Java Quick Reference

OOP, collections, streams, exception handling essentials

Basics

Hello World

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

Compile & Run

```
javac Main.java      # compile
java Main             # run
java Main.java        # single-file (Java 11+)
```

Naming Conventions

`ClassName`	PascalCase for classes and interfaces
`methodName`	camelCase for methods and variables
`CONSTANT_NAME`	UPPER_SNAKE for constants
`com.example.pkg`	Lowercase reverse domain for packages

Data Types

Primitives

`byte`	8-bit signed (-128 to 127)
`short`	16-bit signed
`int`	32-bit signed (default integer)
`long`	64-bit signed (suffix **`L`**)
`float`	32-bit IEEE-754 (suffix **`f`**)
`double`	64-bit IEEE-754 (default decimal)
`boolean`	**`true`** / **`false`**
`char`	16-bit Unicode character

Strings

```
String s = "hello";
String joined = s + " world";      // concatenation
int len = s.length();
String sub = s.substring(0, 3);    // "hel"
boolean eq = s.equals("hello");    // content equality
```

Type Casting

```
int i = (int) 3.14;               // narrowing cast
double d = i;                      // widening (auto)
int n = Integer.parseInt("42");    // string to int
String s = String.valueOf(42);     // int to string
```

Arrays

```
int[] nums = {1, 2, 3};
String[] names = new String[5];
int[][] matrix = new int[3][4];
Arrays.sort(nums);
```

Control Flow

If / Else

```
if (x > 0) {
    System.out.println("positive");
} else if (x == 0) {
    System.out.println("zero");
} else {
    System.out.println("negative");
}
```

Switch

```
// Traditional
switch (day) {
    case "Mon": doWork(); break;
    case "Sat": case "Sun": rest(); break;
    default: routine();
}
// Switch expression (Java 14+)
String type = switch (day) {
    case "Sat", "Sun" -> "weekend";
    default -> "weekday";
};
```

Loops

```
for (int i = 0; i < 10; i++) { }
for (String s : list) { }         // enhanced for
while (condition) { }
do { } while (condition);
```

Methods

Definition

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

Varargs & Overloading

```
static int sum(int... nums) {
    int total = 0;
    for (int n : nums) total += n;
    return total;
}
// sum(1, 2)  sum(1, 2, 3)  both work
```

Access Modifiers

`public`	Accessible from anywhere
`protected`	Same package + subclasses
`(default)`	Same package only (no keyword)
`private`	Same class only

Classes & Objects

Class Definition

```
public class User {
    private String name;
    private int age;
    public User(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public String getName() { return name; }
}
```

Records (Java 16+)

```
public record Point(double x, double y) {
    // auto: constructor, getters, equals, hashCode, toString
    public double distance() {
        return Math.sqrt(x * x + y * y);
    }
}
```

Static & Final

`static`	Belongs to class, not instance
`final field`	Cannot be reassigned after init
`final method`	Cannot be overridden
`final class`	Cannot be subclassed

Inheritance

Extends

```
public class Animal {
    public void speak() { System.out.println("..."); }
}
public class Dog extends Animal {
    @Override
    public void speak() { System.out.println("Woof!"); }
}
```

Abstract Classes

```
public abstract class Shape {
    abstract double area();
    public void describe() {
        System.out.println("Area: " + area());
    }
}
```

Key Concepts

`super`	Call parent constructor or method
`@Override`	Compile-time override check
`instanceof`	Runtime type check
`sealed (17+)`	Restrict which classes can extend

Interfaces

Definition

```
public interface Printable {
    void print();                      // abstract
    default String format() {          // default method
        return toString();
    }
    static Printable of(String s) {  // static method
        return () -> System.out.println(s);
    }
}
```

Implementation

```
public class Report implements Printable, Serializable {
    @Override
    public void print() {
        System.out.println("Report");
    }
}
```

Functional Interfaces

`Runnable`	**`() -> void`**
`Supplier<T>`	**`() -> T`**
`Consumer<T>`	**`T -> void`**
`Function<T,R>`	**`T -> R`**
`Predicate<T>`	**`T -> boolean`**
`Comparator<T>`	**`(T, T) -> int`**

Collections

List

```
List<String> list = new ArrayList<>();
list.add("a");
list.get(0);                  // "a"
list.size();                  // 1
List<String> immutable = List.of("a", "b", "c");
```

Java Quick Reference

Map

```
Map<String, Integer> map = new HashMap<>();
map.put("key", 42);
map.getOrDefault("key", 0);   // 42
map.containsKey("key");       // true
map.forEach((k, v) -> { });
```

Set

```
Set<String> set = new HashSet<>();
set.add("a");
set.contains("a");            // true
Set<String> immutable = Set.of("a", "b", "c");
```

Common Implementations

`ArrayList`	Resizable array, fast random access
`LinkedList`	Doubly-linked, fast insert/remove
`HashMap`	Hash table, O(1) get/put
`TreeMap`	Sorted by key, O(log n)
`HashSet`	Unique elements, O(1) lookup
`LinkedHashMap`	Insertion-ordered HashMap

Exception Handling

Try / Catch / Finally

```
try {
    int result = 10 / 0;
} catch (ArithmeticException e) {
    System.err.println(e.getMessage());
} finally {
    // always executes
}
```

Try-with-Resources

```
try (var reader = new BufferedReader(new FileReader(path))) {
    String line = reader.readLine();
}   // auto-closes reader
```

Exception Hierarchy

`Throwable`	Root of all errors and exceptions
`Error`	Serious problems (OutOfMemoryError)
`Exception`	Checked exceptions (must handle)
`RuntimeException`	Unchecked (NullPointer, IndexOutOfBounds)

Custom Exception

```
public class AppException extends Exception {
    public AppException(String msg) { super(msg); }
    public AppException(String msg, Throwable cause) {
        super(msg, cause);
    }
}
```

Streams & Lambdas

Lambda Syntax

```
Comparator<String> byLen = (a, b) -> a.length() - b.length();
Runnable task = () -> System.out.println("run");
Function<String, Integer> len = String::length;  // method ref
```

Stream Pipeline

```
List<String> result = names.stream()
    .filter(n -> n.length() > 3)
    .map(String::toUpperCase)
    .sorted()
    .collect(Collectors.toList());
```

Common Stream Operations

`.filter(pred)`	Keep elements matching predicate
`.map(func)`	Transform each element
`.flatMap(func)`	Map and flatten nested streams
`.sorted()`	Sort (natural or with Comparator)
`.distinct()`	Remove duplicates
`.limit(n)`	Take first n elements
`.collect()`	Terminal: gather into collection
`.forEach()`	Terminal: perform action on each
`.reduce()`	Terminal: combine into single value
`.count()`	Terminal: count elements

Generics

Generic Class & Method

```
public class Box<T> {
    private T value;
    public Box(T value) { this.value = value; }
    public T get() { return value; }
}
public static <T> List<T> listOf(T... items) {
    return List.of(items);
}
```

Bounded Types & Wildcards

`<T extends Number>`	T must be Number or subclass
`<T extends A & B>`	Multiple bounds (class + interfaces)
`<?>`	Unknown type (read-only)
`<? extends T>`	Upper bound wildcard (producer)
`<? super T>`	Lower bound wildcard (consumer)

Optional & Modern Java

Optional

```
Optional<String> opt = Optional.ofNullable(getValue());
String result = opt.orElse("default");
opt.ifPresent(v -> System.out.println(v));
String upper = opt.map(String::toUpperCase).orElse("");
```

Text Blocks (Java 15+)

```
String json = """
       { "name": "Alice", "age": 30 }
       """;
```

Useful Utilities

`var (10+)`	Local variable type inference
`record (16+)`	Immutable data carrier class
`sealed (17+)`	Restricted class hierarchies
`pattern matching (21+)`	**`instanceof`** with auto-cast
`virtual threads (21+)`	Lightweight threads via **`Thread.ofVirtual()`**