Java Advanced Topics - Complete Study Notes

80. What is Annotation

Annotations provide metadata about code that can be processed at compile-time or runtime.

Key Points:

- Annotations start with `@` symbol
- They don't directly affect program execution
- Can be applied to classes, methods, fields, parameters, etc.

Common Built-in Annotations:

- `@Override` - indicates method overrides parent class method
- `@Deprecated` - marks code as outdated
- `@SuppressWarnings` - suppresses compiler warnings
- `@FunctionalInterface` - marks interface as functional

Custom Annotation Example:

```java
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface MyAnnotation {
    String value() default "";
    int count() default 1;
}
```

81. Types of Interface

Java supports different types of interfaces for various use cases.

1. Regular Interface:

```java
interface Vehicle {
    void start();
    void stop();
}
```

2. Marker Interface (Empty Interface):

```java
interface Serializable {
    // No methods - just marks the class
}
```

3. Functional Interface (Single Abstract Method):

```java
@FunctionalInterface
interface Calculator {
    int calculate(int a, int b);
}
```

4. Default Methods (Java 8+):

```java
interface MyInterface {
    default void defaultMethod() {
        System.out.println("Default implementation");
    }
}
```

82. Functional Interface New

Functional interfaces are interfaces with exactly one abstract method, introduced in Java 8.

Key Features:

- Can have multiple default and static methods
- Used with lambda expressions
- `@FunctionalInterface` annotation is optional but recommended

Common Functional Interfaces:

- `Predicate<T>` - takes T, returns boolean
- `Function<T,R>` - takes T, returns R
- `Consumer<T>` - takes T, returns void
- `Supplier<T>` - takes nothing, returns T

Example:

```java
```

```java
@FunctionalInterface
interface MathOperation {
    int operate(int a, int b);
}

// Usage with lambda
MathOperation addition = (a, b) -> a + b;
```

83. Lambda Expression

Lambda expressions provide a concise way to represent anonymous functions.

Syntax:

```
(parameters) -> expression
(parameters) -> { statements; }
```

Examples:

java

```java
// No parameters
() -> System.out.println("Hello")

// One parameter (parentheses optional)
x -> x * x

// Multiple parameters
(x, y) -> x + y

// Multiple statements
(x, y) -> {
    int result = x + y;
    return result;
}
```

With Collections:

java

```java
List<String> names = Arrays.asList("John", "Jane", "Bob");
names.forEach(name -> System.out.println(name));
```

84. Lambda Expression with Return

Lambda expressions can return values explicitly or implicitly.

Implicit Return (Expression Body):

```java
Function<Integer, Integer> square = x -> x * x;
```

Explicit Return (Statement Body):

```java
Function<Integer, Integer> square = x -> {
    return x * x;
};
```

Complex Example:

```java
Comparator<Person> byAge = (p1, p2) -> {
    if (p1.getAge() > p2.getAge()) return 1;
    else if (p1.getAge() < p2.getAge()) return -1;
    else return 0;
};
```

85. What is Exception

Exceptions are runtime errors that disrupt normal program flow.

Exception Hierarchy:

```
Object
└── Throwable
    ├── Error (System errors)
    └── Exception
        ├── RuntimeException (Unchecked)
        └── Other Exceptions (Checked)
```

Types:

1. **Checked Exceptions** - Must be handled (IOException, SQLException)
2. **Unchecked Exceptions** - Runtime exceptions (NullPointerException, ArrayIndexOutOfBoundsException)

3. **Errors** - System-level problems (OutOfMemoryError)

86. Exception Handling using try-catch

Try-catch blocks handle exceptions gracefully.

Basic Syntax:

```java
try {
    // Code that may throw exception
    int result = 10 / 0;
} catch (ArithmeticException e) {
    System.out.println("Cannot divide by zero: " + e.getMessage());
} finally {
    // Always executes (optional)
    System.out.println("Cleanup code");
}
```

Best Practices:

- Catch specific exceptions first
- Use finally for cleanup
- Don't catch and ignore exceptions

87. Try with Multiple Catch

Handle different types of exceptions with multiple catch blocks.

Example:

```java
try {
    String str = null;
    int length = str.length(); // NullPointerException
    int[] arr = new int[5];
    arr[10] = 20; // ArrayIndexOutOfBoundsException
} catch (NullPointerException e) {
    System.out.println("Null pointer: " + e.getMessage());
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Array index out of bounds: " + e.getMessage());
} catch (Exception e) {
    System.out.println("General exception: " + e.getMessage());
}
```

Multi-catch (Java 7+):

```java
try {
    // risky code
} catch (IOException | SQLException e) {
    System.out.println("IO or SQL exception: " + e.getMessage());
}
```

88. Exception Hierarchy

Understanding the exception hierarchy helps in proper exception handling.

Key Classes:

- `Throwable` - Root class for all errors and exceptions
- `Error` - Serious system problems (usually unrecoverable)
- `Exception` - Application-level problems
- `RuntimeException` - Unchecked exceptions

Common Exceptions:

- `NullPointerException` - Accessing null reference
- `ClassCastException` - Invalid type casting
- `IllegalArgumentException` - Invalid method arguments
- `IOException` - Input/output operations failure

89. Exception Throw Keyword

The `throw` keyword manually throws exceptions.

Syntax:

```java
throw new ExceptionType("Error message");
```

Examples:

```java
```

```java
public void validateAge(int age) {
    if (age < 18) {
        throw new IllegalArgumentException("Age must be 18 or above");
    }
}

public void divide(int a, int b) {
    if (b == 0) {
        throw new ArithmeticException("Division by zero not allowed");
    }
    System.out.println(a / b);
}
```

90. Custom Exception

Create your own exception classes for specific application needs.

Example:

java

```java
public class InsufficientFundsException extends Exception {
    private double amount;

    public InsufficientFundsException(double amount) {
        super("Insufficient funds: " + amount);
        this.amount = amount;
    }

    public double getAmount() {
        return amount;
    }
}

// Usage
public void withdraw(double amount) throws InsufficientFundsException {
    if (amount > balance) {
        throw new InsufficientFundsException(amount);
    }
    balance -= amount;
}
```

91. Ducking Exception using Throws

The `throws` keyword declares that a method may throw exceptions.

Syntax:

```java
public void methodName() throws ExceptionType1, ExceptionType2 {
    // Method body
}
```

Example:

```java
public void readFile(String filename) throws IOException {
    FileReader file = new FileReader(filename);
    // File reading code
}

public void processFile() {
    try {
        readFile("data.txt");
    } catch (IOException e) {
        System.out.println("File error: " + e.getMessage());
    }
}
```

92. User Input using BufferedReader and Scanner

Two main ways to read user input in Java.

BufferedReader:

```java
import java.io.*;

BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
try {
    System.out.print("Enter your name: ");
    String name = reader.readLine();
    System.out.println("Hello " + name);
} catch (IOException e) {
    e.printStackTrace();
}
```

Scanner (Preferred):

```java
```

```
import java.util.Scanner;

Scanner scanner = new Scanner(System.in);
System.out.print("Enter your age: ");
int age = scanner.nextInt();
System.out.print("Enter your name: ");
String name = scanner.next();
scanner.close();
```

93. Try with Resources

Automatically manages resources that implement AutoCloseable.

Syntax:

```
java

try (ResourceType resource = new ResourceType()) {
    // Use resource
} catch (Exception e) {
    // Handle exception
}
// Resource is automatically closed
```

Example:

```
java

try (FileReader file = new FileReader("data.txt");
     BufferedReader reader = new BufferedReader(file)) {

    String line = reader.readLine();
    System.out.println(line);

} catch (IOException e) {
    System.out.println("File error: " + e.getMessage());
}
// Files are automatically closed
```

94. Threads

Threads enable concurrent execution of multiple parts of a program.

Creating Threads:

Method 1 - Extending Thread:

```java
class MyThread extends Thread {
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println("Thread: " + i);
            try { Thread.sleep(1000); } catch (InterruptedException e) {}
        }
    }
}

// Usage
MyThread t1 = new MyThread();
t1.start();
```

Method 2 - Implementing Runnable:

```java
class MyTask implements Runnable {
    public void run() {
        // Task code
    }
}

Thread t2 = new Thread(new MyTask());
t2.start();
```

95. Multiple Threads

Managing multiple threads for concurrent execution.

Example:

```java

```

```
public class MultiThreadExample {
    public static void main(String[] args) {
        Runnable task1 = () -> {
            for (int i = 0; i < 5; i++) {
                System.out.println("Task 1: " + i);
                try { Thread.sleep(500); } catch (InterruptedException e) {}
            }
        };

        Runnable task2 = () -> {
            for (int i = 0; i < 5; i++) {
                System.out.println("Task 2: " + i);
                try { Thread.sleep(700); } catch (InterruptedException e) {}
            }
        };

        Thread t1 = new Thread(task1);
        Thread t2 = new Thread(task2);

        t1.start();
        t2.start();
    }
}
```

96. Thread Priority and Sleep

Control thread execution with priorities and sleep.

Thread Priority:

java

```
Thread t1 = new Thread(() -> {
    for (int i = 0; i < 5; i++) {
        System.out.println("High priority: " + i);
    }
});

t1.setPriority(Thread.MAX_PRIORITY); // 10
t1.start();
```

Thread Sleep:

java

```
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println("Count: " + i);
            try {
                Thread.sleep(1000); // Sleep for 1 second
            } catch (InterruptedException e) {
                System.out.println("Thread interrupted");
            }
        }
    }
```

97. Runnable vs Thread

Runnable Interface (Preferred):

- More flexible (can extend another class)
- Better for task representation
- Can be used with ExecutorService

Thread Class:

- Direct thread control
- Cannot extend another class
- More heavyweight

Best Practice - Use Runnable:

```
java

// Good approach
class Task implements Runnable {
    public void run() {
        // Task logic
    }
}

Thread thread = new Thread(new Task());
thread.start();
```

98. Race Condition

Race conditions occur when multiple threads access shared resources simultaneously.

Problem Example:

```java
class Counter {
    private int count = 0;

    public void increment() {
        count++; // Not thread-safe
    }

    public int getCount() {
        return count;
    }
}
```

Solution - Synchronization:

```java
class SafeCounter {
    private int count = 0;

    public synchronized void increment() {
        count++; // Thread-safe
    }

    public synchronized int getCount() {
        return count;
    }
}
```

99. Thread States

Threads go through various states during their lifecycle.

Thread States:

1. **NEW** - Thread created but not started
2. **RUNNABLE** - Thread is executing or ready to execute
3. **BLOCKED** - Thread blocked waiting for monitor lock
4. **WAITING** - Thread waiting indefinitely
5. **TIMED_WAITING** - Thread waiting for specified time
6. **TERMINATED** - Thread has completed execution

State Transitions:

```java
Thread t = new Thread(() -> {
    try {
        Thread.sleep(2000);
    } catch (InterruptedException e) {}
});

System.out.println(t.getState()); // NEW
t.start();
System.out.println(t.getState()); // RUNNABLE
```

100. Collection API

Java Collections Framework provides data structures and algorithms.

Core Interfaces:

- `Collection` - Root interface
- `List` - Ordered collection (ArrayList, LinkedList)
- `Set` - No duplicates (HashSet, TreeSet)
- `Map` - Key-value pairs (HashMap, TreeMap)

Common Implementations:

```java
// List
List<String> list = new ArrayList<>();
list.add("Java");
list.add("Python");

// Set
Set<Integer> set = new HashSet<>();
set.add(1);
set.add(2);

// Map
Map<String, Integer> map = new HashMap<>();
map.put("John", 25);
map.put("Jane", 30);
```

105. Need of Stream API

Stream API provides functional programming approach for processing collections.

Benefits:

- Declarative programming style
- Parallel processing support
- Lazy evaluation
- Cleaner, more readable code

Before Streams:

```java
List<String> names = Arrays.asList("John", "Jane", "Bob", "Alice");
List<String> result = new ArrayList<>();
for (String name : names) {
    if (name.startsWith("J")) {
        result.add(name.toUpperCase());
    }
}
```

With Streams:

```java
List<String> result = names.stream()
    .filter(name -> name.startsWith("J"))
    .map(String::toUpperCase)
    .collect(Collectors.toList());
```

106. forEach Method

Enhanced iteration using forEach with lambdas.

Examples:

```java
```

```java
List<String> names = Arrays.asList("John", "Jane", "Bob");

// Traditional for-each
for (String name : names) {
    System.out.println(name);
}

// Stream forEach
names.stream().forEach(System.out::println);

// Collection forEach (Java 8+)
names.forEach(System.out::println);

// With lambda
names.forEach(name -> System.out.println("Hello " + name));
```

107. Stream API

Complete guide to Stream operations.

Creating Streams:

```java
// From collection
List<String> list = Arrays.asList("a", "b", "c");
Stream<String> stream1 = list.stream();

// From array
String[] array = {"x", "y", "z"};
Stream<String> stream2 = Arrays.stream(array);

// Static methods
Stream<String> stream3 = Stream.of("1", "2", "3");
```

Intermediate Operations:

```java
list.stream()
    .filter(s -> s.length() > 2)    // Filter elements
    .map(String::toUpperCase)       // Transform elements
    .distinct()                     // Remove duplicates
    .sorted()                       // Sort elements
    .limit(10)                      // Limit results
    .skip(2);                       // Skip elements
```

Terminal Operations:

```java
// Collect results
List<String> result = stream.collect(Collectors.toList());

// Find elements
Optional<String> first = stream.findFirst();
Optional<String> any = stream.findAny();

// Check conditions
boolean allMatch = stream.allMatch(s -> s.length() > 0);
boolean anyMatch = stream.anyMatch(s -> s.startsWith("A"));

// Reduce
Optional<String> reduced = stream.reduce((s1, s2) -> s1 + s2);
```

108. Map Filter Reduce Sorted

Core stream operations for data processing.

Complete Example:

```java
```

```java
List<Person> people = Arrays.asList(
    new Person("John", 25, 50000),
    new Person("Jane", 30, 60000),
    new Person("Bob", 35, 45000),
    new Person("Alice", 28, 55000)
);

// Filter, Map, Sorted, Reduce
double avgSalaryOfYoungEmployees = people.stream()
    .filter(p -> p.getAge() < 30)          // Filter: age < 30
    .map(Person::getSalary)                // Map: extract salary
    .sorted()                              // Sort salaries
    .mapToDouble(Integer::doubleValue)     // Convert to double
    .average()                             // Reduce: calculate average
    .orElse(0.0);

// Collect to new list
List<String> youngNames = people.stream()
    .filter(p -> p.getAge() < 30)
    .map(Person::getName)
    .sorted()
    .collect(Collectors.toList());
```

109. ParallelStream in Java

Parallel processing for improved performance on large datasets.

Creating Parallel Streams:

java

```java
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

// Sequential stream
numbers.stream().forEach(System.out::println);

// Parallel stream
numbers.parallelStream().forEach(System.out::println);

// Convert to parallel
numbers.stream().parallel().forEach(System.out::println);
```

Performance Example:

java

```java
List<Integer> largeList = IntStream.rangeClosed(1, 1000000)
    .boxed()
    .collect(Collectors.toList());

// Sequential processing
long start = System.currentTimeMillis();
long sequentialSum = largeList.stream()
    .mapToLong(Integer::longValue)
    .sum();
long sequentialTime = System.currentTimeMillis() - start;

// Parallel processing
start = System.currentTimeMillis();
long parallelSum = largeList.parallelStream()
    .mapToLong(Integer::longValue)
    .sum();
long parallelTime = System.currentTimeMillis() - start;
```

110. Optional Class in Java

Optional helps avoid NullPointerException and makes null handling explicit.

Creating Optional:

java

```java
// Empty optional
Optional<String> empty = Optional.empty();

// Optional with value
Optional<String> optional = Optional.of("Hello");

// Optional with possible null
Optional<String> nullable = Optional.ofNullable(getString());
```

Using Optional:

java

```java
Optional<String> optional = Optional.of("Hello World");

// Check if present
if (optional.isPresent()) {
    System.out.println(optional.get());
}

// Better approach with ifPresent
optional.ifPresent(System.out::println);

// Provide default value
String result = optional.orElse("Default");

// Provide default with supplier
String result2 = optional.orElseGet(() -> "Generated Default");

// Transform value
Optional<Integer> length = optional.map(String::length);

// Filter value
Optional<String> filtered = optional.filter(s -> s.length() > 5);
```

111. Method Reference

Method references provide shorthand for lambda expressions.

Types of Method References:

1. Static Method Reference:

```java
// Lambda: x -> Math.sqrt(x)
Function<Double, Double> sqrt = Math::sqrt;
```

2. Instance Method Reference:

```java
String str = "Hello";
// Lambda: () -> str.length()
Supplier<Integer> lengthSupplier = str::length;
```

3. Instance Method of Arbitrary Object:

```java
```

```java
List<String> names = Arrays.asList("John", "Jane", "Bob");
// Lambda: name -> name.toUpperCase()
names.stream().map(String::toUpperCase);
```

4. Constructor Reference:

```java
// Lambda: () -> new ArrayList<>()
Supplier<List<String>> listSupplier = ArrayList::new;

// Lambda: capacity -> new ArrayList<>(capacity)
Function<Integer, List<String>> listFunction = ArrayList::new;
```

112. Constructor Reference

Constructor references create new instances using method reference syntax.

Examples:

```java
// No-arg constructor
Supplier<Person> personSupplier = Person::new;
Person p1 = personSupplier.get();

// Single argument constructor
Function<String, Person> personFunction = Person::new;
Person p2 = personFunction.apply("John");

// Multiple arguments
BiFunction<String, Integer, Person> personBiFunction = Person::new;
Person p3 = personBiFunction.apply("Jane", 25);

// With streams
List<String> names = Arrays.asList("John", "Jane", "Bob");
List<Person> people = names.stream()
    .map(Person::new)  // Constructor reference
    .collect(Collectors.toList());

// Array constructor
Function<Integer, String[]> arrayConstructor = String[]::new;
String[] array = arrayConstructor.apply(10);
```

Key Takeaways

1. **Annotations** provide metadata without affecting execution
2. **Functional Interfaces** enable lambda expressions and functional programming
3. **Exception Handling** is crucial for robust applications
4. **Threads** enable concurrent programming but require careful synchronization
5. **Collections** provide efficient data structures
6. **Streams** offer powerful data processing capabilities
7. **Optional** helps avoid null pointer exceptions
8. **Method References** provide cleaner syntax for lambda expressions

Best Practices

- Use appropriate collection types for your use case
- Handle exceptions properly - don't ignore them
- Prefer composition over inheritance
- Use streams for data processing but be mindful of performance
- Always close resources using try-with-resources
- Use Optional to handle potential null values
- Write thread-safe code when dealing with concurrency
- Leverage functional programming features for cleaner code