Class Design

Java SE 8 Programmer II

KNOWLEDGE CHEATSHEET

OBJECT DESIGN

- **State** — Encapsulate
- **Contract** — Interface
- **Behavior** — Override
- **Creation** — Constructor

01 Class design and inheritance

Encapsulation and object contracts

- **Encapsulation** keeps state private and exposes controlled operations. Validate changes through methods; a public getter need not expose a mutable internal object directly.
- **Immutable objects** prevent state changes after construction. Use private final fields, prevent unsafe subclassing, and defensively copy mutable inputs and outputs. A final reference alone does not freeze its object.
- Override **equals(Object)** for logical equality and **hashCode()** consistently: equal objects must have equal hashes. Check null and type safely; equals(MyType) alone overloads rather than overrides equals(Object).
- **toString()** controls an object's text representation. Printing an object does not automatically print its fields; **==** compares reference identity.

Inheritance and initialization

- **Overriding** uses the same parameter types, a compatible return type, and equal or wider access. An override cannot add broader checked exceptions; unchecked exceptions are not restricted by this rule.
- **Instance methods** dispatch using the runtime object. Overload selection, accessible members and checked exceptions are determined from compile-time types. Private methods are not inherited; static methods are hidden.
- A **final class** cannot be extended; a final method cannot be overridden. A concrete subclass must implement inherited abstract methods; an abstract subclass may leave them unimplemented.
- A constructor starts with **this(...)** or **super(...)**, explicitly or implicitly. An implicit super() requires an accessible no-argument superclass constructor. Constructors are not inherited.
- **Static initialization** runs once per class initialization, superclass first. Instance field initializers and initializer blocks run in textual order after the superclass constructor, before the constructor body. Reassigning references does not undo constructor side effects.

Interfaces, nested types and patterns

- An **interface** supplies a contract and multiple inheritance of type. An **abstract class** can share constructors, instance state and nonpublic implementation among related classes. Java classes extend only one class.
- Java 8 interface fields are **public static final**. Abstract and default methods are public; static interface methods belong to the declaring interface. Resolve unrelated inherited default-method conflicts with an explicit override.
- A **local class** or lambda can capture final or effectively final local variables. Captured references cannot be reassigned, but their objects may be mutable. An inner class can access its enclosing object's fields.

Class Design / Generics & Collections

Java SE 8 Programmer II

KNOWLEDGE CHEATSHEET

COLLECTION DECISIONS

- **List** — Positions
- **Set** — Uniqueness
- **Map** — Keys to values
- **Tree** — Sorted order

- **Enum constants** are predefined instances; use Type.CONSTANT, not new. Enum constructors cannot be public or protected. Enums can have fields, constructors, methods and constant-specific behavior.
- An eager **singleton** can combine a private constructor, private static final instance and public static accessor. A **factory** returns an appropriate object while hiding construction details; factory and singleton are separate ideas.

02 Generics and collections

Generic types and bounds

- **Generics** provide compile-time type checking. Use reference types such as Integer, not int, as type arguments. Raw types bypass checks and can cause ClassCastException later.
- Generic types are **invariant**: List<Integer> is not a List<Number>. **? extends T** supports reading as T; **? super T** supports adding T values. You cannot safely add non-null values to List<?>.
- A **bounded type parameter**, such as <T extends Number>, restricts accepted types and exposes bound methods. A generic method declares its own parameters before the return type: static <T> T echo(T value).
- A **static method** cannot use its class's type parameter directly because it belongs to the class rather than an instance. Declare a method type parameter when needed; type erasure limits runtime generic type information.

Collection shapes and mutation

- **List** preserves positions and permits duplicates. **Set** enforces uniqueness. **Map** stores one value per key; putting an equal key replaces its value. Duplicate map values are allowed.
- **HashMap** and **HashSet** promise no iteration order. **LinkedHashMap** normally preserves insertion order. **TreeMap** orders keys and **TreeSet** orders elements by natural order or a supplied comparator.
- **Arrays.asList(...)** returns a fixed-size list backed by its array. set(), sort() and replaceAll() can change elements; add() and remove() throw UnsupportedOperationException. Copy into ArrayList for structural changes.
- **removeIf(Predicate)** removes matching collection elements. **replaceAll(UnaryOperator)** replaces each list value with another value of the same element type. Map.forEach takes a BiConsumer receiving key and value.

Ordering and comparison

- **Comparable<T>.compareTo(T)** defines natural order. **Comparator<T>.compare(T,T)** supplies external order. Return a negative value, zero or a positive value; a boolean is not a comparison result.
- Use **Comparator.comparing(...)**, comparingInt(...) and **thenComparing(...)** for ordered criteria. comparing(first).reversed().thenComparing(last) reverses only the first criterion; reversing the final chain reverses all criteria.

Generics & Collections / Lambdas

Java SE 8 Programmer II

KNOWLEDGE CHEATSHEET

MATCH THE FUNCTION SHAPE

- **Predicate** — T to boolean
- **Consumer** — T to void
- **Supplier** — No input to T
- **Function** — T to R

- Sorted collections require **mutually comparable** values or a suitable comparator. A comparison result of zero determines duplicate keys or elements, even when equals() disagrees. Keep comparison consistent with equality.
- **Collections.sort** and **Arrays.sort** mutate their input. Stream.sorted creates an ordered stream without sorting the source collection. Prefer Integer.compare(a,b) over subtraction to avoid overflow.

03 Lambdas and functional interfaces

Lambda targets and signatures

- A **functional interface** has one abstract method after inherited methods are considered; Object's public method signatures do not count. Default and static methods do not add abstract methods. @FunctionalInterface requests compiler validation.
- A **lambda** needs a target functional-interface type. Its parameter count, parameter types, return value and checked exceptions must match that interface's abstract method.
- Use **x -> expression** for one inferred parameter, or **(x,y) -> expression** for two. In Java 8, declare all parameter types or infer all of them. A value-returning block needs return on every normal completion path.
- A **void-compatible lambda** can invoke a method or perform another statement expression. A bare arithmetic expression such as x -> x + 1 cannot implement Consumer<T>. A lambda cannot redeclare an enclosing local variable.

Functional interface vocabulary

- **Predicate<T>**: boolean test(T). **Consumer<T>**: void accept(T). **Supplier<T>**: T get(). **Function<T,R>**: R apply(T). Match the operation's shape before choosing an interface.
- **BiPredicate<T,U>**, **BiConsumer<T,U>** and **BiFunction<T,U,R>** take two arguments. **UnaryOperator<T>** is Function<T,T>; **BinaryOperator<T>** is BiFunction<T,T,T>.
- **IntFunction<R>** accepts int and returns R; **ToIntFunction<T>** accepts T and returns int through applyAsInt. **IntUnaryOperator** maps int to int. DoubleFunction<R> accepts double and returns R.
- **Predicate.and/or/negate** combine tests with boolean semantics. **Function.compose(g)** applies g first; **andThen(g)** applies g afterward. Composition must preserve compatible input and result types.

Lambdas / Streams

Java SE 8 Programmer II

KNOWLEDGE CHEATSHEET

STREAM PIPELINE

Source Values → **Filter** Select → **Map** Transform → **Terminal** Consume

Method and constructor references

- A **method reference** supplies behavior, not an immediate call. Type::staticMethod, object::instanceMethod and Type::instanceMethod must adapt to the target interface's signature.
- For **String::length**, the functional argument supplies the receiver: Function<String,Integer>. For **text::indexOf**, the receiver is already bound: ToIntFunction<String> can consume the search string.
- **Type::new** refers to a constructor. Supplier<Car> fits an accessible no-argument constructor; Function<String,Car> fits Car(String). Invoke get() or apply(...) to construct the object.
- A lambda may **return another lambda** when the target result is itself functional: IntFunction<IntUnaryOperator> scale = x -> y -> x * y. IntStream.map accepts the returned IntUnaryOperator.

04 Streams and reduction

Pipeline lifecycle

- A **stream pipeline** has a source, intermediate operations and a terminal operation. Intermediate operations are lazy; constructing map(), filter() or peek() stages alone does not process elements.
- **filter** selects values using a Predicate; **map** transforms each value using a Function. **peek** observes values as they pass; **forEach** is terminal and consumes a Consumer.
- Streams are **single-use**. After a terminal operation, obtain a new stream from the source. Stream operations normally leave the source structure intact; mutating source data during traversal violates noninterference.
- **sorted**, distinct, limit and skip are intermediate operations. Infinite streams require a bound or short-circuiting terminal operation to finish. Do not use pipeline side effects as essential program logic.

Flattening and primitive streams

- **flatMap** converts each element into a stream and flattens the results. Stream<List<T>> can become Stream<T> with flatMap(List::stream); map(List::stream) instead produces a stream of streams.
- **IntStream**, **LongStream** and **DoubleStream** process primitives. Use mapToInt/mapToLong/mapToDouble from object streams; flatMapToInt requires a function returning IntStream, not Stream<Integer>.
- **IntStream.map** takes IntUnaryOperator. **boxed()** converts a primitive stream to wrapper objects; mapToObj converts primitives to arbitrary objects. Stream.of(1,2) produces Stream<Integer>, not IntStream.
- Primitive streams provide **sum**, average and summaryStatistics. average returns OptionalDouble; min and max return primitive optionals. count returns long, including for object streams.

Streams / Exceptions

Java SE 8 Programmer II

KNOWLEDGE CHEATSHEET

STREAM PIPELINE

Source Values → **Filter** Select → **Map** Transform → **Terminal** Consume

Matching and optional results

- **findFirst**, findAny, anyMatch, allMatch and noneMatch can short-circuit. findFirst respects encounter order where present; findAny may choose any element, especially in parallel.
- On an **empty stream**, anyMatch is false, while allMatch and noneMatch are true. Match operations return boolean; find operations return Optional<T>. Their predicates may run on only part of the input.
- **Optional.of(value)** rejects null; **ofNullable(value)** makes null empty; empty() creates an empty optional. get() throws NoSuchElementException when empty. Prefer ifPresent or a fallback when absence is expected.
- **orElse(value)** evaluates its argument eagerly; **orElseGet(supplier)** computes a fallback only when empty. Optional.map transforms a present value; flatMap avoids nesting when the mapping already returns Optional.

Reductions and collectors

- **reduce(identity, accumulator)** returns a value; reduce(accumulator) returns Optional. Use an associative operation and a true identity. For three-argument reduce, the combiner must agree with the accumulator.
- **collect** performs mutable reduction with separate containers and a compatible combiner. Use toList(), toSet() or joining() rather than mutating a shared identity object inside reduce.
- **groupingBy(classifier)** returns groups, normally Map<K,List<T>>. A downstream **mapping(mapper,toList())** stores transformed group values. **partitioningBy(predicate)** separates values under Boolean keys.
- **toMap** needs a merge function if mapped keys can collide. Do not assume a collector's map iteration order or list implementation unless explicitly supplied. min/max on object streams require a Comparator.
- **parallel()** changes execution mode; it does not guarantee faster execution. forEach may reorder output; **forEachOrdered** respects encounter order. Keep functions stateless and avoid shared mutable accumulators.

05 Exceptions and assertions

Exception flow and catch rules

- **Checked exceptions** must be caught or declared. RuntimeException and Error subclasses are unchecked. Declaring a superclass exception covers its subclasses; declaring both is redundant.
- Place **specific catches before general catches**. A prior superclass catch makes a later subclass catch unreachable. **Multi-catch** alternatives cannot have a subtype relationship; its exception parameter cannot be reassigned.

Exceptions / Time & Localization

Java SE 8 Programmer II

KNOWLEDGE CHEATSHEET

RESOURCE LIFETIME

Initialize (In order) → **Try body** (Use resources) → **Close** (Reverse order) → **Catch** (Handle failure)

- **finally** normally runs after try/catch, including on return or throw. A new exception or return from finally can replace the earlier outcome. Ordinary try requires catch or finally; try-with-resources can stand alone.
- **throw** raises an exception object; **throws** declares possible exceptions. Code after an unconditional throw is unreachable. Inspect access, override signatures and checked exceptions before tracing runtime output.

Resource lifetime and assertions

- **Try-with-resources** requires AutoCloseable resources, declared in its header in Java 8. close() must be public. Initialized resources close in reverse declaration order, including when the body fails.
- If the body throws, **close failures are suppressed** on the primary exception and available through getSuppressed(). If initialization fails, already initialized resources still close. An alias does not keep a closed resource usable.
- **assert condition : detail** throws AssertionError when enabled and false. The condition must be boolean; detail must yield a value, so a void println call is invalid there.
- Assertions are disabled by default; **-ea** enables and **-da** disables them. Avoid essential side effects in assertions. Use explicit validation for public inputs because assertion checks may never execute.
- A **String switch** compares case values by equality, is case-sensitive and permits fall-through without break. A null selector throws NullPointerException; default is not a null handler.

06 Date, time and localization

Temporal types and arithmetic

- **LocalDate**, LocalTime and LocalDateTime have no time zone. **ZonedDateTime** combines local date-time with zone rules. **Instant** represents a point on the UTC timeline.
- java.time values are **immutable**: plusDays(), plusYears() and with(...) return new values. Save the result. Arithmetic order can matter around leap days and month boundaries; invalid construction throws DateTimeException.
- **Period** expresses years, months and days. **Duration** expresses seconds and nanoseconds. Use Period for calendar differences and Duration for elapsed time; a LocalDate does not support time-based Duration arithmetic.
- For **elapsed time across zones**, compare instants or compatible zoned date-times. Clock labels alone can mislead. A daylight-saving transition can make a calendar day differ from 24 elapsed hours.
- **DateTimeFormatter** formats and parses supported fields. Match the formatter to the temporal type; a date-only value cannot supply time fields. Pattern letters such as MM (month) and mm (minute) are case-sensitive.

Time & Localization / I/O & NIO.2

Java SE 8 Programmer II

KNOWLEDGE CHEATSHEET

FILESYSTEM APIS

- **Path** — Name and location
- **Files.lines** — Text content
- **Files.list** — Direct entries
- **Files.walk** — Recursive paths

Locales, bundles and properties

- A **Locale** identifies language and optional region, not a time zone. Examples include Locale.UK, new Locale("ru","RU") and new Locale.Builder().setLanguage("de").setRegion("DE").build().
- **ResourceBundle** externalizes localized text. Names such as Messages_de_DE.properties specialize a base bundle. Lookup can fall back to less specific bundles; missing keys with no inherited value throw MissingResourceException.
- In a **.properties file**, write key=value entries. Quotes are literal characters, not string delimiters. Properties.getProperty(key) returns null for an absent key; getProperty(key,defaultValue) supplies a fallback.
- Use **NumberFormat.getCurrencyInstance(locale)** for locale-specific currency presentation. Separators, symbols and placement depend on locale; formatting does not perform currency conversion. Externalized resources allow new translations without rewriting application logic.

07 I/O and NIO.2

Readers, writers and console

- **InputStream/OutputStream** process bytes; **Reader/Writer** process characters. InputStreamReader decodes bytes using a charset. BufferedReader adds readLine(); BufferedWriter adds buffering and newLine().
- **read()** returns an int containing a byte or character, or **-1** at end of input. readLine() returns a line without its terminator, or null at end. Parse numeric text with Integer.parseInt(...).
- **ready()** indicates whether character reading can proceed without blocking, not a reliable end-of-file loop condition. **skip(n)** can skip fewer than n units. mark/reset require a supporting stream and a valid mark.
- **flush()** pushes buffered output downstream; close() also closes the resource and normally flushes a writer. Closing a wrapper closes its underlying stream. Keep the resource open through all required reads or writes.
- **System.console()** can return null, including in IDEs. Console.readPassword() returns char[] and suppresses echo. BufferedReader(new InputStreamReader(System.in)) reads console text through readLine().

I/O & NIO.2

Java SE 8 Programmer II

KNOWLEDGE CHEATSHEET

FILESYSTEM APIS

- **Path** — Name and location
- **Files.lines** — Text content
- **Files.list** — Direct entries
- **Files.walk** — Recursive paths

Paths and filesystem operations

- **Paths.get(...)** creates a Path representation without checking existence. getNameCount() excludes the root; getName(0) is the first name element. getFileName() returns the last element; filesystem syntax is platform-dependent.
- **resolve(relative)** appends a relative path; resolving an absolute path returns that absolute path. **normalize()** simplifies redundant dot elements lexically. toRealPath() accesses the filesystem and requires an existing target.
- **Files.copy(source,target)** treats target as the complete destination path. To copy into a directory, resolve the desired filename. An existing target normally requires REPLACE_EXISTING; copying a directory does not recursively copy its contents.
- **Files.move** relocates a path. After a successful move, the old path is absent. **ATOMIC_MOVE** requires filesystem support; existing-target behavior is implementation-specific, and unsupported atomic moves fail rather than silently becoming non-atomic.
- **Files.delete** throws if the path is absent; deleteIfExists returns false for absence. A directory must be empty to delete it. By contrast, File.delete() reports failure with a boolean; check its result.

File content and traversal

- **Files.readAllLines(path)** eagerly returns List<String>; **Files.lines(path)** lazily returns Stream<String>. Close file-backed streams with try-with-resources. A filename extension does not determine whether text decoding succeeds.
- **Files.list(directory)** streams immediate entries, excluding the directory itself. **Files.walk(start)** traverses recursively and includes start. Neither operation reads file contents.
- **Files.find(start,maxDepth,predicate,...)** filters paths using their BasicFileAttributes. Start is depth 0, immediate children depth 1. FOLLOW_LINKS is explicit; traversal may encounter cycles or inaccessible paths.
- **BasicFileAttributes** exposes type, size and timestamps through Files.readAttributes. File.listFiles() may return null on failure. In recursive processing, separate directory traversal from the filename/type test applied to files.

Concurrency

Java SE 8 Programmer II

KNOWLEDGE CHEATSHEET

CONCURRENT TASK LIFECYCLE

Submit Runnable or Callable → **Execute** Worker pool → **Future** Wait for result → **Shutdown** Finish tasks

08 Concurrency

Tasks, executors and futures

- **Thread.start()** schedules a new thread; calling run() directly executes on the caller. Scheduling and printed output order are generally unpredictable without explicit coordination.
- **Runnable.run()** returns void and cannot declare checked exceptions. **Callable<V>.call()** returns V and may throw Exception. ExecutorService.execute accepts Runnable; submit accepts Runnable or Callable and returns Future.
- **Future.get()** waits for completion, returns the Callable result, and wraps task failure in ExecutionException. submit(Runnable) normally produces null on successful get(); submit(Runnable,result) returns the supplied result.
- **shutdown()** rejects new tasks while allowing submitted tasks to finish. **awaitTermination** waits after shutdown but does not initiate it. Shut down executor pools explicitly so their worker threads do not keep the application alive.

Shared state and coordination

- **synchronized** provides exclusive access using one monitor and releases it on normal or exceptional exit. An instance synchronized method locks this; a static synchronized method locks the Class object.
- Threads must use the **same lock** to coordinate shared state. Synchronizing separate Runnable instances does not serialize them. A lock prevents overlap but does not guarantee which waiting thread runs first.
- **volatile** provides visibility and ordering for a field, but count++ is still a compound operation. **AtomicInteger.incrementAndGet()** atomically increments and returns the new value; getAndIncrement returns the old value. Printing can occur in a different order.
- **ReentrantLock** supports explicit locking; unlock in finally after successful acquisition. **CyclicBarrier** waits for its configured number of parties and may run a barrier action before releasing them; it can be reused.
- A **race condition** depends on uncontrolled timing. **Deadlock** involves cyclic waiting; **starvation** prevents a thread from obtaining resources; **livelock** repeatedly changes state without useful progress. Consistent lock order helps prevent deadlock.

Concurrency / JDBC

Java SE 8 Programmer II

KNOWLEDGE CHEATSHEET

DATABASE INTERACTION

DriverManager Connect → **Connection** Session → **Statement** Execute SQL → **ResultSet** Read rows

Fork/join and parallel work

- **ForkJoinPool** uses work-stealing for recursively divided tasks. **RecursiveTask<V>** returns a result; **RecursiveAction** returns none. Implement compute() with a base case and a sensible splitting threshold.
- A common pattern is **fork one task, compute another, then join**. join() waits for the forked result; invoke() submits and waits for the root task. Ensure child work is joined before treating the parent as complete.
- **Parallel work** needs enough independent computation to offset scheduling and combination costs. Blocking I/O, shared state and very small tasks can reduce performance. Parallel execution alone supplies neither output order nor thread safety.

09 JDBC

Connections and SQL execution

- **DriverManager.getConnection(url,user,password)** or getConnection(url,properties) opens a connection through a matching JDBC driver. Properties commonly uses user and password keys; the driver implements Connection, Statement and ResultSet interfaces.
- **JDBC 3** commonly loads a driver using Class.forName(driverClass). **JDBC 4** drivers support automatic discovery through the service-provider mechanism when correctly packaged and available to the application.
- **executeQuery(sql)** returns ResultSet for a query. **executeUpdate(sql)** returns an affected-row count for DML, or zero for statements without a count. **execute(sql)** returns true if the first result is a ResultSet, not merely if execution succeeded.
- A **Connection** is a database session; a Statement executes SQL; a ResultSet exposes results through a cursor. Reexecuting the same Statement normally closes its current ResultSet. Closing the Statement closes its current result set.
- Manage **JDBC resources** with try-with-resources and keep them open while used. Returning a connection from inside its resource block returns a closed connection. **SQLException** is checked and must be caught or declared.

JDBC

Java SE 8 Programmer II

KNOWLEDGE CHEATSHEET

DATABASE INTERACTION

DriverManager Connect → **Connection** Session → **Statement** Execute SQL → **ResultSet** Read rows

Result sets and updates

- A ResultSet cursor starts **before the first row**. next() moves to a row and returns whether one exists. Getter columns are **1-based** and may also use column labels; select a getter compatible with the SQL value.
- Primitive getters return **0 or false for SQL NULL**; call wasNull() immediately afterward to distinguish null. getString() can return null. **ResultSetMetaData** describes columns, not row counts; count rows by iterating with next().
- The default is **TYPE_FORWARD_ONLY** and **CONCUR_READ_ONLY**. Scrollable results support previous(), first(), last() and absolute(n), subject to driver support. Positive absolute positions count from 1; negative positions count from the end.
- For an **updatable ResultSet**, updateString/updateInt stage a row change; **updateRow()** writes it. moveToInsertRow(), setters and insertRow() insert a row; use moveToCurrentRow() to return to the saved cursor position.
- **TYPE_SCROLL_INSENSITIVE** generally does not reflect other transactions' later changes; sensitive behavior depends on the driver and database. Set Statement options such as maxRows before executing the query they should affect.
- **Auto-commit** is normally enabled. With setAutoCommit(false), commit() makes transaction changes durable and rollback() reverses uncommitted work. Result-set update capability and transaction commit are separate requirements.