# **Lambda expressions in Java**

**1. What Is a Lambda Expression?**

A **lambda expression** is an anonymous, function‑style block of code you can pass around as an object. In Java, lambdas target a **functional interface**—an interface with exactly one abstract method. They look like:

(parameters) -> expression

singleParam -> expression

(parameters) -> { statements; }

**Example**:

// Functional interface

@FunctionalInterface

interface Converter<F,T> {

T convert(F from);

}

// Lambda converting String to Integer

Converter<String,Integer> toInt = s -> Integer.parseInt(s);

int x = toInt.convert("123"); // 123

**2. Why Were Lambdas Introduced?**

1. **Reduce Boilerplate**  
   Before Java 8, callbacks and one‑off behaviors required anonymous inner classes:

button.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent e) {

System.out.println("Clicked!");

}

});

Lambdas shrink this to:

button.addActionListener(e -> System.out.println("Clicked!"));

1. **Enable Functional APIs**  
   APIs like **Streams**, **CompletableFuture**, and many third‑party libraries lean on lambdas for concise, chainable pipelines.
2. **Ease Parallelism**  
   By expressing “what to do” (via lambdas) instead of “how,” the runtime can transparently split work across threads (e.g. parallelStream()).

**3. Rules of Lambda Expressions**

1. **Target a Functional Interface**  
   The lambda’s shape must match a single abstract method’s signature.
2. **“Effectively Final” Variables**  
   You can capture (use) local variables only if they are final or never reassigned after initialization.
3. **Cannot Declare Throws (Unless in Interface)**  
   A lambda may only throw checked exceptions declared by its target method.
4. **this Refers to Enclosing Instance**  
   Inside a lambda, this is the surrounding class, not a hidden lambda object.
5. **No New Local Variable Shadowing**  
   You can’t reuse a local variable name that’s in scope around the lambda.
6. **Single Abstract Method Only**  
   Default and static methods in the interface don’t count; only one abstract method is allowed.

**4. Real‑World Use Cases & Examples**

**4.1 Collection Sorting and Filtering**

List<User> users = fetchUsers();

// Sort by age descending

users.sort((a, b) -> Integer.compare(b.getAge(), a.getAge()));

// Get names of active users

List<String> activeNames = users.stream()

.filter(u -> u.isActive())

.map(User::getName)

.collect(Collectors.toList());

**4.2 Thread Creation**

// Before Java 8: new Thread(new Runnable(){ … }).start();

// With lambda:

new Thread(() -> doBackgroundWork()).start();

**4.3 Scheduled Tasks**

ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(1);

scheduler.scheduleAtFixedRate(

() -> System.out.println("Heartbeat at " + Instant.now()),

0, 1, TimeUnit.SECONDS

);

**4.4 Asynchronous Pipelines**

CompletableFuture.supplyAsync(() -> fetchData())

.thenApply(data -> parse(data))

.thenAccept(result -> sendNotification(result));

**4.5 Custom Callback Interfaces**

@FunctionalInterface

interface RetryPolicy {

boolean shouldRetry(int attempt, Exception e);

}

void callWithRetry(RetryPolicy policy) {

for (int i = 1; ; i++) {

try { remoteCall(); break; }

catch (Exception ex) {

if (!policy.shouldRetry(i, ex)) throw ex;

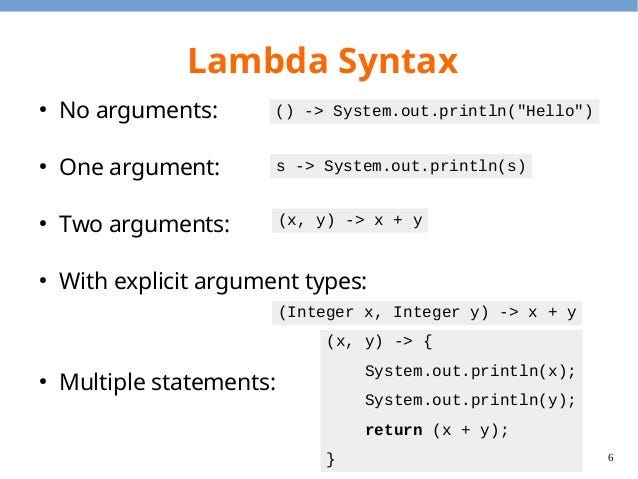
}

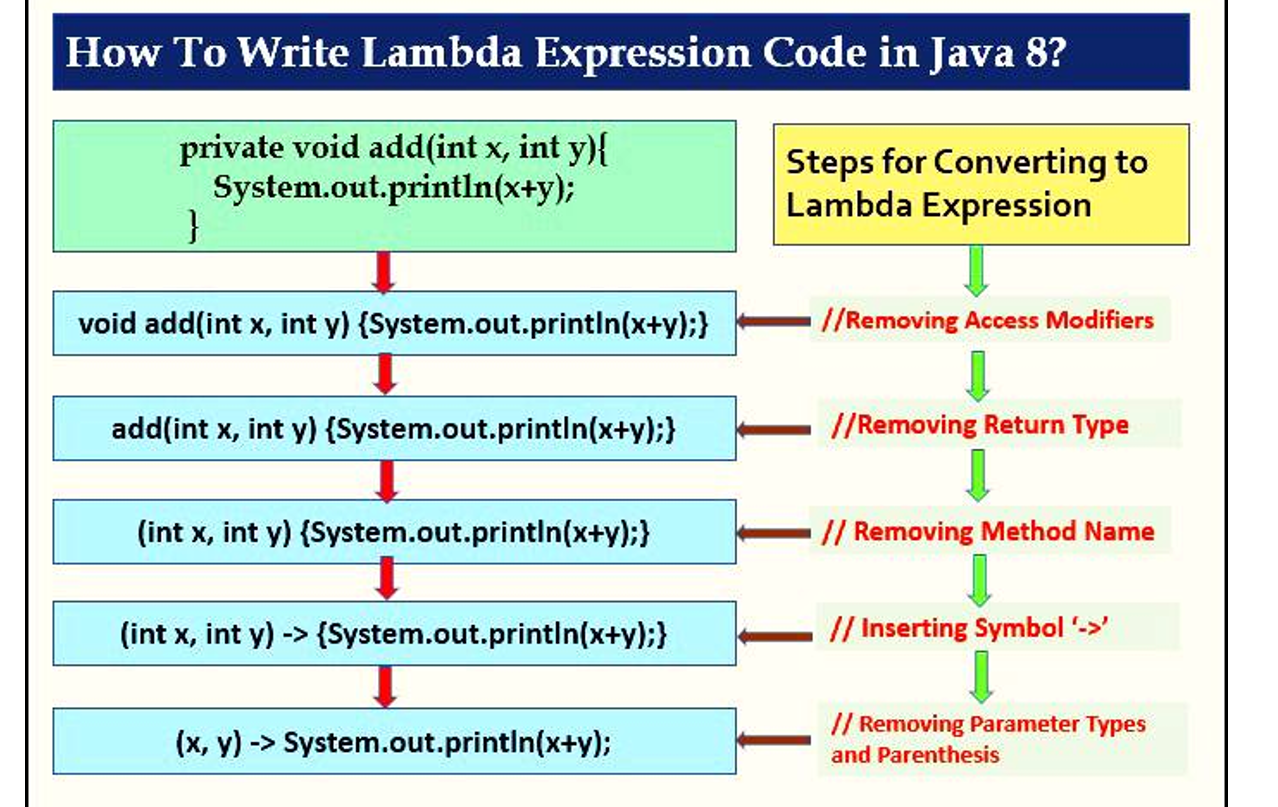
}

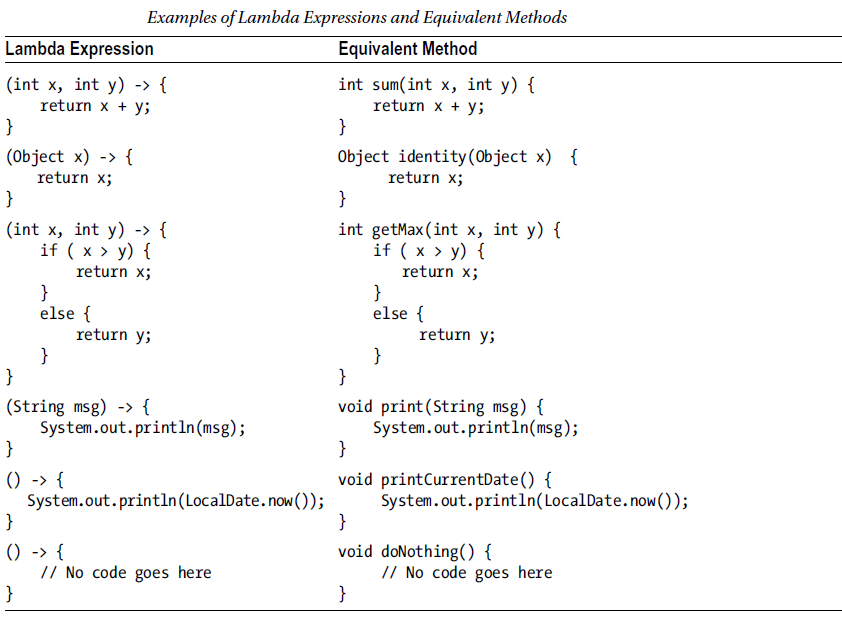
}

// Use a simple policy

callWithRetry((attempt, ex) -> attempt < 3);







**Quiz: Test Your Lambda Knowledge**

**1. Which of these is a valid lambda targeting Function<String,Integer>?**

A. str -> Integer.parseInt(str)  
B. (String s) -> return Integer.parseInt(s);  
C. s -> { Integer.parseInt(s); }  
D. () -> Integer.parseInt("123")

<details><summary>Answer</summary>A is valid. B needs no `return` keyword inside a single‑expression lambda unless you use `{}`; C has no `return`; D has wrong signature.</details>

**2. What does “effectively final” mean for lambda-captured variables?**

A variable is **effectively final** if it’s declared final.  
A variable is **effectively final** if its value never changes after initialization.  
A. Only variables declared final  
B. Variables that aren’t reassigned after initialization  
C. Variables only assigned in the lambda  
D. Local loop indices

<details><summary>Answer</summary>B</details>

**3. Inside a lambda, this refers to…**

A. A hidden instance of the lambda class  
B. The enclosing class instance  
C. null  
D. The functional interface type

<details><summary>Answer</summary>B</details>

**4. Which built‑in interface takes no arguments and returns a value?**

A. Consumer<T>  
B. Supplier<T>  
C. Predicate<T>  
D. Function<T,R>

<details><summary>Answer</summary>B</details>

**5. Can a lambda throw checked exceptions?**

A. Yes, any checked exception  
B. No  
C. Only if the functional interface method declares it  
D. Only RuntimeExceptions

<details><summary>Answer</summary>C</details>

**6. What’s the result of this code?**

Stream.of(1,2,3,4)

.map(n -> n\*2)

.filter(n -> n>5)

.findFirst();

A. Optional.of(6)  
B. Optional.of(8)  
C. Optional.empty()  
D. Compilation error

<details><summary>Answer</summary>A</details>

**7. Which method reference maps to String::toUpperCase?**

A. s -> s.toUpperCase()  
B. String::toLowerCase  
C. x -> x.toUpperCase  
D. String::toUpperCase()

<details><summary>Answer</summary>A and B are both valid but only A maps to `toUpperCase` behavior. The correct target is `String::toUpperCase` without parentheses. </details>

**8. Which of these is NOT a functional interface?**

A. Runnable  
B. Callable<V>  
C. Comparator<T>  
D. Iterable<T>

<details><summary>Answer</summary>D (`Iterable` has more than one abstract method)</details>

**9. What happens if you capture a non-final variable and then reassign it?**

A. Compiler error  
B. Runtime exception  
C. The lambda sees the updated value  
D. It compiles but behaves unpredictably

<details><summary>Answer</summary>A</details>

**10. True or False: Lambdas can be used wherever a target type is a functional interface.**

<details><summary>Answer</summary>True</details>

You should reach for a **lambda** whenever you need to pass a **small piece of behavior** (a “function”) into an API or framework—especially when that API defines a **functional interface** (one single abstract method). Here are the most common real‑world scenarios:

**1. Replacing Anonymous Inner Classes**

**Before** (anonymous class):

button.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent e) {

saveDocument();

}

});

**After** (lambda):

button.addActionListener(e -> saveDocument());

**Why?** Lambdas cut through the boilerplate (no class name, no new, no override).

**2. Collection Processing with Streams**

Whenever you filter, map, group or reduce lists, sets, maps—use lambdas in your stream pipelines:

List<User> users = fetchUsers();

// Find all active user emails sorted alphabetically

List<String> emails = users.stream()

.filter(u -> u.isActive())

.map(User::getEmail)

.sorted()

.collect(Collectors.toList());

**Why?** It reads like a sentence: filter the active ones, map to email, sort, collect.

**3. Sorting & Comparators**

Instead of writing a comparator class:

// Before

Collections.sort(products, new Comparator<Product>() {

public int compare(Product a, Product b) {

return a.getPrice().compareTo(b.getPrice());

}

});

// With lambda

products.sort((a, b) -> a.getPrice().compareTo(b.getPrice()));

**4. Runnable / Callable for Threads**

Thread creation:

// Before

new Thread(new Runnable() {

public void run() {

processQueue();

}

}).start();

// With lambda

new Thread(() -> processQueue()).start();

Or submit to an ExecutorService:

executor.submit(() -> sendReport());

**5. Scheduling Tasks**

Periodic jobs with ScheduledExecutorService:

ScheduledExecutorService svc = Executors.newScheduledThreadPool(1);

svc.scheduleAtFixedRate(

() -> checkHealthMetrics(),

0, 1, TimeUnit.MINUTES

);

**6. Asynchronous Callbacks**

With CompletableFuture:

CompletableFuture.supplyAsync(() -> downloadData())

.thenApply(data -> parse(data))

.thenAccept(result -> notifyUser(result))

.exceptionally(ex -> { logError(ex); return null; });

**7. Strategy or Policy Injection**

Define a tiny interface:

@FunctionalInterface

interface DiscountPolicy {

double apply(double price);

}

// Use it in your code

double checkout(double price, DiscountPolicy policy) {

return policy.apply(price);

}

// Callsite

double finalPrice = checkout(200.0, p -> p > 100 ? p \* 0.9 : p);

**8. Lightweight Listeners & Hooks**

Whenever a library asks you to implement a single‑method listener or hook:

watchService.register(path, ENTRY\_CREATE, ENTRY\_MODIFY, ENTRY\_DELETE, evt ->

handleFileEvent(evt)

);

**Key Guidelines:**

* **Target a functional interface:** Only use lambdas where the API expects one.
* **Keep them short:** If the body grows beyond a few lines, factor it into a named method and reference it (Foo::bar).
* **Leverage method references:** When your lambda just calls one method, Class::method is even more concise.
* **Avoid complex logic in lambdas:** They’re best for succinct behavior; rich workflows belong in full methods or classes.

**Method references**  
Method references in Java are a **shorthand notation** for lambdas whose only action is to call an existing method. They let you **reuse** method implementations without writing the boilerplate of a lambda that simply delegates to it—making code **more concise**, **readable**, and **self‑documenting**.

**1. Why Were Method References Added?**

* **Reduce Boilerplate**: A lambda like s -> s.toUpperCase() can be replaced by String::toUpperCase.
* **Improve Readability**: The intent “call toUpperCase on each element” is clearer than a little inline lambda.
* **Seamless with Functional Interfaces**: They fit right into any API expecting a single‑abstract‑method interface (e.g. Streams, listeners, Executors).
* **Introduced in Java 8** alongside lambdas and the Streams API to complete the functional‑style toolkit.

**2. The Four Kinds of Method References**

| **Kind** | **Syntax** | **Example Lambda** | **Equivalent Method Reference** |
| --- | --- | --- | --- |
| 1. Static method | ClassName::staticMethod | s -> Integer.parseInt(s) | Integer::parseInt |
| 2. Instance method of **a particular object** | instance::instanceMethod | e -> logger.log(e) | logger::log |
| 3. Instance method of an **arbitrary object of a type** | ClassName::instanceMethod | s -> s.trim() | String::trim |
| 4. Constructor | ClassName::new | () -> new ArrayList<>() | ArrayList::new |

**3. Real‑World Examples**

**3.1 Static‑Method Reference**

**Parse a list of Strings into Integers**

List<String> inputs = List.of("10","20","30");

List<Integer> nums = inputs.stream()

.map(Integer::parseInt) // instead of s -> Integer.parseInt(s)

.collect(Collectors.toList());

**3.2 Particular‑Instance Method Reference**

**Logging events**

Logger audit = Logger.getLogger("audit");

events.forEach(audit::info); // instead of e -> audit.info(e)

**3.3 Arbitrary‑Instance Method Reference**

**Trim whitespace from each CSV field**

List<String> fields = rawLine.split(",");

List<String> clean = Arrays.stream(fields)

.map(String::trim) // instead of s -> s.trim()

.collect(Collectors.toList());

**3.4 Constructor Reference**

**Create a new ArrayList for each group**

Map<String, List<User>> byDept = users.stream()

.collect(Collectors.groupingBy(

User::getDepartment,

Collectors.toCollection(ArrayList::new) // instead of () -> new ArrayList<>()

));

**4. When & Where to Use Method References**

1. **Stream Pipelines**
   * Mapping: users.stream().map(User::getName)
   * Filtering: strings.filter(String::isEmpty)
   * For‑each: tasks.forEach(Task::execute)
2. **Comparator Construction**

products.sort(Comparator.comparing(Product::getPrice));

1. **Event Handlers / Listeners**

button.addActionListener(this::onClick);

1. **Executor / Threads**

executor.submit(this::cleanupTempFiles);

1. **Factory Methods**

Stream.generate(Math::random)

.limit(5)

.forEach(System.out::println);

Use them **anywhere** the body of your lambda would be just one method call (possibly on the lambda parameter), and the method’s signature matches the abstract method of your target functional interface.

**5. Key Takeaways**

* **Readability**: Class::method is often clearer than a tiny lambda.
* **Conciseness**: Cuts boilerplate of parameter lists and return statements.
* **Interoperability**: Works with any Java 8+ API expecting a functional interface.
* **Maintainability**: When the method’s implementation changes, all callers automatically pick up the new behavior without touching the lambda code.

**Optional<T>** was introduced in **Java 8** as part of the java.util package to give developers a **type-safe**, **expressive** way of representing “a value *may* be present” instead of using null. It helps you:

* **Avoid NullPointerException** by forcing you to think about absence.
* **Document intent**: method returns “zero or one” result.
* **Chain operations** in a functional, declarative style.

**1. Core Idea**

* An Optional<T> either **contains** a non‑null T or is **empty**.
* You **never** call Optional.get() blindly—only after checking presence.
* You can transform, filter, or supply defaults in one fluent chain.

Optional<String> opt = Optional.of("hello");

Optional<String> emptyOpt = Optional.empty();

Optional<String> nullableOpt = Optional.ofNullable(maybeNullString);

**2. Why and When It Was Added**

1. **Endless NPEs**  
   Prior to Java 8, it was common to return null when something wasn’t found. If the caller forgot to check, you got a NullPointerException at runtime.
2. **Signal Absence Explicitly**  
   A return type of Optional<T> clearly tells the caller: “I might not have a T for you.”
3. **Functional Composition**  
   Optional provides map(), flatMap(), filter() and terminal methods (orElse(), orElseGet(), orElseThrow(), ifPresent()) that let you compose data‑handling pipelines without manual null‑checks.

**Introduced in Java 8** (2014) alongside lambdas and streams to round out the new functional‑style toolkit.

**3. Key Methods & Rules**

| **Method** | **What It Does** |
| --- | --- |
| of(value) | Wraps a **non‑null** value (throws if null) |
| ofNullable(value) | Wraps a value that **may** be null |
| empty() | Returns an **empty** Optional<T> |
| isPresent() | true if a value is present |
| ifPresent(Consumer) | Execute action **only** if value is present |
| get() | Returns the value or throws NoSuchElementException |
| orElse(defaultValue) | Return value if present, otherwise defaultValue |
| orElseGet(Supplier) | Lazy default: invoke supplier only if empty |
| orElseThrow(Supplier<Exc>) | Throw given exception if empty |
| map(Function) | Transform the contained value |
| flatMap(Function) | Transform to **another** Optional<U> |
| filter(Predicate) | Keep value only if it matches predicate |

**Rule of thumb:** Don’t overuse Optional for fields or parameters—its primary design is for **return types**.

**4. Real‑World Examples**

**4.1 Finding a User by ID**

public Optional<User> findUserById(String id) {

User u = userRepository.get(id); // may return null

return Optional.ofNullable(u);

}

// Caller:

findUserById("alice")

.ifPresent(u -> System.out.println("Found: " + u.getName()));

**4.2 Chaining Transformations**

Imagine you fetch an environment variable, parse it as an integer, and use a default:

int maxConnections = Optional.ofNullable(System.getenv("MAX\_CONN"))

.map(Integer::parseInt) // String → Integer

.filter(n -> n > 0) // only positive values

.orElse(10); // default to 10

**4.3 Querying Nested Objects**

Avoid “deep null checks”:

Optional<Address> addrOpt = userRepository.findUser("bob")

.flatMap(User::getPrimaryAddress); // getPrimaryAddress() returns Optional<Address>

String city = addrOpt

.map(Address::getCity)

.orElse("Unknown City");

**4.4 Safe Stream Operations**

List<String> raw = Arrays.*asList*("1", "", "42", null, "100");

List<Integer> ints = raw.stream()

.filter(Objects::nonNull)

.map(s -> Optional.of(s).filter(str -> !str.isBlank()))

.flatMap(Optional::stream) // Java 9+: unwrap non‑empty Optionals

.map(Integer::parseInt)

.collect(Collectors.toList());

// results: [1, 42, 100]

**5. When & Where to Use Optional**

* **Return values** from lookup or calculation methods that may not produce a result.
* **Compose** nullable data safely without littering calls with if (x != null).
* **Configure defaults** or throw custom exceptions cleanly.
* **Avoid** using Optional for **fields** in your domain objects or for **method parameters**—it’s not a serialization‑friendly or performance‑lightweight type in those places.

**In summary**, Optional is your tool of choice whenever you want to:

1. **Explicitly model** “value or no value” in APIs.
2. **Chain** data transforms and null‑checks declaratively.
3. **Prevent** NullPointerException by design rather than by accident.

**Callable<V>** and **Future<V>**

**Callable<V>** and **Future<V>** are complementary interfaces in the **java.util.concurrent** package (added in Java 5) that let you run tasks **asynchronously**, **return a result**, and **handle exceptions** cleanly.

**1. Why They Were Added & When**

* **Pre–Java 5**: You could start a new thread with new Thread(runnable).start(), but Runnable has no return value and cannot throw checked exceptions.
* **Java 5 (JSR‑166)**: Introduced the **java.util.concurrent** framework—among its features:
  + **Callable<V>**: like Runnable, but **returns** a value of type V and may **throw** checked exceptions.
  + **Future<V>**: a handle to the result of an asynchronous computation.

Together, they let you submit tasks to an **ExecutorService** and later retrieve results, cancel tasks, or check completion.

**2. Callable<V> Interface**

@FunctionalInterface

public interface Callable<V> {

V call() throws Exception;

}

* **call()**: invoked by the executor; returns a V or throws an exception.
* **Use When**:
  + Your background task must **produce** a result.
  + You want to **throw** checked exceptions from the task.

**Real‑World Example: Fetching Remote Data**

Callable<String> fetchWeather = () -> {

// simulate HTTP call

HttpResponse resp = httpClient.get("https://api.weather.com/today");

if (resp.getStatus() != 200) throw new IOException("Failed");

return resp.getBody();

};

**3. Future<V> Interface**

public interface Future<V> {

boolean cancel(boolean mayInterruptIfRunning);

boolean isCancelled();

boolean isDone();

V get() throws InterruptedException, ExecutionException;

V get(long timeout, TimeUnit unit)

throws InterruptedException, ExecutionException, TimeoutException;

}

* **get()** blocks until the task completes and returns its V (or wraps an exception in ExecutionException).
* **cancel()** tries to stop execution.
* **isDone()/isCancelled()** let you poll task status.

**4. Rules & Best Practices**

1. **Always shut down** your ExecutorService (shutdown()/shutdownNow()).
2. **Handle exceptions** from get()—it throws ExecutionException (wraps your call() exception).
3. **Timeouts**: prefer get(timeout, unit) to avoid indefinite blocking.
4. **Cancellation**: call future.cancel(true) to attempt an interrupt; check future.isCancelled() before get().
5. **Do not block UI threads**—offload long-running Callables to a background pool.
6. **Avoid heavyweight tasks** in a shared pool—choose pool size appropriate to CPU vs. I/O.

**5. When & Where to Use**

* **I/O‑Bound Tasks** (HTTP calls, file reads): submit as Callable<byte[]> and collect all Future<byte[]>.
* **CPU‑Bound Workloads** (data crunching, simulations): use a fixed‑size pool roughly equal to number of cores.
* **Batch Job Coordination**: submit multiple Callables, then wait on all Futures (e.g., invokeAll()).
* **Timeout‑Sensitive Operations**: Future.get(timeout,…) combined with cancel() for fail‑fast behavior.

**CompletableFuture** and its interface **CompletionStage** were introduced in Java 8 as part of java.util.concurrent package. They represent a powerful and flexible way to handle asynchronous, non-blocking programming using a more functional style.

**✅ Why Were They Introduced?**

**Before Java 8:**

* Future was limited: no way to chain computations, handle errors elegantly, or combine multiple Futures.
* You had to block with get() to retrieve the result.
* It wasn’t suitable for composing dependent async tasks.

**Java 8 introduced CompletableFuture and CompletionStage to:**

* Support non-blocking, composable, asynchronous workflows.
* Allow callback-style programming and functional pipelines with lambdas.

**🚀 What Is CompletableFuture?**

* CompletableFuture<T> implements Future<T> and CompletionStage<T>.
* It can run async tasks, chain computations, handle results or errors, and combine multiple stages.

**🔧 Real World Use Cases**

**1. Async API Calls (e.g., Weather Service)**

CompletableFuture<String> weatherFuture = CompletableFuture.supplyAsync(() -> {

return "Weather from API: Sunny 25°C"; // Simulated external API

});

weatherFuture.thenAccept(result -> {

System.out.println("Received weather: " + result);

});

**2. Chaining Tasks (transform result)**

CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> "apple")

.thenApply(fruit -> fruit.toUpperCase());

future.thenAccept(System.out::println); // Output: APPLE

**3. Combining Multiple Futures**

CompletableFuture<Integer> price = CompletableFuture.supplyAsync(() -> 100);

CompletableFuture<Integer> tax = CompletableFuture.supplyAsync(() -> 15);

CompletableFuture<Integer> total = price.thenCombine(tax, Integer::sum);

total.thenAccept(t -> System.out.println("Total: " + t)); // Output: 115

**4. Handling Exceptions**

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

if (true) throw new RuntimeException("Service failed");

return "Success";

}).exceptionally(ex -> {

return "Fallback response";

});

**future.thenAccept(System.out::println); // Output: Fallback response**

**🔁 Common Methods in CompletableFuture**

| Type | Method | Description |
| --- | --- | --- |
| Source | supplyAsync() | Starts async task with result |
|  | runAsync() | Starts async task with no result |
| Chaining | thenApply() | Transforms result |
|  | thenAccept() | Consumes result |
|  | thenRun() | Runs next action (no input/output) |
| Combining | thenCombine() | Combines two futures |
|  | allOf() / anyOf() | Combines multiple futures |
| Exception | exceptionally() | Handles exceptions |
|  | handle() | Handles both result & exception |
| Blocking (if needed) | get() / join() | Waits for result |

**🧠 What Is CompletionStage?**

* It's the interface implemented by CompletableFuture.
* Represents a stage in a pipeline of async tasks.
* You can write code that uses CompletionStage (more flexible, testable) and return CompletableFuture under the hood.

**📜 Rules & Best Practices**

1. ✅ Use supplyAsync() for tasks that return a value, runAsync() for void tasks.
2. ⚠ Avoid using get() or join() unless necessary (blocks thread).
3. 🧩 Chain dependent tasks using thenApply, thenCompose, etc.

🔄 Combine multiple CompletableFutures with allOf() or anyOf() if parallelism needed.

1. 🚫 Do not block UI or thread-limited environments with .get().
2. ✅ Handle exceptions with exceptionally, handle, or whenComplete.

**💡 Real-World Assignment Idea**

**Build a service that:**

1. Asynchronously fetches product price from multiple vendors.
2. Combines the prices to compute average.
3. If any vendor fails, handles it gracefully and still computes the average from remaining ones.

**📘 Summary**

| Feature | Future | CompletableFuture |
| --- | --- | --- |
| Blocking? | Yes (get()) | No (optional) |
| Combine Futures? | ❌ | ✅ |
| Exception Handling | ❌ | ✅ |
| Async Execution | ✅ (with Executor) | ✅ (easier) |
| Composition | ❌ | ✅ (then...) |

**🔍 What is CompletionStage?**

* CompletionStage<T> is an interface that represents a stage of a possibly asynchronous computation.
* It defines methods to chain computations, such as thenApply, thenCompose, thenAccept, etc.
* The actual implementation of CompletionStage is provided by CompletableFuture.

**✅ Why Use CompletionStage?**

* You can write code that is more abstract and testable.
* It enables flexible chaining of asynchronous operations without tightly coupling with CompletableFuture.
* It allows you to compose async steps like a pipeline.

**✅ Real-World Example: Order Processing Pipeline**

Suppose we have an order processing system:

1. Fetch order details asynchronously.
2. Fetch pricing info.
3. Calculate total.
4. Send a notification email.

import java.util.concurrent.\*;

import java.util.function.Function;

public class OrderProcessingExample {

public static void main(String[] args) {

ExecutorService executor = Executors.newFixedThreadPool(3);

CompletionStage<String> orderStage = CompletableFuture.supplyAsync(() -> {

System.out.println("Fetching order details...");

return "Order123";

}, executor);

CompletionStage<Double> priceStage = orderStage.thenApplyAsync(orderId -> {

System.out.println("Fetching pricing for " + orderId);

return 150.75;

}, executor);

CompletionStage<Double> taxStage = priceStage.thenApplyAsync(price -> {

System.out.println("Calculating tax...");

return price \* 1.18; // adding 18% tax

}, executor);

CompletionStage<Void> notificationStage = taxStage.thenAcceptAsync(total -> {

System.out.println("Sending email for total amount: " + total);

}, executor);

// Wait for final stage to complete (only in demo, don't block in production)

((CompletableFuture<Void>) notificationStage).join();

executor.shutdown();

}

}

**🔧 Breakdown of Key Methods Used**

| Method | Purpose | Example |
| --- | --- | --- |
| supplyAsync() | Start a task that returns a value | CompletableFuture.supplyAsync(() -> "Order") |
| thenApplyAsync() | Transform the result | .thenApplyAsync(order -> getPrice(order)) |
| thenAcceptAsync() | Consume result (void return) | .thenAcceptAsync(total -> sendEmail(total)) |
| join() | Block and get result (only for demo) | future.join() |

**🧠 When to Use CompletionStage**

* When you want to write APIs that return a stage, and let the caller decide how to complete the chain.
* When you're writing asynchronous pipelines in services.
* When you want to abstract out the underlying implementation (e.g., use mocks in tests).

**🎓 Example: API returning CompletionStage**

**public interface OrderService {**

**CompletionStage<Double> calculateTotal(String orderId);**

**}**

Caller code can use .thenApply() or .thenAccept() on it — full decoupling.

**🚀 Assignment: Asynchronous Travel Booking System**

**🎯 Objective:**

Simulate an asynchronous travel booking system that queries different services (like Flights, Hotels, and Car Rentals) **in parallel**, processes their responses using **Lambdas** and **CompletableFuture**, and combines results using **CompletionStage** methods.

**🛠️ Requirements:**

1. **Service Classes** (Simulate external APIs):
   * FlightService: Returns available flight based on source and destination.
   * HotelService: Returns hotel options at the destination.
   * CarRentalService: Returns car rental options at the destination.
   * Each method should return a CompletableFuture<String> after a simulated delay using CompletableFuture.supplyAsync().
2. **TravelBookingService**:
   * Asynchronously request all services using CompletableFuture.
   * Use thenCombine() or thenCompose() to combine results.
   * Use exceptionally() or handle() to manage service failures.
   * Use thenAccept() to display the consolidated result.
3. **Main Class**:
   * Accept user input (source and destination).
   * Trigger booking process and print:

Searching for Flights, Hotels, and Car Rentals to Paris...

✓ Flight booked: Air France 445

✓ Hotel booked: Hilton Paris

✓ Car rented: Hertz Sedan

Booking completed successfully!

**✅ Concepts to Use:**

* ✅ Lambda expressions (for functional style with supplyAsync, thenApply, etc.)
* ✅ CompletableFuture for asynchronous programming
* ✅ CompletionStage methods like thenApply, thenCombine, thenAccept, handle

**🔂 Sample Output:**

Booking trip from New York to Paris...

[FlightService] Searching for flights...

[HotelService] Searching for hotels...

[CarRentalService] Searching for car rentals...

[FlightService] Found: Delta Airlines DL123

[HotelService] Found: Marriott Champs Elysees

[CarRentalService] Found: Renault Scenic

Trip Summary:

Flight: Delta Airlines DL123

Hotel: Marriott Champs Elysees

Car Rental: Renault Scenic

Booking completed successfully!