Mastering Java 8: A Comprehensive Guide

1 Introduction

This guide offers a hands-on exploration of Java 8's core features, designed for learners with basic Java knowledge (OOP, collections). It covers Lambda Expressions, Stream API, Optional, Default/Static Methods, Date/Time API, CompletableFuture, and a practical project, with examples and best practices to deepen your understanding of modern Java development.

2 Lambda Expressions

2.1 What Are Lambda Expressions?

Lambda expressions bring functional programming to Java 8, allowing concise code for single-method interfaces (functional interfaces). They simplify tasks like event handling and sorting by replacing verbose anonymous classes.

2.2 Key Concepts

- **Syntax**: `(parameters) -> expression` or `(parameters) -> { statements; }`.
- **Functional Interfaces**: Interfaces with one abstract method (e.g., `Runnable`, `Comparator`, `Predicate`).
- **Built-in Functional Interfaces**: In `java.util.function` (e.g., `Predicate<T>`, `Function<T,R>`, `Consumer<T>`, `Supplier<T>`).
- **Benefits**: Cleaner code, improved readability, functional programming support.

2.3 Examples

1. **Sorting a List**:

```java
import java.util.*;
List<String> names = Arrays.asList("Bob", "Alice", "Charlie");
Collections.sort(names, (a, b) -> a.length() - b.length());
System.out.println(names); // [Bob, Alice, Charlie]
```

2. **Thread with Lambda**:

```java
new Thread(() -> System.out.println("Running in thread")).start();
```

3. **Using Predicate**:

```java
import java.util.function.Predicate;
Predicate<Integer> isEven = n -> n % 2 == 0;
System.out.println(isEven.test(4)); // true
```

2.4 Best Practices

- Use lambdas for simple logic; avoid complex multi-line lambdas.
- Prefer method references (e.g., `String::toUpperCase`).
- Use `@FunctionalInterface` for custom interfaces.

2.5 Practice

1. Sort a `List<Integer>` in descending order using a lambda.
2. Create a `Predicate<String>` to check if a string starts with ”J”.
3. Convert an anonymous `ActionListener` to a lambda (mocked).
4. Write a custom functional interface for addition.

2.6 Resources

- https://www.baeldung.com/java-8-lambda-expressions-tips
- https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html

3 Stream API Basics

3.1 What Is the Stream API?

The Stream API (`java.util.stream`) enables functional-style processing of collections with lazy evaluation, internal iteration, and pipeline operations.

3.2 Key Concepts

- **Stream Creation**: `list.stream()`, `Stream.of()`, `Arrays.stream()`.
- **Intermediate Operations** (lazy): `filter`, `map`, `sorted`, `distinct`.
- **Terminal Operations** (eager): `collect`, `forEach`, `count`, `reduce`.
- **Lazy Evaluation**: Operations execute only when a terminal operation is called.
- **Immutability**: Streams do not modify the source.

3.3 Examples

1. **Filter and Collect**:

```java
import java.util.*;
import java.util.stream.*;
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
List<Integer> even = numbers.stream()
                    .filter(n -> n % 2 == 0)
                    .collect(Collectors.toList());
System.out.println(even); // [2, 4]
```

2. **Map to Transform**:

```java
List<String> names = Arrays.asList("alice", "bob");
List<String> upper = names.stream()
                   .map(String::toUpperCase)
                   .collect(Collectors.toList());
System.out.println(upper); // [ALICE, BOB]
```

3. **Sort and Count**:

```java
List<String> words = Arrays.asList("apple", "banana", "apricot");
long count = words.stream()
             .filter(s -> s.startsWith("a"))
             .sorted()
             .count();
System.out.println(count); // 2
```

3.4 Best Practices

- Use streams for complex data processing.
- Chain operations logically.
- Avoid side effects in stream operations.

3.5 Practice

1. Filter a `List<String>` for strings longer than 4 characters.
2. Transform a `List<Integer>` to squares using `map`.
3. Sort a `List<Person>` by age in descending order.
4. Count elements in a `List<Double>` greater than 10.0.

3.6 Resources

- https://www.baeldung.com/java-8-streams
- http://tutorials.jenkov.com/java-functional-programming/streams.html

4 Advanced Stream API

4.1 What Are Advanced Stream Operations?

These handle complex data processing like grouping, flattening, and parallel processing.

4.2 Key Concepts

- **flatMap**: Flattens nested collections.
- **groupingBy**: Groups data by a criterion.
- **parallelStream**: Processes data in parallel.
- **reduce**: Combines elements into a single result.

4.3 Examples

1. **FlatMap**:

```java
import java.util.*;
import java.util.stream.*;
List<List<String>> nested = Arrays.asList(Arrays.asList("a", "b"),
    Arrays.asList("c"));
List<String> flat = nested.stream()
                    .flatMap(List::stream)
                    .collect(Collectors.toList());
System.out.println(flat); // [a, b, c]
```

2. **Grouping**:

```java
import java.util.*;
import java.util.stream.*;
class Person {
   String name; int age;
   Person(String name, int age) { this.name = name; this.age = age; }
   public String getName() { return name; }
   public int getAge() { return age; }
}
List<Person> people = Arrays.asList(new Person("Alice", 25), new
    Person("Bob", 25), new Person("Charlie", 30));
Map<Integer, List<String>> byAge = people.stream()
                               .collect(Collectors.groupingBy(
                                  Person::getAge,
                                  Collectors.mapping(Person::getName,
                                     Collectors.toList())
                               ));
System.out.println(byAge); // {25=[Alice, Bob], 30=[Charlie]}
```

3. **Reduce**:

```java
List<Integer> numbers = Arrays.asList(1, 2, 3);
int sum = numbers.stream()
             .reduce(0, (a, b) -> a + b);
System.out.println(sum); // 6
```

4.4 Best Practices

- Use `flatMap` for nested structures.
- Use `parallelStream` for large datasets; profile with VisualVM.
- Ensure `reduce` operations are associative.

4.5 Practice

1. Flatten a `List<List<Integer»` using `flatMap`.
2. Group a `List<String>` by length and collect counts.
3. Sum a `List<Double>` using `reduce`.
4. Compare `stream` vs. `parallelStream` performance.

4.6 Resources

- https://www.baeldung.com/java-8-streams
- https://docs.oracle.com/javase/tutorial/collections/streams/parallelism.html

5 Optional Class

5.1 What Is Optional?

`Optional (java.util.Optional)` is a container for nullable values to prevent `NullPointerException`.

5.2 Key Concepts

- **Creation**: `Optional.of(value)`, `Optional.ofNullable(value)`, `Optional.empty()`.
- **Methods**: `orElse`, `orElseGet`, `ifPresent`, `map`, `filter`, `orElseThrow`.
- **Use Case**: Replace null checks.

5.3 Examples

1. **Basic Optional**:

```
import java.util.Optional;
Optional<String> name = Optional.of("Alice");
System.out.println(name.orElse("Unknown")); // Alice
```

2. **Handle Null**:

```
String value = null;
Optional<String> opt = Optional.ofNullable(value);
System.out.println(opt.orElseGet(() -> "Default")); // Default
```

3. **Chaining**:

```
Optional<String> result = Optional.of("alice")
                          .map(String::toUpperCase)
                          .filter(s -> s.length() > 3);
System.out.println(result.get()); // ALICE
```

5.4 Best Practices

- Use `Optional` for return types, not fields.
- Avoid overusing `Optional.get()`; prefer `orElse`.
- Chain with `map` or `flatMap`.

5.5 Practice

1. Refactor a method with null checks to use `Optional`.
2. Use `map` to get the length of an `Optional<String>`.
3. Filter an `Optional<Integer>` for values > 100.
4. Handle a nested `Optional`.

5.6 Resources

- https://www.baeldung.com/java-optional
- https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html

6 Default and Static Methods in Interfaces

6.1 What Are Default/Static Methods?

- **Default Methods**: Allow interfaces to have implementations.
- **Static Methods**: Utility methods callable without instantiation.

6.2 Examples

1. **Default Method**:

```
interface Vehicle {
   default void start() {
      System.out.println("Vehicle started");
   }
}
class Car implements Vehicle {}
Car car = new Car();
car.start(); // Vehicle started
```

2. **Static Method**:

```
interface MathUtil {
   static int multiply(int a, int b) {
      return a * b;
   }
}
System.out.println(MathUtil.multiply(5, 3)); // 15
```

6.3 Best Practices

- Use default methods for optional behavior.
- Keep static methods simple and stateless.
- Resolve conflicts explicitly.

6.4 Practice

1. Create an interface with a default method and override it.
2. Add a static method to an interface.
3. Resolve conflicting default methods.

6.5 Resources

- https://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html
- https://www.baeldung.com/java-static-default-methods

7 Date and Time API

7.1 What Is the Date/Time API?

The `java.time` package (JSR-310) provides immutable classes for date and time handling.

7.2 Key Concepts

- **Core Classes**: `LocalDate`, `LocalTime`, `LocalDateTime`, `ZonedDateTime`.
- **Formatting**: `DateTimeFormatter` for custom formats.
- **Manipulation**: `plusDays`, `minusMonths`.
- **Time Zones**: `ZonedDateTime` for global applications.

7.3 Examples

1. **LocalDate**:

```java
import java.time.LocalDate;
LocalDate today = LocalDate.now();
LocalDate nextMonth = today.plusMonths(1);
System.out.println(nextMonth); // e.g., 2025-07-17
```

2. **LocalDateTime and Formatting**:

```java
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
LocalDateTime now = LocalDateTime.now();
DateTimeFormatter fmt = DateTimeFormatter.ofPattern("dd-MM-yyyy
    HH:mm:ss");
System.out.println(now.format(fmt)); // e.g., 17-06-2025 00:36:00
```

3. **Time Zones**:

```java
import java.time.ZonedDateTime;
import java.time.ZoneId;
ZonedDateTime zdt = ZonedDateTime.now(ZoneId.of("America/New_York"));
System.out.println(zdt); // e.g.,
    2025-06-16T11:36-04:00[America/New_York]
```

7.4 Best Practices

- Use `LocalDateTime` for most applications.
- Specify `DateTimeFormatter` for consistent formatting.
- Avoid legacy `Date`/`Calendar`.

7.5 Practice

1. Calculate days between two `LocalDate` objects.
2. Format a `LocalDateTime` to "yyyy/MM/dd HH:mm".
3. Convert a `LocalDateTime` to `ZonedDateTime`.
4. Compute hours between two `LocalDateTime` instances.

7.6 Resources

- https://www.baeldung.com/java-8-date-time-intro
- https://docs.oracle.com/javase/8/docs/api/java/time/package-summary.html

8 CompletableFuture

8.1 What Is CompletableFuture?

`CompletableFuture` (`java.util.concurrent`) enables asynchronous programming for non-blocking tasks.

8.2 Key Concepts

- **Async Execution**: `supplyAsync`, `runAsync`.
- **Chaining**: `thenApply`, `thenCompose`, `thenCombine`.
- **Error Handling**: `exceptionally`, `handle`.
- **Combining Futures**: `allOf`, `anyOf`.

8.3 Examples

1. **Async Task**:

```java
import java.util.concurrent.CompletableFuture;
CompletableFuture.supplyAsync(() -> "Hello")
             .thenApply(s -> s + " World")
             .thenAccept(System.out::println); // Hello World
```

2. **Error Handling**:

```java
CompletableFuture.supplyAsync(() -> {
   throw new RuntimeException("Error");
}).exceptionally(e -> "Recovered from " + e.getMessage())
  .thenAccept(System.out::println); // Recovered from
     java.lang.RuntimeException: Error
```

8.4 Best Practices

- Use `supplyAsync` for computations; `runAsync` for side-effects.
- Handle exceptions explicitly.
- Use `thenCompose` for dependent futures.

8.5 Practice

1. Fetch two mock API data with `CompletableFuture` and combine.
2. Handle an exception in a `CompletableFuture` chain.
3. Use `allOf` to wait for multiple futures.

8.6 Resources

- https://www.baeldung.com/java-completablefuture
- https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html

9 Mini Project: Employee Management System

9.1 Overview

This project demonstrates Java 8 features by processing employee data. Objectives:

- Filter employees by age and salary.
- Group by department.
- Format hire dates.
- Process asynchronously with `CompletableFuture`.

9.2 Code

```java
import java.time.LocalDate;
import java.time.format.DateTimeFormatter;
import java.util.*;
import java.util.concurrent.CompletableFuture;
import java.util.stream.*;

class Employee {
   String name; int age; double salary; String department; LocalDate hireDate;
   Employee(String name, int age, double salary, String department, LocalDate
       hireDate) {
      this.name = name; this.age = age; this.salary = salary;
      this.department = department; this.hireDate = hireDate;
   }
   public String getName() { return name; }
   public int getAge() { return age; }
   public double getSalary() { return salary; }
   public String getDepartment() { return department; }
   public LocalDate getHireDate() { return hireDate; }
}

public class EmployeeSystem {
   public static void main(String[] args) {
      List<Employee> employees = Arrays.asList(
         new Employee("Alice", 30, 65000, "HR", LocalDate.of(2020, 1, 15)),
         new Employee("Bob", 40, 70000, "IT", LocalDate.of(2019, 5, 20)),
         new Employee("Charlie", 28, 55000, "HR", LocalDate.of(2021, 3, 10)),
         new Employee("David", 32, 80000, "IT", LocalDate.of(2018, 7, 25))
      );
      Map<String, List<String>> result = employees.stream()
         .filter(e -> e.age >= 25 && e.age <= 35)
         .filter(e -> e.salary > 60000)
         .collect(Collectors.groupingBy(
            Employee::getDepartment,
            Collectors.mapping(e -> e.name + " (Hired: " +
               e.hireDate.format(DateTimeFormatter.ofPattern("dd-MM-yyyy")) +
                  ")",
               Collectors.toList())
         ));
      System.out.println(result);
      CompletableFuture.supplyAsync(() -> employees.stream()
         .mapToDouble(Employee::getSalary)
         .average()
         .orElse(0))
         .thenAccept(avg -> System.out.println("Average Salary: " + avg));
   }
}
```

9.3 Practice

1. Sort employees by hire date.
2. Add `Optional` for missing employee data.

10 Tips for Effective Learning

- **IDE**: Use IntelliJ IDEA with Java 8 SDK.

- **Practice**: Explore LeetCode or HackerRank for exercises.
- **Community**: Engage on Stack Overflow for questions.