**Callable & Future Interfaces**

**Theory Notes**

* Callable<V> is like Runnable but returns a value of type V and can throw an exception.
* Future<V> represents the result of an asynchronous computation — you can check if it’s done, cancel it, or get the result.
* Used in thread-pool / executor contexts when you need a result back.
* Typical flow: submit a Callable to an ExecutorService, you get a Future, then you call future.get() to obtain the result (blocking if necessary).
* Key differences from Runnable: return value, checked exception support.
* Pitfalls: future.get() may block indefinitely unless you handle timeouts; cancellation and interruption require careful handling.

**Code Sample**

import java.util.concurrent.\*;

public class CallableFutureDemo {

public static void main(String[] args) throws InterruptedException, ExecutionException {

ExecutorService executor = Executors.newFixedThreadPool(2);

Callable<Integer> task = () -> {

System.out.println("Starting task in " + Thread.currentThread().getName());

Thread.sleep(1000);

return 123;

};

Future<Integer> future = executor.submit(task);

System.out.println("Do something else while task executes…");

Integer result = future.get(); // blocks until done

System.out.println("Result from Callable: " + result);

executor.shutdown();

}

}

**Training bullets**

* What is Callable<V> vs Runnable.
* What is Future<V> and how to use methods: get(), isDone(), cancel().
* Submit tasks via ExecutorService.
* Handling timeouts and cancellation.
* When to prefer Callable + Future vs Runnable.

**2. CompletableFuture & CompletionStage**

**Theory Notes**

* CompletableFuture<T> implements Future<T> and CompletionStage<T>; it allows you to build asynchronous, non-blocking pipelines of tasks.
* You can attach callbacks (via thenApply, thenAccept, thenCompose, etc), handle exceptions (exceptionally, handle), combine independent tasks (allOf, anyOf), chain tasks.
* CompletionStage<T> is the interface that defines these chaining operations; CompletableFuture is the implementor.
* Benefits: more expressive asynchronous programming, better for functional style, avoids blocking get() where possible.

**Code Sample**

import java.util.concurrent.\*;

public class CompletableFutureDemo {

public static void main(String[] args) {

CompletableFuture<Integer> future = CompletableFuture.supplyAsync(() -> {

System.out.println("Compute in " + Thread.currentThread().getName());

return 50;

});

CompletableFuture<Integer> doubled = future.thenApply(result -> {

System.out.println("Doubling result");

return result \* 2;

});

CompletableFuture<Void> finalStage = doubled.thenAccept(result -> {

System.out.println("Final result: " + result);

});

// wait for completion

finalStage.join();

System.out.println("Done");

}

}

**Training bullets**

* Difference between Future and CompletableFuture.
* How to start tasks (e.g., supplyAsync, runAsync).
* Chaining operations: thenApply, thenAccept, thenCompose.
* Handling exceptions in async.
* Combining futures.
* Use cases: parallel tasks, reactive patterns.

**3. Brief Intro to Java 9–17 Features**

**Theory Notes**  
Here’s a high-level summary of key features from Java 9 through Java 17:

* Java 9: Module system (Project Jigsaw), factory methods for collections (List.of(), etc), improved try-with-resources, etc. [Oracle Docs+2nikgrozev.com+2](https://docs.oracle.com/en/java/javase/23/language/java-language-changes.html?utm_source=chatgpt.com)
* Java 10: var for local variable type inference. [nikgrozev.com](https://nikgrozev.com/2021/12/04/java-9-and-afterwards/?utm_source=chatgpt.com)
* Java 11: More String methods, file-API enhancements (e.g. Files.readString). [GUVI+1](https://www.guvi.in/blog/top-java-9-to-java-17-interview-questions/?utm_source=chatgpt.com)
* Java 12-14: switch expressions (with yield), text blocks. [GUVI+1](https://www.guvi.in/blog/top-java-9-to-java-17-interview-questions/?utm_source=chatgpt.com)
* Java 16–17: records, immutable data classes, additional enhancements. [Medium+1](https://medium.com/%40teggourabdenour/whats-new-in-java-17-and-java-21-unlocking-the-power-of-modern-java-a42da5a759c0?utm_source=chatgpt.com)

**Training bullets**

* Why upgrading from Java 8 matters for developers.
* Modularisation, local variable inference, collection factory methods.
* Enhanced language features (switch, text blocks, records).
* Identify at least 2 features you can start using today in your codebase.

**Code Examples**

// var usage (Java 10+)

var list = List.of("Java", "Kotlin", "Scala");

for (var item : list) {

System.out.println(item);

}

// Text block (Java 13+)

String json = """

{

"name": "Alice",

"age": 30

}

""";

System.out.println(json);

// Record (Java 16+)

record Person(String name, int age) {}

Person p = new Person("Bob", 25);

System.out.println(p.name() + ", age = " + p.age());

**4. Collectors API & Immutable Collections**

**Theory Notes**

* Collectors (in java.util.stream.Collectors) enable you to collect stream results into collections, maps, summarise values, partitioning, grouping, etc.
* Immutable collections: since Java 9 you have List.of(...), Set.of(...), Map.of(...), which produce unmodifiable collections (no adds/removes) and help enforce immutability. [Medium+2nikgrozev.com+2](https://medium.com/javarevisited/java-8-9-11-and-17-what-you-need-to-know-about-the-key-differences-between-these-versions-808dfee652ba?utm_source=chatgpt.com)
* Why immutability matters: thread-safety, fewer bugs, simpler reasoning.

**Code Sample**

import java.util.\*;

import java.util.stream.\*;

public class CollectorsImmutableDemo {

public static void main(String[] args) {

List<String> names = List.of("Alice", "Bob", "Charlie");

Map<String, Integer> nameLengthMap = names.stream()

.collect(Collectors.toMap(name -> name, name -> name.length()));

System.out.println(nameLengthMap);

// Immutable collections

List<String> immList = List.of("Java", "Python");

Set<String> immSet = Set.of("Apple", "Banana");

Map<String, Double> immMap = Map.of("Pi", 3.14, "e", 2.71);

System.out.println(immList);

System.out.println(immSet);

System.out.println(immMap);

// immList.add("C++"); // throws UnsupportedOperationException

}

}

**Training bullets**

* How to use Collectors.toList(), toMap(), groupingBy(), partitioningBy().
* Advantages of immutable collections; when to use them.
* Converting existing modifiable collections to immutable.

**5. var keyword (Local Variable Type Inference)**

**Theory Notes**

* Introduced in Java 10: you can use var to declare local variables; the compiler infers the type. [nikgrozev.com](https://nikgrozev.com/2021/12/04/java-9-and-afterwards/?utm_source=chatgpt.com)
* Benefits: less boilerplate, clearer code when the type is obvious from context.
* Caution: overuse can reduce readability; still good to specify type when it improves clarity.

**Code Sample**

public class VarDemo {

public static void main(String[] args) {

var list = List.of("One", "Two", "Three");

for (var s : list) {

System.out.println(s.toUpperCase());

}

var map = Map.of("A", 1, "B", 2);

map.forEach((var key, var value) -> {

System.out.println(key + " -> " + value);

});

}

}

**Training bullets**

* Syntax of var.
* Rules: only for local variables, must be initialized.
* When *not* to use var.
* Differences between var and explicit types.

**6. File APIs (Modern Java File/Path API)**

**Theory Notes**

* Java 7 introduced java.nio.file.Path, Files APIs; since later versions enhancements added easier methods (Files.readString, writeString in Java 11) [GUVI+1](https://www.guvi.in/blog/top-java-9-to-java-17-interview-questions/?utm_source=chatgpt.com)
* Use Path.of(...), Files.newBufferedReader(), Files.list(), etc. Better and more modern than old File class.

**Code Sample**

import java.nio.file.\*;

import java.io.IOException;

public class FileApiDemo {

public static void main(String[] args) {

Path path = Path.of("sample.txt");

try {

Files.writeString(path, "Hello, modern Java File API!");

String content = Files.readString(path);

System.out.println("File content: " + content);

Files.lines(path)

.forEach(System.out::println);

// list files in current directory

Files.list(Path.of("."))

.filter(p -> p.toString().endsWith(".java"))

.forEach(System.out::println);

} catch (IOException e) {

e.printStackTrace();

}

}

}

**Training bullets**

* Path vs File.
* Key Files methods: readString, writeString, lines, list.
* When to use Streams for files.
* Handling exceptions & resources.

**7. Switch expressions**

**Theory Notes**

* From Java 12/13 onwards: switch can be used as an expression (returns a value) using -> syntax or block with yield. [GUVI+1](https://www.guvi.in/blog/top-java-9-to-java-17-interview-questions/?utm_source=chatgpt.com)
* yield allows returning a value from a block case.
* Advantages: more concise, prevents fall-through, clearer return semantics.

**Code Sample**

public class SwitchExpressionDemo {

public static String getDayType(int day) {

return switch (day) {

case 1, 2, 3, 4, 5 -> "Weekday";

case 6, 7 -> "Weekend";

default -> {

System.out.println("Invalid day: " + day);

yield "Unknown";

}

};

}

public static void main(String[] args) {

System.out.println(getDayType(3)); // Weekday

System.out.println(getDayType(7)); // Weekend

System.out.println(getDayType(10)); // Unknown

}

}

**Training bullets**

* Syntax difference between statement and expression.
* case … -> vs old case … :.
* Use of yield.
* When to choose switch expression vs traditional switch.

**8. Text blocks**

**Theory Notes**

* Introduced as preview in Java 13, finalized in Java 15: triple-quoted string literals (""" … """) for multi-line text. [Medium+1](https://medium.com/%40teggourabdenour/whats-new-in-java-17-and-java-21-unlocking-the-power-of-modern-java-a42da5a759c0?utm_source=chatgpt.com)
* Cleaner for JSON, HTML, SQL embedded strings. Maintains indentation, fewer escape sequences.

**Code Sample**

public class TextBlockDemo {

public static void main(String[] args) {

String html = """

<html>

<body>

<h1>Hello Modern Java</h1>

</body>

</html>

""";

System.out.println(html);

}

}

**Training bullets**

* When to use text blocks.
* Differences vs traditional string literal.
* Handling indent and trimming.
* Compatibility concerns.

**9. Records**

**Theory Notes**

* Since Java 14 as preview, finalized in Java 16: record keyword for immutable data-carrier classes. [Medium+1](https://medium.com/javarevisited/java-8-9-11-and-17-what-you-need-to-know-about-the-key-differences-between-these-versions-808dfee652ba?utm_source=chatgpt.com)
* Records automatically create: private final fields, constructor, getters, equals(), hashCode(), toString().
* Use when you have simple data-holding classes.

**Code Sample**

public record Student(String name, int age, double grade) {}

public class RecordsDemo {

public static void main(String[] args) {

Student s = new Student("Neha", 22, 8.5);

System.out.println(s.name() + " is " + s.age() + " years old, grade " + s.grade());

System.out.println(s);

}

}

**Training bullets**

* Syntax of record.
* When to use record vs class.
* Immutability advantage.
* Limitations (cannot extend other classes, etc).

**10. Log4j Framework Utilization (Logging)**

**Theory Notes**

* Logging is essential for troubleshooting, auditing, performance monitoring.
* Log4j (or newer versions like Log4j2) is a common logging framework in enterprise Java.
* Basic elements: logger, appenders, layouts, configuration (XML/JSON/properties).
* Best practices: avoid System.out.println() for prod apps, use logging levels (DEBUG, INFO, WARN, ERROR), configure log rotation, externalize configuration.

**Code Sample**

import org.apache.logging.log4j.LogManager;

import org.apache.logging.log4j.Logger;

public class LoggingDemo {

private static final Logger logger = LogManager.getLogger(LoggingDemo.class);

public static void main(String[] args) {

logger.info("Application started");

try {

int result = compute(5, 0);

logger.debug("Result = " + result);

} catch (Exception e) {

logger.error("Exception occurred while computing", e);

}

logger.info("Application finished");

}

private static int compute(int a, int b) {

return a / b; // will cause ArithmeticException

}

}

**Training bullets**

* Setup Log4j (dependency + configuration file).
* Logging levels and when to use each.
* External configuration (change log behaviour without code change).
* Avoid logging sensitive info, ensure performance (lazy evaluation, guard log statements).
* Integration with frameworks (Spring Boot default logging, etc).