Modern Java Learning Path (Java 8 to Java 21)

Introduction

Welcome back to the world of Java development! Since you last worked with Java 7 around 2015, the language has undergone significant evolution. This comprehensive guide will walk you through all the important changes from Java 8 through Java 21, with special emphasis on the Long-Term Support (LTS) versions: 8, 11, 17, and 21.

Java has transformed from a primarily object-oriented language to one that elegantly incorporates functional programming concepts while maintaining backward compatibility. Modern Java code is more concise, expressive, and powerful than ever before.

This guide is organized by Java version, with deeper dives into the most impactful features. Each section includes:

- Clear explanations of new features
- Real-world use cases
- Detailed code examples with explanatory comments
- Before/after comparisons
- Common pitfalls to avoid

Let's begin your journey into modern Java!

Table of Contents

Java 8 (LTS)

Java 8, released in March 2014, was a revolutionary update that fundamentally changed how Java code is written. It introduced functional programming concepts to Java, making the language more expressive and concise.

Lambda Expressions

What and Why: Lambda expressions enable you to treat functionality as a method argument, or to create a single method interface instance using an expression. They represent a paradigm shift in Java programming by introducing functional programming capabilities.

Real-world Use Cases:

- Event handlers and callbacks
- Simplifying iteration of collections
- Facilitating parallel processing
- Implementing strategy patterns with less boilerplate

Before (Java 7) / After (Java 8) Comparison:

Before (Java 7):

```
// Creating a Runnable using anonymous inner class
Runnable runnable = new Runnable() {
    @Override
    public void run() {
        System.out.println("Running in a separate thread");
    }
};

// Iterating through a list
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
for (String name : names) {
    System.out.println(name);
}

// Sorting with a custom comparator
Collections.sort(names, new Comparator<String>() {
    @Override
    public int compare(String a, String b) {
        return a.compareTo(b);
    }
});
```

After (Java 8):

```
// Creating a Runnable using lambda expression
Runnable runnable = () -> System.out.println("Running in a separate thread");

// Iterating through a list with forEach and lambda
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
names.forEach(name -> System.out.println(name));

// Sorting with a lambda expression
Collections.sort(names, (a, b) -> a.compareTo(b));
```

Detailed Lambda Example:

```
import java.util.Arrays;
import java.util.List;
import java.util.function.Predicate;
import java.util.stream.Collectors;

public class LambdaExample {
    public static void main(String[] args) {
        // A list of employees
        List<Employee> employees = Arrays.asList(
            new Employee("John", "Engineering", 75000),
            new Employee("Mary", "HR", 65000),
            new Employee("Steve", "Engineering", 95000),
            new Employee("Anna", "Marketing", 80000),
            new Employee("Mike", "HR", 70000)
        );

        // Using lambdas to filter employees in the Engineering department
        // The lambda implements a Predicate<Employee> functional interface
        Predicate<Employee> isEngineer = employee -> "Engineering".equals(employee.getDepartment());

        // Using the predicate with a stream to filter and collect results
        List<Employee> engineers = employees.stream()
                                            .filter(isEngineer) // Apply the lambda predicate
```

```
                                        .collect(Collectors.toList());

        // Print the filtered list using a forEach with lambda
        System.out.println("Engineers:");
        engineers.forEach(engineer -> {
            // Multi-line lambda with explicit block
            System.out.println(engineer.getName() + " - $" + engineer.getSalary());
        });

        // Create another predicate for high-salary employees
        Predicate<Employee> isHighPaid = employee -> employee.getSalary() > 80000;

        // Combining predicates to find high-paid engineers
        Predicate<Employee> isHighPaidEngineer = isEngineer.and(isHighPaid);

        // Using the combined predicate
        List<Employee> highPaidEngineers = employees.stream()
                                                .filter(isHighPaidEngineer)
                                                .collect(Collectors.toList());

        System.out.println("\nHigh-paid engineers:");
        highPaidEngineers.forEach(engineer -> System.out.println(engineer.getName()));
    }
}

class Employee {
    private final String name;
    private final String department;
    private final int salary;

    public Employee(String name, String department, int salary) {
        this.name = name;
        this.department = department;
        this.salary = salary;
    }

    public String getName() { return name; }
    public String getDepartment() { return department; }
    public int getSalary() { return salary; }
}
```

Common Pitfalls and Best Practices:

- **Variable Capture**: Lambdas can only use local variables that are final or effectively final.
- **This Reference**: The `this` keyword inside a lambda refers to the enclosing class, not the lambda itself.
- **Excessive Nesting**: Avoid nesting lambdas too deeply, as this can make code harder to read.
- **Exception Handling**: Lambdas with checked exceptions require special handling.

```
// INCORRECT: Modifying a captured variable
int counter = 0;
Runnable r = () -> {
    counter++; // Compilation error: Variable must be final or effectively final
    System.out.println(counter);
};

// CORRECT: Use an array or AtomicInteger for mutable counters
final int[] counter = {0};
Runnable r = () -> {
    counter[0]++; // This works
    System.out.println(counter[0]);
};

// OR better, use AtomicInteger
AtomicInteger counter = new AtomicInteger(0);
Runnable r = () -> {
    System.out.println(counter.incrementAndGet());
};
```

Functional Interfaces

What and Why: Functional interfaces are interfaces that have exactly one abstract method. They serve as the foundation for lambda expressions in Java. The `@FunctionalInterface` annotation marks an interface as functional and causes a compilation error if the interface doesn't satisfy the requirements.

Common Functional Interfaces:

Java 8 introduced several functional interfaces in the `java.util.function` package:

1. `Function<T, R>`: Takes an input of type T and returns a value of type R
2. `Predicate<T>`: Takes an input of type T and returns a boolean
3. `Consumer<T>`: Takes an input of type T and returns no result (void)
4. `Supplier<T>`: Takes no input and returns a value of type T
5. `UnaryOperator<T>`: A function where the input and output are of the same type
6. `BinaryOperator<T>`: A function that takes two arguments of type T and returns a result of type T

Example with Function<T, R>:

```
import java.util.Arrays;
import java.util.List;
import java.util.function.Function;
import java.util.stream.Collectors;

public class FunctionExample {
    public static void main(String[] args) {
        // A list of strings
        List<String> names = Arrays.asList("John", "Mary", "Steve", "Anna", "Mike");

        // Define a Function that converts a string to its length
        // Function<T, R> takes an input of type T and returns a value of type R
        Function<String, Integer> getStringLength = str -> str.length();

        // Apply the function to each element using map
        List<Integer> nameLengths = names.stream()
                                         .map(getStringLength) // Apply the function to each element
                                         .collect(Collectors.toList());

        // Print the results
        System.out.println("Names: " + names);
        System.out.println("Lengths: " + nameLengths);

        // Function composition: andThen and compose
        // This function takes a number and returns its square
        Function<Integer, Integer> square = n -> n * n;

        // Composing functions: first get length, then square it
        Function<String, Integer> getLengthAndSquare = getStringLength.andThen(square);

        // Apply the composed function
        List<Integer> squaredLengths = names.stream()
                                            .map(getLengthAndSquare)
                                            .collect(Collectors.toList());

        System.out.println("Squared lengths: " + squaredLengths);
    }
}
```

Creating Custom Functional Interfaces:

```
// A custom functional interface
@FunctionalInterface
interface Transformer<T, R> {
    // Exactly one abstract method
    R transform(T input);

    // Default methods are allowed in functional interfaces
    default Transformer<T, R> compose(Transformer<? super R, ? extends R> after) {
        return input -> after.transform(transform(input));
    }

    // Static methods are also allowed
    static <T> Transformer<T, T> identity() {
        return t -> t;
    }
}

public class CustomFunctionalInterfaceExample {
    public static void main(String[] args) {
        // Using our custom functional interface with a lambda
        Transformer<String, Integer> stringToLength = s -> s.length();

        // Using the transformer
        String test = "Hello, Functional Interface!";
        int length = stringToLength.transform(test);
        System.out.println("Length of '" + test + "': " + length);
```

```
        // Using the compose method from our interface
        Transformer<Integer, Integer> doubled = n -> n * 2;
        Transformer<String, Integer> stringToLengthDoubled = stringToLength.compose(doubled);

        int doubledLength = stringToLengthDoubled.transform(test);
        System.out.println("Doubled length: " + doubledLength);

        // Using the static identity method
        Transformer<String, String> identityTransformer = Transformer.identity();
        String same = identityTransformer.transform(test);
        System.out.println("Identity: " + same);
    }
}
```

Common Pitfalls:

- **Overloaded Methods**: When using method references with overloaded methods, the compiler might not be able to infer which one to use.
- **Too Many Custom Interfaces**: Prefer using the standard functional interfaces from `java.util.function` rather than creating custom ones unless necessary.
- **Missing @FunctionalInterface**: While optional, this annotation helps catch errors if the interface doesn't conform to the functional interface contract.

Stream API

What and Why: The Stream API enables functional-style operations on streams of elements, such as map-reduce transformations on collections. Streams promote declarative programming, where you describe what you want to achieve rather than how to achieve it. They also facilitate parallel processing without the complexity of explicit thread management.

Real-world Use Cases:

- Data processing and transformation
- Filtering collections
- Aggregating data (sum, average, etc.)
- Parallel processing of large datasets
- Lazy evaluation of computations

Before (Java 7) / After (Java 8) Comparison:

Before (Java 7):

```
List<Person> people = getPersonList();
List<Person> adults = new ArrayList<>();

// Filter adults
for (Person person : people) {
    if (person.getAge() >= 18) {
        adults.add(person);
    }
}

// Sort by name
Collections.sort(adults, new Comparator<Person>() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.getName().compareTo(p2.getName());
    }
});

// Extract and join names
StringBuilder result = new StringBuilder();
for (int i = 0; i < adults.size(); i++) {
    if (i > 0) {
        result.append(", ");
    }
    result.append(adults.get(i).getName());
}
String adultNames = result.toString();
```

After (Java 8):

```
List<Person> people = getPersonList();

// All of the above in one concise stream operation
String adultNames = people.stream()
    .filter(person -> person.getAge() >= 18)      // Filter adults
    .sorted(Comparator.comparing(Person::getName)) // Sort by name
    .map(Person::getName)                          // Extract names
    .collect(Collectors.joining(", "));           // Join with commas
```

Comprehensive Stream Example:

```
import java.util.*;
import java.util.function.Function;
import java.util.stream.Collectors;
import java.util.stream.IntStream;
import java.util.stream.Stream;

public class StreamAPIExample {
    public static void main(String[] args) {
        // Creating a list of products
        List<Product> products = Arrays.asList(
            new Product(1, "Laptop", 1200.0, "Electronics"),
            new Product(2, "Phone", 800.0, "Electronics"),
            new Product(3, "Desk", 350.0, "Furniture"),
            new Product(4, "Chair", 150.0, "Furniture"),
            new Product(5, "Tablet", 500.0, "Electronics"),
            new Product(6, "Lamp", 50.0, "Furniture"),
            new Product(7, "Keyboard", 100.0, "Electronics"),
            new Product(8, "Mouse", 40.0, "Electronics"),
            new Product(9, "Bookshelf", 200.0, "Furniture"),
            new Product(10, "TV", 900.0, "Electronics")
        );

        // 1. Creating Streams
        System.out.println("==== Stream Creation ====");

        // From a collection
        Stream<Product> productStream = products.stream();

        // From individual objects
        Stream<String> stringStream = Stream.of("a", "b", "c");

        // Empty stream
        Stream<String> emptyStream = Stream.empty();

        // Infinite stream (limited for demonstration)
        Stream<Integer> infiniteStream = Stream.iterate(0, n -> n + 2);
        System.out.println("First 5 even numbers:");
        infiniteStream.limit(5).forEach(n -> System.out.print(n + " "));
        System.out.println();

        // 2. Intermediate Operations
        System.out.println("\n==== Filtering and Mapping ====");

        // Filter electronics products
        List<Product> electronics = products.stream()
                .filter(p -> "Electronics".equals(p.getCategory()))
                .collect(Collectors.toList());
        System.out.println("Electronics products count: " + electronics.size());

        // Map to get all product names
        List<String> productNames = products.stream()
                .map(Product::getName)  // Method reference - same as p -> p.getName()
                .collect(Collectors.toList());
        System.out.println("Product names: " + productNames);

        // 3. FlatMap: dealing with streams of streams
        System.out.println("\n==== FlatMap Example ====");

        // A list of lists
        List<List<Integer>> listOfLists = Arrays.asList(
            Arrays.asList(1, 2, 3),
            Arrays.asList(4, 5, 6),
            Arrays.asList(7, 8, 9)
        );

        // Using flatMap to flatten the list of lists
        List<Integer> allNumbers = listOfLists.stream()
                .flatMap(Collection::stream)  // Flattens each inner list into a single stream
                .collect(Collectors.toList());
        System.out.println("Flattened list: " + allNumbers);

        // 4. Sorting
        System.out.println("\n==== Sorting Products ====");

        // Sort by price
        List<Product> sortedByPrice = products.stream()
```

```
                .sorted(Comparator.comparing(Product::getPrice))
                .collect(Collectors.toList());
        System.out.println("Products sorted by price (lowest first):");
        sortedByPrice.forEach(p -> System.out.println(p.getName() + ": $" + p.getPrice()));

        // 5. Distinct, Skip, Limit
        System.out.println("\n==== Other Intermediate Operations ====");

        // Get distinct categories
        List<String> categories = products.stream()
                .map(Product::getCategory)
                .distinct()  // Remove duplicates
                .collect(Collectors.toList());
        System.out.println("Distinct categories: " + categories);

        // Skip first 3 products and limit to next 3
        List<Product> paginatedProducts = products.stream()
                .skip(3)     // Skip the first 3 products
                .limit(3)    // Take only the next 3 products
                .collect(Collectors.toList());
        System.out.println("Products 4-6:");
        paginatedProducts.forEach(p -> System.out.println(p.getId() + ": " + p.getName()));

        // 6. Terminal Operations
        System.out.println("\n==== Terminal Operations ====");

        // Counting
        long productCount = products.stream().count();
        System.out.println("Total products: " + productCount);

        // Any/All matching
        boolean anyExpensive = products.stream()
                .anyMatch(p -> p.getPrice() > 1000);  // Are any products > $1000?
        boolean allAffordable = products.stream()
                .allMatch(p -> p.getPrice() < 2000);  // Are all products < $2000?
        System.out.println("Any products > $1000: " + anyExpensive);
        System.out.println("All products < $2000: " + allAffordable);

        // 7. Collection Operations
        System.out.println("\n==== Collecting Results ====");

        // Find average price of electronics
        double avgElectronicsPrice = products.stream()
                .filter(p -> "Electronics".equals(p.getCategory()))
                .collect(Collectors.averagingDouble(Product::getPrice));
        System.out.println("Average electronics price: $" + avgElectronicsPrice);

        // Group products by category
        Map<String, List<Product>> productsByCategory = products.stream()
                .collect(Collectors.groupingBy(Product::getCategory));
        System.out.println("Products by category:");
        productsByCategory.forEach((category, prods) -> {
            System.out.println(category + ": " +
                prods.stream().map(Product::getName).collect(Collectors.joining(", ")));
        });

        // 8. Numeric Streams for better performance
        System.out.println("\n==== Numeric Streams ====");

        // Calculate total inventory value using specialized sum operation
        double totalValue = products.stream()
                .mapToDouble(Product::getPrice)  // Convert to DoubleStream
                .sum();                          // Specialized sum method
        System.out.println("Total inventory value: $" + totalValue);

        // Generate statistics in one operation
        DoubleSummaryStatistics priceStats = products.stream()
                .mapToDouble(Product::getPrice)
                .summaryStatistics();  // Gets min, max, average, count, and sum
        System.out.println("Price statistics: " + priceStats);

        // 9. Parallel Streams
        System.out.println("\n==== Parallel Streams ====");

        // Create a large dataset
        List<Integer> largeList = IntStream.rangeClosed(1, 10_000_000)
                .boxed()
                .collect(Collectors.toList());

        // Measure time for sequential processing
        long startSeq = System.currentTimeMillis();
```

```
        long sumSeq = largeList.stream()
                .filter(n -> n % 2 == 0)
                .mapToLong(n -> n)
                .sum();
        long endSeq = System.currentTimeMillis();

        // Measure time for parallel processing
        long startPar = System.currentTimeMillis();
        long sumPar = largeList.parallelStream()  // Use parallel stream
                .filter(n -> n % 2 == 0)
                .mapToLong(n -> n)
                .sum();
        long endPar = System.currentTimeMillis();

        System.out.println("Sequential sum: " + sumSeq + " in " + (endSeq - startSeq) + "ms");
        System.out.println("Parallel sum: " + sumPar + " in " + (endPar - startPar) + "ms");
    }
}

class Product {
    private final int id;
    private final String name;
    private final double price;
    private final String category;

    public Product(int id, String name, double price, String category) {
        this.id = id;
        this.name = name;
        this.price = price;
        this.category = category;
    }

    public int getId() { return id; }
    public String getName() { return name; }
    public double getPrice() { return price; }
    public String getCategory() { return category; }
}
```

Common Stream Operations and Their Uses:

1. **Filtering**: `filter()` - Keep elements that satisfy a predicate
2. **Transformation**: `map()` - Transform each element
3. **Flattening**: `flatMap()` - Transform and flatten nested streams
4. **Ordering**: `sorted()` - Sort elements
5. **Limiting**: `limit()` and `skip()` - Pagination
6. **Uniqueness**: `distinct()` - Remove duplicates
7. **Searching**: `findFirst()`, `findAny()` - Find elements
8. **Matching**: `anyMatch()`, `allMatch()`, `noneMatch()` - Test predicates
9. **Reduction**: `reduce()` - Combine elements
10. **Collection**: `collect()` - Create collections from streams

Common Pitfalls:

- **Stream Consumption**: Streams can be consumed only once. Reusing a stream after a terminal operation will throw an exception.
- **Side Effects**: Avoid side effects in stream operations (like modifying shared state), as this can lead to hard-to-debug issues, especially with parallel streams.
- **Parallel Stream Misuse**: Don't use parallel streams for small collections or when operations are lightweight. The overhead can outweigh the benefits.
- **Order Dependence**: Be aware that some operations may change the order of elements, which can be problematic if order matters.

```
// INCORRECT: Reusing a stream
Stream<String> stream = Arrays.asList("a", "b", "c").stream();
stream.forEach(System.out::println);
stream.forEach(System.out::println); // IllegalStateException: stream has already been operated upon or closed

// INCORRECT: Side effect in a parallel stream
List<String> result = Collections.synchronizedList(new ArrayList<>());
listOfStrings.parallelStream().filter(s -> s.length() > 3).forEach(result::add);
// The order of addition is unpredictable, and you might have synchronization issues

// CORRECT: Use collect() instead
List<String> result = listOfStrings.parallelStream()
    .filter(s -> s.length() > 3)
    .collect(Collectors.toList());
```

Optional

What and Why: The `Optional` class was introduced to provide a better way to handle null values and prevent NullPointerExceptions. It's a container object that may or may not contain a non-null value, forcing developers to explicitly handle both the presence and absence of values.

Real-world Use Cases:

- Representing nullable return values without risking NullPointerExceptions
- Streamlining null checks in method chains
- Enforcing explicit handling of potentially missing values
- API design to indicate optional outcomes or results

Before (Java 7) / After (Java 8) Comparison:

Before (Java 7):

```
// Method that might return null
public String findUsername(int id) {
    // Business logic...
    if (userExists) {
        return username;
    } else {
        return null;
    }
}

// Using the method
String username = findUsername(42);
if (username != null) {
    String transformed = username.toUpperCase();
    System.out.println(transformed);
} else {
    System.out.println("Username not found");
}
```

After (Java 8):

```
// Method that returns an Optional
public Optional<String> findUsername(int id) {
    // Business logic...
    if (userExists) {
        return Optional.of(username);
    } else {
        return Optional.empty();
    }
}

// Using the method
Optional<String> username = findUsername(42);
username.map(String::toUpperCase)
        .ifPresentOrElse(
            transformed -> System.out.println(transformed),
            () -> System.out.println("Username not found")
        );
```

Comprehensive Optional Example:

```
import java.util.HashMap;
import java.util.Map;
import java.util.Optional;

public class OptionalExample {

    // Simulate a user database
    private static final Map<Integer, User> userDatabase = new HashMap<>();

    static {
        userDatabase.put(1, new User(1, "john_doe", "John Doe", "john@example.com"));
        userDatabase.put(2, new User(2, "jane_doe", "Jane Doe", null)); // No email provided
        // User with ID 3 doesn't exist
    }

    public static void main(String[] args) {
        // 1. Creating Optionals

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

```
// Optional with a non-null value
Optional<String> nonEmpty = Optional.of("Hello");

// Optional that might hold a null value
String nullableValue = null;
Optional<String> nullable = Optional.ofNullable(nullableValue);

// 2. Checking if value is present
System.out.println("Empty has value: " + empty.isPresent());
System.out.println("NonEmpty has value: " + nonEmpty.isPresent());
System.out.println("Nullable has value: " + nullable.isPresent());

// 3. Accessing values safely

// Using orElse (provides a default if the Optional is empty)
String result1 = empty.orElse("Default Value");
System.out.println("Empty orElse: " + result1);

// Using orElseGet (lazy evaluation with Supplier)
String result2 = empty.orElseGet(() -> {
    // This code only executes if Optional is empty
    return "Computed Default";
});
System.out.println("Empty orElseGet: " + result2);

// Using orElseThrow (throw exception if empty)
try {
    String result3 = empty.orElseThrow(() -> new IllegalStateException("Optional is empty"));
} catch (IllegalStateException e) {
    System.out.println("Exception caught: " + e.getMessage());
}

// 4. Optional in action: User lookup example
int userId = 1;

// Find a user's email using Optional for cleaner handling of nulls
String email = findUserById(userId)
        .flatMap(User::getEmailOptional) // Safely chain with another Optional
        .orElse("No email found");

System.out.println("\nUser " + userId + " email: " + email);

// Try with a user that has no email
userId = 2;
email = findUserById(userId)
        .flatMap(User::getEmailOptional)
        .orElse("No email found");

System.out.println("User " + userId + " email: " + email);

// Try with a user that doesn't exist
userId = 3;
email = findUserById(userId)
        .flatMap(User::getEmailOptional)
        .orElse("No email found");

System.out.println("User " + userId + " email: " + email);

// 5. Transforming values with map
Optional<User> user = findUserById(1);
Optional<String> username = user.map(User::getUsername);
System.out.println("\nUsername: " + username.orElse("Unknown"));

// 6. Filtering values
Optional<User> filteredUser = findUserById(1)
        .filter(u -> u.getUsername().startsWith("j"));

System.out.println("\nFiltered user present: " + filteredUser.isPresent());

// 7. Combining Optional operations
String displayName = findUserById(1)
        .filter(u -> u.getDisplayName().length() > 5)
        .map(User::getDisplayName)
        .map(String::toUpperCase)
        .orElse("Name too short or user not found");

System.out.println("\nDisplay name: " + displayName);

// 8. Using ifPresent for side effects
System.out.println("\nProcessing user if present:");
```

```
        findUserById(1).ifPresent(u -> {
            System.out.println("Found user: " + u.getDisplayName());
            // Perform other operations with the user
        });

        // 9. Java 9+ additions (ifPresentOrElse, or, stream)
        // ifPresentOrElse lets you handle both the present and empty cases
        findUserById(3).ifPresentOrElse(
            u -> System.out.println("\nFound user: " + u.getDisplayName()),
            () -> System.out.println("\nUser not found")
        );
    }

    // Method returning an Optional
    public static Optional<User> findUserById(int id) {
        return Optional.ofNullable(userDatabase.get(id));
    }

    // User class with an Optional getter for email
    static class User {
        private final int id;
        private final String username;
        private final String displayName;
        private final String email; // Might be null

        User(int id, String username, String displayName, String email) {
            this.id = id;
            this.username = username;
            this.displayName = displayName;
            this.email = email;
        }

        public int getId() { return id; }
        public String getUsername() { return username; }
        public String getDisplayName() { return displayName; }

        // Regular getter that might return null
        public String getEmail() { return email; }

        // Better approach: return an Optional
        public Optional<String> getEmailOptional() {
            return Optional.ofNullable(email);
        }
    }
}
```

Common Pitfalls:

- **Optional as a Field**: Avoid using `Optional` as a field in your classes, as it's not serializable.
- **Optional in Method Parameters**: Don't use `Optional` as a method parameter, as it makes the API confusing. Use overloaded methods instead.
- **Unnecessary Unwrapping**: Don't immediately unwrap an `Optional` with `get()` without checking if a value is present.
- **Missing the Point**: Using `Optional` just to avoid null checks defeats its purpose. It's meant to express that a value may be absent and to force handling of that case.

```
// INCORRECT: Using Optional as a class field
public class User {
    private Optional<String> email; // Avoid this
}

// INCORRECT: Using Optional as a method parameter
public void processUser(Optional<User> user) { // Avoid this
    // ...
}

// INCORRECT: Immediate unwrapping without checking
Optional<User> user = findUser(id);
User unwrapped = user.get(); // May throw NoSuchElementException

// CORRECT: Check before getting
Optional<User> user = findUser(id);
if (user.isPresent()) {
    User unwrapped = user.get();
    // Process user
}

// BETTER: Use methods like map, flatMap, ifPresent, or orElse
User processedUser = findUser(id)
```

```
    .map(this::processUser)
    .orElseGet(this::createDefaultUser);
```

Default and Static Methods in Interfaces

What and Why: Default and static methods in interfaces allow you to add new methods to interfaces without breaking existing implementations. This was crucial for evolving the Java API, especially for adding new methods to the Collection interfaces as part of the Stream API.

Real-world Use Cases:

- Adding new functionality to interfaces in libraries without breaking backward compatibility
- Providing utility methods related to an interface
- Implementing the template method pattern
- Creating utility classes that are tied to an interface

Before (Java 7) / After (Java 8) Comparison:

Before (Java 7):

```
// Interface
public interface PaymentProcessor {
    void processPayment(Payment payment);
}

// Implementation
public class CreditCardProcessor implements PaymentProcessor {
    @Override
    public void processPayment(Payment payment) {
        // Process credit card payment
    }
}

// If you wanted to add a new method to the interface, all implementations would break
```

After (Java 8):

```
// Interface with a default method
public interface PaymentProcessor {
    void processPayment(Payment payment);

    // New method added without breaking existing implementations
    default boolean validatePayment(Payment payment) {
        // Default implementation that subclasses can override if needed
        return payment != null && payment.getAmount() > 0;
    }

    // Static utility method
    static PaymentProcessor getDefaultProcessor() {
        return new DefaultPaymentProcessor();
    }
}

// Existing implementation continues to work without changes
public class CreditCardProcessor implements PaymentProcessor {
    @Override
    public void processPayment(Payment payment) {
        // Process credit card payment
    }

    // Can optionally override the default method
    @Override
    public boolean validatePayment(Payment payment) {
        // Custom validation for credit cards
        return payment != null && payment.getAmount() > 0 && payment.getCreditCardNumber() != null;
    }
}
```

Comprehensive Example:

```
import java.util.Arrays;
import java.util.List;
import java.util.function.Predicate;

public class DefaultAndStaticMethodsExample {
```

```
    public static void main(String[] args) {
        // Create some payment processors
        PaymentProcessor creditCardProcessor = new CreditCardProcessor();
        PaymentProcessor payPalProcessor = new PayPalProcessor();

        // Create a payment
        Payment payment = new Payment(100.0, "USD", "4111111111111111");

        // Process the payment with different processors
        System.out.println("Processing with credit card:");
        creditCardProcessor.processPayment(payment);

        System.out.println("\nProcessing with PayPal:");
        payPalProcessor.processPayment(payment);

        // Use the default validation method
        boolean ccValid = creditCardProcessor.validatePayment(payment);
        boolean ppValid = payPalProcessor.validatePayment(payment);

        System.out.println("\nValidation results:");
        System.out.println("Credit card validation: " + (ccValid ? "Valid" : "Invalid"));
        System.out.println("PayPal validation: " + (ppValid ? "Valid" : "Invalid"));

        // Try with an invalid payment
        Payment invalidPayment = new Payment(-50.0, "USD", null);
        System.out.println("\nValidating invalid payment:");
        System.out.println("Credit card validation: " +
                           (creditCardProcessor.validatePayment(invalidPayment) ? "Valid" : "Invalid"));
        System.out.println("PayPal validation: " +
                           (payPalProcessor.validatePayment(invalidPayment) ? "Valid" : "Invalid"));

        // Use the static method to get a default processor
        PaymentProcessor defaultProcessor = PaymentProcessor.getDefaultProcessor();
        System.out.println("\nProcessing with default processor:");
        defaultProcessor.processPayment(payment);

        // Use the static utility methods
        List<Payment> payments = Arrays.asList(
            new Payment(100.0, "USD", "4111111111111111"),
            new Payment(200.0, "EUR", "5555555555554444"),
            new Payment(-50.0, "USD", null),
            new Payment(300.0, "GBP", "3782822463100005")
        );

        double totalValid = PaymentProcessor.sumValidPayments(payments);
        System.out.println("\nTotal valid payments: " + totalValid);

        // Using the chain method to combine validations
        Predicate<Payment> isLargeAmount = p -> p.getAmount() > 150.0;
        boolean isValidAndLarge = creditCardProcessor
                .validatePaymentWithCondition(payment, isLargeAmount);
        System.out.println("\nIs payment valid and large? " + isValidAndLarge);
    }
}

// Payment class
class Payment {
    private final double amount;
    private final String currency;
    private final String creditCardNumber; // Simplified for the example

    public Payment(double amount, String currency, String creditCardNumber) {
        this.amount = amount;
        this.currency = currency;
        this.creditCardNumber = creditCardNumber;
    }

    public double getAmount() { return amount; }
    public String getCurrency() { return currency; }
    public String getCreditCardNumber() { return creditCardNumber; }
}

// Interface with default and static methods
interface PaymentProcessor {
    // Abstract method that must be implemented
    void processPayment(Payment payment);

    // Default method with an implementation
    default boolean validatePayment(Payment payment) {
        return payment != null && payment.getAmount() > 0;
    }
```

```
    // Another default method that uses the first one and takes additional parameters
    default boolean validatePaymentWithCondition(Payment payment, Predicate<Payment> condition) {
        return validatePayment(payment) && condition.test(payment);
    }

    // Static method to get a default implementation
    static PaymentProcessor getDefaultProcessor() {
        return new DefaultPaymentProcessor();
    }

    // Static utility method
    static double sumValidPayments(List<Payment> payments) {
        return payments.stream()
                .filter(p -> getDefaultProcessor().validatePayment(p))
                .mapToDouble(Payment::getAmount)
                .sum();
    }
}

// Implementation 1
class CreditCardProcessor implements PaymentProcessor {
    @Override
    public void processPayment(Payment payment) {
        System.out.println("Processing credit card payment of "
            + payment.getAmount() + " " + payment.getCurrency());

        if (payment.getCreditCardNumber() == null) {
            System.out.println("Error: Missing credit card number");
        } else {
            System.out.println("Charged to card ending with "
                + payment.getCreditCardNumber().substring(payment.getCreditCardNumber().length() - 4));
        }
    }

    // Override the default method to add credit card specific validation
    @Override
    public boolean validatePayment(Payment payment) {
        boolean baseValidation = PaymentProcessor.super.validatePayment(payment);
        return baseValidation && payment.getCreditCardNumber() != null;
    }
}

// Implementation 2 (doesn't override default method)
class PayPalProcessor implements PaymentProcessor {
    @Override
    public void processPayment(Payment payment) {
        System.out.println("Processing PayPal payment of "
            + payment.getAmount() + " " + payment.getCurrency());
        System.out.println("Redirecting to PayPal for authorization...");
    }
}

// Default implementation provided by the static method
class DefaultPaymentProcessor implements PaymentProcessor {
    @Override
    public void processPayment(Payment payment) {
        System.out.println("Processing payment with default processor");
        if (validatePayment(payment)) {
            System.out.println("Payment of " + payment.getAmount() + " "
                + payment.getCurrency() + " processed successfully");
        } else {
            System.out.println("Payment validation failed");
        }
    }
}
```

Multiple Interface Inheritance Resolution:

```
interface A {
    default void method() {
        System.out.println("Method from A");
    }
}

interface B {
    default void method() {
        System.out.println("Method from B");
    }
```

```
}

// This won't compile: "class C inherits unrelated defaults for method() from interfaces A and B"
// class C implements A, B { }

// Solution 1: Override the method and choose which implementation to use
class C1 implements A, B {
    @Override
    public void method() {
        // Choose implementation from A
        A.super.method();

        // Or do something completely different
        System.out.println("Custom implementation in C1");
    }
}

// Solution 2: Create a new interface that extends both and resolves the conflict
interface C2Interface extends A, B {
    @Override
    default void method() {
        A.super.method(); // Choose A's implementation
    }
}

class C2 implements C2Interface {
    // No conflict here, C2Interface already resolved it
}
```

Common Pitfalls:

- **Multiple Inheritance Conflicts**: When a class implements multiple interfaces with the same default method, you must override the method to resolve the conflict.
- **Shadowing vs. Overriding**: In hierarchical interfaces, a subinterface's default method shadows the parent's default method.
- **Static Methods Accessibility**: Static methods in interfaces cannot be inherited by implementations; they must be called via the interface name.
- **Interface vs. Abstract Class**: Don't forget that interfaces still can't have state, constructors, or non-public methods.

```
// INCORRECT: Trying to call a static interface method through an instance
PaymentProcessor processor = new CreditCardProcessor();
double total = processor.sumValidPayments(payments); // Compilation error

// CORRECT: Call static methods through the interface name
double total = PaymentProcessor.sumValidPayments(payments);

// INCORRECT: Trying to access instance in default method
interface Problematic {
    default void doSomething() {
        this.privateHelper(); // No private methods in interfaces before Java 9
    }

    void privateHelper(); // Forces all implementations to provide this
}

// CORRECT: In Java 9+, use private methods in interfaces
interface BetterSolution {
    default void doSomething() {
        privateHelper(); // Can call private helper method
    }

    // Java 9+ allows private methods in interfaces
    private void privateHelper() {
        // Implementation details
    }
}
```

Method References

What and Why: Method references provide a shorthand notation for lambda expressions that call a single method. They make code more concise and readable when a lambda's primary purpose is to invoke an existing method.

Real-world Use Cases:

- Concise transformation and mapping operations in streams
- Event handling in UIs
- Implementing functional interfaces when direct method calls are sufficient
- Creating factory methods

Before (Lambda) / After (Method Reference) Comparison:

Before (With Lambda):

```
// Using lambda expressions
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

// Print each name
names.forEach(name -> System.out.println(name));

// Get the length of each name
List<Integer> nameLengths = names.stream()
                                 .map(name -> name.length())
                                 .collect(Collectors.toList());

// Sort names by their natural order
names.sort((s1, s2) -> s1.compareTo(s2));
```

After (With Method References):

```
// Using method references
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

// Print each name (instance method reference on System.out)
names.forEach(System.out::println);

// Get the length of each name (instance method reference of arbitrary object)
List<Integer> nameLengths = names.stream()
                                 .map(String::length)
                                 .collect(Collectors.toList());

// Sort names by their natural order (instance method reference)
names.sort(String::compareTo);
```

Types of Method References:

1. **Static method reference**: `ClassName::staticMethodName`
2. **Instance method of a particular object**: `instanceRef::methodName`
3. **Instance method of an arbitrary object of a particular type**: `ClassName::methodName`
4. **Constructor reference**: `ClassName::new`

Comprehensive Example:

```
import java.util.*;
import java.util.function.*;
import java.util.stream.Collectors;

public class MethodReferenceExample {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David", "Eva");

        // 1. Static Method Reference
        System.out.println("\n== Static Method Reference ==");

        // Lambda expression
        names.stream().forEach(s -> MethodReferenceExample.printWithPrefix(s));

        // Equivalent method reference
        names.stream().forEach(MethodReferenceExample::printWithPrefix);

        // 2. Instance Method Reference of a Particular Object
        System.out.println("\n== Instance Method Reference of a Particular Object ==");

        // Create a formatter
        StringFormatter formatter = new StringFormatter();

        // Lambda expression
        names.stream().map(s -> formatter.capitalize(s)).forEach(System.out::println);

        // Equivalent method reference
        names.stream().map(formatter::capitalize).forEach(System.out::println);

        // 3. Instance Method Reference of an Arbitrary Object of a Particular Type
        System.out.println("\n== Instance Method Reference of an Arbitrary Object ==");
```

```
        // Lambda expression
        names.stream().map(s -> s.toUpperCase()).forEach(System.out::println);

        // Equivalent method reference
        names.stream().map(String::toUpperCase).forEach(System.out::println);

        // Another example: sorting
        // Lambda
        names.sort((s1, s2) -> s1.compareToIgnoreCase(s2));

        // Method reference
        names.sort(String::compareToIgnoreCase);

        // 4. Constructor Reference
        System.out.println("\n== Constructor Reference ==");

        // Lambda expression
        Function<String, StringBuilder> builderFunction = s -> new StringBuilder(s);
        StringBuilder sb1 = builderFunction.apply("Hello");
        System.out.println("Using lambda: " + sb1);

        // Equivalent constructor reference
        Function<String, StringBuilder> builderMethodRef = StringBuilder::new;
        StringBuilder sb2 = builderMethodRef.apply("Hello");
        System.out.println("Using constructor reference: " + sb2);

        // 5. Complex example: Combining different types of method references
        System.out.println("\n== Complex Example ==");

        // Create a list of persons
        List<Person> persons = Arrays.asList(
            new Person("John", "Doe", 28),
            new Person("Jane", "Smith", 32),
            new Person("Alice", "Johnson", 25),
            new Person("Bob", "Brown", 40)
        );

        // Sort by age using static method
        System.out.println("\nSorted by age:");
        persons.sort(MethodReferenceExample::compareByAge);
        persons.forEach(System.out::println);

        // Group by age category using a specific instance method
        System.out.println("\nGrouped by age category:");
        AgeClassifier classifier = new AgeClassifier();
        Map<String, List<Person>> personsByAgeCategory = persons.stream()
                .collect(Collectors.groupingBy(classifier::classifyAge));

        personsByAgeCategory.forEach((category, people) -> {
            System.out.println(category + ":");
            people.forEach(p -> System.out.println("  " + p));
        });

        // Transform to DTOs using constructor reference
        System.out.println("\nTransformed to DTOs:");
        List<PersonDTO> dtos = persons.stream()
                .map(PersonDTO::new) // Constructor reference
                .collect(Collectors.toList());

        dtos.forEach(System.out::println);

        // 6. Method reference with method overloading
        System.out.println("\n== Method Reference with Overloading ==");

        // The compiler determines which print method to use based on the context
        Consumer<String> stringPrinter = MethodReferenceExample::print;
        Consumer<Integer> intPrinter = MethodReferenceExample::print;

        stringPrinter.accept("Hello, world!");
        intPrinter.accept(42);
    }

    // Static method for method reference
    public static void printWithPrefix(String s) {
        System.out.println("Name: " + s);
    }

    // Static method for comparing persons by age
    public static int compareByAge(Person p1, Person p2) {
        return Integer.compare(p1.getAge(), p2.getAge());
    }
```

```
    // Overloaded methods to demonstrate context-based resolution
    public static void print(String s) {
        System.out.println("String: " + s);
    }

    public static void print(Integer i) {
        System.out.println("Integer: " + i);
    }
}

// Class for instance method references
class StringFormatter {
    public String capitalize(String s) {
        if (s == null || s.isEmpty()) {
            return s;
        }
        return s.substring(0, 1).toUpperCase() + s.substring(1).toLowerCase();
    }
}

// Class for age classification
class AgeClassifier {
    public String classifyAge(Person person) {
        int age = person.getAge();
        if (age < 30) return "Young";
        if (age < 50) return "Middle-aged";
        return "Senior";
    }
}

// Person class
class Person {
    private final String firstName;
    private final String lastName;
    private final int age;

    public Person(String firstName, String lastName, int age) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
    }

    public String getFirstName() { return firstName; }
    public String getLastName() { return lastName; }
    public int getAge() { return age; }

    @Override
    public String toString() {
        return firstName + " " + lastName + " (" + age + ")";
    }
}

// DTO class for constructor reference
class PersonDTO {
    private final String fullName;
    private final int age;

    // Constructor that takes a Person (used in constructor reference)
    public PersonDTO(Person person) {
        this.fullName = person.getFirstName() + " " + person.getLastName();
        this.age = person.getAge();
    }

    @Override
    public String toString() {
        return "PersonDTO[fullName=" + fullName + ", age=" + age + "]";
    }
}
```

Common Pitfalls:

- **Ambiguous Method References**: If a class has overloaded methods with the same name, the compiler might not be able to determine which one to use.
- **Context Sensitivity**: Method references rely on the context to determine the functional interface they implement.
- **Non-Public Methods**: Method references require that the referenced method be accessible.

```
// INCORRECT: Ambiguous method reference
class StringUtils {
```

```
    public static String process(String s, int mode) { /* ... */ }
    public static String process(String s, String option) { /* ... */ }
}

// This might not compile due to ambiguity
Function<String, String> processor = StringUtils::process; // Error: Ambiguous

// CORRECT: Disambiguate with a lambda
Function<String, String> processor = s -> StringUtils.process(s, 1);

// INCORRECT: Reference to inaccessible method
class Restricted {
    private static void secretMethod(String s) { /* ... */ }
}

// This won't compile: secretMethod is private
Consumer<String> consumer = Restricted::secretMethod; // Error: secretMethod has private access

// CORRECT: Reference to accessible method
class Accessible {
    public static void publicMethod(String s) { /* ... */ }
}

Consumer<String> consumer = Accessible::publicMethod; // Works fine
```

New Date/Time API

What and Why: Java 8 introduced a new date and time API in the `java.time` package to replace the problematic and inconsistent old date/time classes (`java.util.Date`, `java.util.Calendar`, etc.). The new API is immutable, thread-safe, and provides a clearer and more comprehensive approach to handling dates, times, durations, and time zones.

Real-world Use Cases:

- Scheduling applications (appointments, meetings, reminders)
- Financial applications (transaction timestamps, interest calculations)
- Travel applications (flight times, time zone conversions)
- Activity logging and auditing
- Age calculations and date of birth validations

Before (Java 7) / After (Java 8) Comparison:

Before (Java 7):

```
// Create a date for a specific day
Calendar calendar = Calendar.getInstance();
calendar.set(2015, Calendar.JANUARY, 1); // Month is 0-based!
Date january1st2015 = calendar.getTime();

// Add one month
calendar.add(Calendar.MONTH, 1);
Date february1st2015 = calendar.getTime();

// Format a date
SimpleDateFormat formatter = new SimpleDateFormat("yyyy-MM-dd");
String formattedDate = formatter.format(january1st2015);

// Parse a date (requires exception handling)
try {
    Date parsedDate = formatter.parse("2015-01-15");
} catch (ParseException e) {
    e.printStackTrace();
}
```

After (Java 8):

```
// Create a date for a specific day
LocalDate january1st2015 = LocalDate.of(2015, Month.JANUARY, 1);

// Add one month
LocalDate february1st2015 = january1st2015.plusMonths(1);

// Format a date
String formattedDate = january1st2015.format(DateTimeFormatter.ISO_LOCAL_DATE);

// Parse a date
LocalDate parsedDate = LocalDate.parse("2015-01-15");
```

Comprehensive Example:

```
import java.time.*;
import java.time.format.DateTimeFormatter;
import java.time.format.FormatStyle;
import java.time.temporal.ChronoField;
import java.time.temporal.ChronoUnit;
import java.time.temporal.TemporalAdjusters;
import java.util.Locale;

public class DateTimeApiExample {
    public static void main(String[] args) {
        // 1. Core Classes
        System.out.println("\n== Core Classes ==");

        // LocalDate: a date without time or timezone (year, month, day)
        LocalDate today = LocalDate.now();
        System.out.println("Today's date: " + today);

        // LocalTime: a time without date or timezone (hour, minute, second, nanosecond)
        LocalTime currentTime = LocalTime.now();
        System.out.println("Current time: " + currentTime);

        // LocalDateTime: a date and time without timezone
        LocalDateTime currentDateTime = LocalDateTime.now();
        System.out.println("Current date and time: " + currentDateTime);

        // ZonedDateTime: a date and time with timezone
        ZonedDateTime currentZonedDateTime = ZonedDateTime.now();
        System.out.println("Current date and time with timezone: " + currentZonedDateTime);

        // Instant: a point in time (UTC)
        Instant timestamp = Instant.now();
        System.out.println("Current timestamp: " + timestamp);

        // 2. Creating Date-Time Objects
        System.out.println("\n== Creating Date-Time Objects ==");

        // Creating specific dates
        LocalDate dateOfBirth = LocalDate.of(1990, Month.MAY, 15);
        System.out.println("Date of Birth: " + dateOfBirth);

        // Creating specific times
        LocalTime meetingTime = LocalTime.of(13, 30); // 1:30 PM
        System.out.println("Meeting time: " + meetingTime);

        // Combined date and time
        LocalDateTime meetingDateTime = LocalDateTime.of(2023, 1, 10, 13, 30);
        System.out.println("Meeting date and time: " + meetingDateTime);

        // With timezone
        ZoneId newYorkZone = ZoneId.of("America/New_York");
        ZonedDateTime newYorkTime = ZonedDateTime.of(meetingDateTime, newYorkZone);
        System.out.println("Meeting in New York: " + newYorkTime);

        // 3. Date-Time Operations
        System.out.println("\n== Date-Time Operations ==");

        // Adding and subtracting
        LocalDate nextWeek = today.plusWeeks(1);
        System.out.println("Date next week: " + nextWeek);

        LocalDate lastMonth = today.minusMonths(1);
        System.out.println("Date last month: " + lastMonth);

        // Chaining operations
        LocalDateTime futureDateTime = currentDateTime
                .plusYears(1)
                .plusMonths(2)
                .plusDays(3)
                .plusHours(4)
                .plusMinutes(5);
        System.out.println("Future date time: " + futureDateTime);

        // Period: represents a period of days, months, years
        Period period = Period.between(dateOfBirth, today);
        System.out.println("You are " + period.getYears() + " years, " +
                           period.getMonths() + " months, and " +
```

```
                    period.getDays() + " days old");

        // Duration: represents a duration in seconds, nanoseconds
        Duration duration = Duration.between(meetingTime, LocalTime.now());
        System.out.println("Hours until/since meeting: " + duration.toHours());

        // 4. Temporal Adjusters
        System.out.println("\n== Temporal Adjusters ==");

        // Find the next Monday
        LocalDate nextMonday = today.with(TemporalAdjusters.next(DayOfWeek.MONDAY));
        System.out.println("Next Monday: " + nextMonday);

        // First day of next month
        LocalDate firstDayOfNextMonth = today.with(TemporalAdjusters.firstDayOfNextMonth());
        System.out.println("First day of next month: " + firstDayOfNextMonth);

        // Last day of current month
        LocalDate lastDayOfMonth = today.with(TemporalAdjusters.lastDayOfMonth());
        System.out.println("Last day of this month: " + lastDayOfMonth);

        // 5. Date Formatting and Parsing
        System.out.println("\n== Formatting and Parsing ==");

        // Predefined formatters
        String isoDate = today.format(DateTimeFormatter.ISO_LOCAL_DATE);
        System.out.println("ISO formatted date: " + isoDate);

        // Custom formatter
        DateTimeFormatter customFormatter = DateTimeFormatter.ofPattern("MMMM dd, yyyy");
        String formattedDate = today.format(customFormatter);
        System.out.println("Custom formatted date: " + formattedDate);

        // Locale-specific formatting
        DateTimeFormatter frenchFormatter = DateTimeFormatter
                .ofLocalizedDate(FormatStyle.FULL)
                .withLocale(Locale.FRANCE);
        String frenchDate = today.format(frenchFormatter);
        System.out.println("French formatted date: " + frenchDate);

        // Parsing
        LocalDate parsedDate = LocalDate.parse("2023-04-15");
        System.out.println("Parsed date: " + parsedDate);

        LocalDate customParsedDate = LocalDate.parse("April 15, 2023", customFormatter);
        System.out.println("Custom parsed date: " + customParsedDate);

        // 6. Time Zones and Offsets
        System.out.println("\n== Time Zones and Offsets ==");

        // List available time zones (abbreviated for readability)
        System.out.println("Available time zones (sample): " +
                           ZoneId.getAvailableZoneIds().stream().limit(5).toList());

        // Convert between time zones
        ZoneId tokyoZone = ZoneId.of("Asia/Tokyo");
        ZonedDateTime tokyoTime = currentZonedDateTime.withZoneSameInstant(tokyoZone);
        System.out.println("Current time in Tokyo: " + tokyoTime);

        // Fixed offsets
        ZoneOffset offset = ZoneOffset.of("+02:00");
        OffsetDateTime offsetDateTime = OffsetDateTime.of(currentDateTime, offset);
        System.out.println("Date time with +02:00 offset: " + offsetDateTime);

        // 7. Date Queries and Checks
        System.out.println("\n== Date Queries and Checks ==");

        // Check if a year is a leap year
        boolean isLeapYear = today.isLeapYear();
        System.out.println("Is " + today.getYear() + " a leap year? " + isLeapYear);

        // Compare dates
        LocalDate future = LocalDate.of(2030, 1, 1);
        boolean isBefore = today.isBefore(future);
        System.out.println("Is today before 2030-01-01? " + isBefore);

        // Get day of week, month, year
        DayOfWeek dayOfWeek = today.getDayOfWeek();
        int dayOfMonth = today.getDayOfMonth();
        Month month = today.getMonth();
        int year = today.getYear();
```

```
        System.out.println("Today is " + dayOfWeek + ", " + month + " " + dayOfMonth + ", " + year);

        // 8. Real-world Examples
        System.out.println("\n== Real-world Examples ==");

        // Calculate age precisely
        LocalDate birthDate = LocalDate.of(1990, 5, 15);
        long ageInYears = ChronoUnit.YEARS.between(birthDate, today);
        System.out.println("Age: " + ageInYears + " years");

        // Next birthday
        LocalDate nextBirthday = birthDate.withYear(today.getYear());
        if (nextBirthday.isBefore(today) || nextBirthday.isEqual(today)) {
            nextBirthday = nextBirthday.plusYears(1);
        }
        Period untilBirthday = Period.between(today, nextBirthday);
        System.out.println("Next birthday in: " +
                           untilBirthday.getMonths() + " months and " +
                           untilBirthday.getDays() + " days");

        // Business days calculation
        LocalDate startDate = LocalDate.of(2023, 1, 1);
        LocalDate endDate = LocalDate.of(2023, 1, 31);
        long totalDays = ChronoUnit.DAYS.between(startDate, endDate) + 1; // +1 to include end date

        // Count business days (excluding weekends)
        long businessDays = startDate.datesUntil(endDate.plusDays(1))
                .filter(d -> d.getDayOfWeek() != DayOfWeek.SATURDAY &&
                             d.getDayOfWeek() != DayOfWeek.SUNDAY)
                .count();

        System.out.println("Total days in January 2023: " + totalDays);
        System.out.println("Business days in January 2023: " + businessDays);

        // Meeting scheduler across time zones
        LocalDateTime meeting = LocalDateTime.of(2023, 3, 15, 14, 0); // 2 PM
        ZoneId londonZone = ZoneId.of("Europe/London");
        ZoneId newYorkZone2 = ZoneId.of("America/New_York");
        ZoneId tokyoZone2 = ZoneId.of("Asia/Tokyo");

        ZonedDateTime londonMeeting = ZonedDateTime.of(meeting, londonZone);
        ZonedDateTime newYorkMeeting = londonMeeting.withZoneSameInstant(newYorkZone2);
        ZonedDateTime tokyoMeeting = londonMeeting.withZoneSameInstant(tokyoZone2);

        System.out.println("\nInternational Meeting Times:");
        System.out.println("London: " + londonMeeting.format(DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm")));
        System.out.println("New York: " + newYorkMeeting.format(DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm")));
        System.out.println("Tokyo: " + tokyoMeeting.format(DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm")));
    }
}
```

Key Classes in the Date/Time API:

1. **LocalDate**: Represents a date without time or time zone (year, month, day)
2. **LocalTime**: Represents a time without date or time zone (hour, minute, second)
3. **LocalDateTime**: Represents a date and time without time zone
4. **ZonedDateTime**: Represents a date and time with time zone
5. **Instant**: Represents a point in time (UTC)
6. **Duration**: Represents a time-based amount (seconds, nanoseconds)
7. **Period**: Represents a date-based amount (years, months, days)
8. **DateTimeFormatter**: For formatting and parsing date-time objects

Common Pitfalls:

- **Immutability Misunderstanding**: All java.time objects are immutable. Methods like `plusDays()` return new objects, they don't modify the original.
- **Time Zone Confusion**: Be explicit about time zones when dealing with date-times that need time zone context.
- **Month Indexing**: Unlike the old API, months in java.time are 1-based (January is 1, not 0).
- **Instant vs. LocalDateTime**: Understand when to use each one. Instant is for machine timestamps, LocalDateTime is for human date-time representations.

```
// INCORRECT: Ignoring the returned value
LocalDate date = LocalDate.of(2023, 1, 1);
date.plusDays(1); // This doesn't modify date!
System.out.println(date); // Still prints 2023-01-01

// CORRECT: Assign the result to a variable
LocalDate date = LocalDate.of(2023, 1, 1);
LocalDate tomorrow = date.plusDays(1);
```

```
System.out.println(tomorrow); // Prints 2023-01-02

// INCORRECT: Forgetting time zones when needed
LocalDateTime localMeeting = LocalDateTime.of(2023, 3, 15, 14, 0);
// Converting to another time zone without proper context
LocalDateTime incorrectNewYorkMeeting = localMeeting.minusHours(5); // Wrong approach!

// CORRECT: Use ZonedDateTime for timezone conversions
ZonedDateTime londonMeeting = ZonedDateTime.of(localMeeting, ZoneId.of("Europe/London"));
ZonedDateTime newYorkMeeting = londonMeeting.withZoneSameInstant(ZoneId.of("America/New_York"));
```

CompletableFuture

What and Why: CompletableFuture provides a way to write asynchronous, non-blocking code in Java. It extends the Future interface with methods to check if a computation is complete, get the result, and chain asynchronous operations together in a fluent way. This was a significant upgrade from the older Future API, which was limited in composability.

Real-world Use Cases:

- Parallel API calls in web applications
- Background processing in UIs
- Orchestrating multiple independent operations
- Implementing timeout and fallback mechanisms
- Build pipelines of asynchronous operations

Before (Java 7) / After (Java 8) Comparison:

Before (Java 7 with ExecutorService and Future):

```
// Create an ExecutorService
ExecutorService executor = Executors.newFixedThreadPool(5);

// Submit a task that returns a result
Future<String> future = executor.submit(() -> {
    Thread.sleep(1000);
    return "Result of the asynchronous computation";
});

// Try to get the result, blocking until it's available
try {
    String result = future.get(2, TimeUnit.SECONDS);
    System.out.println(result);

    // If we want to do something with the result, we need to do it here
    String transformed = result.toUpperCase();
    System.out.println(transformed);
} catch (InterruptedException | ExecutionException | TimeoutException e) {
    e.printStackTrace();
}

// Shut down the executor
executor.shutdown();
```

After (Java 8 with CompletableFuture):

```
// Create a CompletableFuture
CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {
    try {
        Thread.sleep(1000);
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }
    return "Result of the asynchronous computation";
});

// Chain operations to be performed when the result is available
future.thenApply(String::toUpperCase)
      .thenAccept(System.out::println)
      .exceptionally(ex -> {
          System.err.println("Error: " + ex.getMessage());
          return null;
      });

// No explicit blocking, but we can wait for completion if needed
future.join();
```

Comprehensive Example:

```
import java.util.Arrays;
import java.util.List;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.ThreadLocalRandom;
import java.util.concurrent.TimeUnit;
import java.util.stream.Collectors;

public class CompletableFutureExample {
    public static void main(String[] args) {
        // For executing our CompletableFutures
        ExecutorService executor = Executors.newFixedThreadPool(5);

        try {
            // 1. Creating CompletableFutures
            System.out.println("== Creating CompletableFutures ==");

            // With runAsync (no result)
            CompletableFuture<Void> runFuture = CompletableFuture.runAsync(() -> {
                System.out.println("Running in a separate thread");
                sleep(500);
                System.out.println("Completed running");
            }, executor);

            // With supplyAsync (returns a result)
            CompletableFuture<String> supplyFuture = CompletableFuture.supplyAsync(() -> {
                System.out.println("Supplying a result from a separate thread");
                sleep(700);
                return "Hello, CompletableFuture!";
            }, executor);

            // Wait for both to complete
            CompletableFuture.allOf(runFuture, supplyFuture).join();

            // Get the result from supplyFuture
            System.out.println("Result from supplyFuture: " + supplyFuture.get());

            // 2. Transforming and acting on results
            System.out.println("\n== Transforming Results ==");

            // thenApply - transform the result
            CompletableFuture<String> transformedFuture = CompletableFuture
                .supplyAsync(() -> "hello", executor)
                .thenApply(s -> s + " world")
                .thenApply(String::toUpperCase);

            System.out.println("Transformed result: " + transformedFuture.get());

            // thenAccept - consume the result without returning anything
            CompletableFuture<Void> consumedFuture = CompletableFuture
                .supplyAsync(() -> "Hello, Consumer!", executor)
                .thenAccept(s -> System.out.println("Consumed: " + s));

            // thenRun - run an action after completion, ignoring the result
            CompletableFuture<Void> runAfterFuture = CompletableFuture
                .supplyAsync(() -> "Hello, Runner!", executor)
                .thenRun(() -> System.out.println("Processing complete!"));

            // Wait for both to complete
            CompletableFuture.allOf(consumedFuture, runAfterFuture).join();

            // 3. Chaining asynchronous operations
            System.out.println("\n== Chaining Asynchronous Operations ==");

            // thenApplyAsync - transform asynchronously
            CompletableFuture<String> asyncTransformedFuture = CompletableFuture
                .supplyAsync(() -> {
                    System.out.println("First stage: " + Thread.currentThread().getName());
                    sleep(500);
                    return "First result";
                }, executor)
                .thenApplyAsync(s -> {
                    System.out.println("Second stage: " + Thread.currentThread().getName());
                    sleep(500);
                    return s + " -> Second result";
                }, executor);
```

```
System.out.println("Async transformed result: " + asyncTransformedFuture.get());

// thenComposeAsync - flatmap to another CompletableFuture
CompletableFuture<String> composedFuture = CompletableFuture
    .supplyAsync(() -> {
        System.out.println("Computing first value...");
        sleep(500);
        return "Hello";
    }, executor)
    .thenComposeAsync(s -> CompletableFuture.supplyAsync(() -> {
        System.out.println("Computing second value based on: " + s);
        sleep(500);
        return s + ", Composed World!";
    }, executor), executor);

System.out.println("Composed result: " + composedFuture.get());

// 4. Combining multiple futures
System.out.println("\n== Combining Multiple Futures ==");

// thenCombine - combine two independent futures
CompletableFuture<String> future1 = CompletableFuture
    .supplyAsync(() -> {
        System.out.println("Future 1 executing...");
        sleep(500);
        return "Hello";
    }, executor);

CompletableFuture<String> future2 = CompletableFuture
    .supplyAsync(() -> {
        System.out.println("Future 2 executing...");
        sleep(700);
        return "World";
    }, executor);

CompletableFuture<String> combinedFuture = future1
    .thenCombine(future2, (result1, result2) -> result1 + ", " + result2 + "!");

System.out.println("Combined result: " + combinedFuture.get());

// allOf - wait for all futures to complete
List<String> dataIds = Arrays.asList("1", "2", "3", "4", "5");

List<CompletableFuture<String>> futures = dataIds.stream()
    .map(id -> CompletableFuture.supplyAsync(() -> {
        System.out.println("Fetching data for ID: " + id);
        sleep(300); // Simulate network call
        return "Data_" + id;
    }, executor))
    .collect(Collectors.toList());

// Create a single future that completes when all futures complete
CompletableFuture<Void> allFutures = CompletableFuture.allOf(
    futures.toArray(new CompletableFuture[0])
);

// When all futures complete, collect all results
CompletableFuture<List<String>> allResults = allFutures.thenApply(v ->
    futures.stream()
        .map(CompletableFuture::join) // Safe to call join() here
        .collect(Collectors.toList())
);

System.out.println("All results: " + allResults.get());

// anyOf - complete when any future completes
CompletableFuture<Object> anyFuture = CompletableFuture.anyOf(
    CompletableFuture.supplyAsync(() -> {
        sleep(500);
        return "Fast Result";
    }, executor),
    CompletableFuture.supplyAsync(() -> {
        sleep(300); // This should complete first
        return "Faster Result";
    }, executor),
    CompletableFuture.supplyAsync(() -> {
        sleep(700);
        return "Slow Result";
    }, executor)
);
```

```
System.out.println("First completed result: " + anyFuture.get());

// 5. Exception Handling
System.out.println("\n== Exception Handling ==");

// exceptionally - recover from exceptions
CompletableFuture<String> recoveringFuture = CompletableFuture
    .supplyAsync(() -> {
        if (Math.random() > 0.5) {
            throw new RuntimeException("Something went wrong!");
        }
        return "Success";
    }, executor)
    .exceptionally(ex -> {
        System.out.println("Recovered from: " + ex.getMessage());
        return "Fallback Value";
    });

System.out.println("Result with recovery: " + recoveringFuture.get());

// handle - process both successful and failed outcomes
CompletableFuture<String> handledFuture = CompletableFuture
    .supplyAsync(() -> {
        if (Math.random() > 0.5) {
            throw new RuntimeException("Error occurred!");
        }
        return "Direct Success";
    }, executor)
    .handle((result, ex) -> {
        if (ex != null) {
            System.out.println("Handled exception: " + ex.getMessage());
            return "Handled Fallback";
        } else {
            return result;
        }
    });

System.out.println("Result after handling: " + handledFuture.get());

// whenComplete - take action on completion without changing the result
CompletableFuture<String> completionActionFuture = CompletableFuture
    .supplyAsync(() -> {
        if (Math.random() > 0.5) {
            throw new RuntimeException("Random failure!");
        }
        return "Final Success";
    }, executor)
    .whenComplete((result, ex) -> {
        // This doesn't affect the result, just performs an action
        if (ex != null) {
            System.out.println("Operation failed with: " + ex.getMessage());
        } else {
            System.out.println("Operation completed with: " + result);
        }
    })
    .exceptionally(ex -> "Recovered from failure");

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

// 6. Timeouts
System.out.println("\n== Adding Timeouts ==");

// Using the built-in timeout mechanisms
CompletableFuture<String> timeoutFuture = CompletableFuture
    .supplyAsync(() -> {
        System.out.println("Starting long operation...");
        sleep(2000); // Long operation
        return "Finally completed!";
    }, executor)
    .orTimeout(1, TimeUnit.SECONDS)
    .exceptionally(ex -> {
        if (ex instanceof java.util.concurrent.TimeoutException) {
            return "Operation timed out!";
        }
        return "Other error occurred: " + ex.getMessage();
    });

try {
    System.out.println("Result with timeout: " + timeoutFuture.get());
} catch (Exception e) {
    System.out.println("Error: " + e.getMessage());
```

```
            }

            // 7. Real-world example: Fetching user profile with fallback
            System.out.println("\n== Real-world Example ==");

            CompletableFuture<UserProfile> userProfileFuture = getUserDetails(123)
                .thenCompose(user -> {
                    // Concurrent fetches of additional data
                    CompletableFuture<List<String>> friendsFuture = getFriendsList(user.getId());
                    CompletableFuture<List<String>> photosFuture = getRecentPhotos(user.getId());

                    // Combine the results
                    return friendsFuture.thenCombine(photosFuture, (friends, photos) ->
                        new UserProfile(user, friends, photos)
                    );
                })
                .completeOnTimeout(
                    new UserProfile(new User(123, "Guest User"), List.of(), List.of()),
                    2,
                    TimeUnit.SECONDS
                );

            // Process the profile when ready
            UserProfile profile = userProfileFuture.get();
            System.out.println("User profile: " + profile);

        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            // Ensure the executor is shut down
            executor.shutdown();
        }
    }

    // Utility method to simulate a delay
    private static void sleep(long millis) {
        try {
            Thread.sleep(millis);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    // Simulated API methods for the real-world example
    private static CompletableFuture<User> getUserDetails(int userId) {
        return CompletableFuture.supplyAsync(() -> {
            System.out.println("Fetching user details for: " + userId);
            sleep(500); // Simulate network call
            return new User(userId, "User_" + userId);
        });
    }

    private static CompletableFuture<List<String>> getFriendsList(int userId) {
        return CompletableFuture.supplyAsync(() -> {
            System.out.println("Fetching friends list for: " + userId);
            sleep(700); // Simulate network call
            return Arrays.asList("Friend1", "Friend2", "Friend3");
        });
    }

    private static CompletableFuture<List<String>> getRecentPhotos(int userId) {
        return CompletableFuture.supplyAsync(() -> {
            System.out.println("Fetching recent photos for: " + userId);
            sleep(600); // Simulate network call
            return Arrays.asList("Photo1", "Photo2");
        });
    }

    // Helper classes for the real-world example
    static class User {
        private final int id;
        private final String name;

        User(int id, String name) {
            this.id = id;
            this.name = name;
        }

        public int getId() { return id; }
        public String getName() { return name; }
```

```
        @Override
        public String toString() {
            return "User{id=" + id + ", name='" + name + "'}";
        }
    }

    static class UserProfile {
        private final User user;
        private final List<String> friends;
        private final List<String> recentPhotos;

        UserProfile(User user, List<String> friends, List<String> recentPhotos) {
            this.user = user;
            this.friends = friends;
            this.recentPhotos = recentPhotos;
        }

        @Override
        public String toString() {
            return "UserProfile{user=" + user +
                   ", friends=" + friends +
                   ", recentPhotos=" + recentPhotos + "}";
        }
    }
}
```

Key CompletableFuture Methods:

1. **Creation**:
   - `CompletableFuture.runAsync(Runnable)`: Executes the Runnable asynchronously, returning a future that completes when the Runnable completes
   - `CompletableFuture.supplyAsync(Supplier)`: Executes the Supplier asynchronously, returning a future with the supplier's result
2. **Transformation**:
   - `thenApply(Function)`: Transforms the result synchronously
   - `thenApplyAsync(Function)`: Transforms the result asynchronously
   - `thenCompose(Function)`: Chains with another CompletableFuture (flatMap)
3. **Consuming results**:
   - `thenAccept(Consumer)`: Consumes the result without returning anything
   - `thenRun(Runnable)`: Runs an action after completion, ignoring the result
4. **Combining futures**:
   - `thenCombine(CompletableFuture, BiFunction)`: Combines two independent futures when both complete
   - `allOf(CompletableFuture...)`: Returns a future that completes when all given futures complete
   - `anyOf(CompletableFuture...)`: Returns a future that completes when any of the given futures completes
5. **Error handling**:
   - `exceptionally(Function)`: Handles exceptions and provides a fallback value
   - `handle(BiFunction)`: Handles both the result and exception
   - `whenComplete(BiConsumer)`: Performs an action when the future completes, with access to both result and exception
6. **Timing**:
   - `orTimeout(long, TimeUnit)`: Completes exceptionally if not completed within the timeout
   - `completeOnTimeout(T, long, TimeUnit)`: Completes with the given value if not completed within the timeout

Common Pitfalls:

- **Ignoring Exceptions**: Not handling exceptions properly. Use `exceptionally()`, `handle()`, or `whenComplete()`.
- **Blocking Unintentionally**: Calling `get()` or `join()` on a CompletableFuture causes the current thread to block.
- **Executor Confusion**: By default, many CompletableFuture methods use the common ForkJoinPool, which might not be suitable for blocking operations.
- **Not Completing Manually**: When creating a CompletableFuture directly, remember to complete it with `complete()` or `completeExceptionally()`.

```
// INCORRECT: Not handling exceptions
CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {
    if (Math.random() > 0.5) {
        throw new RuntimeException("Error");
    }
    return "Success";
});
// No exception handling - the exception will be propagated when future.get() is called

// CORRECT: Handle exceptions
```

```
CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {
    if (Math.random() > 0.5) {
        throw new RuntimeException("Error");
    }
    return "Success";
}).exceptionally(ex -> "Fallback");

// INCORRECT: Blocking the current thread unnecessarily
CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {
    // Long operation
    return "Result";
});
String result = future.get(); // Blocks the current thread!

// BETTER: Use callbacks instead of blocking
CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {
    // Long operation
    return "Result";
});
future.thenAccept(result -> {
    // Process the result asynchronously
});

// INCORRECT: Using the common pool for blocking operations
CompletableFuture.supplyAsync(() -> {
    try {
        Thread.sleep(10000); // Blocks a thread from common pool for 10 seconds!
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }
    return "Result";
});

// CORRECT: Provide a dedicated executor for blocking operations
ExecutorService executor = Executors.newFixedThreadPool(10);
```

Java 9

Java 9, released in September 2017, introduced several significant features, with the module system (Project Jigsaw) being the most substantial architectural change.

Module System

What and Why: The Java Platform Module System (JPMS) was introduced to solve the problems of strong encapsulation, reliable configuration, and scalable Java platforms. It allows Java applications to be organized as modules, improving maintainability and performance.

Real-world Use Cases:

- Building large-scale applications with clear boundaries
- Creating more secure applications by explicitly controlling what's exposed
- Improved performance through more efficient class loading
- Custom runtime images containing only needed modules

Module Basics:

A module in Java 9+ is defined by a `module-info.java` file in the root directory:

```
// file: module-info.java
module com.example.mymodule {
    // Modules this module depends on
    requires java.base;          // Implicit dependency, could be omitted
    requires java.sql;           // This module depends on java.sql

    // Packages this module exposes to other modules
    exports com.example.api;             // Exports to all modules
    exports com.example.internal to      // Exports only to specific modules
        com.example.othermodule;

    // Services this module uses
    uses com.example.spi.MyService;

    // Services this module provides
    provides com.example.spi.MyService
        with com.example.impl.MyServiceImpl;
}
```

Example Project Structure with Modules:

```
my-application/
├── main-app/
│   ├── src/
│   │   ├── main/
│   │   │   ├── java/
│   │   │   │   ├── module-info.java
│   │   │   │   └── com/
│   │   │   │       └── example/
│   │   │   │           └── app/
│   │   │   │               └── Main.java
│   │   │   └── resources/
│   │   └── test/
│   └── build.gradle
├── user-module/
│   ├── src/
│   │   ├── main/
│   │   │   ├── java/
│   │   │   │   ├── module-info.java
│   │   │   │   └── com/
│   │   │   │       └── example/
│   │   │   │           └── user/
│   │   │   │               ├── User.java
│   │   │   │               └── UserService.java
│   │   │   └── resources/
│   │   └── test/
│   └── build.gradle
└── settings.gradle
```

Comprehensive Example:

Let's create a simple modular application with three modules:

1. A service module defining an interface
2. A provider module implementing the interface
3. A consumer module using the service

```
// Module 1: Service Definition (com.example.service)
// file: service-module/src/main/java/module-info.java
module com.example.service {
    exports com.example.service.api;
}

// file: service-module/src/main/java/com/example/service/api/MessageService.java
package com.example.service.api;

public interface MessageService {
    String getMessage();
}

// Module 2: Service Implementation (com.example.provider)
// file: provider-module/src/main/java/module-info.java
module com.example.provider {
    requires com.example.service;
    provides com.example.service.api.MessageService
        with com.example.provider.impl.SimpleMessageService;
}

// file: provider-module/src/main/java/com/example/provider/impl/SimpleMessageService.java
package com.example.provider.impl;

import com.example.service.api.MessageService;

public class SimpleMessageService implements MessageService {
    @Override
    public String getMessage() {
        return "Hello from SimpleMessageService!";
    }
}

// Module 3: Consumer Application (com.example.consumer)
// file: consumer-module/src/main/java/module-info.java
module com.example.consumer {
    requires com.example.service;
    uses com.example.service.api.MessageService;
}

// file: consumer-module/src/main/java/com/example/consumer/app/ConsumerApp.java
package com.example.consumer.app;
```

```
import com.example.service.api.MessageService;
import java.util.ServiceLoader;

public class ConsumerApp {
    public static void main(String[] args) {
        // Use ServiceLoader to find implementations of MessageService
        ServiceLoader<MessageService> serviceLoader = ServiceLoader.load(MessageService.class);

        // Print messages from all available implementations
        serviceLoader.forEach(service ->
            System.out.println("Received: " + service.getMessage())
        );

        // If no service is found
        if (!serviceLoader.iterator().hasNext()) {
            System.out.println("No MessageService implementation found!");
        }
    }
}
```

Building and Running a Modular Application:

To compile and run the modular application, use the following commands:

```
# Compile the service module
javac -d mods/com.example.service \
    service-module/src/main/java/module-info.java \
    service-module/src/main/java/com/example/service/api/MessageService.java

# Compile the provider module
javac -d mods/com.example.provider \
    --module-path mods \
    provider-module/src/main/java/module-info.java \
    provider-module/src/main/java/com/example/provider/impl/SimpleMessageService.java

# Compile the consumer module
javac -d mods/com.example.consumer \
    --module-path mods \
    consumer-module/src/main/java/module-info.java \
    consumer-module/src/main/java/com/example/consumer/app/ConsumerApp.java

# Run the application
java --module-path mods -m com.example.consumer/com.example.consumer.app.ConsumerApp
```

Creating Custom Runtime Images with jlink:

One of the benefits of the module system is the ability to create custom runtime images with just the modules your application needs:

```
jlink --module-path $JAVA_HOME/jmods:mods \
      --add-modules com.example.consumer \
      --launcher run=com.example.consumer/com.example.consumer.app.ConsumerApp \
      --output myapp
```

This creates a directory `myapp` containing a minimal Java runtime with just the modules needed to run your application. You can then run the application directly:

```
myapp/bin/run
```

Migration and Backwards Compatibility:

For existing applications, Java 9 introduced the concept of the unnamed module and automatic modules to ease migration:

- **Unnamed Module**: All classes on the classpath that don't belong to an explicit module
- **Automatic Modules**: JAR files placed on the module path automatically become modules, with their name derived from the JAR filename

Common Pitfalls:

- **Reflection Access Issues**: Reflection might not work if the accessed class is not exported.
- **Split Package Problem**: The same package cannot be in multiple modules.
- **Circular Dependencies**: Module dependencies must be acyclic.
- **Misconception about Visibility**: `exports` controls access at compile-time AND runtime.

```
// INCORRECT: Trying to access an unexported package via reflection
// Module A does not export com.example.internal
Class<?> clazz = Class.forName("com.example.internal.SomeClass");
clazz.getDeclaredConstructor().newInstance(); // IllegalAccessException

// SOLUTION: Use --add-opens or make sure the package is exported

// INCORRECT: Circular dependency
// module-info.java of module A
module com.example.a {
    requires com.example.b;
    exports com.example.a.api;
}

// module-info.java of module B
module com.example.b {
    requires com.example.a; // This creates a circular dependency!
    exports com.example.b.api;
}

// SOLUTION: Refactor to eliminate the circular dependency, possibly
// by introducing a third module that both depend on
```

Collection Factory Methods

What and Why: Java 9 introduced convenient factory methods for creating small, immutable collections. These methods provide a more concise way to create collections compared to the traditional approach.

Before (Java 8) / After (Java 9) Comparison:

Before (Java 8):

```
// Creating an immutable List
List<String> list = Collections.unmodifiableList(Arrays.asList("one", "two", "three"));

// Creating an immutable Set
Set<String> set = Collections.unmodifiableSet(new HashSet<>(Arrays.asList("one", "two", "three")));

// Creating an immutable Map
Map<String, Integer> map = Collections.unmodifiableMap(new HashMap<String, Integer>() {{
    put("one", 1);
    put("two", 2);
    put("three", 3);
}});
```

After (Java 9):

```
// Creating an immutable List
List<String> list = List.of("one", "two", "three");

// Creating an immutable Set
Set<String> set = Set.of("one", "two", "three");

// Creating an immutable Map
Map<String, Integer> map = Map.of(
    "one", 1,
    "two", 2,
    "three", 3
);
// For more entries, use Map.ofEntries
Map<String, Integer> largeMap = Map.ofEntries(
    Map.entry("one", 1),
    Map.entry("two", 2),
    Map.entry("three", 3),
    Map.entry("four", 4)
    // Can add more entries
);
```

Comprehensive Example:

```
import java.util.*;

public class CollectionFactoryMethodsExample {
```

```
    public static void main(String[] args) {
        // 1. Creating immutable Lists
        System.out.println("=== Immutable Lists ===");

        // Empty List
        List<String> emptyList = List.of();
        System.out.println("Empty list: " + emptyList);

        // Single element List
        List<String> singletonList = List.of("only one");
        System.out.println("Singleton list: " + singletonList);

        // Multiple elements List
        List<String> namesList = List.of("Alice", "Bob", "Charlie", "David");
        System.out.println("Names list: " + namesList);

        try {
            // Lists created with List.of() are immutable
            namesList.add("Eve"); // This will throw UnsupportedOperationException
        } catch (UnsupportedOperationException e) {
            System.out.println("Cannot modify an immutable list!");
        }

        try {
            // Also can't set elements
            namesList.set(0, "Anna"); // This will throw UnsupportedOperationException
        } catch (UnsupportedOperationException e) {
            System.out.println("Cannot replace elements in an immutable list!");
        }

        try {
            // Null elements are not allowed
            List.of("Valid", null); // This will throw NullPointerException
        } catch (NullPointerException e) {
            System.out.println("Cannot include null elements in factory-created collections!");
        }

        // 2. Creating immutable Sets
        System.out.println("\n=== Immutable Sets ===");

        // Empty Set
        Set<String> emptySet = Set.of();
        System.out.println("Empty set: " + emptySet);

        // Set with elements
        Set<String> colorSet = Set.of("Red", "Green", "Blue");
        System.out.println("Color set: " + colorSet);

        try {
            // Duplicate elements are not allowed in Sets
            Set.of("Red", "Green", "Red"); // This will throw IllegalArgumentException
        } catch (IllegalArgumentException e) {
            System.out.println("Cannot include duplicate elements in a set!");
        }

        // 3. Creating immutable Maps
        System.out.println("\n=== Immutable Maps ===");

        // Empty Map
        Map<String, Integer> emptyMap = Map.of();
        System.out.println("Empty map: " + emptyMap);

        // Map with key-value pairs (up to 10 pairs with Map.of)
        Map<String, Integer> dayMap = Map.of(
            "Monday", 1,
            "Wednesday", 3,
            "Friday", 5
        );
        System.out.println("Day map: " + dayMap);

        // For more than 10 pairs, use Map.ofEntries
        Map<String, Integer> monthMap = Map.ofEntries(
            Map.entry("January", 1),
            Map.entry("February", 2),
            Map.entry("March", 3),
            Map.entry("April", 4),
            Map.entry("May", 5),
            Map.entry("June", 6),
            Map.entry("July", 7),
            Map.entry("August", 8),
            Map.entry("September", 9),
```

```
            Map.entry("October", 10),
            Map.entry("November", 11),
            Map.entry("December", 12)
        );
        System.out.println("Month map: " + monthMap);

        try {
            // Maps created with factory methods are immutable
            dayMap.put("Sunday", 0); // This will throw UnsupportedOperationException
        } catch (UnsupportedOperationException e) {
            System.out.println("Cannot modify an immutable map!");
        }

        try {
            // Duplicate keys are not allowed
            Map.of("Key", 1, "Key", 2); // This will throw IllegalArgumentException
        } catch (IllegalArgumentException e) {
            System.out.println("Cannot include duplicate keys in a map!");
        }

        // 4. Practical examples and use cases
        System.out.println("\n=== Practical Examples ===");

        // Configuration constants
        final Map<String, String> CONFIG = Map.of(
            "host", "localhost",
            "port", "8080",
            "user", "admin",
            "timeout", "30000"
        );
        System.out.println("Application config: " + CONFIG);

        // Fixed set of options
        final Set<String> VALID_STATES = Set.of(
            "PENDING", "PROCESSING", "COMPLETED", "FAILED"
        );

        String currentState = "PROCESSING";
        System.out.println("Is " + currentState + " a valid state? " +
                           VALID_STATES.contains(currentState));

        // Method returning immutable lists - no defensive copying needed
        List<String> availableOptions = getOptions("USER");
        System.out.println("Available options: " + availableOptions);

        // Using in stream operations
        List<Integer> result = List.of(1, 2, 3, 4, 5)
                                   .stream()
                                   .filter(n -> n % 2 == 0)
                                   .map(n -> n * n)
                                   .collect(Collectors.toList());
        System.out.println("Processed numbers: " + result);
    }

    // Method returning an immutable collection
    private static List<String> getOptions(String role) {
        // Return different immutable lists based on role
        if ("ADMIN".equals(role)) {
            return List.of("Create", "Read", "Update", "Delete");
        } else if ("USER".equals(role)) {
            return List.of("Read", "Comment");
        } else {
            return List.of(); // Empty immutable list
        }
    }
}
```

Common Pitfalls:

- **Nulls Not Allowed**: Unlike traditional collections, factory methods do not allow null elements.
- **Immutability**: Collections created with factory methods are immutable.
- **Duplicates in Sets**: `Set.of()` rejects duplicate elements (unlike new HashSet(Arrays.asList(...))).
- **Maximum Entries for Map.of()**: Limited to 10 key-value pairs. Use `Map.ofEntries()` for more.

```
// INCORRECT: Adding nulls
List<String> list = List.of("one", null); // NullPointerException

// INCORRECT: Adding duplicates to Sets
```

```
Set<String> set = Set.of("one", "one"); // IllegalArgumentException

// INCORRECT: Trying to modify
List<String> list = List.of("one", "two");
list.add("three"); // UnsupportedOperationException

// INCORRECT: Using more than 10 pairs with Map.of
Map<String, Integer> map = Map.of(
    "k1", 1, "k2", 2, "k3", 3, "k4", 4, "k5", 5,
    "k6", 6, "k7", 7, "k8", 8, "k9", 9, "k10", 10,
    "k11", 11 // Error: Too many arguments
);

// CORRECT: Using Map.ofEntries for more than 10 pairs
Map<String, Integer> map = Map.ofEntries(
    Map.entry("k1", 1),
    Map.entry("k2", 2),
    // ... as many as you need
    Map.entry("k20", 20)
);
```

Stream API Improvements

What and Why: Java 9 enhanced the Stream API with several new methods, making certain operations more convenient and expressive.

New Stream Methods in Java 9:

1. `takeWhile(Predicate)`: Takes elements from the stream while the predicate is true, then stops.
2. `dropWhile(Predicate)`: Drops elements from the stream while the predicate is true, then takes the rest.
3. `iterate(seed, hasNext, next)`: Creates a finite stream using the provided predicate to determine when to stop.
4. `ofNullable(T t)`: Creates a stream of 0 or 1 elements, with 0 if the element is null.

Example:

```
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.IntStream;
import java.util.stream.Stream;

public class StreamImprovementsExample {
    public static void main(String[] args) {
        // 1. takeWhile and dropWhile
        System.out.println("=== takeWhile and dropWhile ===");

        List<Integer> numbers = List.of(2, 4, 6, 8, 9, 10, 12);

        // takeWhile: Take elements while they're even
        List<Integer> takeWhileEven = numbers.stream()
                                        .takeWhile(n -> n % 2 == 0)
                                        .collect(Collectors.toList());
        System.out.println("Take while even: " + takeWhileEven);

        // dropWhile: Drop elements while they're even
        List<Integer> dropWhileEven = numbers.stream()
                                       .dropWhile(n -> n % 2 == 0)
                                       .collect(Collectors.toList());
        System.out.println("Drop while even: " + dropWhileEven);

        // Contrast with filter (which processes ALL elements)
        List<Integer> filtered = numbers.stream()
                                    .filter(n -> n % 2 == 0)
                                    .collect(Collectors.toList());
        System.out.println("Filter even: " + filtered);

        // 2. iterate with predicate (finite streams)
        System.out.println("\n=== Iterate with Predicate ===");

        // In Java 8: infinite stream, must limit explicitly
        List<Integer> java8Way = Stream.iterate(1, n -> n + 1)
                                   .limit(10)
                                   .collect(Collectors.toList());
        System.out.println("Java 8 way: " + java8Way);

        // In Java 9: finite stream with predicate
        List<Integer> java9Way = Stream.iterate(1, n -> n <= 10, n -> n + 1)
                                   .collect(Collectors.toList());
        System.out.println("Java 9 way: " + java9Way);
```

```
        // Example: Generate Fibonacci sequence up to 100
        List<Integer> fibonacci = Stream.iterate(
                                    new int[]{0, 1},
                                    arr -> arr[1] <= 100,
                                    arr -> new int[]{arr[1], arr[0] + arr[1]})
                                  .map(arr -> arr[0])
                                  .collect(Collectors.toList());
        System.out.println("Fibonacci sequence up to 100: " + fibonacci);

        // 3. ofNullable
        System.out.println("\n=== ofNullable ===");

        // Java 8 way: Need to check and create empty or singleton stream
        String nullableValue = Math.random() > 0.5 ? "Not Null" : null;
        Stream<String> java8Nullable = nullableValue == null ?
                                       Stream.empty() :
                                       Stream.of(nullableValue);

        // Java 9 way: Simple ofNullable
        Stream<String> java9Nullable = Stream.ofNullable(nullableValue);

        // Usage in flatMap
        List<String> possiblyNull = List.of("a", null, "b", null, "c");
        List<String> nonNull = possiblyNull.stream()
                                    .flatMap(Stream::ofNullable)
                                    .collect(Collectors.toList());
        System.out.println("Original with nulls: " + possiblyNull);
        System.out.println("After removing nulls: " + nonNull);

        // 4. Real-world example
        System.out.println("\n=== Real-world Example ===");

        // Log processing: Parse until error encountered
        List<String> logLines = List.of(
            "INFO: System starting",
            "INFO: User logged in",
            "INFO: Processing data",
            "ERROR: Unexpected exception",
            "INFO: Continuing process",
            "INFO: Completed"
        );

        // Get all log lines until first error
        List<String> preErrorLogs = logLines.stream()
                                        .takeWhile(line -> !line.startsWith("ERROR"))
                                        .collect(Collectors.toList());
        System.out.println("Logs until error: " + preErrorLogs);

        // Get all log lines after first error
        List<String> postErrorLogs = logLines.stream()
                                         .dropWhile(line -> !line.startsWith("ERROR"))
                                         .skip(1) // Skip the error itself
                                         .collect(Collectors.toList());
        System.out.println("Logs after error: " + postErrorLogs);

        // 5. Performance example with takeWhile
        System.out.println("\n=== Performance Advantage ===");

        // Find the first 5 even numbers under 1000

        // Using limit and filter (processes many elements)
        long startFilter = System.nanoTime();
        List<Integer> evenWithFilter = IntStream.range(0, 1000)
                                           .filter(n -> {
                                               boolean isEven = n % 2 == 0;
                                               // In real code, this could be an expensive operation
                                               return isEven;
                                           })
                                           .limit(5)
                                           .boxed()
                                           .collect(Collectors.toList());
        long endFilter = System.nanoTime();

        // Using iterate with predicate (only processes necessary elements)
        long startIterate = System.nanoTime();
        List<Integer> evenWithIterate = Stream.iterate(
                                            0,
                                            n -> n < 1000,
                                            n -> n + 2)
                                        .limit(5)
                                        .collect(Collectors.toList());
```

```
        long endIterate = System.nanoTime();

        System.out.println("First 5 even numbers (filter): " + evenWithFilter);
        System.out.println("First 5 even numbers (iterate): " + evenWithIterate);
        System.out.println("Filter time: " + (endFilter - startFilter) / 1000 + " µs");
        System.out.println("Iterate time: " + (endIterate - startIterate) / 1000 + " µs");
    }
}
```

Common Pitfalls:

- **Ordered vs. Unordered Streams**: `takeWhile` and `dropWhile` are sensitive to the stream order.
- **State in Predicates**: Be careful about maintaining state in predicates used with `takeWhile` and `dropWhile`.
- **Infinite Streams**: The old `iterate` method produces infinite streams, while the new three-arg version can produce finite streams.

```
// INCORRECT UNDERSTANDING: takeWhile with unordered stream
Set<Integer> unorderedSet = new HashSet<>(List.of(2, 4, 6, 8, 9, 10, 12));
List<Integer> result = unorderedSet.stream()
                                .takeWhile(n -> n % 2 == 0)
                                .collect(Collectors.toList());
// Result is unpredictable because HashSet doesn't guarantee order

// CORRECT: Use with ordered sources or first make the stream ordered
List<Integer> result = unorderedSet.stream()
                                .sorted()
                                .takeWhile(n -> n % 2 == 0)
                                .collect(Collectors.toList());
```

Private Interface Methods

What and Why: Java 9 allows private methods in interfaces, extending the default method functionality introduced in Java 8. This enables code reuse within an interface, keeping the implementation details hidden from implementing classes.

Before (Java 8) / After (Java 9) Comparison:

Before (Java 8):

```
// In Java 8, you could have default methods but couldn't extract common code
public interface Logger {
    default void logInfo(String message) {
        System.out.println("INFO: " + message);
    }

    default void logWarning(String message) {
        System.out.println("WARNING: " + message);
    }

    default void logError(String message) {
        System.out.println("ERROR: " + message);
    }
}
```

After (Java 9):

```
// In Java 9, you can extract common code to private methods
public interface Logger {
    default void logInfo(String message) {
        log("INFO", message);
    }

    default void logWarning(String message) {
        log("WARNING", message);
    }

    default void logError(String message) {
        log("ERROR", message);
    }

    // Private method - only visible within the interface
    private void log(String level, String message) {
        System.out.println(level + ": " + message);
    }
}
```

Comprehensive Example:

```
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;

public class PrivateInterfaceMethodsExample {

    public static void main(String[] args) {
        // Using the Logger interface through different implementations
        Logger consoleLogger = new ConsoleLogger();
        consoleLogger.logInfo("Application started");
        consoleLogger.logWarning("Low memory detected");
        consoleLogger.logError("Database connection failed");

        System.out.println();

        // Another implementation with different behavior
        Logger detailedLogger = new DetailedLogger();
        detailedLogger.logInfo("User logged in");
        detailedLogger.logWarning("Session timeout approaching");
        detailedLogger.logError("Authentication failed");

        System.out.println();

        // Using the enhanced calculator interface
        Calculator calc = new BasicCalculator();
        System.out.println("5 + 3 = " + calc.add(5, 3));
        System.out.println("5 - 3 = " + calc.subtract(5, 3));
        System.out.println("5 * 3 = " + calc.multiply(5, 3));
        System.out.println("5 / 3 = " + calc.divide(5, 3));

        // Using debug mode
        calc.setDebugMode(true);
        System.out.println("\nWith debug mode:");
        System.out.println("5 + 3 = " + calc.add(5, 3));
        System.out.println("5 - 3 = " + calc.subtract(5, 3));
    }

    // Interface with private methods
    interface Logger {
        default void logInfo(String message) {
            log("INFO", message);
        }

        default void logWarning(String message) {
            log("WARNING", message);
        }

        default void logError(String message) {
            log("ERROR", message);
        }

        // Private method for internal code reuse
        private void log(String level, String message) {
            System.out.println(formatMessage(level, message));
        }

        // Another private method
        private String formatMessage(String level, String message) {
            LocalDateTime now = LocalDateTime.now();
            return now.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME)
                   + " [" + level + "] " + message;
        }
    }

    // Simple implementation
    static class ConsoleLogger implements Logger {
        // No need to implement anything, uses the defaults
    }

    // Custom implementation that overrides the default methods
    static class DetailedLogger implements Logger {
        @Override
        public void logInfo(String message) {
            System.out.println("ℹ INFO: " + message);
        }

        @Override
        public void logWarning(String message) {
            System.out.println("⚠ WARNING: " + message);
```

```
        }

        @Override
        public void logError(String message) {
            System.out.println("❌ ERROR: " + message);
        }
    }

    // Another interface with private methods, demonstrating more complex usage
    interface Calculator {
        // Abstract methods that need to be implemented
        double add(double a, double b);
        double subtract(double a, double b);
        double multiply(double a, double b);
        double divide(double a, double b);

        // State that can be controlled
        default void setDebugMode(boolean debug) {
            DebugContext.setDebugMode(debug);
        }

        // Default implementation that uses private methods
        default double power(double base, double exponent) {
            validate(base, exponent);
            double result = Math.pow(base, exponent);
            logOperation("power", base, exponent, result);
            return result;
        }

        // Private static method
        private static void validate(double... values) {
            for (double value : values) {
                if (Double.isNaN(value) || Double.isInfinite(value)) {
                    throw new IllegalArgumentException(
                        "Invalid argument: " + value);
                }
            }
        }

        // Private instance method
        private void logOperation(String operation, double a, double b, double result) {
            if (DebugContext.isDebugMode()) {
                System.out.println("DEBUG: " + operation + "(" + a + ", " + b + ") = " + result);
            }
        }
    }

    // Helper class for storing debug state
    static class DebugContext {
        private static boolean debugMode = false;

        static void setDebugMode(boolean debug) {
            debugMode = debug;
        }

        static boolean isDebugMode() {
            return debugMode;
        }
    }

    // Implementation of the Calculator interface
    static class BasicCalculator implements Calculator {
        @Override
        public double add(double a, double b) {
            double result = a + b;
            // We can't directly call private interface methods here
            if (DebugContext.isDebugMode()) {
                System.out.println("DEBUG: add(" + a + ", " + b + ") = " + result);
            }
            return result;
        }

        @Override
        public double subtract(double a, double b) {
            double result = a - b;
            if (DebugContext.isDebugMode()) {
                System.out.println("DEBUG: subtract(" + a + ", " + b + ") = " + result);
            }
            return result;
        }
```

```
        @Override
        public double multiply(double a, double b) {
            return a * b;
        }

        @Override
        public double divide(double a, double b) {
            if (b == 0) {
                throw new ArithmeticException("Division by zero");
            }
            return a / b;
        }
    }
}
```

Common Pitfalls:

- **Accessibility**: Private interface methods are only accessible within the interface itself.
- **Inheritance**: Unlike default methods, private methods cannot be overridden or directly accessed by implementing classes.
- **Static vs. Instance**: Private methods can be either static or instance methods, but they can only call other private methods of the same kind (static or instance).

```
// INCORRECT: Calling a private interface method from the implementing class
interface Processor {
    default void process(String data) {
        String processed = preprocess(data);
        // Further processing...
    }

    private String preprocess(String data) {
        return data.trim().toLowerCase();
    }
}

class CustomProcessor implements Processor {
    public void customProcess(String data) {
        String preprocessed = preprocess(data); // Compilation error: preprocess has private access
    }
}

// CORRECT: Use the default method that internally calls the private method
class BetterCustomProcessor implements Processor {
    public void customProcess(String data) {
        process(data); // This works because process() is accessible
    }
}
```

JShell

What and Why: JShell is a Read-Eval-Print Loop (REPL) tool introduced in Java 9 that allows developers to execute Java code snippets interactively without the need for a complete program structure. It's designed to make learning and exploration easier.

Real-world Use Cases:

- Learning and exploring Java features
- Quickly testing code snippets
- Prototyping and experimentation
- Teaching and demonstrations
- API exploration

JShell Basic Usage:

Starting JShell:

```
$ jshell
|  Welcome to JShell -- Version 9
|  For an introduction type: /help intro
jshell>
```

Simple expression evaluation:

```
jshell> 2 + 2
$1 ==> 4
```

```
jshell> "Hello, " + "world!"
$2 ==> "Hello, world!"
```

Variable declarations:

```
jshell> int x = 10
x ==> 10

jshell> var y = 20
y ==> 20

jshell> x + y
$3 ==> 30
```

Method declarations:

```
jshell> int add(int a, int b) {
   ...>     return a + b;
   ...> }
|  created method add(int,int)

jshell> add(5, 7)
$4 ==> 12
```

Importing classes:

```
jshell> import java.util.List

jshell> List<String> names = List.of("Alice", "Bob", "Charlie")
names ==> [Alice, Bob, Charlie]

jshell> names.size()
$5 ==> 3
```

JShell Commands:

```
/list         List all snippets you've entered
/vars         List all variables
/methods      List all methods
/imports      List all imports
/edit         Edit a snippet
/drop         Delete a snippet
/save         Save snippets to a file
/open         Load snippets from a file
/exit         Exit JShell
/help         Display help information
```

Example JShell Session:

```
jshell> // Define a Person class
jshell> class Person {
   ...>     private String name;
   ...>     private int age;
   ...>
   ...>     public Person(String name, int age) {
   ...>         this.name = name;
   ...>         this.age = age;
   ...>     }
   ...>
   ...>     public String getName() { return name; }
   ...>     public int getAge() { return age; }
   ...>
   ...>     @Override
   ...>     public String toString() {
   ...>         return "Person{name='" + name + "', age=" + age + "}";
   ...>     }
   ...> }
|  created class Person

jshell> // Create a list of persons
```

```
jshell> import java.util.List
jshell> import java.util.stream.Collectors

jshell> var people = List.of(
   ...>     new Person("Alice", 25),
   ...>     new Person("Bob", 30),
   ...>     new Person("Charlie", 35),
   ...>     new Person("David", 28)
   ...> )
people ==> [Person{name='Alice', age=25}, Person{name='Bob', age=30}, Person{name='Charlie', age=35}, Person{name='David',
age=28}]

jshell> // Filter people above 30
jshell> var above30 = people.stream()
   ...>                     .filter(p -> p.getAge() > 30)
   ...>                     .collect(Collectors.toList())
above30 ==> [Person{name='Charlie', age=35}]

jshell> // Calculate average age
jshell> var avgAge = people.stream()
   ...>                   .mapToInt(Person::getAge)
   ...>                   .average()
   ...>                   .orElse(0)
avgAge ==> 29.5

jshell> // Try out the new Java 9 features
jshell> var namesList = people.stream()
   ...>                      .map(Person::getName)
   ...>                      .collect(Collectors.toList())
namesList ==> [Alice, Bob, Charlie, David]

jshell> // Using takeWhile to get names until we reach Charlie
jshell> namesList.stream()
   ...>          .takeWhile(name -> !name.equals("Charlie"))
   ...>          .forEach(System.out::println)
Alice
Bob

jshell> // List current variables
jshell> /vars
|    Person people = [Person{name='Alice', age=25}, Person{name='Bob', age=30}, Person{name='Charlie', age=35},
Person{name='David', age=28}]
|    List<Person> above30 = [Person{name='Charlie', age=35}]
|    double avgAge = 29.5
|    List<String> namesList = [Alice, Bob, Charlie, David]

jshell> // Exit JShell
jshell> /exit
|  Goodbye
```

Common Pitfalls and Tips:

- **Code Completion**: Press Tab for code completion suggestions.
- **Command-Line History**: Use up and down arrows to navigate through previously entered commands.
- **Semicolons are Optional**: JShell makes semicolons optional for single-line statements.
- **Feedback Control**: Use `/set feedback` to control the verbosity of feedback.
- **Importing by Default**: JShell imports common packages automatically, like `java.lang.*`, `java.util.*`, etc.

JShell is particularly valuable for quickly testing new Java features, API exploration, and learning. It eliminates the need to set up a full project with a main method just to test simple code snippets.

Java 10

Java 10, released in March 2018, was the first release under the new six-month cadence. While it had fewer features than the major releases, it introduced some important enhancements.

Local Variable Type Inference

What and Why: The `var` keyword was introduced to allow type inference for local variables, reducing verbosity while maintaining type safety. The compiler infers the type from the initialization expression.

Real-world Use Cases:

- Reducing boilerplate in variable declarations
- Improving readability with complex generic types
- Making code more concise for local variables with obvious types
- Working with anonymous classes with cleaner syntax

Before (Java 9) / After (Java 10) Comparison:

Before (Java 9):

```
// Explicit type declaration
String message = "Hello, World!";

// Verbose declarations with generics
Map<String, List<String>> usersByCountry = new HashMap<>();

// Long declaration for stream collectors
Stream<String> stream = getNames().stream();
Map<String, Integer> nameLengths = stream.collect(Collectors.toMap(
    Function.identity(),
    String::length
));
```

After (Java 10):

```
// Type inferred by compiler
var message = "Hello, World!";

// Cleaner declaration with generics
var usersByCountry = new HashMap<String, List<String>>();

// More concise stream operation
var stream = getNames().stream();
var nameLengths = stream.collect(Collectors.toMap(
    Function.identity(),
    String::length
));
```

Comprehensive Example:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.URL;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.time.LocalDate;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.function.Predicate;
import java.util.stream.Collectors;

public class LocalVarTypeInferenceExample {
    public static void main(String[] args) throws IOException {
        // 1. Basic usage
        System.out.println("=== Basic Usage ===");

        // Before var
        String name = "John Doe";
        int age = 30;

        // With var
        var newName = "Jane Smith";
        var newAge = 28;

        System.out.println("newName is of type: " + newName.getClass().getName());
        System.out.println("newName: " + newName);
        System.out.println("newAge: " + newAge);

        // 2. Complex generic types
        System.out.println("\n=== Complex Generic Types ===");

        // Before var
        Map<String, List<Map<String, String>>> complexMap1 = new HashMap<>();

        // With var - more readable, especially with the construction on the right
        var complexMap2 = new HashMap<String, List<Map<String, String>>>();
```

```
        // 3. Enhanced for loop iterations
        System.out.println("\n=== Enhanced For Loops ===");

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

        // Before var
        for (String n : names) {
            System.out.println("Name: " + n);
        }

        // With var
        for (var n : names) {
            System.out.println("Name (var): " + n);
        }

        // 4. Try-with-resources
        System.out.println("\n=== Try-with-resources ===");

        // Before var
        try (BufferedReader reader = new BufferedReader(
                new InputStreamReader(
                    LocalVarTypeInferenceExample.class.getResourceAsStream("/dummy.txt")))) {
            // Process reader
        } catch (IOException | NullPointerException e) {
            System.out.println("Resource not found or error reading");
        }

        // With var - cleaner nesting
        try (var reader = new BufferedReader(
                new InputStreamReader(
                    LocalVarTypeInferenceExample.class.getResourceAsStream("/dummy.txt")))) {
            // Process reader
        } catch (IOException | NullPointerException e) {
            System.out.println("Resource not found or error reading");
        }

        // 5. Lambda expressions with var (Java 11+)
        System.out.println("\n=== Lambda with var (Java 11+) ===");

        // Before var (Java 10)
        Predicate<String> startsWithA = s -> s.startsWith("A");

        // With var in lambda parameters (Java 11+)
        // Predicate<String> startsWithAVar = (var s) -> s.startsWith("A");

        // 6. Anonymous classes
        System.out.println("\n=== Anonymous Classes ===");

        // Before var
        Runnable runnable1 = new Runnable() {
            @Override
            public void run() {
                System.out.println("Runnable without var");
            }
        };

        // With var - cleaner syntax
        var runnable2 = new Runnable() {
            @Override
            public void run() {
                System.out.println("Runnable with var");
            }
        };

        runnable1.run();
        runnable2.run();

        // 7. Working with streams
        System.out.println("\n=== Streams ===");

        // Before var
        List<String> filteredNames = names.stream()
                                          .filter(s -> s.length() > 4)
                                          .collect(Collectors.toList());

        // With var
        var filteredNamesVar = names.stream()
                                    .filter(s -> s.length() > 4)
                                    .collect(Collectors.toList());

        System.out.println("Filtered names: " + filteredNamesVar);
```

```
        // 8. Practical example: File processing
        System.out.println("\n=== Practical Example: File Processing ===");

        // Create a sample file
        var path = Paths.get("sample.txt");
        var content = List.of("Line 1", "Line 2", "Line 3");
        Files.write(path, content);

        // Process the file
        try {
            var lines = Files.readAllLines(path);
            var lineCount = lines.size();
            var charCount = lines.stream().mapToInt(String::length).sum();

            System.out.println("File stats:");
            System.out.println("- Lines: " + lineCount);
            System.out.println("- Characters: " + charCount);

            // Process each line
            for (var line : lines) {
                var words = line.split("\\s+");
                System.out.println(line + " -> " + words.length + " words");
            }
        } catch (IOException e) {
            System.out.println("Error reading file: " + e.getMessage());
        } finally {
            // Clean up
            Files.deleteIfExists(path);
        }

        // 9. Creating objects with inference
        System.out.println("\n=== Creating Objects ===");

        // Date handling with var
        var today = LocalDate.now();
        var tomorrow = today.plusDays(1);

        System.out.println("Today: " + today);
        System.out.println("Tomorrow: " + tomorrow);

        // Custom object
        var person = new Person("John", 30);
        System.out.println("Person: " + person);
    }

    // Helper class
    static class Person {
        private final String name;
        private final int age;

        public Person(String name, int age) {
            this.name = name;
            this.age = age;
        }

        @Override
        public String toString() {
            return "Person{name='" + name + "', age=" + age + "}";
        }
    }
}
```

Limitations and Restrictions:

1. `var` can only be used for local variables, not for fields, method parameters, or return types
2. The variable must be initialized in the same statement where it's declared
3. The initializer must not be `null`
4. The initializer cannot be an array initializer (without new)
5. You cannot use `var` in a compound declaration

Common Pitfalls:

- **Readability Issues**: Using `var` with expressions that don't clearly indicate the type can make code harder to understand.
- **Diamond Operator Confusion**: When using `var` with the diamond operator, type arguments must be on the variable type or constructor.
- **Inference Limitations**: The compiler infers the most specific type from the right-hand side, which might not be what you expect.

```
// INCORRECT: Using var without a clear type context
var result = getResult(); // What type is this?

// BETTER: Either use explicit type or ensure context is clear
var result = getStringResult(); // Better - method name gives context
String result = getResult(); // Explicit type

// INCORRECT: Using var with null
var value = null; // Error: Cannot infer type

// CORRECT: Specify the type when using null
Integer value = null;

// INCORRECT: Using var with array initializers
var array = { 1, 2, 3 }; // Error: Cannot infer array type

// CORRECT: Use explicit array creation
var array = new int[] { 1, 2, 3 };

// INCORRECT: Diamond operator without type arguments
var list = new ArrayList<>(); // Infers ArrayList<Object>

// CORRECT: Specify type arguments
var list = new ArrayList<String>(); // Infers ArrayList<String>
```

Best Practices:

1. Use `var` when the type is obvious from the context or initialization
2. Avoid `var` when the initialization expression is complex and the type isn't clear
3. Consider readability and maintainability for your team
4. Give variables descriptive names when using `var`
5. Use `var` consistently throughout your codebase

Java 11 (LTS)

Java 11, released in September 2018, was the second Long-Term Support (LTS) release after Java 8. It introduced several important features and improvements.

HTTP Client

What and Why: Java 11 standardized the HTTP Client API (previously introduced as an incubator module in Java 9) to provide a modern replacement for the legacy `HttpURLConnection`. The new client supports HTTP/2, WebSocket, and is designed with both synchronous and asynchronous programming models.

Real-world Use Cases:

- RESTful API integration
- Web service consumption
- Parallel HTTP requests
- WebSocket communication
- File uploads and downloads

Before (Java 8) / After (Java 11) Comparison:

Before (Java 8 with HttpURLConnection):

```
URL url = new URL("https://api.example.com/data");
HttpURLConnection connection = (HttpURLConnection) url.openConnection();
connection.setRequestMethod("GET");
connection.setRequestProperty("Accept", "application/json");

int responseCode = connection.getResponseCode();
if (responseCode == HttpURLConnection.HTTP_OK) {
    BufferedReader in = new BufferedReader(new InputStreamReader(connection.getInputStream()));
    String inputLine;
    StringBuilder response = new StringBuilder();

    while ((inputLine = in.readLine()) != null) {
        response.append(inputLine);
    }
    in.close();

    // Process response
    System.out.println(response.toString());
}
```

After (Java 11 with HttpClient):

```
HttpClient client = HttpClient.newHttpClient();
HttpRequest request = HttpRequest.newBuilder()
        .uri(URI.create("https://api.example.com/data"))
        .header("Accept", "application/json")
        .GET()
        .build();

HttpResponse<String> response = client.send(request, HttpResponse.BodyHandlers.ofString());
System.out.println(response.statusCode());
System.out.println(response.body());
```

Comprehensive Example:

```
import java.io.IOException;
import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;
import java.net.http.WebSocket;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.time.Duration;
import java.util.List;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.CompletionStage;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Flow;
import java.util.concurrent.SubmissionPublisher;
import java.util.concurrent.TimeUnit;
import java.util.stream.Collectors;

public class HttpClientExample {

    // Use a public API for demonstration
    private static final String API_URL = "https://jsonplaceholder.typicode.com";

    public static void main(String[] args) throws Exception {

        // 1. Create an HTTP Client (can be reused for multiple requests)
        HttpClient client = HttpClient.newBuilder()
                .version(HttpClient.Version.HTTP_2)  // Prefer HTTP/2
                .connectTimeout(Duration.ofSeconds(10))  // Set connection timeout
                .followRedirects(HttpClient.Redirect.NORMAL)  // Follow redirects
                .build();

        // 2. Basic GET request (synchronous)
        System.out.println("=== Basic GET Request ===");
        HttpRequest getRequest = HttpRequest.newBuilder()
                .uri(URI.create(API_URL + "/posts/1"))
                .header("Accept", "application/json")
                .GET()  // GET is the default, so this is optional
                .build();

        HttpResponse<String> getResponse = client.send(getRequest, HttpResponse.BodyHandlers.ofString());

        System.out.println("Status Code: " + getResponse.statusCode());
        System.out.println("Response Body: " + getResponse.body());

        // 3. POST request with JSON body
        System.out.println("\n=== POST Request ===");
        String jsonBody = "{\"title\":\"foo\",\"body\":\"bar\",\"userId\":1}";

        HttpRequest postRequest = HttpRequest.newBuilder()
                .uri(URI.create(API_URL + "/posts"))
                .header("Content-Type", "application/json")
                .POST(HttpRequest.BodyPublishers.ofString(jsonBody))
                .build();

        HttpResponse<String> postResponse = client.send(postRequest, HttpResponse.BodyHandlers.ofString());

        System.out.println("Status Code: " + postResponse.statusCode());
        System.out.println("Response Body: " + postResponse.body());

        // 4. Asynchronous GET request
        System.out.println("\n=== Asynchronous GET Request ===");
        HttpRequest asyncRequest = HttpRequest.newBuilder()
                .uri(URI.create(API_URL + "/users/1"))
```

```
                .GET()
                .build();

        CompletableFuture<HttpResponse<String>> futureResponse =
                client.sendAsync(asyncRequest, HttpResponse.BodyHandlers.ofString());

        // Do other work while the request is in progress...
        System.out.println("Request sent asynchronously, doing other work...");

        // Get the response when it's ready
        futureResponse.thenAccept(response -> {
            System.out.println("Async Response Status Code: " + response.statusCode());
            System.out.println("Async Response Body: " + response.body());
        }).join(); // wait for completion

        // 5. Multiple asynchronous requests in parallel
        System.out.println("\n=== Multiple Parallel Requests ===");
        List<URI> uris = List.of(
                URI.create(API_URL + "/posts/1"),
                URI.create(API_URL + "/posts/2"),
                URI.create(API_URL + "/posts/3")
        );

        List<CompletableFuture<HttpResponse<String>>> futures = uris.stream()
                .map(uri -> HttpRequest.newBuilder(uri).build())
                .map(request -> client.sendAsync(request, HttpResponse.BodyHandlers.ofString()))
                .collect(Collectors.toList());

        // Wait for all responses and process them
        CompletableFuture.allOf(futures.toArray(new CompletableFuture[0]))
                .thenRun(() -> {
                    System.out.println("All requests completed");
                    futures.forEach(future -> {
                        try {
                            HttpResponse<String> response = future.get();
                            System.out.println("URI: " + response.uri());
                            System.out.println("Status: " + response.statusCode());
                            System.out.println("First 50 chars: " +
                                    response.body().substring(0, Math.min(50, response.body().length())) + "...\n");
                        } catch (InterruptedException | ExecutionException e) {
                            e.printStackTrace();
                        }
                    });
                })
                .join(); // wait for completion

        // 6. File download
        System.out.println("\n=== File Download ===");
        HttpRequest downloadRequest = HttpRequest.newBuilder()
                .uri(URI.create("https://www.w3.org/WAI/ER/tests/xhtml/testfiles/resources/pdf/dummy.pdf"))
                .build();

        Path downloadPath = Paths.get("dummy.pdf");
        HttpResponse<Path> fileResponse = client.send(downloadRequest,
                HttpResponse.BodyHandlers.ofFile(downloadPath));

        System.out.println("File downloaded to: " + fileResponse.body());
        System.out.println("File size: " + Files.size(fileResponse.body()) + " bytes");

        // Clean up
        Files.delete(downloadPath);

        // 7. Setting request timeout
        System.out.println("\n=== Request Timeout ===");
        HttpRequest timeoutRequest = HttpRequest.newBuilder()
                .uri(URI.create(API_URL + "/posts/1"))
                .timeout(Duration.ofSeconds(10))
                .build();

        HttpResponse<String> timeoutResponse = client.send(timeoutRequest,
                HttpResponse.BodyHandlers.ofString());
        System.out.println("Response received within timeout: " + timeoutResponse.statusCode());

        // 8. Using HTTP/1.1 explicitly
        System.out.println("\n=== Using HTTP/1.1 ===");
        HttpClient http11Client = HttpClient.newBuilder()
                .version(HttpClient.Version.HTTP_1_1)
                .build();

        HttpResponse<String> http11Response = http11Client.send(getRequest,
                HttpResponse.BodyHandlers.ofString());
```

```
        System.out.println("HTTP/1.1 Response Status: " + http11Response.statusCode());
        System.out.println("HTTP Version Used: " + http11Response.version());

        // 9. POST with form data
        System.out.println("\n=== POST with Form Data ===");
        HttpRequest formRequest = HttpRequest.newBuilder()
                .uri(URI.create(API_URL + "/posts"))
                .header("Content-Type", "application/x-www-form-urlencoded")
                .POST(HttpRequest.BodyPublishers.ofString("title=Test&body=Body&userId=1"))
                .build();

        HttpResponse<String> formResponse = client.send(formRequest,
                HttpResponse.BodyHandlers.ofString());
        System.out.println("Form POST Response: " + formResponse.body());
    }

    // WebSocket example - Note: requires a WebSocket server to test with
    static class WebSocketExample implements WebSocket.Listener {

        public void demonstrateWebSocket() throws Exception {
            HttpClient client = HttpClient.newHttpClient();
            WebSocket webSocket = client.newWebSocketBuilder()
                    .buildAsync(URI.create("wss://echo.websocket.org"), this)
                    .join();

            webSocket.sendText("Hello WebSocket!", true);

            // Keep connection open for a while
            TimeUnit.SECONDS.sleep(5);
            webSocket.sendClose(WebSocket.NORMAL_CLOSURE, "Closing");
        }

        @Override
        public void onOpen(WebSocket webSocket) {
            System.out.println("WebSocket opened");
            webSocket.request(1);
        }

        @Override
        public CompletionStage<?> onText(WebSocket webSocket, CharSequence data, boolean last) {
            System.out.println("Received: " + data);
            webSocket.request(1);
            return CompletableFuture.completedFuture(null);
        }

        @Override
        public CompletionStage<?> onClose(WebSocket webSocket, int statusCode, String reason) {
            System.out.println("WebSocket closed: " + statusCode + " " + reason);
            return CompletableFuture.completedFuture(null);
        }

        @Override
        public void onError(WebSocket webSocket, Throwable error) {
            System.out.println("WebSocket error: " + error.getMessage());
            error.printStackTrace();
        }
    }
}
```

Key Features of the HTTP Client API:

1. **Builder Pattern**: Both `HttpClient` and `HttpRequest` use the builder pattern for configuration.
2. **HTTP/2 Support**: By default, the client uses HTTP/2 if available, falling back to HTTP/1.1.
3. **Synchronous and Asynchronous APIs**: `send()` for synchronous requests and `sendAsync()` for asynchronous requests.
4. **BodyHandlers**: Various body handlers for different response types (string, byte array, file, etc.).
5. **BodyPublishers**: Different publishers for request bodies (string, byte array, file, etc.).
6. **WebSocket Support**: Built-in WebSocket client with `Listener` interface.
7. **Timeouts**: Both connection and request timeouts can be configured.

Common Pitfalls:

- **Response Handling**: Always check the status code before processing the response body.
- **Resource Management**: The HTTP client manages connections for you, but you need to handle response resources properly.
- **Exception Handling**: Different kinds of exceptions can be thrown during HTTP operations.
- **Thread Management**: Asynchronous requests use the client's executor, which may need to be configured for large-scale applications.

```
// INCORRECT: Not checking response status before processing
HttpResponse<String> response = client.send(request, HttpResponse.BodyHandlers.ofString());
```

```
String data = response.body(); // May be an error message if request failed

// CORRECT: Check status code first
if (response.statusCode() >= 200 && response.statusCode() < 300) {
    // Success
    String data = response.body();
} else {
    // Handle error
    System.err.println("Error: " + response.statusCode() + " - " + response.body());
}

// INCORRECT: Blocking in an asynchronous context
client.sendAsync(request, HttpResponse.BodyHandlers.ofString())
      .get(); // Blocks the current thread, defeating the purpose of async

// CORRECT: Use CompletableFuture methods for async processing
client.sendAsync(request, HttpResponse.BodyHandlers.ofString())
      .thenAccept(response -> {
          // Process response when it's ready
      })
      .exceptionally(ex -> {
          // Handle exceptions
          return null;
      });
```

String API Enhancements

What and Why: Java 11 introduced several new methods to the String class, making common string operations more convenient and reducing the need for external libraries or custom utility methods.

New String Methods in Java 11:

1. `isBlank()`: Checks if a string is empty or contains only white space.
2. `lines()`: Returns a stream of lines extracted from the string, separated by line terminators.
3. `strip()`, `stripLeading()`, `stripTrailing()`: Remove leading and trailing white space, understanding Unicode white space better than `trim()`.
4. `repeat(int)`: Returns a string consisting of the original string repeated n times.

Before (Java 10) / After (Java 11) Comparison:

Before (Java 10):

```
// Check if string is blank
boolean isBlank = s.trim().isEmpty();

// Split string into lines
List<String> lines = Arrays.asList(s.split("\\R"));

// Trim whitespace
String trimmed = s.trim();

// Repeat a string
String repeated = String.join("", Collections.nCopies(3, "hello"));
```

After (Java 11):

```
// Check if string is blank
boolean isBlank = s.isBlank();

// Split string into lines
Stream<String> lines = s.lines();

// Trim whitespace (with better Unicode support)
String stripped = s.strip();
String strippedLeading = s.stripLeading();
String strippedTrailing = s.stripTrailing();

// Repeat a string
String repeated = "hello".repeat(3);
```

Comprehensive Example:

```
import java.util.stream.Collectors;

public class StringEnhancementsExample {
```

```
    public static void main(String[] args) {
        // 1. isBlank() - Checks if string is empty or contains only whitespace
        System.out.println("=== isBlank() ===");
        String empty = "";
        String blank = "   \t   \n   ";
        String notBlank = "Hello";

        System.out.println("empty.isBlank(): " + empty.isBlank());        // true
        System.out.println("blank.isBlank(): " + blank.isBlank());        // true
        System.out.println("notBlank.isBlank(): " + notBlank.isBlank());  // false

        // Comparison with isEmpty()
        System.out.println("empty.isEmpty(): " + empty.isEmpty());        // true
        System.out.println("blank.isEmpty(): " + blank.isEmpty());        // false

        // 2. lines() - Splits string by line terminators and returns a Stream
        System.out.println("\n=== lines() ===");
        String multiline = "Line 1\nLine 2\r\nLine 3\nLine 4";

        System.out.println("Original multiline string:");
        System.out.println(multiline);

        System.out.println("\nProcessing lines as a stream:");
        multiline.lines()
                .map(line -> "- " + line)
                .forEach(System.out::println);

        // Count lines
        long lineCount = multiline.lines().count();
        System.out.println("\nNumber of lines: " + lineCount);

        // Filter and transform lines
        System.out.println("\nLines containing '3' or '4':");
        multiline.lines()
                .filter(line -> line.contains("3") || line.contains("4"))
                .forEach(System.out::println);

        // 3. strip(), stripLeading(), stripTrailing() - Better whitespace handling
        System.out.println("\n=== strip() methods ===");
        String whitespace = "  \u2005Hello World\u2005  ";

        System.out.println("Original: '" + whitespace + "'");
        System.out.println("After strip(): '" + whitespace.strip() + "'");
        System.out.println("After stripLeading(): '" + whitespace.stripLeading() + "'");
        System.out.println("After stripTrailing(): '" + whitespace.stripTrailing() + "'");

        // Compare with trim()
        System.out.println("After trim(): '" + whitespace.trim() + "'");

        // Demonstrate trim() vs strip() with Unicode whitespace
        String unicodeWhitespace = "\u2005Unicode whitespace\u2005";
        System.out.println("\nUnicode whitespace string: '" + unicodeWhitespace + "'");
        System.out.println("After trim(): '" + unicodeWhitespace.trim() + "'");  // Doesn't trim Unicode whitespace
        System.out.println("After strip(): '" + unicodeWhitespace.strip() + "'"); // Properly handles Unicode whitespace

        // 4. repeat() - Repeats the string n times
        System.out.println("\n=== repeat() ===");
        String star = "*";
        System.out.println("star.repeat(1): " + star.repeat(1));
        System.out.println("star.repeat(5): " + star.repeat(5));

        String abc = "abc";
        System.out.println("abc.repeat(0): '" + abc.repeat(0) + "'");   // Empty string
        System.out.println("abc.repeat(3): " + abc.repeat(3));          // "abcabcabc"

        // Creating indentation using repeat
        String indentation = "    ";
        String codeSnippet = indentation + "if (condition) {" + "\n" +
                             indentation.repeat(2) + "doSomething();" + "\n" +
                             indentation + "}";
        System.out.println("\nFormatted code snippet:");
        System.out.println(codeSnippet);

        // 5. Practical Examples
        System.out.println("\n=== Practical Examples ===");

        // Parse CSV data with lines()
        String csvData = "id,name,age\n1,Alice,30\n2,Bob,25\n3,Charlie,35";
        System.out.println("CSV Data:");
        System.out.println(csvData);
```

```
        System.out.println("\nParsed CSV data:");
        csvData.lines()
              .skip(1)  // Skip header
              .map(line -> line.split(","))
              .forEach(fields -> System.out.println("Name: " + fields[1] + ", Age: " + fields[2]));

        // Clean user input with strip()
        String userInput = "   input with spaces   ";
        String cleanInput = userInput.strip();
        System.out.println("\nUser input: '" + userInput + "'");
        System.out.println("Cleaned input: '" + cleanInput + "'");

        // Create a string pyramid using repeat()
        System.out.println("\nString pyramid:");
        for (int i = 1; i <= 5; i++) {
            System.out.println(" ".repeat(5 - i) + "*".repeat(i * 2 - 1));
        }

        // Validate form input
        String username = "  ";
        if (username.isBlank()) {
            System.out.println("\nUsername cannot be blank!");
        }

        // Process multiline text
        String log = "2023-01-01 INFO: System started\n" +
                     "2023-01-01 DEBUG: Loading configuration\n" +
                     "2023-01-01 ERROR: Failed to connect to database\n" +
                     "2023-01-01 INFO: Retrying connection\n" +
                     "2023-01-01 INFO: Connected successfully";

        System.out.println("\nError logs:");
        log.lines()
           .filter(line -> line.contains("ERROR"))
           .forEach(System.out::println);

        // Count word occurrences
        String text = "Java is a programming language. Java is class-based and object-oriented.";
        long javaCount = text.toLowerCase()
                             .lines()
                             .flatMap(line -> Stream.of(line.split("\\s+")))
                             .map(word -> word.strip().replaceAll("[^a-zA-Z]", ""))
                             .filter(word -> word.equalsIgnoreCase("java"))
                             .count();

        System.out.println("\nOccurrences of 'Java': " + javaCount);
    }
}
```

Common Pitfalls:

- **trim() vs. strip()**: `strip()` handles Unicode whitespace better than `trim()`, which only removes characters <= U+0020.
- **Performance Considerations**: `lines()` creates a Stream, which can be more efficient for large strings but has overhead for small ones.
- **repeat() with negative values**: Calling `repeat()` with a negative count will throw an exception.

```
// INCORRECT: Using trim() with Unicode whitespace
String unicodeWhitespace = "\u2005Hello\u2005";
String trimmed = unicodeWhitespace.trim(); // Unicode whitespace remains

// CORRECT: Use strip() for Unicode whitespace
String stripped = unicodeWhitespace.strip(); // All whitespace removed

// INCORRECT: Expecting repeat() to work with negative values
String repeated = "abc".repeat(-1); // IllegalArgumentException

// CORRECT: Ensure count is non-negative
int count = Math.max(0, someValue);
String repeated = "abc".repeat(count);
```

Files API Enhancements

What and Why: Java 11 added new methods to the `Files` class to simplify common file operations, particularly reading and writing String content.

New Files Methods in Java 11:

1. `readString(Path)`: Reads all content from a file as a String.
2. `writeString(Path, String, OpenOption...)`: Writes a String to a file.

Before (Java 10) / After (Java 11) Comparison:

Before (Java 10):

```
// Read file to string
String content = new String(Files.readAllBytes(path), StandardCharsets.UTF_8);

// Write string to file
Files.write(path, content.getBytes(StandardCharsets.UTF_8));
```

After (Java 11):

```
// Read file to string
String content = Files.readString(path);

// Write string to file
Files.writeString(path, content);
```

Example:

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardOpenOption;
import java.nio.charset.StandardCharsets;

public class FilesApiEnhancementsExample {
    public static void main(String[] args) {
        // 1. Reading and writing Strings with the new methods
        try {
            // Create a temporary file
            Path tempFile = Files.createTempFile("java11-demo", ".txt");
            System.out.println("Created temporary file: " + tempFile);

            // Write a string to the file
            String content = "Hello, Java 11!\nThis is a test file.\nWritten with Files.writeString()";
            Files.writeString(tempFile, content, StandardCharsets.UTF_8);
            System.out.println("Content written to file");

            // Read the string back from the file
            String readContent = Files.readString(tempFile, StandardCharsets.UTF_8);
            System.out.println("\nRead content from file:");
            System.out.println(readContent);

            // Append to the file
            String additionalContent = "\nThis line was appended with writeString()";
            Files.writeString(tempFile, additionalContent, StandardOpenOption.APPEND);

            // Read the updated content
            readContent = Files.readString(tempFile);
            System.out.println("\nUpdated content with append:");
            System.out.println(readContent);

            // Clean up
            Files.delete(tempFile);
            System.out.println("\nTemporary file deleted");

        } catch (IOException e) {
            System.err.println("Error: " + e.getMessage());
            e.printStackTrace();
        }

        // 2. Comparison with older methods
        try {
            // Create a file for comparison
            Path comparisonFile = Files.createTempFile("java-comparison", ".txt");

            // Old way (Java 8-10)
            long startOld = System.nanoTime();
            String oldContent = "Using pre-Java 11 methods";
            Files.write(comparisonFile, oldContent.getBytes(StandardCharsets.UTF_8));
            String readOldContent = new String(Files.readAllBytes(comparisonFile), StandardCharsets.UTF_8);
            long endOld = System.nanoTime();
```

```
            // New way (Java 11+)
            long startNew = System.nanoTime();
            Files.writeString(comparisonFile, "Using Java 11 methods", StandardCharsets.UTF_8);
            String readNewContent = Files.readString(comparisonFile, StandardCharsets.UTF_8);
            long endNew = System.nanoTime();

            System.out.println("\nPerformance comparison:");
            System.out.println("Old way execution time: " + (endOld - startOld) / 1000 + " µs");
            System.out.println("New way execution time: " + (endNew - startNew) / 1000 + " µs");

            // Clean up
            Files.delete(comparisonFile);

        } catch (IOException e) {
            System.err.println("Error in comparison: " + e.getMessage());
        }

        // 3. Practical example: Simple text file processing
        try {
            // Create a sample configuration file
            Path configFile = Files.createTempFile("config", ".properties");
            Files.writeString(configFile,
                "# Application Configuration\n" +
                "app.name=My Java 11 App\n" +
                "app.version=1.0.0\n" +
                "app.debug=true\n" +
                "app.server.host=localhost\n" +
                "app.server.port=8080"
            );

            System.out.println("\nSimple config file processing:");

            // Read and process the configuration
            String config = Files.readString(configFile);

            // Using Java 11 string methods with Files API
            config.lines()
                  .filter(line -> !line.isBlank() && !line.strip().startsWith("#"))
                  .map(line -> {
                      String[] parts = line.split("=", 2);
                      return parts.length == 2 ?
                          Map.entry(parts[0].strip(), parts[1].strip()) :
                          null;
                  })
                  .filter(Objects::nonNull)
                  .forEach(entry -> System.out.println("Key: " + entry.getKey() + ", Value: " + entry.getValue()));

            // Update a value in the config
            String updatedConfig = config.lines()
                .map(line -> line.startsWith("app.version") ?
                    "app.version=1.0.1" : line)
                .collect(Collectors.joining("\n"));

            // Write the updated config back to the file
            Files.writeString(configFile, updatedConfig, StandardOpenOption.TRUNCATE_EXISTING);

            System.out.println("\nUpdated version in config file:");
            System.out.println(Files.readString(configFile));

            // Clean up
            Files.delete(configFile);

        } catch (IOException e) {
            System.err.println("Error in config processing: " + e.getMessage());
        }
    }
}
```

Common Pitfalls:

- **Character Encoding**: Default encoding is used if not specified, which may vary across platforms.
- **File Size**: For very large files, these methods read the entire file into memory, which can cause OutOfMemoryError.

```
// INCORRECT: Not specifying character encoding
String content = Files.readString(path); // Uses default charset

// CORRECT: Explicitly specify the character encoding
String content = Files.readString(path, StandardCharsets.UTF_8);
```

```
// INCORRECT: Reading very large files as strings
String hugeContent = Files.readString(hugePath); // May cause OutOfMemoryError

// CORRECT: Use streaming approaches for large files
try (Stream<String> lines = Files.lines(hugePath)) {
    lines.forEach(line -> processLine(line));
}
```

Java 12-16 (Selected Features)

Java 12 through 16 were non-LTS releases with several important features that were later stabilized in Java 17.

Switch Expressions

What and Why: Switch expressions were introduced as a preview feature in Java 12, enhanced in Java 13, and finalized in Java 14. They allow switch constructs to be used as expressions rather than just statements, enabling more concise and less error-prone code.

Real-world Use Cases:

- Mapping inputs to outputs
- Pattern-based data extraction
- Replacing complex if-else chains
- State machine transitions
- Command processing

Before (Java 8) / After (Java 14) Comparison:

Before (Java 8):

```
String day = "MONDAY";
String result;

switch (day) {
    case "MONDAY":
    case "TUESDAY":
    case "WEDNESDAY":
    case "THURSDAY":
    case "FRIDAY":
        result = "Weekday";
        break;
    case "SATURDAY":
    case "SUNDAY":
        result = "Weekend";
        break;
    default:
        result = "Invalid day";
        break;
}
```

After (Java 14):

```
String day = "MONDAY";
String result = switch (day) {
    case "MONDAY", "TUESDAY", "WEDNESDAY", "THURSDAY", "FRIDAY" -> "Weekday";
    case "SATURDAY", "SUNDAY" -> "Weekend";
    default -> "Invalid day";
};
```

Comprehensive Example:

```
public class SwitchExpressionsExample {
    enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY }

    public static void main(String[] args) {
        // 1. Basic switch expression with arrow syntax
        System.out.println("=== Basic Switch Expression ===");
        Day today = Day.WEDNESDAY;

        String typeOfDay = switch (today) {
            case MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY -> "Weekday";
            case SATURDAY, SUNDAY -> "Weekend";
        };

        System.out.println(today + " is a " + typeOfDay);
```

```
        // 2. Switch expression with multiple statements using blocks
        System.out.println("\n=== Switch with Blocks ===");

        int numLetters = switch (today) {
            case MONDAY, FRIDAY, SUNDAY -> {
                System.out.println("Six letters in " + today);
                yield 6;
            }
            case TUESDAY -> {
                System.out.println("Seven letters in " + today);
                yield 7;
            }
            case THURSDAY, SATURDAY -> {
                System.out.println("Eight letters in " + today);
                yield 8;
            }
            case WEDNESDAY -> {
                System.out.println("Nine letters in " + today);
                yield 9;
            }
        };

        System.out.println("Number of letters: " + numLetters);

        // 3. Traditional switch statement (still supported)
        System.out.println("\n=== Traditional Switch Statement ===");

        switch (today) {
            case MONDAY:
                System.out.println("Start of work week");
                break;
            case TUESDAY:
            case WEDNESDAY:
            case THURSDAY:
                System.out.println("Midweek");
                break;
            case FRIDAY:
                System.out.println("End of work week");
                break;
            case SATURDAY:
            case SUNDAY:
                System.out.println("Weekend");
                break;
        }

        // 4. Mixing old and new syntax (not recommended but possible)
        System.out.println("\n=== Mixed Syntax (Not Recommended) ===");

        int result = switch (today) {
            case MONDAY, TUESDAY -> 1;
            case WEDNESDAY -> 2;
            case THURSDAY:
            case FRIDAY:
                yield 3;
            default:
                System.out.println("Weekend detected");
                yield 4;
        };

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

        // 5. Switch expressions with null handling
        System.out.println("\n=== Handling Null Values ===");

        Day nullDay = null;
        try {
            String nullDayType = switch (nullDay) {
                case MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY -> "Weekday";
                case SATURDAY, SUNDAY -> "Weekend";
                default -> "Unknown";
                // null -> "Null day"; // This syntax isn't supported yet
            };
        } catch (NullPointerException e) {
            System.out.println("NullPointerException caught: " + e.getMessage());
            System.out.println("Switch expressions still can't directly handle null");
        }

        // Safe handling of potentially null values
        String message = (nullDay == null) ? "Null day" : switch (nullDay) {
            case MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY -> "Weekday";
```

```
            case SATURDAY, SUNDAY -> "Weekend";
        };

        System.out.println("Safe null handling: " + message);

        // 6. Real-world examples
        System.out.println("\n=== Real-world Examples ===");

        // Example: HTTP status code classification
        int statusCode = 404;
        String statusCategory = switch (statusCode / 100) {
            case 1 -> "Informational";
            case 2 -> "Success";
            case 3 -> "Redirection";
            case 4 -> "Client Error";
            case 5 -> "Server Error";
            default -> "Unknown Status Code";
        };

        System.out.println("HTTP " + statusCode + " is a " + statusCategory + " status");

        // Example: Command processor
        String command = "SAVE";
        boolean processResult = switch (command.toUpperCase()) {
            case "OPEN" -> {
                System.out.println("Opening file...");
                yield true;
            }
            case "SAVE" -> {
                System.out.println("Saving file...");
                yield true;
            }
            case "CLOSE" -> {
                System.out.println("Closing file...");
                yield true;
            }
            default -> {
                System.out.println("Unknown command: " + command);
                yield false;
            }
        };

        System.out.println("Command processed successfully: " + processResult);

        // Example: Different calculation based on shape type
        String shape = "circle";
        double dimension = 5.0;
        double area = switch (shape.toLowerCase()) {
            case "square" -> dimension * dimension;
            case "circle" -> Math.PI * dimension * dimension;
            case "triangle" -> (Math.sqrt(3) / 4) * dimension * dimension;
            default -> throw new IllegalArgumentException("Unknown shape: " + shape);
        };

        System.out.printf("Area of %s with dimension %.1f: %.2f%n", shape, dimension, area);
    }
}
```

Common Pitfalls:

- **Exhaustiveness**: Switch expressions must be exhaustive (cover all possible values or include a default case).
- **Yield vs. Return**: Use `yield` (not `return`) to provide a value from a block in a switch expression.
- **Fall-through**: Arrow syntax (`->`) doesn't fall through, but colon syntax (`:`) still does.
- **Null Handling**: Switch expressions don't directly support null values as case labels (until pattern matching in later versions).

```
// INCORRECT: Non-exhaustive switch expression
enum Color { RED, GREEN, BLUE }
Color color = Color.RED;

String intensity = switch (color) { // Error: switch expression not exhaustive
    case RED -> "High";
    case GREEN -> "Medium";
    // Missing BLUE case
};

// CORRECT: Exhaustive switch expression
String intensity = switch (color) {
    case RED -> "High";
```

```
    case GREEN -> "Medium";
    case BLUE -> "Low";
};

// INCORRECT: Using return instead of yield
int value = switch (color) {
    case RED -> {
        System.out.println("Red detected");
        return 1; // Error: return not allowed in switch expressions
    }
    case GREEN, BLUE -> 0;
};

// CORRECT: Using yield
int value = switch (color) {
    case RED -> {
        System.out.println("Red detected");
        yield 1; // Use yield to produce a value
    }
    case GREEN, BLUE -> 0;
};
```

Text Blocks

What and Why: Text blocks, introduced as a preview feature in Java 13 and finalized in Java 15, provide a way to define multi-line string literals with improved readability and handling of indentation. They're especially useful for writing HTML, XML, SQL, JSON and other structured text without excessive escape sequences.

Real-world Use Cases:

- SQL queries
- HTML/XML templates
- JSON structures
- File templates
- Code generation

Before (Java 12) / After (Java 15) Comparison:

Before (Java 12):

```
String html = "<html>\n" +
              "    <body>\n" +
              "        <p>Hello, World!</p>\n" +
              "    </body>\n" +
              "</html>";

String json = "{\n" +
              "    \"name\": \"John\",\n" +
              "    \"age\": 30,\n" +
              "    \"city\": \"New York\"\n" +
              "}";
```

After (Java 15):

```
String html = """
              <html>
                  <body>
                      <p>Hello, World!</p>
                  </body>
              </html>
              """;

String json = """
              {
                  "name": "John",
                  "age": 30,
                  "city": "New York"
              }
              """;
```

Comprehensive Example:

```
public class TextBlocksExample {
    public static void main(String[] args) {
        // 1. Basic Text Block
```

```
        System.out.println("=== Basic Text Block ===");

        String textBlock = """
                Hello,
                World!
                This is a text block.
                """;

        System.out.println("Text block content:");
        System.out.println(textBlock);

        // 2. Comparing with traditional strings
        System.out.println("\n=== Comparison with Traditional Strings ===");

        String traditional = "Line 1\n" +
                             "Line 2\n" +
                             "Line 3";

        String withTextBlock = """
                Line 1
                Line 2
                Line 3""";

        System.out.println("Traditional string:");
        System.out.println(traditional);

        System.out.println("\nText block:");
        System.out.println(withTextBlock);

        System.out.println("\nAre they equal? " + traditional.equals(withTextBlock));

        // 3. Handling indentation
        System.out.println("\n=== Handling Indentation ===");

        // The indentation of the closing delimiter determines the incidental indentation
        String indented = """
                This line has no extra indentation.
                    This line is indented with 4 spaces.
                        This line is indented with 8 spaces.
                """;

        System.out.println("Indented text block:");
        System.out.println(indented);

        // 4. Escaping delimiters and controlling newlines
        System.out.println("\n=== Escaping and Newline Control ===");

        // Escape sequence for triple quotes
        String escapedQuotes = """
                This text block contains \""" three quotes\""".
                """;

        System.out.println("Escaped quotes:");
        System.out.println(escapedQuotes);

        // Controlling trailing newline with \ at the end
        String noTrailingNewline = """
                This text block has \
                no newline between these \
                three lines, and no trailing newline.\
                """;

        System.out.println("\nNo trailing newline:");
        System.out.println(noTrailingNewline);
        System.out.println("(Text after)");

        // 5. String interpolation and expressions
        System.out.println("\n=== String Interpolation ===");

        String name = "John";
        int age = 30;

        String userInfo = """
                User details:
                  Name: %s
                  Age: %d
                """.formatted(name, age);

        System.out.println("Interpolated string:");
        System.out.println(userInfo);
```

```
        // 6. Real-world examples
        System.out.println("\n=== Real-world Examples ===");

        // HTML Example
        String html = """
                <!DOCTYPE html>
                <html>
                  <head>
                    <title>Text Blocks Demo</title>
                  </head>
                  <body>
                    <h1>Hello, Text Blocks!</h1>
                    <p>This HTML is much easier to read and write with text blocks.</p>
                  </body>
                </html>
                """;

        System.out.println("HTML Example:");
        System.out.println(html);

        // SQL Example
        String sql = """
                SELECT e.employee_id, e.first_name, e.last_name, d.department_name
                FROM employees e
                JOIN departments d ON e.department_id = d.department_id
                WHERE e.salary > 5000
                ORDER BY e.salary DESC, e.last_name ASC
                """;

        System.out.println("\nSQL Example:");
        System.out.println(sql);

        // JSON Example
String json = """
        {
          "user": {
            "id": 12345,
            "name": "John Doe",
            "email": "john.doe@example.com",
            "roles": ["user", "admin"],
            "settings": {
              "theme": "dark",
              "notifications": true,
              "twoFactorAuth": false
            }
          }
        }
        """;

System.out.println("\nJSON Example:");
System.out.println(json);

// XML Example
String xml = """
        <?xml version="1.0" encoding="UTF-8"?>
        <configuration>
          <appSettings>
            <add key="apiUrl" value="https://api.example.com" />
            <add key="apiTimeout" value="30000" />
            <add key="maxRetries" value="3" />
          </appSettings>
          <connectionStrings>
            <add name="mainDB" connectionString="server=db1;database=myapp;uid=user;pwd=password" />
          </connectionStrings>
        </configuration>
        """;

System.out.println("\nXML Example:");
System.out.println(xml);

// 7. Advanced techniques
System.out.println("\n=== Advanced Techniques ===");

// Concatenating text blocks with traditional strings
String header = "REPORT HEADER";
String footer = "REPORT FOOTER";

String report = header + "\n" + """
        This is the body of the report.
        It contains multiple lines.
        This is the last line of the body.
```

```
        """ + footer;

System.out.println("Concatenated report:");
System.out.println(report);

// Using text blocks in method calls
boolean isValid = validateSQL("""
        SELECT id, name, email
        FROM users
        WHERE active = true
        """);

System.out.println("\nSQL validation result: " + isValid);

// Creating templates
String emailTemplate = """
        Dear %s,

        Thank you for your recent purchase of %s.
        Your order #%d has been processed and will be shipped on %s.

        Best regards,
        The Example Store Team
        """;

String personalizedEmail = emailTemplate.formatted(
        "Alice Smith",
        "Wireless Headphones",
        12345,
        "2023-06-15"
);

System.out.println("\nPersonalized email:");
System.out.println(personalizedEmail);
```

Common Pitfalls:

- **Incidental Whitespace**: The indentation of the closing delimiter determines the incidental whitespace removed from each line.
- **Trailing Newline**: Text blocks include a trailing newline by default, which can be prevented with a trailing `\`.
- **Escaped Delimiters**: You need to escape triple quotes within the text block.
- **Line Breaks**: Each line break in the source code becomes a `\n` in the resulting string, regardless of the platform.

```
// INCORRECT: Misalignment affects content
String misaligned = """
        First line
        Second line
    Third line
        """; // Third line will have extra spaces because the closing """ is indented differently

// CORRECT: Consistent alignment
String aligned = """
        First line
        Second line
        Third line
        """;

// INCORRECT: Forgetting about the trailing newline
String noNewlineNeeded = """
        This text has an extra trailing newline
        """;
printWithBrackets(noNewlineNeeded); // Output: [This text has an extra trailing newline
                                    // ]

// CORRECT: Remove the trailing newline when needed
String betterNoNewline = """
        This text has no trailing newline\
        """;
printWithBrackets(betterNoNewline); // Output: [This text has no trailing newline]
```

Records

What and Why: Records, introduced as a preview feature in Java 14 and finalized in Java 16, are a special kind of class that simplifies the creation of immutable data-carrying classes. They automatically provide constructors, accessor methods, `equals()`, `hashCode()`, and `toString()` implementations.

Real-world Use Cases:

- Data transfer objects (DTOs)

- API request/response objects
- Value objects in domain-driven design
- Configuration settings
- Immutable data carriers in streams

Before (Java 11) / After (Java 16) Comparison:

Before (Java 11):

```
public final class Point {
    private final int x;
    private final int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() { return x; }
    public int getY() { return y; }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Point point = (Point) o;
        return x == point.x && y == point.y;
    }

    @Override
    public int hashCode() {
        return Objects.hash(x, y);
    }

    @Override
    public String toString() {
        return "Point[x=" + x + ", y=" + y + "]";
    }
}
```

After (Java 16):

```
public record Point(int x, int y) { }
```

Comprehensive Example:

```
import java.time.LocalDate;
import java.util.*;

public class RecordsExample {
    public static void main(String[] args) {
        // 1. Basic Record Usage
        System.out.println("=== Basic Record Usage ===");

        // Create a record instance
        Person person = new Person("John Doe", 30, "john.doe@example.com");

        // Access fields using accessor methods (automatically generated)
        System.out.println("Name: " + person.name());
        System.out.println("Age: " + person.age());
        System.out.println("Email: " + person.email());

        // Automatically generated toString()
        System.out.println("\ntoString():");
        System.out.println(person);

        // 2. Equals and HashCode
        System.out.println("\n=== Equals and HashCode ===");

        Person person1 = new Person("Jane Smith", 25, "jane.smith@example.com");
        Person person2 = new Person("Jane Smith", 25, "jane.smith@example.com");
        Person person3 = new Person("Jane Smith", 26, "jane.smith@example.com");

        System.out.println("person1 equals person2: " + person1.equals(person2)); // true
        System.out.println("person1 equals person3: " + person1.equals(person3)); // false
```

```
// Using records with HashSet
Set<Person> personSet = new HashSet<>();
personSet.add(person1);
personSet.add(person2); // Won't be added since it's equal to person1
personSet.add(person3); // Will be added since it's different

System.out.println("\nSet size: " + personSet.size()); // 2
System.out.println("Set contents: " + personSet);

// 3. Compact Constructor
System.out.println("\n=== Compact Constructor ===");

try {
    Employee employee = new Employee("", -5, "HR"); // Invalid data
} catch (IllegalArgumentException e) {
    System.out.println("Validation error: " + e.getMessage());
}

Employee validEmployee = new Employee("Alice Johnson", 35, "Engineering");
System.out.println("Valid employee: " + validEmployee);

// 4. Canonical Constructor
System.out.println("\n=== Canonical Constructor ===");

Order order = new Order(12345, LocalDate.now(), List.of(
    new OrderItem("Product 1", 2, 19.99),
    new OrderItem("Product 2", 1, 29.99)
));

System.out.println("Order: " + order);
System.out.println("Total items: " + order.getTotalItems());
System.out.println("Total value: $" + order.getTotalValue());

// 5. Records with Methods
System.out.println("\n=== Records with Methods ===");

Point point1 = new Point(3, 4);
Point point2 = new Point(6, 8);

System.out.println("Distance: " + point1.distance(point2));
System.out.println("Distance from origin: " + point1.distanceFromOrigin());

// 6. Nested Records
System.out.println("\n=== Nested Records ===");

Address address = new Address("123 Main St", "Anytown", "CA", "12345");
Contact contact = new Contact("Bob Smith", address, "555-1234");

System.out.println("Contact: " + contact);
System.out.println("City: " + contact.address().city());

// 7. Records with Generics
System.out.println("\n=== Records with Generics ===");

Pair<String, Integer> pair = new Pair<>("Answer", 42);
System.out.println("Pair: " + pair);

Result<Integer> success = Result.success(100);
Result<Integer> error = Result.error("Process failed");

System.out.println("Success result: " + success);
System.out.println("Error result: " + error);

// Process results
processResult(success);
processResult(error);

// 8. Record inheritance and interfaces
System.out.println("\n=== Record Inheritance and Interfaces ===");

Identified identifiedPerson = new PersonWithId("Charlie Brown", 27, "charlie@example.com", 1001);
System.out.println("Person with ID: " + identifiedPerson);
System.out.println("ID: " + identifiedPerson.id());

// 9. Records in real-world scenarios
System.out.println("\n=== Real-world Usage ===");

// JSON-like data structure using records
UserProfile userProfile = new UserProfile(
    101,
```

```
            "dave_wilson",
            new UserProfile.Name("Dave", "Wilson"),
            List.of("admin", "user"),
            Map.of(
                "theme", "light",
                "language", "en",
                "notifications", "enabled"
            )
        );

        System.out.println("User profile: " + userProfile);

        // Using records for query parameters
        QueryParams params = new QueryParams("products", 10, 20,
            Map.of("category", "electronics", "minPrice", "100"));

        System.out.println("\nQuery: " + buildQuery(params));
    }

    // Helper method for processing Result records
    private static <T> void processResult(Result<T> result) {
        if (result.success()) {
            System.out.println("Processing success value: " + result.value());
        } else {
            System.out.println("Handling error: " + result.error());
        }
    }

    // Helper method for building a query from QueryParams
    private static String buildQuery(QueryParams params) {
        StringBuilder query = new StringBuilder();
        query.append("SELECT * FROM ").append(params.table());

        if (params.filters() != null && !params.filters().isEmpty()) {
            query.append(" WHERE ");

            int i = 0;
            for (Map.Entry<String, String> entry : params.filters().entrySet()) {
                if (i > 0) {
                    query.append(" AND ");
                }
                query.append(entry.getKey()).append(" = '").append(entry.getValue()).append("'");
                i++;
            }
        }

        query.append(" LIMIT ").append(params.limit());
        query.append(" OFFSET ").append(params.offset());

        return query.toString();
    }

    // 1. Basic record definition
    record Person(String name, int age, String email) { }

    // 2. Record with validation using compact constructor
    record Employee(String name, int age, String department) {
        // Compact constructor for validation
        public Employee {
            Objects.requireNonNull(name, "Name cannot be null");
            Objects.requireNonNull(department, "Department cannot be null");

            if (name.isBlank()) {
                throw new IllegalArgumentException("Name cannot be empty");
            }

            if (age < 0) {
                throw new IllegalArgumentException("Age cannot be negative");
            }
        }
    }

    // 3. Records with additional methods
    record Point(int x, int y) {
        // Additional methods
        public double distanceFromOrigin() {
            return Math.sqrt(x * x + y * y);
        }

        public double distance(Point other) {
            int dx = this.x - other.x;
```

```
            int dy = this.y - other.y;
            return Math.sqrt(dx * dx + dy * dy);
        }
    }

    // 4. Record with canonical constructor
    record Order(int orderId, LocalDate orderDate, List<OrderItem> items) {
        // Canonical constructor with additional logic
        public Order(int orderId, LocalDate orderDate, List<OrderItem> items) {
            this.orderId = orderId;
            this.orderDate = orderDate;
            // Create defensive copy of the list
            this.items = List.copyOf(items);
        }

        // Additional methods
        public int getTotalItems() {
            return items.stream().mapToInt(OrderItem::quantity).sum();
        }

        public double getTotalValue() {
            return items.stream()
                    .mapToDouble(item -> item.quantity() * item.unitPrice())
                    .sum();
        }
    }

    record OrderItem(String productName, int quantity, double unitPrice) { }

    // 5. Nested records
    record Address(String street, String city, String state, String zipCode) { }

    record Contact(String name, Address address, String phoneNumber) { }

    // 6. Generic records
    record Pair<K, V>(K first, V second) { }

    // A more complex generic record example
    record Result<T>(T value, String error, boolean success) {
        // Factory methods
        public static <T> Result<T> success(T value) {
            return new Result<>(value, null, true);
        }

        public static <T> Result<T> error(String error) {
            return new Result<>(null, error, false);
        }
    }

    // 7. Records with interfaces
    interface Identified {
        int id();
    }

    // Record implementing an interface
    record PersonWithId(String name, int age, String email, int id) implements Identified { }

    // 8. Complex record structure for JSON-like data
    record UserProfile(
        int id,
        String username,
        Name name,
        List<String> roles,
        Map<String, String> preferences
    ) {
        // Nested record inside another record
        record Name(String first, String last) { }
    }

    // 9. Record for query parameters
    record QueryParams(String table, int limit, int offset, Map<String, String> filters) { }
}
```

Limitations of Records:

1. Records are implicitly final and cannot be extended
2. Records cannot extend other classes (they implicitly extend `java.lang.Record`)
3. Records cannot declare instance fields other than the components in their state description
4. Records cannot have mutable fields (though they can contain references to mutable objects)

5. Records cannot have custom instance field declarations

Common Pitfalls:

- **Mutability**: Records themselves are immutable, but they may contain references to mutable objects.
- **Serialization**: Records are serializable if their components are serializable, but the serialized form may change in future JDK versions.
- **Reflection**: Records have a different structure than regular classes, which might affect reflection-based frameworks.
- **Validation**: For input validation, use compact constructors or canonical constructors.

```
// INCORRECT: Trying to extend a record
record Student(String name, int age) { }
record GraduateStudent(String name, int age, String thesis) extends Student { } // Error: Records cannot extend other classes

// CORRECT: Create a new record with all fields
record GraduateStudent(String name, int age, String thesis) { }

// INCORRECT: Adding instance fields outside the record header
record Rectangle(double width, double height) {
    private double area; // Error: Records cannot have instance fields outside the record components

    public Rectangle {
        this.area = width * height; // Cannot inject fields
    }
}

// CORRECT: Calculate area in a method
record Rectangle(double width, double height) {
    public double area() {
        return width * height;
    }
}

// INCORRECT: Not creating defensive copies for mutable objects
record Team(String name, List<String> members) { }
// Later usage
List<String> players = new ArrayList<>();
players.add("Player 1");
Team team = new Team("TeamA", players);
players.add("Player 2"); // This changes the internal state of the team record!

// CORRECT: Creating defensive copies
record Team(String name, List<String> members) {
    public Team(String name, List<String> members) {
        this.name = name;
        this.members = List.copyOf(members); // Defensive copy
    }
}
```

Pattern Matching for instanceof

What and Why: Pattern matching for `instanceof`, introduced as a preview feature in Java 14 and finalized in Java 16, simplifies the common pattern of type checking and casting with `instanceof` by allowing variable declaration in the same statement.

Real-world Use Cases:

- Processing heterogeneous collections
- Implementing visitor patterns
- Working with class hierarchies
- Handling polymorphic methods
- Parsing and processing structured data

Before (Java 15) / After (Java 16) Comparison:

Before (Java 15):

```
Object obj = getObject();
if (obj instanceof String) {
    String s = (String) obj;
    // Use s
    System.out.println(s.length());
}
```

After (Java 16):

```
Object obj = getObject();
if (obj instanceof String s) {
    // Use s directly
    System.out.println(s.length());
}
```

Comprehensive Example:

```
import java.util.*;

public class PatternMatchingInstanceofExample {
    public static void main(String[] args) {
        // 1. Basic pattern matching with instanceof
        System.out.println("=== Basic Pattern Matching ===");

        Object obj1 = "Hello, Pattern Matching!";
        Object obj2 = Integer.valueOf(42);
        Object obj3 = List.of(1, 2, 3);

        printObjectInfo(obj1);
        printObjectInfo(obj2);
        printObjectInfo(obj3);

        // 2. Flow scoping
        System.out.println("\n=== Flow Scoping ===");

        Object obj = "Flow scoping example";

        // The variable s is in scope only when obj is a String
        if (obj instanceof String s) {
            System.out.println("Length: " + s.length());
        } else {
            // s is not in scope here
            System.out.println("Not a string");
        }

        // Using flow scoping with conditional expressions
        if (!(obj instanceof String s)) {
            System.out.println("Not a string");
        } else {
            System.out.println("String value: " + s);
        }

        // Variable in scope in the "then" statement only when the condition is true
        if (obj instanceof String s && s.length() > 10) {
            System.out.println("Long string: " + s);
        }

        // Variable not in scope in the "then" statement when used with ||
        if (!(obj instanceof String s) || s.length() < 5) {
            System.out.println("Either not a string or a short string");
        }

        // 3. Pattern matching in loops
        System.out.println("\n=== Pattern Matching in Loops ===");

        List<Object> objects = List.of(
            "First string",
            42,
            "Second string",
            List.of("Nested", "List"),
            Map.of("key", "value"),
            "Third string"
        );

        // Count strings using pattern matching
        int stringCount = 0;
        for (Object o : objects) {
            if (o instanceof String s) {
                stringCount++;
                System.out.println("Found string: " + s);
            }
        }
        System.out.println("Total strings: " + stringCount);

        // 4. Nested pattern matching
        System.out.println("\n=== Nested Pattern Matching ===");
```

```
        Object complexObj = Map.of(
            "name", "Product",
            "properties", List.of("Property1", "Property2", "Property3")
        );

        processComplexObject(complexObj);

        // 5. Real-world example: Shape hierarchy
        System.out.println("\n=== Shape Hierarchy Example ===");

        List<Shape> shapes = List.of(
            new Circle(5.0),
            new Rectangle(4.0, 6.0),
            new Triangle(3.0, 4.0, 5.0),
            new Square(4.0)
        );

        double totalArea = 0;
        for (Shape shape : shapes) {
            totalArea += calculateArea(shape);
        }

        System.out.println("Total area of all shapes: " + totalArea);

        // 6. Processing a tree structure
        System.out.println("\n=== Tree Structure Example ===");

        TreeNode root = new TreeNode("Root",
            List.of(
                new TreeNode("Child1", List.of()),
                new TreeNode("Child2",
                    List.of(
                        new TreeNode("Grandchild1", List.of()),
                        new TreeNode("Grandchild2", List.of())
                    )
                ),
                new TreeNode("Child3", List.of())
            )
        );

        // Print tree structure
        printTree(root, 0);

        // Count nodes
        System.out.println("Total nodes: " + countNodes(root));
    }

    // Example 1: Basic usage with different types
    private static void printObjectInfo(Object obj) {
        if (obj instanceof String s) {
            System.out.println("String of length " + s.length() + ": \"" + s + "\"");
        } else if (obj instanceof Integer i) {
            System.out.println("Integer with value: " + i);
        } else if (obj instanceof List<?> list) {
            System.out.println("List with " + list.size() + " elements: " + list);
        } else {
            System.out.println("Unknown type: " + obj.getClass().getName());
        }
    }

    // Example 4: Processing complex object structures
    private static void processComplexObject(Object obj) {
        if (obj instanceof Map<?, ?> map) {
            System.out.println("Processing map with " + map.size() + " entries");

            // Check for specific key and value type
            if (map.get("name") instanceof String name) {
                System.out.println("Name: " + name);
            }

            // Navigate deeper in the structure
            if (map.get("properties") instanceof List<?> properties) {
                System.out.println("Properties (" + properties.size() + "):");

                for (Object property : properties) {
                    if (property instanceof String propName) {
                        System.out.println("- " + propName);
                    }
                }
            }
        }
```

```
    }

    // Example 5: Shape hierarchy
    interface Shape { }

    static class Circle implements Shape {
        private final double radius;

        Circle(double radius) {
            this.radius = radius;
        }

        public double getRadius() {
            return radius;
        }
    }

    static class Rectangle implements Shape {
        private final double width;
        private final double height;

        Rectangle(double width, double height) {
            this.width = width;
            this.height = height;
        }

        public double getWidth() {
            return width;
        }

        public double getHeight() {
            return height;
        }
    }

    static class Square extends Rectangle {
        Square(double side) {
            super(side, side);
        }
    }

    static class Triangle implements Shape {
        private final double a, b, c; // sides

        Triangle(double a, double b, double c) {
            this.a = a;
            this.b = b;
            this.c = c;
        }

        public double getA() { return a; }
        public double getB() { return b; }
        public double getC() { return c; }
    }

    // Calculate area using pattern matching
    private static double calculateArea(Shape shape) {
        if (shape instanceof Circle c) {
            return Math.PI * c.getRadius() * c.getRadius();
        } else if (shape instanceof Rectangle r) {
            return r.getWidth() * r.getHeight();
        } else if (shape instanceof Triangle t) {
            // Heron's formula
            double s = (t.getA() + t.getB() + t.getC()) / 2;
            return Math.sqrt(s * (s - t.getA()) * (s - t.getB()) * (s - t.getC()));
        } else {
            throw new IllegalArgumentException("Unknown shape type");
        }
    }

    // Example 6: Tree structure
    static class TreeNode {
        private final String name;
        private final List<TreeNode> children;

        TreeNode(String name, List<TreeNode> children) {
            this.name = name;
            this.children = children;
        }

        public String getName() { return name; }
```

```
        public List<TreeNode> getChildren() { return children; }
    }

    // Print tree recursively with pattern matching
    private static void printTree(Object node, int depth) {
        if (node instanceof TreeNode n) {
            System.out.println("  ".repeat(depth) + "- " + n.getName());

            for (Object child : n.getChildren()) {
                printTree(child, depth + 1);
            }
        }
    }

    // Count nodes recursively with pattern matching
    private static int countNodes(Object node) {
        if (node instanceof TreeNode n) {
            int count = 1; // Count this node

            for (Object child : n.getChildren()) {
                count += countNodes(child);
            }

            return count;
        } else {
            return 0;
        }
    }
}
```

Scope and Control Flow: Pattern variables are subject to "flow scoping" – they are in scope where the compiler can determine they are definitely assigned, following the control flow of the program.

Common Pitfalls:

- **Scope Confusion**: Understanding exactly where pattern variables are in scope can be tricky.
- **Complex Boolean Expressions**: In expressions with `&&` and `||`, the scope rules can be subtle.
- **Overreliance**: Not every `instanceof`/cast combination should be replaced – sometimes explicit casting is clearer.

```
// INCORRECT: Misunderstanding scope with logical operators
if (obj instanceof String s || s.length() > 10) { // Error: s not in scope for the right side of ||
    // ...
}

// CORRECT: Restructure to ensure proper scoping
if (obj instanceof String s) {
    if (s.length() > 10) {
        // ...
    }
}

// ALSO CORRECT: Use && when the right side depends on the left side being true
if (obj instanceof String s && s.length() > 10) {
    // ...
}

// INCORRECT: Attempting to use pattern variable outside its scope
if (!(obj instanceof String s)) {
    System.out.println("Not a string");
} else {
    System.out.println("Length: " + s.length()); // s is in scope here
}
System.out.println(s); // Error: s is not in scope here

// CORRECT: Only use pattern variables within their scope
if (obj instanceof String s) {
    System.out.println("Length: " + s.length()); // s is in scope here
}
```

Helpful NullPointerExceptions

What and Why: Introduced in Java 14, helpful NullPointerExceptions (NPEs) provide more detailed error messages that precisely identify which variable was null in an expression that caused the NPE. This significantly improves debugging by eliminating guesswork about the null source.

Real-world Use Cases:

- Debugging complex expressions

- Identifying causes of NPEs in production logs
- Training new developers
- Troubleshooting code in unfamiliar codebases

Before (Java 13) / After (Java 14) Comparison:

Before (Java 13):

```
Exception in thread "main" java.lang.NullPointerException
    at com.example.MyClass.method(MyClass.java:15)
    at com.example.MyClass.main(MyClass.java:5)
```

After (Java 14):

```
Exception in thread "main" java.lang.NullPointerException: Cannot invoke "String.length()" because "customer.address.street" is
null
    at com.example.MyClass.method(MyClass.java:15)
    at com.example.MyClass.main(MyClass.java:5)
```

Example:

```
public class HelpfulNPEExample {

    public static void main(String[] args) {
        // 1. Simple null pointer
        System.out.println("=== Simple Null Pointer ===");

        try {
            String str = null;
            int length = str.length(); // NPE
        } catch (NullPointerException e) {
            System.out.println("Error 1: " + e.getMessage());
            // Output: Cannot invoke "String.length()" because "str" is null
        }

        // 2. Nested property access
        System.out.println("\n=== Nested Property Access ===");

        try {
            Person person = new Person("John", null); // Null address
            String city = person.getAddress().getCity(); // NPE
        } catch (NullPointerException e) {
            System.out.println("Error 2: " + e.getMessage());
            // Output: Cannot invoke "HelpfulNPEExample$Address.getCity()" because the return value of
"HelpfulNPEExample$Person.getAddress()" is null
        }

        // 3. Method call chain
        System.out.println("\n=== Method Call Chain ===");

        try {
            Person person = getPerson(); // Returns null
            String name = person.getName().toUpperCase(); // NPE
        } catch (NullPointerException e) {
            System.out.println("Error 3: " + e.getMessage());
            // Output: Cannot invoke "HelpfulNPEExample$Person.getName()" because "person" is null
        }

        // 4. Array access
        System.out.println("\n=== Array Access ===");

        try {
            String[] names = null;
            String firstNameUpper = names[0].toUpperCase(); // NPE
        } catch (NullPointerException e) {
            System.out.println("Error 4: " + e.getMessage());
            // Output: Cannot load from object array because "names" is null
        }

        // 5. Array element null
        System.out.println("\n=== Array Element Null ===");

        try {
            String[] names = new String[3]; // All elements initialized to null
            String firstNameUpper = names[0].toUpperCase(); // NPE
```

```
        } catch (NullPointerException e) {
            System.out.println("Error 5: " + e.getMessage());
            // Output: Cannot invoke "String.toUpperCase()" because "names[0]" is null
        }

        // 6. Complex expressions
        System.out.println("\n=== Complex Expression ===");

        try {
            Person person = new Person("Alice", new Address("123 Main St", null)); // Null city
            int cityLength = person.getAddress().getCity().length(); // NPE
        } catch (NullPointerException e) {
            System.out.println("Error 6: " + e.getMessage());
            // Output: Cannot invoke "String.length()" because "person.getAddress().getCity()" is null
        }

        // 7. Method parameters
        System.out.println("\n=== Method Parameters ===");

        try {
            processCity(new Person("Bob", null)); // Null address
        } catch (NullPointerException e) {
            System.out.println("Error 7: " + e.getMessage());
        }
    }

    private static Person getPerson() {
        return null;
    }

    private static void processCity(Person person) {
        String city = person.getAddress().getCity(); // NPE
    }

    // Helper classes
    static class Person {
        private final String name;
        private final Address address;

        Person(String name, Address address) {
            this.name = name;
            this.address = address;
        }

        public String getName() {
            return name;
        }

        public Address getAddress() {
            return address;
        }
    }

    static class Address {
        private final String street;
        private final String city;

        Address(String street, String city) {
            this.street = street;
            this.city = city;
        }

        public String getStreet() {
            return street;
        }

        public String getCity() {
            return city;
        }
    }
}
```

How It Works:

1. The JVM analyzes the bytecode and source information when an NPE occurs
2. It identifies exactly which reference was null
3. It constructs a descriptive message including the variable name and operation that failed
4. The enhanced message is included in the exception

Limitations:

- Works only with standard NullPointerExceptions thrown by the JVM, not those explicitly created with `new NullPointerException()`
- Might not work in highly optimized code
- May not be helpful with lambda expressions where variable names are synthesized
- Decompiled code or code without proper debug information will have less helpful messages

This feature is enabled by default but can be disabled with the JVM flag `-XX:-ShowCodeDetailsInExceptionMessages`.

Java 17 (LTS)

Java 17, released in September 2021, is an LTS (Long-Term Support) release that finalized several features that had been in preview in earlier releases.

Sealed Classes

What and Why: Sealed classes, introduced as a preview feature in Java 15 and finalized in Java 17, allow class authors to restrict which other classes can extend or implement them. This creates a closed set of subtypes, offering a middle ground between traditional inheritance and enums.

Real-world Use Cases:

- Domain modeling with a fixed set of implementations
- Creating type-safe API responses
- Implementing algebraic data types
- Pattern matching optimization
- State machine modeling

Before (Java 16) / After (Java 17) Comparison:

Before (Java 16):

```
// No built-in way to restrict which classes can extend Shape
public abstract class Shape {
    // Javadoc might say: "This class should only be extended by Circle, Rectangle, and Triangle"
    // but nothing prevents someone from extending it
}

public class Circle extends Shape { }
public class Rectangle extends Shape { }
public class Triangle extends Shape { }
public class Hexagon extends Shape { } // Nothing prevents this
```

After (Java 17):

```
// Explicitly define permitted subclasses
public sealed class Shape permits Circle, Rectangle, Triangle {
    // Only the listed classes can extend Shape
}

public final class Circle extends Shape { }
public final class Rectangle extends Shape { }
public final class Triangle extends Shape { }
public class Hexagon extends Shape { } // Compilation error
```

Comprehensive Example:

```
import java.util.List;
import java.util.function.Function;

public class SealedClassesExample {

    public static void main(String[] args) {
        // 1. Basic usage of sealed classes
        System.out.println("=== Basic Sealed Class Usage ===");

        Shape circle = new Circle(5.0);
        Shape rectangle = new Rectangle(4.0, 6.0);
        Shape triangle = new Triangle(3.0, 4.0, 5.0);

        List<Shape> shapes = List.of(circle, rectangle, triangle);

        double totalArea = 0;
        for (Shape shape : shapes) {
            totalArea += calculateArea(shape);
        }
```

```
        System.out.println("Total area: " + totalArea);

        // 2. Pattern matching with sealed classes (using instanceof for now)
        System.out.println("\n=== Pattern Matching with Sealed Classes ===");

        for (Shape shape : shapes) {
            describeShape(shape);
        }

        // 3. Using sealed interfaces
        System.out.println("\n=== Sealed Interfaces ===");

        List<Expr> expressions = List.of(
            new Expr.Constant(42),
            new Expr.Variable("x"),
            new Expr.Sum(new Expr.Constant(10), new Expr.Variable("y")),
            new Expr.Product(new Expr.Constant(2), new Expr.Variable("z"))
        );

        // Evaluate expressions with different variable values
        var evaluator = new ExpressionEvaluator(Map.of("x", 5, "y", 7, "z", 3));

        for (Expr expr : expressions) {
            System.out.println(expr + " = " + evaluator.evaluate(expr));
        }

        // 4. Implementing visitor pattern with sealed classes
        System.out.println("\n=== Visitor Pattern with Sealed Classes ===");

        ExprVisitor<String> printer = new ExprVisitor<>() {
            @Override
            public String visitConstant(Expr.Constant constant) {
                return Double.toString(constant.value());
            }

            @Override
            public String visitVariable(Expr.Variable variable) {
                return variable.name();
            }

            @Override
            public String visitSum(Expr.Sum sum) {
                return "(" + visit(sum.left()) + " + " + visit(sum.right()) + ")";
            }

            @Override
            public String visitProduct(Expr.Product product) {
                return visit(product.left()) + " * " + visit(product.right());
            }
        };

        for (Expr expr : expressions) {
            System.out.println("Expression: " + printer.visit(expr));
        }

        // 5. Non-sealed subclass examples
        System.out.println("\n=== Non-sealed Subclasses ===");

        Vehicle car = new Car("Toyota", "Corolla");
        Vehicle bike = new Bicycle("Mountain");
        CustomCar customCar = new CustomCar("Tesla", "Model S", "Custom Paint");

        List<Vehicle> vehicles = List.of(car, bike, customCar);

        for (Vehicle vehicle : vehicles) {
            System.out.println(vehicle.getDescription());
        }

        // 6. Type patterns with sealed hierarchies (Pattern matching)
        System.out.println("\n=== Type Pattern with Sealed Types ===");

        for (Expr expr : expressions) {
            String result = switch (expr) {
                case Expr.Constant c -> "Constant: " + c.value();
                case Expr.Variable v -> "Variable: " + v.name();
                case Expr.Sum s -> "Sum of " + s.left() + " and " + s.right();
                case Expr.Product p -> "Product of " + p.left() + " and " + p.right();
                // No default needed since Expr is sealed and all cases are covered
            };
            System.out.println(result);
```

```
        }
    }

    // Calculate area based on shape type
    private static double calculateArea(Shape shape) {
        if (shape instanceof Circle c) {
            return Math.PI * c.radius() * c.radius();
        } else if (shape instanceof Rectangle r) {
            return r.width() * r.height();
        } else if (shape instanceof Triangle t) {
            // Heron's formula
            double s = (t.a() + t.b() + t.c()) / 2;
            return Math.sqrt(s * (s - t.a()) * (s - t.b()) * (s - t.c()));
        } else {
            throw new IllegalArgumentException("Unknown shape: " + shape.getClass());
        }
    }

    // Describe shape using pattern matching
    private static void describeShape(Shape shape) {
        if (shape instanceof Circle c) {
            System.out.println("Circle with radius " + c.radius());
        } else if (shape instanceof Rectangle r) {
            if (r.width() == r.height()) {
                System.out.println("Square with side " + r.width());
            } else {
                System.out.println("Rectangle with width " + r.width() + " and height " + r.height());
            }
        } else if (shape instanceof Triangle t) {
            System.out.println("Triangle with sides " + t.a() + ", " + t.b() + ", and " + t.c());
        }
    }

    // 1. Basic sealed class with record subclasses
    sealed abstract class Shape permits Circle, Rectangle, Triangle {
        // Common methods or fields for all shapes could go here
    }

    record Circle(double radius) implements Shape { }
    record Rectangle(double width, double height) implements Shape { }
    record Triangle(double a, double b, double c) implements Shape { }

    // 2. Sealed interface for expressions
    sealed interface Expr permits Expr.Constant, Expr.Variable, Expr.Sum, Expr.Product {
        // Nested classes/records as permitted implementations
        record Constant(double value) implements Expr { }
        record Variable(String name) implements Expr { }
        record Sum(Expr left, Expr right) implements Expr { }
        record Product(Expr left, Expr right) implements Expr { }
    }

    // Visitor interface for expressions
    interface ExprVisitor<T> {
        T visitConstant(Expr.Constant constant);
        T visitVariable(Expr.Variable variable);
        T visitSum(Expr.Sum sum);
        T visitProduct(Expr.Product product);

        default T visit(Expr expr) {
            return switch (expr) {
                case Expr.Constant c -> visitConstant(c);
                case Expr.Variable v -> visitVariable(v);
                case Expr.Sum s -> visitSum(s);
                case Expr.Product p -> visitProduct(p);
            };
        }
    }

    // Expression evaluator using the visitor pattern
    class ExpressionEvaluator {
        private final Map<String, Double> variables;

        ExpressionEvaluator(Map<String, Double> variables) {
            this.variables = Map.copyOf(variables);
        }

        double evaluate(Expr expr) {
            return switch (expr) {
                case Expr.Constant c -> c.value();
                case Expr.Variable v -> variables.getOrDefault(v.name(), 0.0);
                case Expr.Sum s -> evaluate(s.left()) + evaluate(s.right());
```

```
                case Expr.Product p -> evaluate(p.left()) * evaluate(p.right());
            };
        }
    }

    // 3. Sealed class with non-sealed subclass
    sealed abstract class Vehicle permits Car, Bicycle, Truck {
        protected final String type;

        protected Vehicle(String type) {
            this.type = type;
        }

        public abstract String getDescription();
    }

    // Final class - cannot be extended
    final class Bicycle extends Vehicle {
        private final String bikeType;

        public Bicycle(String bikeType) {
            super("Bicycle");
            this.bikeType = bikeType;
        }

        @Override
        public String getDescription() {
            return "A " + bikeType + " bicycle";
        }
    }

    // Non-sealed class - can be extended by any class
    non-sealed class Car extends Vehicle {
        protected final String make;
        protected final String model;

        public Car(String make, String model) {
            super("Car");
            this.make = make;
            this.model = model;
        }

        @Override
        public String getDescription() {
            return "A " + make + " " + model + " car";
        }
    }

    // Extending a non-sealed class (allowed)
    class CustomCar extends Car {
        private final String customization;

        public CustomCar(String make, String model, String customization) {
            super(make, model);
            this.customization = customization;
        }

        @Override
        public String getDescription() {
            return super.getDescription() + " with " + customization;
        }
    }

    // Sealed by default (permits nothing) - can't be extended
    final class Truck extends Vehicle {
        private final String capacity;

        public Truck(String capacity) {
            super("Truck");
            this.capacity = capacity;
        }

        @Override
        public String getDescription() {
            return "A truck with " + capacity + " capacity";
        }
    }
}
```

Defining Sealed Classes:

There are three steps to using sealed classes:

1. Declare a class or interface with the `sealed` modifier.
2. Specify the permitted subclasses with the `permits` clause.
3. Ensure each permitted subclass has one of these modifiers:
    - `final`: The class cannot be extended further.
    - `sealed`: The class is also sealed and specifies its own permitted subclasses.
    - `non-sealed`: The class can be extended by any class.

Common Pitfalls:

- **Missing Required Modifiers**: All direct subclasses of a sealed class must be declared as `final`, `sealed`, or `non-sealed`.
- **Location Constraints**: By default, permitted classes must be in the same module (or package if in an unnamed module) as the sealed class, unless explicitly named in the `permits` clause.
- **Exhaustiveness Checks**: The main benefit of sealed classes comes with pattern matching, which may not be fully utilized until pattern matching for switch is finalized.

```
// INCORRECT: Missing required modifier
sealed class Animal permits Dog, Cat { }
class Dog extends Animal { } // Error: Class must be final, sealed, or non-sealed
class Cat extends Animal { } // Error: Class must be final, sealed, or non-sealed

// CORRECT: Adding required modifiers
sealed class Animal permits Dog, Cat { }
final class Dog extends Animal { }
final class Cat extends Animal { }

// INCORRECT: Trying to extend a class not in the permits clause
sealed class Shape permits Circle, Rectangle { }
final class Circle extends Shape { }
final class Rectangle extends Shape { }
final class Triangle extends Shape { } // Error: Not in the permits clause

// CORRECT: Update the permits clause
sealed class Shape permits Circle, Rectangle, Triangle { }
final class Circle extends Shape { }
final class Rectangle extends Shape { }
final class Triangle extends Shape { }
```

Pattern Matching for switch (Preview)

What and Why: Pattern matching for switch, initially previewed in Java 17, extends switch expressions and statements to work with patterns rather than just constants. This allows for more powerful type checking and data extraction in a single statement.

Note: While this was a preview feature in Java 17, it was finalized in Java 21. The following examples show the syntax as it evolved.

Real-world Use Cases:

- Processing heterogeneous collections
- Implementing type-based dispatch
- Working with algebraic data types
- Data validation and conversion
- Command processing

Before (Java 16) / After (Java 17 Preview) Comparison:

Before (Java 16):

```
Object obj = getSomeObject();
if (obj instanceof String s) {
    System.out.println("String of length " + s.length());
} else if (obj instanceof Integer i) {
    System.out.println("Integer with value " + i);
} else if (obj instanceof List<?> list && !list.isEmpty()) {
    System.out.println("Non-empty list: " + list);
} else {
    System.out.println("Something else");
}
```

After (Java 17 Preview):

```
Object obj = getSomeObject();
switch (obj) {
    case String s -> System.out.println("String of length " + s.length());
```

```
    case Integer i -> System.out.println("Integer with value " + i);
    case List<?> list && !list.isEmpty() -> System.out.println("Non-empty list: " + list);
    default -> System.out.println("Something else");
}
```

Example:

```
import java.time.LocalDate;
import java.util.*;

public class PatternMatchingSwitchExample {
    public static void main(String[] args) {
        // Example objects to test
        List<Object> testObjects = List.of(
            "Hello, Pattern Matching!",
            42,
            List.of(1, 2, 3),
            Map.of("key", "value"),
            new int[] {1, 2, 3},
            null,
            LocalDate.now()
        );

        // Test pattern matching
        System.out.println("=== Pattern Matching on Different Types ===");
        for (Object obj : testObjects) {
            String result = formatObject(obj);
            System.out.println("Object: " + obj + " -> " + result);
        }

        // Example with nested patterns
        System.out.println("\n=== Nested Pattern Matching ===");

        Object nestedObj = Map.of(
            "name", "Product",
            "details", Map.of(
                "price", 29.99,
                "inStock", true
            )
        );

        processNestedObject(nestedObj);

        // Example with sealed types
        System.out.println("\n=== Matching on Sealed Types ===");

        List<Shape> shapes = List.of(
            new Circle(5.0),
            new Rectangle(4.0, 6.0),
            new Rectangle(5.0, 5.0), // Square
            new Triangle(3.0, 4.0, 5.0)
        );

        for (Shape shape : shapes) {
            System.out.println(describeShape(shape));
        }

        // Example with guards
        System.out.println("\n=== Pattern Matching with Guards ===");

        List<Integer> numbers = List.of(-5, 0, 3, 10, 42);

        for (Integer number : numbers) {
            System.out.println("Number " + number + " is " + categorizeNumber(number));
        }
    }

    // Example 1: Basic pattern matching for switch
    static String formatObject(Object obj) {
        return switch (obj) {
            case String s -> "String with length " + s.length();
            case Integer i -> "Integer with value " + i;
            case Long l -> "Long with value " + l;
            case Double d -> "Double with value " + d;
            case List<?> list -> "List with " + list.size() + " elements";
            case Map<?, ?> map -> "Map with " + map.size() + " entries";
            case int[] arr -> "Int array with length " + arr.length;
            case null -> "Null object";
            default -> "Unrecognized object of type " + obj.getClass().getName();
```

```
        };
    }

    // Example 2: Nested pattern matching
    static void processNestedObject(Object obj) {
        switch (obj) {
            case Map<?, ?> map -> {
                System.out.println("Processing map with " + map.size() + " entries");

                // Check for nested structures
                if (map.get("details") instanceof Map<?, ?> details) {
                    System.out.println("Details found:");
                    details.forEach((k, v) -> System.out.println("  " + k + ": " + v));
                }

                // Process specific entries
                if (map.get("name") instanceof String name) {
                    System.out.println("Name: " + name);
                }
            }
            default -> System.out.println("Not a recognized format");
        }
    }

    // Example 3: Matching on sealed types
    static String describeShape(Shape shape) {
        return switch (shape) {
            // Using type patterns
            case Circle c -> "Circle with radius " + c.radius();

            // Using guards for refinement
            case Rectangle r && r.width() == r.height() ->
                "Square with side " + r.width();
            case Rectangle r ->
                "Rectangle with width " + r.width() + " and height " + r.height();

            // Normal case
            case Triangle t -> "Triangle with sides " + t.a() + ", " + t.b() + ", " + t.c();

            // No default needed with sealed types - if a subtype is added,
            // the compiler will detect the non-exhaustive switch
        };
    }

    // Example 4: Pattern matching with guards
    static String categorizeNumber(Integer num) {
        return switch (num) {
            case null -> "null";
            case Integer i && i < 0 -> "negative";
            case Integer i && i == 0 -> "zero";
            case Integer i && i > 0 && i < 10 -> "small positive";
            case Integer i && i >= 10 -> "large positive";
            // With proper preview feature usage, this would not need a default
            default -> "impossible";
        };
    }

    // Sealed class hierarchy for Example 3
    sealed interface Shape permits Circle, Rectangle, Triangle { }
    record Circle(double radius) implements Shape { }
    record Rectangle(double width, double height) implements Shape { }
    record Triangle(double a, double b, double c) implements Shape { }
}
```

Key Features:

1. **Type Patterns**: `case String s ->`
2. **Pattern Guards**: `case String s && s.length() > 5 ->`
3. **Null Handling**: `case null ->`
4. **Exhaustiveness**: With sealed types, all possible cases must be covered
5. **Flow Scoping**: The pattern variable is only in scope within the branch

Common Pitfalls:

- **Different Semantics**: Switch pattern matching doesn't "fall through" by default, unlike traditional switch statements with the colon syntax.
- **Null Handling**: Pattern matching switch expressions automatically handle null values, unlike traditional switch statements.
- **Exhaustiveness Requirements**: You need to handle all potential cases, either with explicit cases or a default clause.

```
// INCORRECT: Forgetting about null
Object obj = null;
switch (obj) {
    case String s -> System.out.println(s.toUpperCase());
    case Integer i -> System.out.println(i * 2);
    // Missing null case and default
}

// CORRECT: Include null handling
switch (obj) {
    case null -> System.out.println("Null input");
    case String s -> System.out.println(s.toUpperCase());
    case Integer i -> System.out.println(i * 2);
    default -> System.out.println("Unhandled type");
}

// INCORRECT: Trying to access pattern variable outside its scope
switch (obj) {
    case String s -> s.length(); // s is in scope
    default -> s.length(); // Error: s is not in scope here
}
```

Enhanced Random Number Generators

What and Why: Java 17 enhanced the random number generation API with the introduction of new random number generator interfaces and implementations, providing better performance, more options for randomization, and improved handling of random streams.

Real-world Use Cases:

- Simulations and modeling
- Game development
- Cryptographic applications
- Statistical sampling
- Testing with randomized inputs

Before (Java 16) / After (Java 17) Comparison:

Before (Java 16):

```
// Basic random number generation
Random random = new Random();
int randomInt = random.nextInt(100); // 0-99

// Thread-safe random with potentially lower contention
SecureRandom secureRandom = new SecureRandom();
int secureRandomInt = secureRandom.nextInt(100);

// For reproducible sequences
Random seededRandom = new Random(12345L);
int seededRandomInt = seededRandom.nextInt(100);
```

After (Java 17):

```
// Basic random number generator
RandomGenerator random = RandomGenerator.getDefault();
int randomInt = random.nextInt(100); // 0-99

// Specific algorithm with good statistical properties
RandomGenerator xoroshiro = RandomGenerator.of("Xoroshiro128PlusPlus");
int xoroshiroInt = xoroshiro.nextInt(100);

// Splitting for parallel streams
SplittableRandomGenerator splittable = SplittableRandomGenerator.of("L128X256MixRandom");
SplittableRandomGenerator split1 = splittable.split();
SplittableRandomGenerator split2 = splittable.split();
```

Comprehensive Example:

```
import java.util.List;
import java.util.random.*;
import java.util.stream.Collectors;
import java.util.stream.IntStream;
```

```
public class EnhancedRandomExample {
    public static void main(String[] args) {
        // 1. Basic RandomGenerator usage
        System.out.println("=== Basic RandomGenerator ===");

        // Get the default generator
        RandomGenerator defaultRandom = RandomGenerator.getDefault();
        System.out.println("Default algorithm: " + defaultRandom.getClass().getSimpleName());

        // Generate some random values
        System.out.println("Random int: " + defaultRandom.nextInt());
        System.out.println("Random int (0-99): " + defaultRandom.nextInt(100));
        System.out.println("Random long: " + defaultRandom.nextLong());
        System.out.println("Random double (0.0-1.0): " + defaultRandom.nextDouble());
        System.out.println("Random boolean: " + defaultRandom.nextBoolean());

        // 2. Different algorithms
        System.out.println("\n=== Random Number Algorithms ===");

        // List all available algorithms
        List<String> algorithms = RandomGenerator.all()
                .map(provider -> provider.name())
                .sorted()
                .collect(Collectors.toList());

        System.out.println("Available algorithms: " + algorithms);

        // Create generators with specific algorithms
        RandomGenerator xoshiro = RandomGenerator.of("Xoshiro256PlusPlus");
        RandomGenerator l64 = RandomGenerator.of("L64X128StarStarRandom");

        System.out.println("\nXoshiro256PlusPlus vs L64X128StarStarRandom:");
        for (int i = 0; i < 5; i++) {
            System.out.printf("Xoshiro: %d, L64: %d%n",
                    xoshiro.nextInt(100), l64.nextInt(100));
        }

        // 3. Creating seeded generators
        System.out.println("\n=== Seeded Generators ===");

        // Create two generators with the same seed
        long seed = 12345L;
        RandomGenerator seeded1 = RandomGenerator.of("Xoshiro256PlusPlus").create(seed);
        RandomGenerator seeded2 = RandomGenerator.of("Xoshiro256PlusPlus").create(seed);

        System.out.println("Same sequence from identical seeds:");
        for (int i = 0; i < 5; i++) {
            int value1 = seeded1.nextInt(100);
            int value2 = seeded2.nextInt(100);
            System.out.printf("Generator 1: %d, Generator 2: %d, Equal: %b%n",
                    value1, value2, value1 == value2);
        }

        // 4. Splittable random generators
        System.out.println("\n=== Splittable Random Generators ===");

        SplittableRandomGenerator splittable = SplittableRandomGenerator.of("L128X256MixRandom");

        // Create two independent streams from the original
        SplittableRandomGenerator split1 = splittable.split();
        SplittableRandomGenerator split2 = splittable.split();

        System.out.println("Original vs splits:");
        for (int i = 0; i < 3; i++) {
            System.out.printf("Original: %d, Split1: %d, Split2: %d%n",
                    splittable.nextInt(100), split1.nextInt(100), split2.nextInt(100));
        }

        // 5. JumpableRandomGenerator
        System.out.println("\n=== Jumpable Random Generators ===");

        JumpableRandomGenerator jumpable = JumpableRandomGenerator.of("Xoshiro256PlusPlus");

        System.out.println("Original sequence:");
        for (int i = 0; i < 3; i++) {
            System.out.println(jumpable.nextInt(100));
        }

        // Jump ahead in the sequence - equivalent to many nextInt() calls
        jumpable.jump();
```

```
        System.out.println("After jump:");
        for (int i = 0; i < 3; i++) {
            System.out.println(jumpable.nextInt(100));
        }

        // Long jump - even further ahead
        jumpable.longJump();

        System.out.println("After long jump:");
        for (int i = 0; i < 3; i++) {
            System.out.println(jumpable.nextInt(100));
        }

        // 6. LeapableRandomGenerator
        System.out.println("\n=== Leapable Random Generators ===");

        LeapableRandomGenerator leapable = LeapableRandomGenerator.of("Xoshiro256PlusPlus");

        System.out.println("Original leapable sequence:");
        for (int i = 0; i < 3; i++) {
            System.out.println(leapable.nextInt(100));
        }

        // Create a new generator that's far ahead in the sequence
        LeapableRandomGenerator leaped = leapable.leap(1_000_000);

        System.out.println("After leaping 1,000,000 steps:");
        for (int i = 0; i < 3; i++) {
            System.out.println(leaped.nextInt(100));
        }

        // 7. ArbitrarilyJumpableRandomGenerator
        System.out.println("\n=== Arbitrarily Jumpable Generators ===");

        ArbitrarilyJumpableRandomGenerator arbitrary =
                ArbitrarilyJumpableRandomGenerator.of("Xoshiro256PlusPlus");

        System.out.println("Original sequence:");
        for (int i = 0; i < 3; i++) {
            System.out.println(arbitrary.nextInt(100));
        }

        // Jump ahead by a specific power of 2
        arbitrary.jumpPowerOfTwo(10); // Jump ahead by 2^10 steps

        System.out.println("After jumping 2^10 steps:");
        for (int i = 0; i < 3; i++) {
            System.out.println(arbitrary.nextInt(100));
        }

        // 8. Using with streams for parallel processing
        System.out.println("\n=== Parallel Streams ===");

        // Create a splittable generator for parallel streams
        SplittableRandomGenerator parallelGen = SplittableRandomGenerator.of("L128X256MixRandom");

        // Use in a parallel stream
        List<Integer> randomNumbers = parallelGen.ints(0, 1000)
                .parallel()
                .limit(10)
                .boxed()
                .collect(Collectors.toList());

        System.out.println("Random numbers from parallel stream: " + randomNumbers);

        // 9. Performance comparison
        System.out.println("\n=== Performance Comparison ===");

        // Generate 10 million random numbers with different generators
        int iterations = 10_000_000;

        RandomGenerator standardRandom = RandomGenerator.of("Random");
        RandomGenerator xoshiro256 = RandomGenerator.of("Xoshiro256PlusPlus");
        RandomGenerator l128 = RandomGenerator.of("L128X128MixRandom");

        // Test each generator
        benchmarkGenerator("Standard Random", standardRandom, iterations);
        benchmarkGenerator("Xoshiro256PlusPlus", xoshiro256, iterations);
        benchmarkGenerator("L128X128MixRandom", l128, iterations);
    }
```

```
    // Helper method to benchmark random generators
    private static void benchmarkGenerator(String name, RandomGenerator generator, int iterations) {
        long start = System.nanoTime();
        double sum = 0;

        for (int i = 0; i < iterations; i++) {
            sum += generator.nextDouble();
        }

        long end = System.nanoTime();
        double timeMs = (end - start) / 1_000_000.0;

        System.out.printf("%s: %.2f ms (avg: %.6f)%n", name, timeMs, sum / iterations);
    }
}
```

Key Interfaces in the New Random API:

1. `RandomGenerator`: The base interface for all random number generators.
2. `SplittableRandomGenerator`: Can split into multiple independent generators, useful for parallel streams.
3. `JumpableRandomGenerator`: Can jump ahead in the sequence by a large, fixed number of steps.
4. `LeapableRandomGenerator`: Can create a new generator advanced by a specific number of steps.
5. `ArbitrarilyJumpableRandomGenerator`: Can jump ahead by an arbitrary power of 2 steps.

Common Pitfalls:

- **Algorithm Selection**: Different algorithms have different performance and statistical properties.
- **Predictability**: Standard random generators are not suitable for security-critical applications.
- **Thread Safety**: Some generators are not thread-safe by default.
- **State Size**: The period of a generator depends on its state size.

```
// INCORRECT: Using the default generator for cryptographic purposes
RandomGenerator insecure = RandomGenerator.getDefault();
byte[] key = new byte[32];
insecure.nextBytes(key); // Not suitable for cryptographic keys

// CORRECT: Use SecureRandom for security needs
SecureRandom secure = new SecureRandom();
byte[] secureKey = new byte[32];
secure.nextBytes(secureKey);

// INCORRECT: Assuming all generators have the same properties
RandomGenerator any = RandomGenerator.all().findAny().get().create();
// The above could give you any generator with unknown characteristics

// CORRECT: Select a specific algorithm with known properties
RandomGenerator specific = RandomGenerator.of("Xoshiro256PlusPlus");
```

Java 18-20 (Selected Features)

Java 18-20 were non-LTS releases with several important features, some of which were finalized in Java 21.

UTF-8 by Default

What and Why: Java 18 made UTF-8 the default charset for APIs that depend on the default charset when no explicit charset is specified. This ensures consistent encoding behavior across different platforms and operating systems.

Real-world Use Cases:

- Cross-platform applications
- Internationalization
- File I/O operations
- Network communication
- Text processing

Before (Java 17) / After (Java 18) Comparison:

Before (Java 17):

```
// The default charset was platform-dependent
Charset defaultCharset = Charset.defaultCharset(); // Might be different on Windows vs. Linux/MacOS

// Reading a file without specifying charset could produce different results on different platforms
String content = new String(Files.readAllBytes(path)); // Uses platform default
```

After (Java 18):

```
// The default charset is now UTF-8 on all platforms
Charset defaultCharset = Charset.defaultCharset(); // Always UTF-8

// Reading a file without specifying charset now consistently uses UTF-8
String content = new String(Files.readAllBytes(path)); // Uses UTF-8
```

Example:

```
import java.io.*;
import java.nio.charset.Charset;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.Arrays;

public class Utf8DefaultExample {
    public static void main(String[] args) throws IOException {
        // 1. Show the default charset
        System.out.println("=== Default Charset ===");
        System.out.println("Default charset: " + Charset.defaultCharset());

        // 2. Demonstrate writing with default charset
        System.out.println("\n=== File I/O with Default Charset ===");

        // Create a string with non-ASCII characters
        String text = "Hello, UTF-8! 你好, UTF-8! こんにちは UTF-8! Привет UTF-8!";
        System.out.println("Original text: " + text);

        // Create a temporary file
        Path tempFile = Files.createTempFile("utf8-demo", ".txt");
        System.out.println("Created temp file: " + tempFile);

        // Write using default charset (UTF-8)
        Files.writeString(tempFile, text); // Uses the default charset (UTF-8)

        // Read back using default charset (UTF-8)
        String readText = Files.readString(tempFile);
        System.out.println("Read back text: " + readText);
        System.out.println("Texts match: " + text.equals(readText));

        // Read as bytes to visualize the UTF-8 encoding
        byte[] bytes = Files.readAllBytes(tempFile);
        System.out.println("File bytes: " + Arrays.toString(Arrays.copyOf(bytes, 20)) + "... (" +
                           bytes.length + " bytes total)");

        // 3. Compare with explicit charsets
        System.out.println("\n=== Charset Comparison ===");

        // Create more temporary files
        Path utf8File = Files.createTempFile("utf8-explicit", ".txt");
        Path iso8859File = Files.createTempFile("iso8859", ".txt");

        // Write using different explicit charsets
        Files.writeString(utf8File, text, StandardCharsets.UTF_8);
        try {
            Files.writeString(iso8859File, text, StandardCharsets.ISO_8859_1);
            System.out.println("Wrote to ISO-8859-1 file (some characters may not be represented correctly)");
        } catch (Exception e) {
            System.out.println("Error writing to ISO-8859-1: " + e.getMessage());
        }

        // Read back from each file with corresponding charset
        String utf8Read = Files.readString(utf8File, StandardCharsets.UTF_8);
        String iso8859Read = Files.readString(iso8859File, StandardCharsets.ISO_8859_1);

        System.out.println("UTF-8 read matches original: " + text.equals(utf8Read));
        System.out.println("ISO-8859-1 read matches original: " + text.equals(iso8859Read));
        System.out.println("ISO-8859-1 text: " + iso8859Read);

        // Read with the wrong charset to demonstrate encoding problems
        String utf8ReadAsIso = Files.readString(utf8File, StandardCharsets.ISO_8859_1);
        String iso8859ReadAsUtf8 = Files.readString(iso8859File, StandardCharsets.UTF_8);

        System.out.println("\nReading with the wrong charset:");
        System.out.println("UTF-8 file read as ISO-8859-1: " + utf8ReadAsIso);
```

```
        System.out.println("ISO-8859-1 file read as UTF-8: " + iso8859ReadAsUtf8);

        // 4. Stream-based I/O
        System.out.println("\n=== Stream-based I/O ===");

        // Writer without explicit charset (uses UTF-8)
        try (BufferedWriter writer = Files.newBufferedWriter(tempFile)) {
            writer.write("This uses the default charset (UTF-8)");
        }

        // Reader without explicit charset (uses UTF-8)
        try (BufferedReader reader = Files.newBufferedReader(tempFile)) {
            String line = reader.readLine();
            System.out.println("Read with default charset: " + line);
        }

        // 5. Clean up
        Files.delete(tempFile);
        Files.delete(utf8File);
        Files.delete(iso8859File);
        System.out.println("\nTemporary files deleted.");

        // 6. Create and demonstrate a helper method that highlights the benefit
        System.out.println("\n=== Cross-Platform Text Processing ===");

        String mixedText = "English, 中文, 日本語, Русский, Français";
        processTextSafely(mixedText);
    }

    // Helper method that processes text consistently across platforms
    private static void processTextSafely(String text) {
        try {
            System.out.println("Processing text: " + text);

            // Convert to bytes and back (with default charset)
            byte[] bytes = text.getBytes(); // Uses default charset (UTF-8)
            String reconstructed = new String(bytes); // Uses default charset (UTF-8)

            System.out.println("Successfully reconstructed: " +
                               reconstructed.equals(text));

            // Count characters by language/script
            long latinChars = text.chars().filter(c -> c >= 'A' && c <= 'z').count();
            long nonLatinChars = text.length() - latinChars -
                                 text.chars().filter(c -> c == ' ' || c == ',').count();

            System.out.println("Latin characters: " + latinChars);
            System.out.println("Non-Latin characters: " + nonLatinChars);

            System.out.println("This code now works consistently on all platforms!");

        } catch (Exception e) {
            System.err.println("Error processing text: " + e.getMessage());
        }
    }
}
```

Key Benefits:

1. **Consistency**: Same behavior across all platforms
2. **Modern Web Compatibility**: UTF-8 is the dominant encoding on the web
3. **Internationalization**: Better support for international character sets
4. **Simplified Code**: Less need for explicit charset specification

Common Pitfalls:

- **Legacy Applications**: Applications that relied on platform-specific behavior might need updates
- **Explicit Platform Dependency**: Code that explicitly depends on platform-specific charsets
- **Performance with Large Files**: UTF-8 processing can be more resource-intensive than some other encodings for certain operations

```
// INCORRECT: Relying on specific platform behaviors
// This code assumed Windows platform would use a specific encoding
String content = new String(bytes); // Previously might use Windows-1252 on Windows

// CORRECT: Explicitly specify the charset if you need specific behavior
String content = new String(bytes, StandardCharsets.ISO_8859_1); // Clear intention

// INCORRECT: Loading properties files with non-ASCII content from code written before Java 18
```

```
Properties props = new Properties();
try (InputStream in = new FileInputStream("config.properties")) {
    props.load(in); // Previously used platform default, now uses UTF-8
}

// CORRECT: Specify encoding expectations explicitly for legacy files
Properties props = new Properties();
try (InputStreamReader in = new InputStreamReader(
        new FileInputStream("config.properties"), StandardCharsets.ISO_8859_1)) {
    props.load(in);
}
```

Simple Web Server

What and Why: Java 18 introduced a command-line tool called `jwebserver` that can start a minimal web server serving static files from the current directory. This is useful for testing, development, and simple file sharing scenarios without the need for a full web server.

Real-world Use Cases:

- Quick testing of web applications
- Sharing files during development
- Running simple demos
- Testing REST API clients
- Local documentation hosting

Example:

To start the web server from the command line:

```
$ jwebserver
```

This starts a web server on port 8000 (by default) serving files from the current directory.

You can also specify options:

```
$ jwebserver -p 9000 -d /path/to/directory -o
```

This starts a server on port 9000, serving files from the specified directory, and opens a browser.

Using the Simple Web Server from Java:

```
import com.sun.net.httpserver.*;
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

public class SimpleWebServerExample {
    public static void main(String[] args) throws IOException {
        // Create a simple HTTP server on port 8000
        HttpServer server = HttpServer.create(new InetSocketAddress(8000), 0);

        // Create a file handler for the current directory
        String currentDir = System.getProperty("user.dir");
        System.out.println("Serving files from: " + currentDir);

        // Create a context for handling file requests
        server.createContext("/", new FileHandler(currentDir));

        // Set up a simple handler for API requests
        server.createContext("/api", new ApiHandler());

        // Start the server
        server.start();

        System.out.println("Server started on http://localhost:8000/");
        System.out.println("API endpoint: http://localhost:8000/api");
        System.out.println("Press Ctrl+C to stop the server");
    }

    // Handler for serving static files
    static class FileHandler implements HttpHandler {
        private final String rootDir;
```

```
        public FileHandler(String rootDir) {
            this.rootDir = rootDir;
        }

        @Override
        public void handle(HttpExchange exchange) throws IOException {
            String path = exchange.getRequestURI().getPath();

            // Default to index.html for root path
            if (path.equals("/")) {
                path = "/index.html";
            }

            // Create the full path
            Path filePath = Paths.get(rootDir, path.substring(1));

            // Check if the file exists
            if (Files.exists(filePath) && !Files.isDirectory(filePath)) {
                // Set content type based on file extension
                String fileName = filePath.getFileName().toString();
                String contentType = getContentType(fileName);
                exchange.getResponseHeaders().set("Content-Type", contentType);

                // Send the file
                byte[] fileContent = Files.readAllBytes(filePath);
                exchange.sendResponseHeaders(200, fileContent.length);
                exchange.getResponseBody().write(fileContent);
            } else {
                // File not found
                String response = "404 Not Found";
                exchange.sendResponseHeaders(404, response.length());
                exchange.getResponseBody().write(response.getBytes());
            }

            exchange.getResponseBody().close();
        }

        // Simple method to determine content type
        private String getContentType(String fileName) {
            if (fileName.endsWith(".html") || fileName.endsWith(".htm")) {
                return "text/html";
            } else if (fileName.endsWith(".css")) {
                return "text/css";
            } else if (fileName.endsWith(".js")) {
                return "application/javascript";
            } else if (fileName.endsWith(".jpg") || fileName.endsWith(".jpeg")) {
                return "image/jpeg";
            } else if (fileName.endsWith(".png")) {
                return "image/png";
            } else if (fileName.endsWith(".gif")) {
                return "image/gif";
            } else if (fileName.endsWith(".json")) {
                return "application/json";
            } else {
                return "text/plain";
            }
        }
    }

    // Simple API handler for demonstration
    static class ApiHandler implements HttpHandler {
        @Override
        public void handle(HttpExchange exchange) throws IOException {
            // Set response headers
            exchange.getResponseHeaders().set("Content-Type", "application/json");

            // Create a simple JSON response
            String response = """
                {
                    "message": "Hello from Simple Web Server API",
                    "timestamp": "%s",
                    "method": "%s",
                    "path": "%s"
                }
                """.formatted(
                    java.time.LocalDateTime.now(),
                    exchange.getRequestMethod(),
                    exchange.getRequestURI().getPath()
                );
```

```
            // Send the response
            exchange.sendResponseHeaders(200, response.length());
            exchange.getResponseBody().write(response.getBytes());
            exchange.getResponseBody().close();
        }
    }
}
```

Creating a Sample HTML File for Testing:

```
// Create a sample index.html file to serve
Path indexHtml = Paths.get(System.getProperty("user.dir"), "index.html");
if (!Files.exists(indexHtml)) {
    String htmlContent = """
        <!DOCTYPE html>
        <html>
        <head>
            <title>Simple Web Server Demo</title>
            <style>
                body {
                    font-family: Arial, sans-serif;
                    margin: 40px;
                    line-height: 1.6;
                }
                h1 {
                    color: #333;
                }
                .container {
                    max-width: 800px;
                    margin: 0 auto;
                    padding: 20px;
                    border: 1px solid #ddd;
                    border-radius: 5px;
                }
                code {
                    background-color: #f4f4f4;
                    padding: 2px 5px;
                    border-radius: 3px;
                }
            </style>
        </head>
        <body>
            <div class="container">
                <h1>Simple Web Server Demo</h1>
                <p>This is a demo page served by Java's Simple Web Server.</p>
                <p>Current time: <span id="current-time"></span></p>
                <h2>API Example</h2>
                <p>Try accessing the <a href="/api">/api</a> endpoint to see the JSON response.</p>
                <div id="api-result">
                    <p>API Result will appear here when you click the button:</p>
                    <button id="fetch-btn">Fetch API Data</button>
                    <pre id="api-data"></pre>
                </div>
            </div>

            <script>
                // Update current time
                document.getElementById('current-time').textContent = new Date().toLocaleString();

                // Fetch API data when button is clicked
                document.getElementById('fetch-btn').addEventListener('click', async () => {
                    try {
                        const response = await fetch('/api');
                        const data = await response.json();
                        document.getElementById('api-data').textContent = JSON.stringify(data, null, 2);
                    } catch (error) {
                        document.getElementById('api-data').textContent = 'Error: ' + error.message;
                    }
                });
            </script>
        </body>
        </html>
        """;

    Files.writeString(indexHtml, htmlContent);
    System.out.println("Created sample index.html file");
}
```

Key Benefits:

1. **Simplicity**: No need for external web servers or complex setups
2. **Built-in**: Comes with the JDK, no additional dependencies
3. **Fast Startup**: Ideal for quick testing and development
4. **Customizable**: Can be configured via command-line options

Limitations:

1. **Static Files Only**: No built-in support for server-side processing (though you can add it programmatically)
2. **Basic Features**: Limited security features compared to production web servers
3. **Performance**: Not designed for high-load scenarios
4. **No HTTPS**: No built-in support for HTTPS (though you can implement it programmatically)

Vector API (Incubator)

What and Why: The Vector API, introduced as an incubator module in Java 16 and continuing to evolve through Java 21, provides a way to express vector computations that can be compiled at runtime to optimal vector hardware instructions on supported CPU architectures, enabling significant performance improvements for computationally intensive tasks.

Real-world Use Cases:

- Scientific computing
- Machine learning algorithms
- Signal processing
- Financial calculations
- Image and video processing

Example:

```
import jdk.incubator.vector.*;
import java.util.Arrays;
import java.util.random.RandomGenerator;

public class VectorAPIExample {
    public static void main(String[] args) {
        // Need to run with: --add-modules jdk.incubator.vector

        // 1. Basic vector operations
        System.out.println("=== Basic Vector Operations ===");

        // Choose appropriate vector size for the platform
        VectorSpecies<Float> SPECIES = FloatVector.SPECIES_PREFERRED;
        System.out.println("Vector species: " + SPECIES);
        System.out.println("Vector length: " + SPECIES.length());

        // Create sample arrays
        int arraySize = 1000;
        float[] a = new float[arraySize];
        float[] b = new float[arraySize];
        float[] result = new float[arraySize];

        // Initialize with random data
        RandomGenerator rnd = RandomGenerator.getDefault();
        for (int i = 0; i < arraySize; i++) {
            a[i] = rnd.nextFloat();
            b[i] = rnd.nextFloat();
        }

        // 2. Vector addition
        System.out.println("\n=== Vector Addition ===");

        // Scalar implementation
        long scalarStart = System.nanoTime();
        for (int i = 0; i < arraySize; i++) {
            result[i] = a[i] + b[i];
        }
        long scalarTime = System.nanoTime() - scalarStart;

        // Reset result array
        Arrays.fill(result, 0);

        // Vector implementation
        long vectorStart = System.nanoTime();
        int i = 0;
        int upperBound = SPECIES.loopBound(arraySize);

        // Process vectors of SPECIES.length() elements at a time
```

```
        for (; i < upperBound; i += SPECIES.length()) {
            // Load vectors from arrays
            var va = FloatVector.fromArray(SPECIES, a, i);
            var vb = FloatVector.fromArray(SPECIES, b, i);

            // Add vectors
            var vr = va.add(vb);

            // Store result back to array
            vr.intoArray(result, i);
        }

        // Handle remaining elements (if any)
        for (; i < arraySize; i++) {
            result[i] = a[i] + b[i];
        }

        long vectorTime = System.nanoTime() - vectorStart;

        System.out.println("Scalar time: " + scalarTime + " ns");
        System.out.println("Vector time: " + vectorTime + " ns");
        System.out.println("Speedup: " + (double) scalarTime / vectorTime + "x");

        // 3. More complex example: Vector dot product
        System.out.println("\n=== Vector Dot Product ===");

        // Scalar implementation
        scalarStart = System.nanoTime();
        float dotProductScalar = 0.0f;
        for (i = 0; i < arraySize; i++) {
            dotProductScalar += a[i] * b[i];
        }
        scalarTime = System.nanoTime() - scalarStart;

        // Vector implementation
        vectorStart = System.nanoTime();
        var sumVector = FloatVector.zero(SPECIES);

        // Process vectors
        i = 0;
        upperBound = SPECIES.loopBound(arraySize);

        for (; i < upperBound; i += SPECIES.length()) {
            var va = FloatVector.fromArray(SPECIES, a, i);
            var vb = FloatVector.fromArray(SPECIES, b, i);

            // Multiply and accumulate
            sumVector = va.fma(vb, sumVector);
        }

        // Reduce vector to single value
        float dotProductVector = sumVector.reduceLanes(VectorOperators.ADD);

        // Handle remaining elements
        for (; i < arraySize; i++) {
            dotProductVector += a[i] * b[i];
        }

        vectorTime = System.nanoTime() - vectorStart;

        System.out.println("Scalar result: " + dotProductScalar);
        System.out.println("Vector result: " + dotProductVector);
        System.out.println("Scalar time: " + scalarTime + " ns");
        System.out.println("Vector time: " + vectorTime + " ns");
        System.out.println("Speedup: " + (double) scalarTime / vectorTime + "x");

        // 4. Matrix multiplication
        System.out.println("\n=== Matrix Multiplication ===");

        int size = 256;
        float[][] matrixA = new float[size][size];
        float[][] matrixB = new float[size][size];
        float[][] resultScalar = new float[size][size];
        float[][] resultVector = new float[size][size];

        // Initialize matrices with random data
        for (i = 0; i < size; i++) {
            for (int j = 0; j < size; j++) {
                matrixA[i][j] = rnd.nextFloat();
                matrixB[i][j] = rnd.nextFloat();
            }
```

```
        }

        // Scalar implementation of matrix multiplication
        scalarStart = System.nanoTime();
        for (int r = 0; r < size; r++) {
            for (int c = 0; c < size; c++) {
                float sum = 0.0f;
                for (int k = 0; k < size; k++) {
                    sum += matrixA[r][k] * matrixB[k][c];
                }
                resultScalar[r][c] = sum;
            }
        }
        scalarTime = System.nanoTime() - scalarStart;

        // Vector implementation of matrix multiplication
        vectorStart = System.nanoTime();
        for (int r = 0; r < size; r++) {
            for (int c = 0; c < size; c++) {
                i = 0;
                var sum = FloatVector.zero(SPECIES);
                upperBound = SPECIES.loopBound(size);

                // Process vectors
                for (; i < upperBound; i += SPECIES.length()) {
                    var va = FloatVector.fromArray(SPECIES, matrixA[r], i);

                    // We need to gather values from matrix B since they're not contiguous
                    float[] bColumn = new float[SPECIES.length()];
                    for (int j = 0; j < SPECIES.length() && i + j < size; j++) {
                        bColumn[j] = matrixB[i + j][c];
                    }
                    var vb = FloatVector.fromArray(SPECIES, bColumn, 0);

                    // Multiply and accumulate
                    sum = va.fma(vb, sum);
                }

                // Reduce to get result
                resultVector[r][c] = sum.reduceLanes(VectorOperators.ADD);

                // Handle remaining elements
                for (; i < size; i++) {
                    resultVector[r][c] += matrixA[r][i] * matrixB[i][c];
                }
            }
        }
        vectorTime = System.nanoTime() - vectorStart;

        // Verify results (check a few elements)
        boolean correct = true;
        for (i = 0; i < Math.min(5, size); i++) {
            for (int j = 0; j < Math.min(5, size); j++) {
                if (Math.abs(resultScalar[i][j] - resultVector[i][j]) > 1e-4) {
                    correct = false;
                    System.out.println("Mismatch at [" + i + "][" + j + "]: " +
                                       resultScalar[i][j] + " vs " + resultVector[i][j]);
                }
            }
        }
        System.out.println("Results match: " + correct);
        System.out.println("Scalar time: " + scalarTime / 1_000_000.0 + " ms");
        System.out.println("Vector time: " + vectorTime / 1_000_000.0 + " ms");
        System.out.println("Speedup: " + (double) scalarTime / vectorTime + "x");
    }
}
```

Key Concepts:

1. **Vector Species**: Represents the shape and element type of vector values
2. **Vector Operations**: Basic arithmetic operations (add, subtract, multiply, etc.)
3. **Lane-wise Operations**: Operations applied independently to each element in the vector
4. **Cross-lane Operations**: Operations that work across elements of a vector (like reduction)
5. **Masking**: Conditional operations on selected elements of a vector
6. **Auto-vectorization**: The compiler's ability to convert scalar operations to vector operations automatically

More Advanced Example:

```
import jdk.incubator.vector.*;
import java.util.Arrays;

public class AdvancedVectorAPIExample {

    // Define species based on element type and preferred size
    private static final VectorSpecies<Float> FLOAT_SPECIES = FloatVector.SPECIES_PREFERRED;
    private static final VectorSpecies<Integer> INT_SPECIES = IntVector.SPECIES_PREFERRED;

    public static void main(String[] args) {
        // 1. Conditional operations with masks
        System.out.println("=== Conditional Operations with Masks ===");

        float[] values = new float[16];
        float[] results = new float[16];

        // Initialize array
        for (int i = 0; i < values.length; i++) {
            values[i] = i;
        }

        // Create a mask for even indices
        var mask = FLOAT_SPECIES.indexInRange(0, values.length)
                                .and(FLOAT_SPECIES.loadMask(values, 0)
                                    .toLong() % 2 == 0);

        System.out.println("Mask: " + mask);

        // Load vector with mask
        var vector = FloatVector.fromArray(FLOAT_SPECIES, values, 0, mask);

        // Double values and store with mask
        vector.mul(2.0f).intoArray(results, 0, mask);

        System.out.println("Original values: " + Arrays.toString(values));
        System.out.println("Results (even indices only): " + Arrays.toString(results));

        // 2. Blend operation
        System.out.println("\n=== Blend Operation ===");

        float[] a = {1, 2, 3, 4, 5, 6, 7, 8};
        float[] b = {10, 20, 30, 40, 50, 60, 70, 80};
        float[] blendResult = new float[8];

        // Create a mask for the blend operation
        VectorMask<Float> blendMask = VectorMask.fromLong(FLOAT_SPECIES, 0b10101010);

        // Load vectors
        var va = FloatVector.fromArray(FLOAT_SPECIES, a, 0);
        var vb = FloatVector.fromArray(FLOAT_SPECIES, b, 0);

        // Blend operation (select from va or vb based on mask)
        var blended = va.blend(vb, blendMask);
        blended.intoArray(blendResult, 0);

        System.out.println("Vector A: " + Arrays.toString(a));
        System.out.println("Vector B: " + Arrays.toString(b));
        System.out.println("Blend mask: " + blendMask);
        System.out.println("Blended result: " + Arrays.toString(blendResult));

        // 3. Data rearrangement
        System.out.println("\n=== Data Rearrangement ===");

        int[] data = {0, 1, 2, 3, 4, 5, 6, 7};
        int[] shuffled = new int[8];

        // Create a shuffle pattern
        var indices = new int[]{7, 6, 5, 4, 3, 2, 1, 0}; // Reverse
        var shuffle = IntVector.fromArray(INT_SPECIES, indices, 0);

        // Load data and shuffle
        var vdata = IntVector.fromArray(INT_SPECIES, data, 0);
        var reshuffled = vdata.rearrange(shuffle);
        reshuffled.intoArray(shuffled, 0);

        System.out.println("Original data: " + Arrays.toString(data));
        System.out.println("Shuffle indices: " + Arrays.toString(indices));
        System.out.println("Shuffled result: " + Arrays.toString(shuffled));

        // 4. Image processing example (grayscale conversion)
```

```
        System.out.println("\n=== Image Processing Example ===");

        // Simulate RGB pixel data (3 values per pixel: R, G, B)
        byte[] rgbData = new byte[300]; // 100 pixels

        // Fill with sample colors
        for (int i = 0; i < rgbData.length; i += 3) {
            rgbData[i] = (byte)(i % 256);       // R
            rgbData[i + 1] = (byte)((i + 85) % 256); // G
            rgbData[i + 2] = (byte)((i + 170) % 256); // B
        }

        // Output array for grayscale (1 value per pixel)
        byte[] grayscale = new byte[rgbData.length / 3];

        // Convert using Vector API
        convertToGrayscale(rgbData, grayscale);

        // Print a few pixels
        System.out.println("Sample pixels (R,G,B -> Grayscale):");
        for (int i = 0; i < Math.min(15, grayscale.length); i++) {
            int rgbIndex = i * 3;
            System.out.printf("(%d,%d,%d) -> %d%n",
                rgbData[rgbIndex] & 0xFF,
                rgbData[rgbIndex + 1] & 0xFF,
                rgbData[rgbIndex + 2] & 0xFF,
                grayscale[i] & 0xFF);
        }
    }

    // Grayscale conversion using vector operations
    // Uses the formula: gray = 0.3*R + 0.59*G + 0.11*B
    private static void convertToGrayscale(byte[] rgb, byte[] grayscale) {
        // Define constants for the conversion formula
        var rWeight = FloatVector.broadcast(FLOAT_SPECIES, 0.3f);
        var gWeight = FloatVector.broadcast(FLOAT_SPECIES, 0.59f);
        var bWeight = FloatVector.broadcast(FLOAT_SPECIES, 0.11f);

        int vectorSize = FLOAT_SPECIES.length();

        for (int i = 0; i < grayscale.length; i += vectorSize) {
            // Bound check
            int upperBound = Math.min(vectorSize, grayscale.length - i);
            var mask = FLOAT_SPECIES.maskAll(true).indexInRange(0, upperBound);

            // Temporary arrays for R, G, B components
            float[] r = new float[vectorSize];
            float[] g = new float[vectorSize];
            float[] b = new float[vectorSize];

            // Extract RGB components
            for (int j = 0; j < upperBound; j++) {
                int rgbIndex = (i + j) * 3;
                r[j] = rgb[rgbIndex] & 0xFF; // Unsigned byte value
                g[j] = rgb[rgbIndex + 1] & 0xFF;
                b[j] = rgb[rgbIndex + 2] & 0xFF;
            }

            // Load R, G, B components into vectors
            var rVector = FloatVector.fromArray(FLOAT_SPECIES, r, 0, mask);
            var gVector = FloatVector.fromArray(FLOAT_SPECIES, g, 0, mask);
            var bVector = FloatVector.fromArray(FLOAT_SPECIES, b, 0, mask);

            // Calculate weighted sum
            var result = rVector.mul(rWeight)
                                .add(gVector.mul(gWeight))
                                .add(bVector.mul(bWeight));

            // Convert to byte grayscale values
            var intResult = result.convertShape(VectorOperators.F2I, INT_SPECIES, 0);
            byte[] byteResult = new byte[vectorSize];

            for (int j = 0; j < upperBound; j++) {
                byteResult[j] = (byte) Math.min(255, intResult.lane(j));
            }

            // Store in the result array
            for (int j = 0; j < upperBound; j++) {
                grayscale[i + j] = byteResult[j];
            }
        }
```

```
    }
}
```

Use Cases for Vector API:

1. **Machine Learning**: Matrix operations, neural network calculations
2. **Scientific Computing**: Physics simulations, computational chemistry
3. **Image/Video Processing**: Filters, transformations, compression
4. **Financial Applications**: Option pricing, risk analysis, Monte Carlo simulations
5. **Data Processing**: Statistical analysis, data transformation

Common Pitfalls:

- **Incorrect Species**: Using a vector species that doesn't match the hardware capabilities
- **Alignment Issues**: Memory alignment affecting performance
- **Overhead for Small Arrays**: Vector operations have overhead that may outweigh benefits for small arrays
- **Platform Dependency**: Different performance characteristics across CPU architectures

```
// INCORRECT: Ignoring return values (vectors are immutable)
FloatVector v = FloatVector.fromArray(SPECIES, array, 0);
v.add(FloatVector.broadcast(SPECIES, 1.0f)); // Does nothing!
// The result is not stored anywhere

// CORRECT: Store the result
FloatVector v = FloatVector.fromArray(SPECIES, array, 0);
FloatVector result = v.add(FloatVector.broadcast(SPECIES, 1.0f));
result.intoArray(array, 0);

// INCORRECT: Not handling array bounds properly
FloatVector v = FloatVector.fromArray(SPECIES, array, array.length - 2);
// This will throw ArrayIndexOutOfBoundsException

// CORRECT: Check bounds
int i = 0;
for (; i <= (array.length - SPECIES.length()); i += SPECIES.length()) {
    FloatVector v = FloatVector.fromArray(SPECIES, array, i);
    // Process vector
}
// Handle remaining elements
for (; i < array.length; i++) {
    // Scalar processing
}
```

Java 21 (LTS)

Java 21, released in September 2023, is a Long-Term Support (LTS) release with several important features that significantly enhance Java's capabilities, particularly in the areas of concurrency, pattern matching, and collections.

Virtual Threads

What and Why: Virtual threads are lightweight threads that are managed by the JVM rather than the operating system. They enable a high degree of concurrency with minimal resource usage, making it easier to write, maintain, and debug concurrent applications that scale with near-optimal hardware utilization.

Real-world Use Cases:

- High-throughput server applications
- Microservices handling many concurrent requests
- I/O-bound applications
- Reactive programming patterns
- Connection pools

Before (Java 20) / After (Java 21) Comparison:

Before (Java 20):

```
// Using platform threads
ExecutorService executor = Executors.newFixedThreadPool(100); // Limited by OS capabilities

Future<String> future = executor.submit(() -> {
    // Blocking operations tie up the thread
    return callExpensiveOperation();
});

// Each thread consumes substantial memory (~1MB stack)
```

```
// Thread scheduling is expensive
// Limited practical concurrency (thousands)
```

After (Java 21):

```
// Using virtual threads
ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor();

Future<String> future = executor.submit(() -> {
    // Blocking doesn't waste resources
    return callExpensiveOperation();
});

// Each virtual thread uses minimal memory
// Scheduling is lightweight
// Massive concurrency possible (millions)
```

Comprehensive Example:

```
import java.time.Duration;
import java.time.Instant;
import java.util.concurrent.*;
import java.util.stream.IntStream;

public class VirtualThreadsExample {

    public static void main(String[] args) throws Exception {
        // 1. Creating a single virtual thread
        System.out.println("=== Creating a Single Virtual Thread ===");

        Thread vThread = Thread.ofVirtual().name("my-virtual-thread").start(() -> {
            System.out.println("Running in: " + Thread.currentThread());
            System.out.println("Is virtual: " + Thread.currentThread().isVirtual());
        });

        vThread.join();

        // 2. Creating many virtual threads
        System.out.println("\n=== Creating Many Virtual Threads ===");

        // Track memory usage before
        long beforeMem = getMemoryUsage();

        // Create 10,000 virtual threads
        Thread[] threads = new Thread[10_000];
        for (int i = 0; i < threads.length; i++) {
            final int id = i;
            threads[i] = Thread.ofVirtual().name("vt-" + i).start(() -> {
                try {
                    // Simulate some work
                    Thread.sleep(Duration.ofMillis(100));
                    if (id % 1000 == 0) {
                        System.out.println("Completed virtual thread " + id);
                    }
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            });
        }

        // Wait for all threads to complete
        for (Thread t : threads) {
            t.join();
        }

        // Track memory usage after
        long afterMem = getMemoryUsage();
        System.out.println("Memory used: " + (afterMem - beforeMem) / (1024.0 * 1024.0) + " MB");
        System.out.println("Average per thread: " + (afterMem - beforeMem) / (threads.length * 1024.0) + " KB");

        // 3. Virtual threads vs platform threads performance
        System.out.println("\n=== Virtual vs Platform Threads Performance ===");

        // Platform thread executor (fixed pool)
        int platformThreadCount = 200;
        ExecutorService platformExecutor =
```

```
            Executors.newFixedThreadPool(platformThreadCount);

        // Virtual thread executor (unlimited)
        ExecutorService virtualExecutor =
            Executors.newVirtualThreadPerTaskExecutor();

        // Benchmark platform threads
        System.out.println("Running with " + platformThreadCount + " platform threads...");
        Instant platformStart = Instant.now();
        runTasks(platformExecutor, 10_000);
        Duration platformDuration = Duration.between(platformStart, Instant.now());

        // Benchmark virtual threads
        System.out.println("Running with virtual threads...");
        Instant virtualStart = Instant.now();
        runTasks(virtualExecutor, 10_000);
        Duration virtualDuration = Duration.between(virtualStart, Instant.now());

        System.out.println("Platform threads: " + platformDuration.toMillis() + " ms");
        System.out.println("Virtual threads: " + virtualDuration.toMillis() + " ms");

        platformExecutor.shutdown();
        virtualExecutor.shutdown();

        // 4. Pinned virtual threads demonstration
        System.out.println("\n=== Pinned Virtual Threads ===");

        demonstratePinning();

        // 5. Structured concurrency example
        System.out.println("\n=== Combining with Structured Concurrency ===");

        try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
            // Launch multiple subtasks
            StructuredTaskScope.Subtask<String> task1 =
                scope.fork(() -> fetchUserData(123));
            StructuredTaskScope.Subtask<String> task2 =
                scope.fork(() -> fetchOrderHistory(123));

            // Wait for all subtasks to complete
            scope.join();
            // Handle any exceptions
            scope.throwIfFailed(e -> new RuntimeException("Task failed", e));

            // Process results
            String userData = task1.get();
            String orderHistory = task2.get();

            System.out.println("User data: " + userData);
            System.out.println("Order history: " + orderHistory);
        }

        // 6. Real-world example: HTTP server with virtual threads
        System.out.println("\n=== HTTP Server with Virtual Threads ===");

        simulateHttpServer();
    }

    // Helper method to benchmark task execution
    private static void runTasks(ExecutorService executor, int taskCount)
                                throws InterruptedException {
        CountDownLatch latch = new CountDownLatch(taskCount);

        for (int i = 0; i < taskCount; i++) {
            executor.submit(() -> {
                try {
                    // Simulate I/O-bound work (e.g., network call)
                    Thread.sleep(Duration.ofMillis(100));
                    return "Task completed";
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                    return "Task interrupted";
                } finally {
                    latch.countDown();
                }
            });
        }

        // Wait for all tasks to complete
        latch.await();
    }
```

```
    // Helper to get memory usage
    private static long getMemoryUsage() {
        Runtime runtime = Runtime.getRuntime();
        runtime.gc(); // Suggestion to run garbage collector
        return runtime.totalMemory() - runtime.freeMemory();
    }

    // Demonstrate pinned virtual threads
    private static void demonstratePinning() throws Exception {
        Thread vt = Thread.ofVirtual().start(() -> {
            System.out.println("Starting virtual thread execution");

            // Unpinned operation
            try {
                Thread.sleep(100); // Just uses a timer, doesn't pin
                System.out.println("After sleep (not pinned)");
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }

            // Pinned operation (synchronized on a monitor)
            Object monitor = new Object();
            synchronized (monitor) {
                // While in this block, the virtual thread is "pinned" to a carrier thread
                System.out.println("Inside synchronized block (pinned)");

                try {
                    Thread.sleep(100); // The carrier thread is blocked here
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            }

            System.out.println("After synchronized block (unpinned)");
        });

        vt.join();
    }

    // Mock HTTP server simulation with virtual threads
    private static void simulateHttpServer() throws Exception {
        // Simulate an HTTP server that handles concurrent requests using virtual threads
        int requestCount = 100;
        CountDownLatch latch = new CountDownLatch(requestCount);

        // Metrics
        long[] responseTimes = new long[requestCount];

        // Process incoming requests (each in its own virtual thread)
        for (int i = 0; i < requestCount; i++) {
            final int requestId = i;
            Thread.ofVirtual().name("request-handler-" + i).start(() -> {
                try {
                    long start = System.currentTimeMillis();

                    // Simulate request processing
                    System.out.println("Processing request: " + requestId);

                    // Simulate database query (I/O bound)
                    String userData = fetchUserData(requestId);

                    // Simulate third-party API call (I/O bound)
                    String additionalData = fetchExternalAPI(requestId);

                    // Simulate response creation
                    String response = createResponse(userData, additionalData);

                    // Calculate response time
                    long end = System.currentTimeMillis();
                    responseTimes[requestId] = end - start;

                    if (requestId % 10 == 0) {
                        System.out.println("Completed request " + requestId +
                                " in " + responseTimes[requestId] + " ms");
                    }
                } finally {
                    latch.countDown();
                }
            });
        }
```

```
        // Wait for all requests to complete
        latch.await();

        // Calculate average response time
        double avgResponseTime = IntStream.range(0, requestCount)
                                    .mapToLong(i -> responseTimes[i])
                                    .average()
                                    .orElse(0);

        System.out.println("All requests completed");
        System.out.println("Average response time: " + avgResponseTime + " ms");
    }

    // Mock methods to simulate I/O operations
    private static String fetchUserData(int userId) throws InterruptedException {
        // Simulate database query
        Thread.sleep(Duration.ofMillis(50 + ThreadLocalRandom.current().nextInt(50)));
        return "User_" + userId;
    }

    private static String fetchOrderHistory(int userId) throws InterruptedException {
        // Simulate another database query
        Thread.sleep(Duration.ofMillis(30 + ThreadLocalRandom.current().nextInt(70)));
        return "Orders_" + userId;
    }

    private static String fetchExternalAPI(int requestId) throws InterruptedException {
        // Simulate external API call
        Thread.sleep(Duration.ofMillis(70 + ThreadLocalRandom.current().nextInt(30)));
        return "ExternalData_" + requestId;
    }

    private static String createResponse(String userData, String additionalData) {
        // Simulate response creation
        return "Response: " + userData + " - " + additionalData;
    }
}
```

Key Concepts and Features:

1. **Creation Methods**:
   - `Thread.ofVirtual()`: Creates a builder for virtual threads
   - `Thread.startVirtualThread()`: Creates and starts a virtual thread
   - `Executors.newVirtualThreadPerTaskExecutor()`: Creates an executor that creates a new virtual thread for each task
2. **Characteristics**:
   - Very lightweight (a few hundred bytes vs. ~1MB for platform threads)
   - Not managed by the OS (scheduled by JVM onto platform threads)
   - Support all thread operations like `join()`, `interrupt()`, etc.
   - Use the same Thread API as platform threads
3. **Carrier Threads**:
   - Virtual threads are scheduled onto platform threads (called carriers)
   - When a virtual thread blocks, it unmounts from its carrier, freeing it for other virtual threads
   - The number of carriers typically equals the number of CPU cores
4. **Pinning**:
   - "Pinning" occurs when a virtual thread cannot unmount from its carrier thread
   - Common causes: synchronized blocks, native methods, or legacy thread-local code
   - Excessive pinning defeats the purpose of virtual threads

Common Pitfalls:

- **Thread Pools**: Traditional thread pools (`newFixedThreadPool`) don't make sense with virtual threads
- **Thread-Local Variables**: Can cause memory leaks with millions of virtual threads
- **Synchronization**: `synchronized` blocks cause pinning; use `java.util.concurrent` locks instead
- **Resource Exhaustion**: It's possible to create too many virtual threads if each consumes external resources

```
// INCORRECT: Using thread pools with virtual threads
ExecutorService executor = Executors.newFixedThreadPool(
    100, Thread.ofVirtual().factory());
// ^^ This defeats the purpose of virtual threads by limiting concurrency
```

```
// CORRECT: Use unlimited executor
ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor();

// INCORRECT: Heavy use of ThreadLocal with virtual threads
ThreadLocal<ExpensiveResource> resourceCache = ThreadLocal.withInitial(() -> {
    return new ExpensiveResource(); // Creates one per thread, may cause memory issues
});

// CORRECT: Use a scoped value or explicit passing
// Java 21 preview: ScopedValue<ExpensiveResource> scopedResource = ...

// INCORRECT: Using synchronized blocks (causes pinning)
synchronized (lock) {
    // Virtual thread is pinned while in this block
    performLongOperation();
}

// CORRECT: Use java.util.concurrent locks
Lock lock = new ReentrantLock();
lock.lock();
try {
    // Virtual thread can be unmounted if it blocks
    performLongOperation();
} finally {
    lock.unlock();
}

// INCORRECT: Assuming unlimited resources
for (int i = 0; i < 1_000_000; i++) {
    Thread.startVirtualThread(() -> {
        // Each opens a database connection
        try (Connection conn = dataSource.getConnection()) {
            // ... use connection
        }
    });
}

// CORRECT: Control resource usage
Semaphore dbConnections = new Semaphore(100); // Limit to 100 concurrent DB connections
for (int i = 0; i < 1_000_000; i++) {
    Thread.startVirtualThread(() -> {
        try {
            dbConnections.acquire();
            try (Connection conn = dataSource.getConnection()) {
                // ... use connection
            } finally {
                dbConnections.release();
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    });
}
```

Structured Concurrency

What and Why: Structured Concurrency, introduced as an incubator feature in Java 21, provides a framework for treating multiple related tasks as a single unit of work. It simplifies error handling, cancellation, and resource management in concurrent programs by ensuring that tasks executing in different threads can be coordinated safely.

Real-world Use Cases:

- Handling multiple concurrent API calls
- Aggregating results from multiple services
- Timeout management across related operations
- Parallel data processing with controlled cancellation
- Coordinated background tasks

Before (Java 20) / After (Java 21) Comparison:

Before (Java 20):

```
// Manual task coordination
ExecutorService executor = Executors.newCachedThreadPool();

try {
    Future<String> userFuture = executor.submit(() -> fetchUserData(userId));
    Future<List<Order>> ordersFuture = executor.submit(() -> fetchOrders(userId));
```

```
    try {
        String userData = userFuture.get(1, TimeUnit.SECONDS);
        List<Order> orders = ordersFuture.get(1, TimeUnit.SECONDS);

        // Process results
        processUserData(userData, orders);

    } catch (Exception e) {
        // Cancel individual tasks
        userFuture.cancel(true);
        ordersFuture.cancel(true);
        throw e;
    }
} finally {
    executor.shutdown();
}
```

After (Java 21):

```
// Structured concurrency
try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
    Subtask<String> userTask = scope.fork(() -> fetchUserData(userId));
    Subtask<List<Order>> ordersTask = scope.fork(() -> fetchOrders(userId));

    // Wait for all tasks to complete or one to fail
    scope.join();
    scope.throwIfFailed(e -> new RuntimeException("Task failed", e));

    // Process results (all tasks completed successfully)
    String userData = userTask.get();
    List<Order> orders = ordersTask.get();
    processUserData(userData, orders);
}
```

Comprehensive Example:

```
import java.time.Duration;
import java.util.List;
import java.util.concurrent.*;
import jdk.incubator.concurrent.*;

public class StructuredConcurrencyExample {

    public static void main(String[] args) {
        try {
            // Example 1: Basic usage with ShutdownOnFailure
            System.out.println("=== Basic ShutdownOnFailure Example ===");

            String result1 = basicExample(123);
            System.out.println("Basic example result: " + result1);

            // Example 2: ShutdownOnSuccess - Take the first successful result
            System.out.println("\n=== ShutdownOnSuccess Example ===");

            String result2 = firstSuccessfulExample();
            System.out.println("First successful result: " + result2);

            // Example 3: Custom scope policies
            System.out.println("\n=== Custom Scope Policies Example ===");

            try {
                customPolicyExample();
            } catch (Exception e) {
                System.out.println("Expected exception: " + e.getMessage());
            }

            // Example 4: With timeout handling
            System.out.println("\n=== Timeout Handling Example ===");

            try {
                timeoutExample();
            } catch (Exception e) {
                System.out.println("Expected timeout exception: " + e.getMessage());
            }
```

```
            // Example 5: Real-world scenario
            System.out.println("\n=== Real-world Scenario Example ===");

            aggregateUserDataExample(42);

        } catch (Exception e) {
            System.out.println("Main exception: " + e);
            e.printStackTrace();
        }
    }

    // Example 1: Basic usage with ShutdownOnFailure
    static String basicExample(int userId) throws Exception {
        try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
            // Fork two tasks
            StructuredTaskScope.Subtask<String> userTask =
                scope.fork(() -> fetchUserData(userId));

            StructuredTaskScope.Subtask<List<String>> ordersTask =
                scope.fork(() -> fetchOrderIds(userId));

            // Join all tasks and propagate any exceptions
            scope.join();
            scope.throwIfFailed(e -> new RuntimeException("Task failed", e));

            // All tasks completed successfully, get their results
            String userData = userTask.get();
            List<String> orderIds = ordersTask.get();

            return "User: " + userData + ", Orders: " + orderIds;
        }
    }

    // Example 2: ShutdownOnSuccess - Take the first successful result
    static String firstSuccessfulExample() throws Exception {
        try (var scope = new StructuredTaskScope.ShutdownOnSuccess<String>()) {
            // Fork multiple tasks to different servers/services
            scope.fork(() -> callService("Service A")); // Might be slow or fail
            scope.fork(() -> callService("Service B")); // Try an alternative
            scope.fork(() -> callService("Service C")); // Yet another alternative

            // Wait for the first successful result
            scope.join();

            // Get the first successful result
            return scope.result();
        }
    }

    // Example 3: Custom scope policies
    static void customPolicyExample() throws Exception {
        // Create a custom scope that fails if more than 2 subtasks fail
        try (var scope = new StructuredTaskScope<String>("Custom-Policy",
                task -> {}, // onComplete handler is a no-op
                task -> {})) { // onError handler is a no-op

            // Track task state manually
            var failedTasks = new ConcurrentLinkedQueue<Throwable>();
            var successfulTasks = new ConcurrentLinkedQueue<String>();

            // Fork several tasks
            for (int i = 0; i < 5; i++) {
                final int id = i;
                scope.fork(() -> {
                    try {
                        if (id % 2 == 0) {
                            // Even tasks succeed
                            String result = "Task " + id + " result";
                            successfulTasks.add(result);
                            return result;
                        } else {
                            // Odd tasks fail
                            throw new RuntimeException("Task " + id + " failed");
                        }
                    } catch (Throwable t) {
                        failedTasks.add(t);
                        throw t;
                    }
                });
            }
```

```
            // Wait for all tasks to complete
            scope.join();

            System.out.println("Successful tasks: " + successfulTasks.size());
            System.out.println("Failed tasks: " + failedTasks.size());

            // Implement custom policy: fail if more than 2 tasks failed
            if (failedTasks.size() > 2) {
                throw new RuntimeException("Too many tasks failed: " + failedTasks.size());
            }

            // Process successful results
            for (String result : successfulTasks) {
                System.out.println("Result: " + result);
            }
        }
    }

    // Example 4: With timeout handling
    static void timeoutExample() throws Exception {
        try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
            // Fork tasks
            scope.fork(() -> fetchWithTimeout("Quick task", 500));
            scope.fork(() -> fetchWithTimeout("Slow task", 2000)); // This will be too slow

            // Wait with timeout
            boolean allDone = scope.joinUntil(Instant.now().plusMillis(1000));

            if (!allDone) {
                scope.shutdown();
                throw new TimeoutException("Tasks did not complete within timeout");
            }

            scope.throwIfFailed(e -> new RuntimeException("Task failed", e));

            // We won't reach here due to timeout
            System.out.println("All tasks completed successfully!");
        }
    }

    // Example 5: Real-world scenario - Aggregate user data
    static void aggregateUserDataExample(int userId) throws Exception {
        try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
            // Fork multiple tasks to fetch different aspects of user data
            StructuredTaskScope.Subtask<UserProfile> profileTask =
                scope.fork(() -> fetchUserProfile(userId));

            StructuredTaskScope.Subtask<List<Order>> ordersTask =
                scope.fork(() -> fetchUserOrders(userId));

            StructuredTaskScope.Subtask<List<Recommendation>> recommendationsTask =
                scope.fork(() -> fetchRecommendations(userId));

            StructuredTaskScope.Subtask<Metrics> metricsTask =
                scope.fork(() -> fetchUserMetrics(userId));

            // Wait with timeout
            boolean completed = scope.joinUntil(Instant.now().plusMillis(500));

            if (!completed) {
                // Some tasks are still running - we can decide what to do
                System.out.println("Some tasks did not complete in time");
            }

            try {
                // Check for failures
                scope.throwIfFailed(e -> new AggregationException("Failed to aggregate user data", e));

                // All tasks completed successfully
                UserProfile profile = profileTask.get();
                List<Order> orders = ordersTask.get();
                List<Recommendation> recommendations = recommendationsTask.get();
                Metrics metrics = metricsTask.get();

                // Create aggregate response
                AggregatedUserData userData = new AggregatedUserData(
                    profile, orders, recommendations, metrics);

                System.out.println("Successfully aggregated user data:");
                System.out.println(userData);
```

```
            } catch (AggregationException e) {
                // Log the error and return partial data if possible
                System.out.println("Error aggregating data: " + e.getMessage());

                // Get whatever data is available
                try {
                    if (profileTask.state() == StructuredTaskScope.Subtask.State.SUCCESS) {
                        System.out.println("Profile data available: " + profileTask.get());
                    }
                } catch (Exception ex) {
                    // Ignore
                }

                try {
                    if (ordersTask.state() == StructuredTaskScope.Subtask.State.SUCCESS) {
                        System.out.println("Orders available: " + ordersTask.get().size());
                    }
                } catch (Exception ex) {
                    // Ignore
                }
            }
        }
    }

    // Simulated service methods

    static String fetchUserData(int userId) throws InterruptedException {
        Thread.sleep(ThreadLocalRandom.current().nextInt(200, 300));
        return "User " + userId;
    }

    static List<String> fetchOrderIds(int userId) throws InterruptedException {
        Thread.sleep(ThreadLocalRandom.current().nextInt(100, 200));
        return List.of("Order-1", "Order-2", "Order-3");
    }

    static String callService(String serviceName) throws InterruptedException {
        int delay = switch (serviceName) {
            case "Service A" -> ThreadLocalRandom.current().nextInt(400, 600);
            case "Service B" -> ThreadLocalRandom.current().nextInt(200, 400);
            case "Service C" -> ThreadLocalRandom.current().nextInt(100, 300);
            default -> 300;
        };

        Thread.sleep(delay);

        // Randomly fail some services
        if (ThreadLocalRandom.current().nextBoolean() && !"Service C".equals(serviceName)) {
            throw new RuntimeException(serviceName + " failed");
        }

        return "Response from " + serviceName;
    }

    static String fetchWithTimeout(String taskName, int delay) throws InterruptedException {
        System.out.println("Starting " + taskName);
        Thread.sleep(delay);
        System.out.println("Completed " + taskName);
        return taskName + " result";
    }

    // Data classes for the real-world example
    static class UserProfile {
        private final int userId;
        private final String name;

        UserProfile(int userId, String name) {
            this.userId = userId;
            this.name = name;
        }

        @Override
        public String toString() {
            return "UserProfile{userId=" + userId + ", name='" + name + "'}";
        }
    }

    static class Order {
        private final String orderId;
        private final double amount;
```

```
        Order(String orderId, double amount) {
            this.orderId = orderId;
            this.amount = amount;
        }

        @Override
        public String toString() {
            return "Order{orderId='" + orderId + "', amount=" + amount + "}";
        }
    }

    static class Recommendation {
        private final String productId;
        private final double score;

        Recommendation(String productId, double score) {
            this.productId = productId;
            this.score = score;
        }

        @Override
        public String toString() {
            return "Recommendation{productId='" + productId + "', score=" + score + "}";
        }
    }

    static class Metrics {
        private final int visitCount;
        private final int orderCount;

        Metrics(int visitCount, int orderCount) {
            this.visitCount = visitCount;
            this.orderCount = orderCount;
        }

        @Override
        public String toString() {
            return "Metrics{visitCount=" + visitCount + ", orderCount=" + orderCount + "}";
        }
    }

    static class AggregatedUserData {
        private final UserProfile profile;
        private final List<Order> orders;
        private final List<Recommendation> recommendations;
        private final Metrics metrics;

        AggregatedUserData(UserProfile profile, List<Order> orders,
                           List<Recommendation> recommendations, Metrics metrics) {
            this.profile = profile;
            this.orders = orders;
            this.recommendations = recommendations;
            this.metrics = metrics;
        }

        @Override
        public String toString() {
            return "AggregatedUserData{\n" +
                   "  profile=" + profile + ",\n" +
                   "  orders=" + orders + ",\n" +
                   "  recommendations=" + recommendations + ",\n" +
                   "  metrics=" + metrics + "\n" +
                   "}";
        }
    }

    static class AggregationException extends Exception {
        AggregationException(String message, Throwable cause) {
            super(message, cause);
        }
    }

    // Real-world example service methods

    static UserProfile fetchUserProfile(int userId) throws InterruptedException {
        Thread.sleep(50);
        return new UserProfile(userId, "User " + userId);
    }

    static List<Order> fetchUserOrders(int userId) throws InterruptedException {
        Thread.sleep(100);
```

```
        return List.of(
            new Order("Order-1", 99.99),
            new Order("Order-2", 49.99)
        );
    }

    static List<Recommendation> fetchRecommendations(int userId) throws InterruptedException {
        Thread.sleep(200);
        return List.of(
            new Recommendation("Product-1", 0.9),
            new Recommendation("Product-2", 0.8),
            new Recommendation("Product-3", 0.7)
        );
    }

    static Metrics fetchUserMetrics(int userId) throws InterruptedException {
        Thread.sleep(70);
        return new Metrics(42, 5);
    }
}
```

Key Concepts and Features:

1. **StructuredTaskScope**:
   - The base class for managing a structured concurrent task
   - Provides a try-with-resources block to define the scope's lifetime
   - Takes care of task cancellation when the scope is closed
2. **Shutdown Policies**:
   - **ShutdownOnFailure**: If any task fails, shutdown the scope and cancel other tasks
   - **ShutdownOnSuccess**: Shutdown after the first successful task completes
   - Custom policies can be implemented by extending StructuredTaskScope
3. **Task State**:
   - `UNAVAILABLE`: Task hasn't completed yet
   - `SUCCESS`: Task completed successfully
   - `FAILED`: Task completed with an exception
4. **Fork and Join Model**:
   - Fork: Start a new task within the scope
   - Join: Wait for all tasks to complete
   - joinUntil: Wait with a timeout
5. **Containment**:
   - Ensures that all subtasks are completed (either successfully or with failure) before the scope closes
   - Automatically cleans up any unfinished work when the scope is closed

Common Pitfalls:

- **Resource Leaks**: Not using try-with-resources, which could cause task leaks
- **Deadlocks**: Tasks that wait for each other in a circular dependency
- **Forgotten Exceptions**: Not checking the task state or handling exceptions properly
- **Missing Error Propagation**: Not using `throwIfFailed()` when necessary

```
// INCORRECT: Not using try-with-resources
StructuredTaskScope<String> scope = new StructuredTaskScope.ShutdownOnFailure();
scope.fork(() -> someTask());
scope.join();
// Missing scope.close() leads to potential resource leaks

// CORRECT: Use try-with-resources
try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
    scope.fork(() -> someTask());
    scope.join();
    // Scope automatically closed
}

// INCORRECT: Not handling exceptions after join
try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
    var task1 = scope.fork(() -> maybeFail());
    var task2 = scope.fork(() -> anotherTask());
    scope.join();
    // Missing throwIfFailed() - we don't know if tasks failed
```

```
    String result1 = task1.get(); // Might throw if task failed
    String result2 = task2.get(); // Might throw if task failed
}

// CORRECT: Proper exception handling
try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
    var task1 = scope.fork(() -> maybeFail());
    var task2 = scope.fork(() -> anotherTask());
    scope.join();
    scope.throwIfFailed(e -> new RuntimeException("Task failed", e));
    // Now safe to get results
    String result1 = task1.get();
    String result2 = task2.get();
}
```

Record Patterns

What and Why: Record patterns, introduced in Java 21, extend pattern matching to enable destructuring of record values. This feature simplifies extracting components from complex nested data structures, making code more concise and readable.

Real-world Use Cases:

- Processing hierarchical data structures
- Parsing and processing JSON/XML data
- Working with nested records
- Simplifying data extraction and transformation
- Protocol buffer or message handling

Before (Java 20) / After (Java 21) Comparison:

Before (Java 20):

```
// Accessing nested record fields
if (obj instanceof Point p) {
    int x = p.x();
    int y = p.y();
    // Use x and y
}

// Nested data access
if (obj instanceof Rectangle r) {
    Point upperLeft = r.upperLeft();
    Point lowerRight = r.lowerRight();
    int x1 = upperLeft.x();
    int y1 = upperLeft.y();
    int x2 = lowerRight.x();
    int y2 = lowerRight.y();
    // Use coordinates
}
```

After (Java 21):

```
// Accessing nested record fields
if (obj instanceof Point(int x, int y)) {
    // Use x and y directly
}

// Nested data access with destructuring
if (obj instanceof Rectangle(Point(int x1, int y1), Point(int x2, int y2))) {
    // Use coordinates directly
}
```

Comprehensive Example:

```
import java.util.*;

public class RecordPatternsExample {

    // Sample record hierarchy
    record Point(int x, int y) {}
    record Rectangle(Point upperLeft, Point lowerRight) {}
    record Circle(Point center, int radius) {}
    record Triangle(Point a, Point b, Point c) {}
```

```
    // Nested records for more complex examples
    record ColoredShape(Shape shape, Color color) {}
    record Color(int red, int green, int blue) {}

    // Records for JSON-like data
    record Person(String name, int age, Address address) {}
    record Address(String street, String city, String country, ZipCode zip) {}
    record ZipCode(String code, int extension) {}

    // Records for response handling
    record ApiResponse<T>(int statusCode, String message, T data) {}
    record UserData(String userId, String username, List<String> roles) {}

    // Sealed interface for pattern matching example
    sealed interface Shape permits Rectangle, Circle, Triangle {}

    public static void main(String[] args) {
        // 1. Basic record pattern matching
        System.out.println("=== Basic Record Pattern Matching ===");

        Object obj1 = new Point(10, 20);
        Object obj2 = new Rectangle(new Point(0, 0), new Point(100, 100));
        Object obj3 = new Circle(new Point(50, 50), 25);

        printObjectInfo(obj1);
        printObjectInfo(obj2);
        printObjectInfo(obj3);

        // 2. Record patterns in switch statements
        System.out.println("\n=== Record Patterns in Switch ===");

        printShapeDetails(new Rectangle(new Point(10, 20), new Point(30, 40)));
        printShapeDetails(new Circle(new Point(50, 60), 15));
        printShapeDetails(new Triangle(
            new Point(0, 0),
            new Point(10, 20),
            new Point(20, 0)
        ));

        // 3. Nested record patterns
        System.out.println("\n=== Nested Record Patterns ===");

        Object nestedObj = new ColoredShape(
            new Rectangle(new Point(5, 10), new Point(25, 30)),
            new Color(255, 0, 0) // Red
        );

        processNestedPattern(nestedObj);

        // 4. Record patterns with guards
        System.out.println("\n=== Record Patterns with Guards ===");

        List<Point> points = List.of(
            new Point(0, 0),
            new Point(10, 0),
            new Point(0, 10),
            new Point(10, 10),
            new Point(-5, -5)
        );

        for (Point point : points) {
            classifyPoint(point);
        }

        // 5. Processing complex hierarchical data
        System.out.println("\n=== Processing Complex Data ===");

        Person person = new Person(
            "John Doe",
            35,
            new Address(
                "123 Main St",
                "New York",
                "USA",
                new ZipCode("10001", 1234)
            )
        );

        processPersonData(person);

        // 6. Using record patterns with generics
```

```
        System.out.println("\n=== Record Patterns with Generics ===");

        ApiResponse<UserData> successResponse = new ApiResponse<>(
            200,
            "Success",
            new UserData("U123", "johndoe", List.of("USER", "ADMIN"))
        );

        ApiResponse<String> errorResponse = new ApiResponse<>(
            404,
            "Not Found",
            "User does not exist"
        );

        processApiResponse(successResponse);
        processApiResponse(errorResponse);
    }

    // 1. Basic record pattern matching
    static void printObjectInfo(Object obj) {
        // Using record patterns to destructure the object
        if (obj instanceof Point(int x, int y)) {
            System.out.println("Point at (" + x + ", " + y + ")");
        } else if (obj instanceof Rectangle(Point(int x1, int y1), Point(int x2, int y2))) {
            System.out.println("Rectangle from (" + x1 + ", " + y1 +
                               ") to (" + x2 + ", " + y2 + ")");
            System.out.println("  Width: " + (x2 - x1) + ", Height: " + (y2 - y1));
        } else if (obj instanceof Circle(Point(int x, int y), int r)) {
            System.out.println("Circle at (" + x + ", " + y + ") with radius " + r);
            System.out.println("  Area: " + Math.PI * r * r);
        } else {
            System.out.println("Unknown object: " + obj);
        }
    }

    // 2. Record patterns in switch statements
    static void printShapeDetails(Shape shape) {
        // Using record patterns in switch
        switch (shape) {
            case Rectangle(Point(var x1, var y1), Point(var x2, var y2)) -> {
                int width = x2 - x1;
                int height = y2 - y1;
                System.out.println("Rectangle with width " + width + " and height " + height);
                System.out.println("  Area: " + (width * height));
            }
            case Circle(Point(var x, var y), var radius) -> {
                System.out.println("Circle at (" + x + ", " + y + ") with radius " + radius);
                System.out.println("  Circumference: " + (2 * Math.PI * radius));
            }
            case Triangle(Point(var x1, var y1), Point(var x2, var y2), Point(var x3, var y3)) -> {
                System.out.println("Triangle with vertices at:");
                System.out.println("  (" + x1 + ", " + y1 + ")");
                System.out.println("  (" + x2 + ", " + y2 + ")");
                System.out.println("  (" + x3 + ", " + y3 + ")");
            }
        }
    }

    // 3. Nested record patterns
    static void processNestedPattern(Object obj) {
        if (obj instanceof ColoredShape(
                Rectangle(Point(int x1, int y1), Point(int x2, int y2)),
                Color(int r, int g, int b)
            )) {
            System.out.println("Colored rectangle:");
            System.out.println("  From (" + x1 + ", " + y1 + ") to (" + x2 + ", " + y2 + ")");
            System.out.println("  Color: RGB(" + r + ", " + g + ", " + b + ")");
        } else if (obj instanceof ColoredShape(
                Circle(Point(int x, int y), int radius),
                Color(int r, int g, int b)
            )) {
            System.out.println("Colored circle:");
            System.out.println("  Center: (" + x + ", " + y + ") with radius " + radius);
            System.out.println("  Color: RGB(" + r + ", " + g + ", " + b + ")");
        } else {
            System.out.println("Unknown or unsupported shape: " + obj);
        }
    }

    // 4. Record patterns with guards
    static void classifyPoint(Point point) {
```

```
        // Using record pattern with guards (conditional expressions)
        if (point instanceof Point(int x, int y) && x >= 0 && y >= 0) {
            System.out.println("Point (" + x + ", " + y + ") is in the first quadrant");
        } else if (point instanceof Point(int x, int y) && x < 0 && y >= 0) {
            System.out.println("Point (" + x + ", " + y + ") is in the second quadrant");
        } else if (point instanceof Point(int x, int y) && x < 0 && y < 0) {
            System.out.println("Point (" + x + ", " + y + ") is in the third quadrant");
        } else if (point instanceof Point(int x, int y) && x >= 0 && y < 0) {
            System.out.println("Point (" + x + ", " + y + ") is in the fourth quadrant");
        } else {
            System.out.println("Point classification error");
        }
    }

    // 5. Processing complex hierarchical data
    static void processPersonData(Object data) {
        // Using deeply nested patterns to extract specific information
        if (data instanceof Person(
                String name,
                int age,
                Address(String street, String city, String country, ZipCode(String code, int ext))
            )) {
            System.out.println("Person details:");
            System.out.println("  Name: " + name);
            System.out.println("  Age: " + age);
            System.out.println("  Address: " + street + ", " + city + ", " + country);
            System.out.println("  Zip code: " + code + "-" + ext);

            // Business logic based on extracted data
            if (age >= 18 && country.equals("USA")) {
                System.out.println("  Status: Adult US resident");
            } else {
                System.out.println("  Status: Not an adult US resident");
            }
        } else {
            System.out.println("Invalid person data format");
        }
    }

    // 6. Using record patterns with generics
    static void processApiResponse(Object response) {
        if (response instanceof ApiResponse(int status, String msg, UserData(String id, String username, List<String> roles))) {
            System.out.println("User API Response (Status: " + status + ")");
            System.out.println("  Message: " + msg);
            System.out.println("  User ID: " + id);
            System.out.println("  Username: " + username);
            System.out.println("  Roles: " + roles);

            if (roles.contains("ADMIN")) {
                System.out.println("  Admin access granted");
            }
        } else if (response instanceof ApiResponse(int status, String msg, String errorMsg)) {
            System.out.println("Error API Response (Status: " + status + ")");
            System.out.println("  Message: " + msg);
            System.out.println("  Error details: " + errorMsg);
        } else {
            System.out.println("Unknown API response format: " + response);
        }
    }
}
```

Key Features and Benefits:

1. **Nested Pattern Matching**: Destructure deeply nested record structures in a single expression
2. **Type Patterns**: Combine with type patterns to safely cast and destructure
3. **Readability**: Eliminates repetitive accessor method calls
4. **Expressiveness**: Makes code more declarative and focused on the data structure
5. **Safety**: Pattern matching is type-safe and verified at compile time

Common Pitfalls:

- **Naming Conflicts**: Component variables can shadow existing variables in the scope
- **Overuse of Nesting**: Excessively deep nesting can reduce readability
- **Limited to Records**: Pattern matching works only with records, not with regular classes
- **Generic Types**: Care must be taken with generic type parameters

```
// INCORRECT: Shadowing variables
int x = 10;
```

```
if (obj instanceof Point(int x, int y)) { // Shadowing the outer x
    // x here refers to Point's x component, not the outer x
    System.out.println(x); // Prints Point's x, not 10
}

// CORRECT: Use different variable names
int outerX = 10;
if (obj instanceof Point(int x, int y)) {
    System.out.println("Outer: " + outerX + ", Point: " + x);
}

// INCORRECT: Too much nesting reducing readability
if (obj instanceof Response(
        Header(String protocol, int version, Status(int code, String message)),
        Body(Content(String type, byte[] data, Metadata(long timestamp, String signature)))
    )) {
    // Too complex to follow easily
}

// CORRECT: Break down complex patterns
if (obj instanceof Response(Header header, Body body)) {
    if (header instanceof Header(String protocol, int version, Status status)) {
        if (status instanceof Status(int code, String message)) {
            // Process status
        }
    }
    if (body instanceof Body(Content content)) {
        // Process content
    }
}
```

Pattern Matching for switch (Final)

What and Why: Pattern matching for switch statements, previewed in earlier releases and finalized in Java 21, extends the switch expression to work with patterns rather than just constants. This feature enables more powerful type-checking and data extraction in a single expression, making code more concise and readable.

Real-world Use Cases:

- Processing polymorphic data structures
- Converting between different data types
- Implementing visitor patterns
- Handling command patterns
- Validating and transforming data

Before (Java 20) / After (Java 21) Comparison:

Before (Java 20):

```
// Chain of instanceof checks
Object obj = getValue();
String result;

if (obj instanceof Integer i) {
    result = "Integer: " + i;
} else if (obj instanceof String s) {
    result = "String: " + s;
} else if (obj instanceof Double d) {
    result = "Double: " + d;
} else {
    result = "Unknown type";
}
```

After (Java 21):

```
// Pattern matching switch expression
Object obj = getValue();
String result = switch (obj) {
    case Integer i -> "Integer: " + i;
    case String s -> "String: " + s;
    case Double d -> "Double: " + d;
    default -> "Unknown type";
};
```

Comprehensive Example:

```
import java.time.*;
import java.util.*;

public class PatternMatchingSwitchExample {

    // Sealed interface for shapes
    sealed interface Shape permits Circle, Rectangle, Triangle {}
    record Circle(double radius) implements Shape {}
    record Rectangle(double width, double height) implements Shape {}
    record Triangle(double a, double b, double c) implements Shape {}

    // Records for nested pattern examples
    record Point(int x, int y) {}
    record ColoredShape(Shape shape, String color) {}

    // Records for command pattern
    sealed interface Command permits AddCommand, RemoveCommand, UpdateCommand {}
    record AddCommand(String item, int quantity) implements Command {}
    record RemoveCommand(String item) implements Command {}
    record UpdateCommand(String item, int newQuantity) implements Command {}

    // Records for API response handling
    sealed interface Response<T> permits SuccessResponse, ErrorResponse {}
    record SuccessResponse<T>(T data) implements Response<T> {}
    record ErrorResponse<T>(int code, String message) implements Response<T> {}

    public static void main(String[] args) {
        // 1. Basic pattern matching for types
        System.out.println("=== Basic Pattern Matching ===");

        List<Object> objects = List.of(
            Integer.valueOf(42),
            "Hello, pattern matching!",
            LocalDate.now(),
            Double.valueOf(3.14),
            List.of(1, 2, 3)
        );

        for (Object obj : objects) {
            String description = getDescription(obj);
            System.out.println(obj + " -> " + description);
        }

        // 2. Pattern matching with guards
        System.out.println("\n=== Pattern Matching with Guards ===");

        List<Integer> numbers = List.of(-10, -1, 0, 1, 42, 100);
        for (Integer num : numbers) {
            String category = categorizeNumber(num);
            System.out.println(num + " is " + category);
        }

        // 3. Pattern matching with sealed types
        System.out.println("\n=== Pattern Matching with Sealed Types ===");

        List<Shape> shapes = List.of(
            new Circle(5.0),
            new Rectangle(4.0, 6.0),
            new Rectangle(5.0, 5.0),
            new Triangle(3.0, 4.0, 5.0)
        );

        for (Shape shape : shapes) {
            double area = calculateArea(shape);
            String description = describeShape(shape);
            System.out.println(description + " - Area: " + area);
        }

        // 4. Nested pattern matching
        System.out.println("\n=== Nested Pattern Matching ===");

        List<Object> nestedObjects = List.of(
            new Point(3, 4),
            new ColoredShape(new Circle(7.0), "Red"),
            new ColoredShape(new Rectangle(2.0, 3.0), "Blue"),
            new Rectangle(1.0, 2.0)
        );

        for (Object obj : nestedObjects) {
            processNestedObject(obj);
```

```
        }

        // 5. Command pattern example
        System.out.println("\n=== Command Pattern Example ===");

        List<Command> commands = List.of(
            new AddCommand("Apple", 5),
            new RemoveCommand("Banana"),
            new UpdateCommand("Orange", 10)
        );

        for (Command cmd : commands) {
            String action = processCommand(cmd);
            System.out.println(action);
        }

        // 6. Handling null values
        System.out.println("\n=== Handling Null Values ===");

        List<String> stringsWithNull = new ArrayList<>();
        stringsWithNull.add("Hello");
        stringsWithNull.add(null);
        stringsWithNull.add("World");

        for (String str : stringsWithNull) {
            String result = processStringWithNull(str);
            System.out.println("Result: " + result);
        }

        // 7. Pattern matching with generics
        System.out.println("\n=== Pattern Matching with Generics ===");

        Response<User> successResponse = new SuccessResponse<>(new User("john", "John Doe"));
        Response<User> errorResponse = new ErrorResponse<>(404, "User not found");

        System.out.println(processResponse(successResponse));
        System.out.println(processResponse(errorResponse));
    }

    // 1. Basic pattern matching for different types
    static String getDescription(Object obj) {
        return switch (obj) {
            case Integer i -> "an Integer with value " + i;
            case String s -> "a String of length " + s.length();
            case LocalDate d -> "a Date: " + d.getDayOfWeek();
            case Double d when d > 0 -> "a positive Double";
            case Double d -> "a non-positive Double";
            case List<?> list -> "a List with " + list.size() + " elements";
            case null -> "a null reference";
            default -> "something else";
        };
    }

    // 2. Pattern matching with guards (conditional expressions)
    static String categorizeNumber(Integer num) {
        return switch (num) {
            case null -> "null";
            case Integer i when i < 0 -> "negative";
            case Integer i when i == 0 -> "zero";
            case Integer i when i > 0 && i < 10 -> "small positive";
            case Integer i when i >= 10 && i < 100 -> "medium positive";
            case Integer i when i >= 100 -> "large positive";
            default -> "impossible case";
        };
    }

    // 3. Pattern matching with sealed types
    static double calculateArea(Shape shape) {
        return switch (shape) {
            case Circle c -> Math.PI * c.radius() * c.radius();
            case Rectangle r -> r.width() * r.height();
            case Triangle t -> {
                // Heron's formula
                double s = (t.a() + t.b() + t.c()) / 2;
                yield Math.sqrt(s * (s - t.a()) * (s - t.b()) * (s - t.c()));
            }
        };
    }

    static String describeShape(Shape shape) {
        return switch (shape) {
```

```
            case Circle c -> "Circle with radius " + c.radius();
            case Rectangle r when r.width() == r.height() ->
                "Square with side " + r.width();
            case Rectangle r ->
                "Rectangle with width " + r.width() + " and height " + r.height();
            case Triangle t -> "Triangle with sides " + t.a() + ", " + t.b() + ", " + t.c();
        };
    }

    // 4. Nested pattern matching
    static void processNestedObject(Object obj) {
        switch (obj) {
            case Point p ->
                System.out.println("Point at (" + p.x() + ", " + p.y() + ")");

            case ColoredShape(Circle c, String color) ->
                System.out.println(color + " Circle with radius " + c.radius());

            case ColoredShape(Rectangle r, String color) when r.width() == r.height() ->
                System.out.println(color + " Square with side " + r.width());

            case ColoredShape(Rectangle r, String color) ->
                System.out.println(color + " Rectangle with dimensions " +
                                   r.width() + "x" + r.height());

            case ColoredShape(var s, var color) ->
                System.out.println(color + " Shape: " + s);

            case Rectangle r ->
                System.out.println("Plain Rectangle with dimensions " +
                                   r.width() + "x" + r.height());

            default ->
                System.out.println("Unknown object: " + obj);
        }
    }

    // 5. Command pattern example
    static String processCommand(Command cmd) {
        return switch (cmd) {
            case AddCommand ac ->
                "Adding " + ac.quantity() + " units of " + ac.item();

            case RemoveCommand rc ->
                "Removing all units of " + rc.item();

            case UpdateCommand uc ->
                "Updating " + uc.item() + " quantity to " + uc.newQuantity();
        };
    }

    // 6. Handling null values
    static String processStringWithNull(String str) {
        return switch (str) {
            case null -> "String was null";
            case String s when s.isEmpty() -> "String was empty";
            case String s when s.length() < 5 -> "Short string: " + s;
            default -> "Long string: " + str;
        };
    }

    // 7. Pattern matching with generics
    static String processResponse(Response<User> response) {
        return switch (response) {
            case SuccessResponse<User>(User user) ->
                "Success: User " + user.username() + " (" + user.name() + ")";

            case ErrorResponse<User>(int code, String message) ->
                "Error " + code + ": " + message;
        };
    }

    // Helper class for the generics example
    record User(String username, String name) {}
}
```

Key Features and Benefits:

1. **Type Patterns**: Match and cast based on type, extracting variables in a single step
2. **Pattern Guards**: Add conditions to pattern matches using `when` clauses

3. **Null Handling**: Explicit case for handling null values
4. **Exhaustiveness**: Compiler enforces handling all possible cases for sealed types
5. **Record Patterns**: Combine with record patterns for powerful destructuring

Common Pitfalls:

- **Dominance**: More specific patterns must come before more general patterns
- **Fall-through**: Pattern matching switch doesn't fall through by default
- **Exhaustiveness**: Must handle all cases for sealed hierarchies
- **Scope**: Pattern variables are scoped to their respective cases only

```
// INCORRECT: Pattern dominance problem
switch (obj) {
    case Object o -> "An object"; // This case dominates all others!
    case String s -> "A string"; // This case is unreachable
    default -> "Something else";
}

// CORRECT: More specific patterns first
switch (obj) {
    case String s -> "A string";
    case Integer i -> "An integer";
    case Object o -> "Some other object";
    case null -> "A null reference";
}

// INCORRECT: Missing case in a sealed hierarchy
sealed interface Vehicle permits Car, Truck, Motorcycle {}

// Compiler error: not exhaustive, missing Motorcycle
switch (vehicle) {
    case Car c -> "A car";
    case Truck t -> "A truck";
}

// CORRECT: Handle all cases
switch (vehicle) {
    case Car c -> "A car";
    case Truck t -> "A truck";
    case Motorcycle m -> "A motorcycle";
}

// INCORRECT: Using pattern variable outside its scope
switch (obj) {
    case String s -> {
        System.out.println(s.length());
    }
    case Integer i -> {
        System.out.println(i + 1);
    }
}
System.out.println(s); // Error: s is not in scope here

// CORRECT: Variables are scoped to their case
String result = switch (obj) {
    case String s -> "String of length " + s.length();
    case Integer i -> "Integer with value " + i;
    default -> "Unknown";
};
```

String Templates (Preview)

What and Why: String templates, introduced as a preview feature in Java 21, provide a more powerful way to create formatted strings by allowing embedded expressions directly within string literals. This makes string interpolation more readable and less error-prone compared to string concatenation or formatting methods.

Real-world Use Cases:

- Generating formatted output
- Creating SQL queries
- Building JSON/XML strings
- URL construction
- Template rendering

Before (Java 20) / After (Java 21) Comparison:

Before (Java 20):

```
String name = "John";
int age = 30;

// Using string concatenation
String message1 = "Hello, " + name + "! You are " + age + " years old.";

// Using String.format
String message2 = String.format("Hello, %s! You are %d years old.", name, age);

// Using formatted()
String message3 = "Hello, %s! You are %d years old.".formatted(name, age);
```

After (Java 21):

```
String name = "John";
int age = 30;

// Using string template (preview feature)
String message = STR."Hello, \{name}! You are \{age} years old.";
```

Example:

```
import java.time.LocalDate;
import java.util.*;
import static java.lang.StringTemplate.RAW;
import static java.util.FormatProcessor.FMT;
import static java.lang.StringTemplate.STR;

public class StringTemplatesExample {
    public static void main(String[] args) {
        // Basic variables for examples
        String name = "Alice";
        int age = 30;
        double salary = 75000.50;
        LocalDate birthdate = LocalDate.of(1993, 5, 15);

        // 1. Basic string template usage
        System.out.println("=== Basic String Templates ===");

        // Using STR template processor
        String greeting = STR."Hello, \{name}! You are \{age} years old.";
        System.out.println(greeting);

        // Using expressions in templates
        String info = STR."Age in months: \{age * 12}";
        System.out.println(info);

        // Method calls in expressions
        String upperName = STR."Uppercase name: \{name.toUpperCase()}";
        System.out.println(upperName);

        // 2. Formatted output with FMT
        System.out.println("\n=== Formatted Output with FMT ===");

        // Currency formatting
        String salaryInfo = FMT."Salary: $\{salary%.2f} per year";
        System.out.println(salaryInfo);

        // Date formatting
        String birthdateInfo = FMT."Birthdate: \{birthdate%td %tb, %tY}";
        System.out.println(birthdateInfo);

        // Number formatting with width
        System.out.println(FMT."ID: \{123%05d}");

        // 3. Multi-line templates
        System.out.println("\n=== Multi-line Templates ===");

        String multiline = STR."""
            User Profile:
            -------------
            Name: \{name}
            Age: \{age}
            Birthdate: \{birthdate}
            Annual Income: $\{salary}
            """;
```

```
        System.out.println(multiline);

        // 4. Templates with conditional logic
        System.out.println("\n=== Templates with Conditional Logic ===");

        boolean isPremium = true;

        String status = STR."Account status: \{isPremium ? "Premium" : "Standard"}";
        System.out.println(status);

        // More complex condition
        String ageCategory = STR."Age category: \{
            age < 18 ? "Minor" :
            age < 65 ? "Adult" :
                       "Senior"
        }";
        System.out.println(ageCategory);

        // 5. Templates with loops (embedded expressions)
        System.out.println("\n=== Templates with Collection Processing ===");

        List<String> fruits = List.of("Apple", "Banana", "Cherry", "Date");

        String fruitList = STR."Fruits: \{
            fruits.stream()
                  .map(String::toUpperCase)
                  .reduce((a, b) -> a + ", " + b)
                  .orElse("")
        }";
        System.out.println(fruitList);

        // 6. Templates for structured data
        System.out.println("\n=== Templates for Structured Data ===");

        // JSON example
        String jsonUser = STR."""
            {
              "name": "\{name}",
              "age": \{age},
              "premium": \{isPremium},
              "birthdate": "\{birthdate}",
              "contact": {
                "email": "\{name.toLowerCase()}@example.com",
                "phone": "+1-555-123-4567"
              }
            }
            """;

        System.out.println("JSON:\n" + jsonUser);

        // HTML example
        String htmlProfile = STR."""
            <div class="user-profile">
              <h2>\{name}'s Profile</h2>
              <ul>
                <li><strong>Age:</strong> \{age}</li>
                <li><strong>Birthdate:</strong> \{birthdate}</li>
                <li><strong>Status:</strong> \{isPremium ? "Premium" : "Standard"}</li>
              </ul>
            </div>
            """;

        System.out.println("\nHTML:\n" + htmlProfile);

        // SQL example
        String sqlQuery = STR."""
            SELECT *
            FROM users
            WHERE name = '\{name}'
              AND age >= \{age}
              AND is_premium = \{isPremium ? 1 : 0}
            LIMIT 10
            """;

        System.out.println("\nSQL:\n" + sqlQuery);

        // 7. Custom template processor
        System.out.println("\n=== Raw Templates and Custom Processors ===");

        // Using RAW processor
```

```
        StringTemplate template = RAW."User: \{name}, Age: \{age}";
        System.out.println("Raw template: " + template);
        System.out.println("Fragments: " + template.fragments());
        System.out.println("Values: " + Arrays.toString(template.values()));

        // Custom processing (uppercase all values)
        String upperCased = processUppercase(RAW."Hello, \{name}! Your age is \{age}.");
        System.out.println("\nCustom processed: " + upperCased);
    }

    // Example of custom template processing
    static String processUppercase(StringTemplate template) {
        StringBuilder result = new StringBuilder();

        // Get the fragments and values
        List<String> fragments = template.fragments();
        Object[] values = template.values();

        // Combine fragments and uppercased values
        for (int i = 0; i < fragments.size(); i++) {
            result.append(fragments.get(i));

            if (i < values.length) {
                // Convert value to uppercase string
                String valueStr = String.valueOf(values[i]).toUpperCase();
                result.append(valueStr);
            }
        }

        return result.toString();
    }
}
```

Key Concepts and Features:

1. **Template Processors**:
   - `STR`: Basic string interpolation
   - `FMT`: Formatted output similar to `String.format()`
   - `RAW`: Raw template objects for custom processing
2. **Syntax**:
   - Embedded expressions: `\{expression}`
   - Format specifiers: `\{value%format}`
   - Multi-line templates: Triple-quoted strings with embedded expressions
3. **Advantages**:
   - More readable than string concatenation
   - Type-safe unlike format strings
   - Less verbose than `String.format()`
   - Supports text blocks with interpolation

Common Pitfalls:

- **Preview Status**: Being a preview feature, the syntax or API might change in future releases
- **Performance**: String templates might have different performance characteristics compared to traditional methods
- **Complex Expressions**: Embedding complex expressions can make templates harder to read
- **Escaping Braces**: Need to use escapes when actual curly braces are needed in the output

```
// INCORRECT: Overcomplex embedded expressions
String message = STR."Result: \{
    list.stream()
        .filter(s -> s.length() > 3)
        .map(String::toUpperCase)
        .collect(Collectors.joining(", "))
}";

// BETTER: Extract complex logic to variables
String processed = list.stream()
                    .filter(s -> s.length() > 3)
                    .map(String::toUpperCase)
                    .collect(Collectors.joining(", "));

String message = STR."Result: \{processed}";

// INCORRECT: Missing escape for literal braces
```

```
String json = STR."{ "name": "\{name}" }"; // Syntax error

// CORRECT: Escape literal braces
String json = STR."\\{ \"name\": \"\{name}\" \\}";

// BETTER: Use text blocks
String json = STR."""
    {
      "name": "\{name}"
    }
    """;
```

Sequenced Collections

What and Why: Sequenced Collections, introduced in Java 21, is a new collections framework enhancement that provides a standard way to work with collections that have a defined encounter order. It adds new interfaces that support operations for accessing the first and last elements and obtaining views of the collection in both forward and reverse order.

Real-world Use Cases:

- Queue processing with access to both ends
- Navigation history (forward/backward)
- Undo/redo operations
- Tree traversal algorithms
- UI component ordering

Before (Java 20) / After (Java 21) Comparison:

Before (Java 20):

```
// Different methods for different collection types
List<String> list = new ArrayList<>();
String first = list.isEmpty() ? null : list.get(0);
String last = list.isEmpty() ? null : list.get(list.size() - 1);

Deque<String> deque = new ArrayDeque<>();
String first = deque.peekFirst();
String last = deque.peekLast();

// No standard way to reverse order for all collections
List<String> reversed = new ArrayList<>(list);
Collections.reverse(reversed);
```

After (Java 21):

```
// Unified interface for all sequenced collections
SequencedCollection<String> sequenced = new ArrayList<>();
String first = sequenced.getFirst();
String last = sequenced.getLast();

// Standard way to get reversed view
SequencedCollection<String> reversed = sequenced.reversed();
```

Comprehensive Example:

```
import java.util.*;

public class SequencedCollectionsExample {
    public static void main(String[] args) {
        // 1. SequencedCollection examples
        System.out.println("=== SequencedCollection Examples ===");

        // ArrayList implements SequencedCollection
        SequencedCollection<String> fruits = new ArrayList<>();
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Cherry");
        fruits.add("Date");

        System.out.println("Fruits: " + fruits);

        // Access first and last elements
        System.out.println("First fruit: " + fruits.getFirst());
        System.out.println("Last fruit: " + fruits.getLast());
```

```
        // Add to beginning or end
        fruits.addFirst("Apricot");
        fruits.addLast("Elderberry");
        System.out.println("After adding at both ends: " + fruits);

        // Remove from beginning or end
        String firstRemoved = fruits.removeFirst();
        String lastRemoved = fruits.removeLast();
        System.out.println("Removed first: " + firstRemoved);
        System.out.println("Removed last: " + lastRemoved);
        System.out.println("After removal: " + fruits);

        // Get reversed view
        SequencedCollection<String> reversedFruits = fruits.reversed();
        System.out.println("Reversed view: " + reversedFruits);

        // Changes to the reversed view reflect in the original
        reversedFruits.addFirst("Zucchini");
        System.out.println("Original after adding to reversed: " + fruits);
        System.out.println("Reversed after adding: " + reversedFruits);

        // 2. SequencedSet examples
        System.out.println("\n=== SequencedSet Examples ===");

        // LinkedHashSet implements SequencedSet
        SequencedSet<String> colors = new LinkedHashSet<>();
        colors.add("Red");
        colors.add("Green");
        colors.add("Blue");

        System.out.println("Colors: " + colors);
        System.out.println("First color: " + colors.getFirst());
        System.out.println("Last color: " + colors.getLast());

        // Add elements (will be ignored if they already exist)
        colors.addFirst("Red"); // No effect, already exists
        colors.addLast("Purple");
        System.out.println("After additions: " + colors);

        // Reversed set view
        SequencedSet<String> reversedColors = colors.reversed();
        System.out.println("Reversed colors: " + reversedColors);

        // 3. SequencedMap examples
        System.out.println("\n=== SequencedMap Examples ===");

        // LinkedHashMap implements SequencedMap
        SequencedMap<Integer, String> employees = new LinkedHashMap<>();
        employees.put(101, "Alice");
        employees.put(102, "Bob");
        employees.put(103, "Charlie");

        System.out.println("Employees: " + employees);

        // Access first and last entries
        System.out.println("First entry: " + employees.firstEntry());
        System.out.println("Last entry: " + employees.lastEntry());

        // Get first and last keys
        System.out.println("First key: " + employees.firstKey());
        System.out.println("Last key: " + employees.lastKey());

        // Add at beginning or end
        employees.putFirst(100, "Alice's Manager");
        employees.putLast(104, "New Hire");
        System.out.println("After adding at both ends: " + employees);

        // Remove from beginning or end
        Map.Entry<Integer, String> firstEntry = employees.pollFirstEntry();
        Map.Entry<Integer, String> lastEntry = employees.pollLastEntry();

        System.out.println("Removed first: " + firstEntry);
        System.out.println("Removed last: " + lastEntry);
        System.out.println("After removal: " + employees);

        // Get sequenced views
        SequencedMap<Integer, String> reversedMap = employees.reversed();
        System.out.println("Reversed map: " + reversedMap);

        SequencedSet<Integer> keySet = employees.sequencedKeySet();
```

```
        System.out.println("Sequenced key set: " + keySet);

        SequencedCollection<String> values = employees.sequencedValues();
        System.out.println("Sequenced values: " + values);

        SequencedSet<Map.Entry<Integer, String>> entrySet = employees.sequencedEntrySet();
        System.out.println("Sequenced entry set: " + entrySet);

        // 4. Practical examples
        System.out.println("\n=== Practical Examples ===");

        // Example: Command history with undo/redo functionality
        commandHistoryExample();

        // Example: Navigation breadcrumbs
        navigationBreadcrumbsExample();

        // Example: Processing queue from both ends
        processingQueueExample();
    }

    // Example: Command history with undo/redo
    static void commandHistoryExample() {
        System.out.println("\nCommand History Example:");

        // Create a command history using a LinkedList (implements SequencedCollection)
        SequencedCollection<String> commandHistory = new LinkedList<>();

        // Add some commands
        commandHistory.addLast("Create File");
        commandHistory.addLast("Edit Text");
        commandHistory.addLast("Save File");
        commandHistory.addLast("Close File");

        System.out.println("Command History: " + commandHistory);

        // Simulate undo (remove last command)
        String undoneCommand = commandHistory.removeLast();
        System.out.println("Undo: " + undoneCommand);
        System.out.println("After undo: " + commandHistory);

        // Add new command
        commandHistory.addLast("Print File");
        System.out.println("After new command: " + commandHistory);

        // Undo multiple commands
        System.out.println("\nUndo multiple commands:");
        String undo1 = commandHistory.removeLast();
        String undo2 = commandHistory.removeLast();

        // Store in redo stack
        SequencedCollection<String> redoStack = new LinkedList<>();
        redoStack.addFirst(undo1);
        redoStack.addFirst(undo2);

        System.out.println("After multiple undos: " + commandHistory);
        System.out.println("Redo stack: " + redoStack);

        // Redo a command
        String redoCommand = redoStack.removeFirst();
        commandHistory.addLast(redoCommand);

        System.out.println("After redo: " + commandHistory);
    }

    // Example: Navigation breadcrumbs
    static void navigationBreadcrumbsExample() {
        System.out.println("\nNavigation Breadcrumbs Example:");

        // Create a breadcrumb trail using LinkedHashSet (implements SequencedSet)
        SequencedSet<String> breadcrumbs = new LinkedHashSet<>();

        // User navigates through pages
        breadcrumbs.addLast("Home");
        breadcrumbs.addLast("Products");
        breadcrumbs.addLast("Electronics");
        breadcrumbs.addLast("Computers");
        breadcrumbs.addLast("Laptops");

        // Display breadcrumb trail
        System.out.println("Current path: " + String.join(" > ", breadcrumbs));
```

```
        // User clicks "Electronics" in the breadcrumb trail (navigate back)
        // Remove all breadcrumbs after Electronics
        while (!breadcrumbs.getLast().equals("Electronics")) {
            breadcrumbs.removeLast();
        }

        System.out.println("After clicking 'Electronics': " + String.join(" > ", breadcrumbs));

        // Now navigate to a different path
        breadcrumbs.addLast("Smartphones");

        System.out.println("New path: " + String.join(" > ", breadcrumbs));

        // Display reverse navigation history
        System.out.println("Reverse history: " + String.join(" < ", breadcrumbs.reversed()));
    }

    // Example: Processing queue from both ends
    static void processingQueueExample() {
        System.out.println("\nProcessing Queue Example:");

        // Work queue that supports priority items
        SequencedCollection<String> workQueue = new LinkedList<>();

        // Add regular tasks to the end
        workQueue.addLast("Regular Task 1");
        workQueue.addLast("Regular Task 2");
        workQueue.addLast("Regular Task 3");

        // Add high priority task to the front
        workQueue.addFirst("URGENT: Fix critical bug");

        // Add another regular task
        workQueue.addLast("Regular Task 4");

        // Add another high priority task
        workQueue.addFirst("URGENT: Server down");

        System.out.println("Work Queue: " + workQueue);

        // Process the queue (always take from the front)
        System.out.println("\nProcessing queue:");
        while (!workQueue.isEmpty()) {
            String task = workQueue.removeFirst();
            System.out.println("Processing: " + task);

            // Simulate occasionally deferring tasks to the end of the queue
            if (task.contains("Task 2")) {
                System.out.println("Deferring task to later");
                workQueue.addLast(task + " (deferred)");
            }
        }
    }
}
```

Key Interfaces and Methods:

1. **SequencedCollection**:
   - `getFirst()`: Returns the first element
   - `getLast()`: Returns the last element
   - `addFirst(E)`: Adds an element at the beginning
   - `addLast(E)`: Adds an element at the end
   - `removeFirst()`: Removes and returns the first element
   - `removeLast()`: Removes and returns the last element
   - `reversed()`: Returns a view of the collection in reverse order
2. **SequencedSet**:
   - Extends SequencedCollection
   - Maintains the set property (no duplicates)
3. **SequencedMap**:
   - `firstEntry()`, `lastEntry()`: Returns the first/last entry
   - `firstKey()`, `lastKey()`: Returns the first/last key
   - `putFirst(K, V)`, `putLast(K, V)`: Puts an entry at the beginning/end
   - `pollFirstEntry()`, `pollLastEntry()`: Removes and returns the first/last entry

- `sequencedKeySet()`: Returns a SequencedSet view of the keys
- `sequencedValues()`: Returns a SequencedCollection view of the values
- `sequencedEntrySet()`: Returns a SequencedSet view of the entries
- `reversed()`: Returns a reversed view of the map

Implementations:

- **SequencedCollection**: ArrayList, LinkedList, ArrayDeque
- **SequencedSet**: LinkedHashSet, TreeSet
- **SequencedMap**: LinkedHashMap, TreeMap

Common Pitfalls:

- **Optional Operations**: Some methods might throw UnsupportedOperationException for certain implementations
- **Performance Variations**: Different implementations have different performance characteristics for operations
- **Unmodifiable Views**: Some collection views might not support modification operations
- **Concurrent Modification**: Changes during iteration might cause ConcurrentModificationException

```
// INCORRECT: Assuming all implementations support all operations
SequencedCollection<String> unmodifiable =
    Collections.unmodifiableSequencedCollection(someCollection);
unmodifiable.addFirst("item"); // Throws UnsupportedOperationException

// CORRECT: Check implementation or wrap in try-catch
try {
    collection.addFirst("item");
} catch (UnsupportedOperationException e) {
    System.out.println("Operation not supported by this implementation");
}

// INCORRECT: Not considering performance characteristics
SequencedCollection<String> linkedList = new LinkedList<>();
// Frequent random access is inefficient for LinkedList
for (int i = 0; i < linkedList.size(); i++) {
    String item = ((List<String>)linkedList).get(i);
    // Process item
}

// CORRECT: Use iterator for sequential access
for (String item : linkedList) {
    // Process item
}

// INCORRECT: Modifying during iteration
for (String item : sequencedSet) {
    if (item.startsWith("temp")) {
        sequencedSet.remove(item); // ConcurrentModificationException
    }
}

// CORRECT: Use iterator's remove method or collect items to remove
Iterator<String> iterator = sequencedSet.iterator();
while (iterator.hasNext()) {
    String item = iterator.next();
    if (item.startsWith("temp")) {
        iterator.remove(); // Safe removal during iteration
    }
}
```

Learning Path Progression

Now that we've explored the major features introduced from Java 8 to Java 21, let's create a structured learning path to help you systematically build your knowledge. This progression is designed to be incremental, with each step building upon the previous ones.

Phase 1: Java 8 Fundamentals

1. **Lambda Expressions and Functional Interfaces**
    - Start with understanding how to write basic lambda expressions
    - Learn about built-in functional interfaces in `java.util.function`
    - Practice converting anonymous classes to lambdas
2. **Stream API**
    - Begin with simple stream operations (filter, map, collect)
    - Move to more complex operations (flatMap, reduce)
    - Learn about specialized streams (IntStream, LongStream, etc.)

- Practice combining multiple stream operations

3. **Optional**
   - Understand the purpose and proper use cases
   - Learn the core methods (map, flatMap, orElse, etc.)
   - Practice refactoring null-checking code to use Optional
4. **Default Methods and Static Methods in Interfaces**
   - Understand how to extend interfaces without breaking implementations
   - Practice creating interfaces with default methods
   - Learn about method resolution rules with multiple inheritance
5. **New Date/Time API**
   - Learn the core classes (LocalDate, LocalTime, LocalDateTime, etc.)
   - Practice date/time manipulations and calculations
   - Understand time zones and conversions
6. **CompletableFuture**
   - Start with basic asynchronous operations
   - Learn about composition and chaining
   - Practice error handling and timeout management

Phase 2: Java 9-16 Features

1. **Java 9 Module System**
   - Understand module basics (requires, exports)
   - Create a simple modular application
   - Learn about services and providers
   - Explore migration strategies for existing applications
2. **Collection Factory Methods**
   - Replace collection initialization code with factory methods
   - Understand the limitations (immutability, null handling)
3. **Try with Resources Enhancements**
   - Update existing resource management code
4. **Private Interface Methods**
   - Refactor interfaces to extract common code into private methods
5. **Convenience APIs**
   - Explore enhancements to Stream, Optional, etc.
6. **Records**
   - Identify and refactor data carrier classes to records
   - Learn about compact constructors and custom methods in records
7. **Switch Expressions**
   - Refactor complex switch statements to expressions
   - Use the new arrow syntax and yield statement
8. **Text Blocks**
   - Improve readability of SQL, HTML, JSON strings
   - Learn about escape sequences and formatting
9. **Pattern Matching for instanceof**
   - Simplify type checking and casting operations
   - Combine with other features for cleaner code
10. **Local Variable Type Inference**
    - Apply var where it improves readability
    - Understand when not to use var

Phase 3: Java 17-21 Features

1. **Sealed Classes**

- Identify hierarchies that can benefit from sealing
- Practice implementing permitted subclasses
- Combine with pattern matching for exhaustive handling

2. **Pattern Matching for switch**
   - Refactor type-checking code to use pattern matching
   - Use guards for conditional matching
   - Combine with sealed classes for compile-time exhaustiveness checking
3. **Record Patterns**
   - Apply nested pattern matching for record hierarchies
   - Refactor deep accessor chains with record patterns
4. **Virtual Threads**
   - Identify blocking code that could benefit from virtual threads
   - Replace thread pools with virtual threads
   - Understand and address pinning issues
5. **Structured Concurrency**
   - Refactor concurrent code to use StructuredTaskScope
   - Learn about different shutdown policies
   - Implement timeout handling
6. **String Templates**
   - Replace string concatenation and formatting with templates
   - Use different processors (STR, FMT, RAW)
   - Create custom template processors
7. **Sequenced Collections**
   - Update collection code to use new sequenced APIs
   - Implement bidirectional access patterns

Integration Projects

To solidify your understanding, consider working on these integration projects that combine multiple Java features:

1. **RESTful API Service**
   - Use virtual threads for handling requests
   - Apply records for request/response objects
   - Use pattern matching for request processing
   - Implement structured concurrency for parallel operations
   - Apply sequenced collections for managing ordered data
2. **Data Processing Pipeline**
   - Use Stream API for data transformations
   - Apply CompletableFuture for asynchronous processing
   - Use records for data containers
   - Apply pattern matching for data filtering
   - Implement module system for pipeline components
3. **Reactive Dashboard Application**
   - Use virtual threads for background processing
   - Apply CompletableFuture for non-blocking operations
   - Use record patterns for event handling
   - Implement string templates for dynamic content generation
4. **Configuration Management System**
   - Use sealed classes for configuration types
   - Apply pattern matching for config processing
   - Use records for configuration entries
   - Implement sequenced collections for ordered configs

Reference Table

Here's a comprehensive reference table summarizing the key syntax and use cases for the most important features from Java 8 to Java 21.

Java 8 Features

Feature	Syntax	Use Cases	Example
Lambda Expressions	`(parameters) -> expression` or `(parameters) -> { statements; }`	Event handlers, filters, functional logic	`(int x, int y) -> x + y` or `s -> s.length()`
Functional Interfaces	`@FunctionalInterface interface Name { returnType method(params); }`	Define SAM interfaces, API design	`Function<String, Integer>`, `Predicate<T>`, `Consumer<T>`
Method References	`ClassName::staticMethod`, `instance::instanceMethod`, `ClassName::instanceMethod`, `ClassName::new`	Concise lambdas, factory methods	`String::length`, `System.out::println`
Stream API	`collection.stream().operation().collect()`	Data processing, transformations	`list.stream().filter(s -> s.length() > 3).collect(Collector`
Optional	`Optional.of(value)`, `Optional.ofNullable(value)`, `optional.orElse(default)`	Null handling, expressing absent values	`Optional.ofNullable(user).map(User::getAddress).orElse(DEFA`
Default Methods	`default returnType methodName(params) { }`	Interface evolution, API enhancement	`default void sort() { Collections.sort(this); }`
New Date/Time	`LocalDate.now()`, `ZonedDateTime.of()`	Date manipulation, time zones	`LocalDate.now().plusDays(7)`
CompletableFuture	`CompletableFuture.supplyAsync(() -> computation)`	Async operations, parallel execution	`future.thenApply(String::toUpperCase).thenAccept(System.out`

Java 9-16 Features

Feature	Syntax	Use Cases	Example
Module System	`module name { requires module; exports package; }`	Encapsulation, dependency management	`module com.app { requires java.sql; exports com.app.api; }`
Collection Factory	`List.of()`, `Set.of()`, `Map.of()`	Immutable collection creation	`List.of("a", "b", "c")`, `Map.of("k1", "v1", "k2", "v2")`
Private Interface Methods	`private returnType methodName(params) { }`	Code reuse in interfaces	`private void helperMethod() { }`
var Type Inference	`var variableName = initializer;`	Local variable declarations	`var list = new ArrayList<String>()`
try-with-resources Enhancement	`try (resource)` (resource declared outside)	Resource management	`try (inputStream)`
Records	`record Name(Type field, Type field2) { }`	Data carrier classes, DTOs	`record Point(int x, int y) { }`
Switch Expressions	`switch(value) { case pattern -> result; }`	Pattern-based dispatch	`return switch(day) { case MONDAY -> "Work"; default -> "Rest"; };`
Text Blocks	`"""multi-line string"""`	Multi-line strings, SQL, HTML	`String html = """<div>content</div>"""`
instanceof Pattern Matching	`if (obj instanceof Type name)`	Type checking, casting	`if (obj instanceof String s) { use(s); }`

Java 17-21 Features

Feature	Syntax	Use Cases	Example
Sealed Classes	`sealed class/interface Name permits Sub1, Sub2 { }`	Restricted hierarchies, exhaustiveness	`sealed interface Shape permits Circle, Rectangle { }`
Pattern Matching for switch	`switch(obj) { case Pattern p -> result; }`	Type-based dispatch, data extraction	`switch(shape) { case Circle c -> c.radius(); }`
Record Patterns	`if (obj instanceof Record(Type1 field1, Type2 field2))`	Data destructuring, nested patterns	`if (obj instanceof Point(int x, int y))`

Feature	Syntax	Use Cases	Example
Virtual Threads	`Thread.startVirtualThread(() -> task)`	Concurrent applications, I/O-bound operations	`ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor()`
Structured Concurrency	`try (var scope = new StructuredTaskScope<>())`	Coordinated concurrent tasks	`scope.fork(() -> task); scope.join();`
String Templates	`STR."Text \{expression}"`	String interpolation, formatting	`STR."Hello, \{name}! Today is \{date}."`
Sequenced Collections	`sequencedCollection.getFirst()`, `sequencedCollection.getLast()`	Ordered data access, bidirectional iteration	`LinkedHashMap<K,V> map = ...; Entry<K,V> first = map.firstEntry();`

Conclusion

Congratulations on completing this comprehensive guide to modern Java features! You've covered significant ground, from the functional programming revolution in Java 8 to the cutting-edge concurrency and pattern matching improvements in Java 21.

As you continue your Java journey, remember these key principles:

1. **Be Incremental**: Start with the fundamentals (lambdas, streams, optionals) before moving to advanced features.
2. **Be Practical**: For each feature, look for opportunities to apply it in your current codebase.
3. **Be Balanced**: Not every new feature is appropriate for every situation. Choose the right tool for the job.
4. **Be Current**: Follow the JDK Enhancement Proposals (JEPs) and Early Access builds to stay ahead of upcoming features.
5. **Be Community-Oriented**: Engage with the Java community through forums, blogs, and conferences to learn best practices and patterns.

The evolution of Java from 8 to 21 represents a remarkable transformation that has preserved backward compatibility while introducing modern paradigms. By mastering these features, you're well-equipped to write more concise, expressive, and powerful Java code for years to come.

Happy coding!