote the exact method/field and explain *which contract it breaks* (a strengthened precondition, a broken invariant, a second unrelated responsibility, etc.).
3. Show the concrete failure: construct a short piece of client code that behaves correctly for the supertype/other case but incorrectly once the violating subtype/responsibility is involved.
4. Propose a fix: this is very often *restructuring the hierarchy* (e.g. don't make `Square extends Rectangle`; instead both implement a common `Shape` interface) or *splitting the class* (for SRP) rather than patching the existing method.
5. Re-check your fix satisfies the principle by re-running the client-code test from step 3 mentally.

16 Exam Strategy: Choosing Your 2 Questions

16.1 Paper Structure

Aspect	Detail
Format	2 questions on the paper (Q3 and Q4 in recent papers), answer **both**
Marks	Typically 20 marks per question, multi-part (a–e/f), later parts often build on earlier ones
Recurring skeleton	Q3 tends to open with a conceptual/definitional sub-part, then a code-design task. Q4 (recent years) often centres on a worked class hierarchy, exception design, or a code trace
High-yield topics	Encapsulation/access modifiers, immutability, and design-pattern application appear almost every year

16.2 Recommended Strategy

1. **Read both questions fully before writing anything.** Later parts frequently hint at, or depend on, earlier parts' answers (e.g. "using your class from (b)...").
2. **Master the four recurring pillars first:** encapsulation/access modifiers, immutability, design patterns (Singleton, Observer, Decorator, State, Strategy), and Generics (erasure/wildcards/covariance). Together these cover the large majority of marks across the entire 2010–2025 archive.
3. **Practise "trace the code" questions explicitly** — they are pure method-marks-for-reasoning territory: state the static-vs-dynamic resolution rule, then apply it line by line.
4. **When asked to critique or compare designs**, always structure your answer as advantages *and* disadvantages, then a justified recommendation — partial credit rewards seeing both sides even when the "expected" answer favours one option.
5. **Write idiomatic Java, not pseudocode**, when code is asked for — correct syntax (access modifiers, `@Override`, exception types) is explicitly marked in several past papers.

16 Consolidated Cheat Sheet

16.2 Static vs Dynamic Resolution

The Single Most Tested Rule

Fields and **overloaded methods** resolve using the **declared (static) type** of the reference, at compile time.
Overridden methods resolve using the **runtime (dynamic) type** of the object, at call time (dynamic dispatch).

16.2 Access Modifiers

Visibility

`private` ⊂ *default (package)* ⊂ `protected` ⊂ `public`
`protected` = package + subclasses anywhere. Interface members are implicitly `public` (fields also `static final`).

16.2 Immutability Recipe

Four Steps

`final` class → `private final` fields → no setters → defensive copies of mutable fields (constructor *and* getters).

16.2 Equality

== vs equals() vs hashCode()

`==` on objects: reference identity, never content. `.equals()`: override for value equality. **Contract:** `a.equals(b)` ⇒ `a.hashCode() == b.hashCode()`. Always override both together.

16.2 Generics

Erasure, Covariance, Wildcards

Type erasure: `T` → `Object`/bound at compile time; no runtime generic type info; no `new T()`; no primitive `T`; overloads that differ only by generic parameter clash.
Arrays are **covariant** & **reified** (runtime-checked, `ArrayStoreException` possible). Generics are **invariant** & **erased** (compile-time-checked only).
PECS: `? extends T` = producer (read-only); `? super T` = consumer (write-only).

16.2 Design Pattern Index

One-liner per pattern

Composite: treat one object and a group uniformly. **Decorator:** add behaviour to one object dynamically. **State:** object changes its own behaviour as its state changes. **Strategy:** client swaps an algorithm implementation externally. **Singleton:** exactly one instance, global access (private constructor + static factory). **Observer:** notify many dependents automatically on change.

16.2 Exceptions

Checked vs Unchecked

`Throwable` → `Error` (don't catch) / `Exception` → {checked (compiler-enforced, recoverable) , `RuntimeException` (unchecked, programmer errors)}. Never swallow silently; never catch bare `Exception`; never use for control flow.

16.2 Memory

Pass-by-Value, Always

Primitives: value copied. Objects: **reference** copied (not the object). Reassigning a parameter never affects the caller; mutating through it does. Stack = frames/locals; Heap = objects, reclaimed by mark-and-sweep from GC roots.

16 Tripos Exam Strategy & Common Pitfalls

16.2 General Exam Technique

1. **Read all parts of a question before starting** — later parts often reveal what earlier parts are building towards.
2. **Show your reasoning, not just conclusions.** Method marks are awarded for correctly stating principles (e.g. "fields resolve statically") even if a final detail is wrong.
3. **Write compilable, idiomatic Java** for code answers — correct modifiers, `@Override`, exception types, and generic syntax are explicitly marked.
4. **For "compare and contrast" questions,** always structure as a table or clearly parallel paragraphs — one column/paragraph per option — rather than prose that blends them together.
5. **For "explain why" questions on principles** (SRP, LSP, OCP), name the principle precisely, point to the exact line/field/method responsible, then explain the consequence with a concrete example.
6. **Don't forget the "and justify" part** of any design question — a correct design without justification loses marks in nearly every past paper.

Top 10 OOP-Specific Tripos Traps

1. **"Java is pass-by-reference for objects":** false. It is always pass-by-value; for objects, the value passed is the reference.
2. **Confusing field shadowing with method overriding:** fields never override, only shadow, and resolve statically — a very common "what does this print" trap.
3. **`clone()` doing only a shallow copy:** must manually deep-copy every mutable reference field after `super.clone()`.
4. **Forgetting `hashCode()` when overriding `equals()`:** breaks `HashMap`/`HashSet` silently.
5. **Assuming generics are covariant like arrays:** `List<Dog>` is *not* a `List<Animal>` — this is intentional (compile-time safety over array's runtime `ArrayStoreException` risk).
6. **Auto-unboxing a `null` wrapper:** throws `NullPointerException`, easy to miss in a code trace.
7. **Believing `assert` statements always run:** they are disabled by default (`-ea` flag needed) — never rely on them for production validation.
8. **`Square extends Rectangle`-style LSP violations:** geometric "is-a" does not imply behavioural substitutability.
9. **Static methods trying to access instance fields:** does not compile — no implicit `this` in a static context.
10. **Mutating a collection while iterating it with a for-each loop:** throws `ConcurrentModificationException` — must use `Iterator.remove()` instead (see Tripos 2024 Q4c).

16 Final Exam Checklist

Before submitting your answers, verify these 8 things:

1. **Static vs dynamic resolution:** for any code-trace question, have you explicitly stated which rule applies to each line (field/overload = static; override = dynamic)?
2. **Access modifiers:** does your code use the *correct*, most restrictive modifier that still satisfies the design (fields `private`, API `public`)?
3. **Immutability recipe:** if asked for an immutable class, did you include *all four* steps, including defensive copies in getters?
4. **equals()/hashCode() pairing:** did you override both together wherever you overrode one?

5. **Design pattern justification:** did you name the pattern, sketch its roles, *and* explain why it fits this scenario specifically (not a generic description)?
6. **Generics correctness:** did you use wildcards (PECS) correctly, and correctly reason about erasure where relevant?
7. **Exception design:** did you distinguish checked vs unchecked appropriately, and avoid swallowing exceptions silently?
8. **Interpretation:** for every "explain" or "discuss" sub-part, did you give a genuine justification sentence rather than just a definition?

Final Advice

The OOP Tripos paper rewards **precise use of terminology** and **concrete, scenario-specific code** over generic textbook answers. Always tie a design pattern or principle back to the *specific* classes and methods named in the question. Two well-structured, fully-justified answers, each engaging honestly with trade-offs, will comfortably earn a First.

Good luck in your Tripos!