Iraqi Kurdistan Region
Salahaddin University - Erbil

College of Engineering

Department of Software & Informatics

ACP Report

Functional method programming

Prepared by:

Olivia Salim

Aynda Mohammedkher

Lava Ramzi

Bawar Surdash

Supervisor by

Dr. Polla Fattah

2024 - 2025

Contents

1. Introduction

In Java programming, a method can be defined as a block of code with a name that controls the behavior of an object. In the Java programming language, a method might be just a collection of statements that are executed for a specific task. Code snippets are very useful to improve modularity while minimizing repetition and organizing a codebase. To facilitate the design and testing of software, methods as helpful for developers to simplify and separate operations or calculations.

This report will take you through the structure, types and implementation of methods in Java. Above all, it contains their syntax, purpose and flow of execution. The article discusses instance methods, static methods, abstract methods, constructors, and their differences based on design needs and development use cases. The report also shows how methods promote object-oriented principles via the movement of a class or object from one entity to another through interfaces or others.

Additionally, it shows how methods work in real-world Java. Furthermore, it demonstrates how methods help write code which is scalable, readable, and maintainable. Methods are a flexible and effective mechanism to implement any functionality. In fact, whether it is receiving user input, performing calculations, or managing internal state you can use methods to complete the tasks. By means of examples and analysis this report gives a clear understanding of Java methods. The report provides an insight on how the Java methods can be used to make a robust and professional software system.

2. Basics and Anatomy of Methods

A Java method comprises several critical elements:

- **Access Modifier:** Defines visibility (public, private, protected).
- **Return Type:** Data type of value returned (void, int, String, etc.).
- **Method Name:** Identifier for the method (e.g., calculateSum).
- **Parameter List:** Inputs accepted by the method.
  Method Body: Enclosed by {} and contains executable statements.
  Each method exists inside a class or an interface, embodying Java's object-oriented paradigm where behavior is tied to data.

3. Types of Methods

Java offers several categories of methods:
Predefined Methods: Provided by Java libraries (e.g., Math.sqrt()).
User-Defined Methods: Created by developers to perform specific tasks.
Static Methods: Belong to the class and do not require object creation.
Instance Methods: Belong to objects and operate on instance data.
Abstract Methods: Declared without implementation, meant for subclasses.
Final Methods: Cannot be overridden in subclasses.
Synchronized Methods: Control access in multithreaded programs.

Each method type has specific applications that improve modularity, reuse, and clarity.

4. Static vs Instance Methods

Static Methods:
- Invoked without an object.
- Cannot access instance variables directly.
- Useful for utility or helper operations.

Instance Methods:
- Require an object to be called.
- Can access instance variables and other instance methods.
- Represent object-specific behaviors.

5. Abstract Methods

Abstract methods serve as placeholders for functionality that must be implemented by subclasses. They:
- Have no body.
- Must be inside an abstract class or an interface.
- Enforce a contract for derived classes to provide their own implementations.

Example:

```
abstract class Shape {
    abstract void draw();
}
```

6. Final Methods

Final methods are declared using the final keyword to prevent overriding:

```
public final void displayInfo() { ... }
```

They protect core logic by ensuring subclasses do not modify critical method behavior, enhancing system stability.

7. Synchronized Methods

In multithreaded environments, synchronized methods ensure that only one thread accesses the method at a time:

```
public synchronized void updateCounter() { ... }
```

- This prevents race conditions, ensuring data integrity across concurrent threads.

8. Default Methods in Interfaces

Default methods were introduced in Java 8 to allow adding new methods to interfaces without breaking existing implementations:

```
interface Vehicle {
   default void start() {
      System.out.println("Vehicle is starting");
   }
```

}

- They provide flexibility in extending APIs without affecting older codebases.

9. Native and Strictfp Methods

- **Native Methods**: Interface with non-Java code (C, C++) for performance or system-specific operations.

- **Strictfp Methods**: Enforce strict floating-point calculations across platforms for consistency.

Example:

```
public strictfp double calculate() { ... }
```

10. Method Overloading and Overriding

- **Overloading**: Same method name, different parameter lists within the same class.

- **Overriding**: Same method signature redefined in a subclass.

Overloading supports compile-time polymorphism, while overriding enables runtime polymorphism.

11. Method Overloading

Method overloading allows flexibility within a class:

```
public int sum(int a, int b) { ... }

public int sum(int a, int b, int c) { ... }
```

- Different parameter sets accommodate varying input scenarios, enhancing code readability and utility.

12. Method Overriding

In overriding, a subclass modifies the behavior of a superclass method:

```
class Animal {
   void sound() { System.out.println("Animal sound"); }
}
class Dog extends Animal {
   void sound() { System.out.println("Bark"); }
}
```

- It enables polymorphic behavior, improving software flexibility.

13. Recursion in Methods

Recursion occurs when a method calls itself to solve smaller instances of a problem:

```
int factorial(int n) {
   if (n == 1) return 1;
   else return n * factorial(n - 1);
```

```
}
```

- It is fundamental to divide-and-conquer strategies, backtracking, and tree traversals.

14. Best Practices for Method Design

- Keep methods short and focused.
- Use meaningful names that convey method purpose.
- Minimize the number of parameters.
- Avoid side effects where possible.
- Document the purpose and usage clearly.
- Good method design enhances maintainability, readability, and testing.

15. Scope and Lifetime of Variables in Methods

- **Local Variables**: Declared inside methods; exist only during method execution.

- **Instance Variables**: Belong to objects; exist as long as the object exists.

- **Static Variables**: Shared among all instances; tied to the class.

Understanding variable scope improves memory management and error prevention.

16. Lambda Expressions and Functional Interfaces

Java 8 introduced lambda expressions to simplify passing behavior as parameters:

```
Runnable r = () -> System.out.println("Running...");
```

Functional interfaces (interfaces with a single abstract method) are key to supporting lambdas (e.g., Runnable, Callable).

17. Exception Handling inside Methods

Methods may throw exceptions or handle them internally:

```
public void readFile()  throws IOException { ... }
```

- Use try-catch-finally blocks for robust error handling and declare thrown exceptions where necessary for compile-time safety.

18. Access Modifiers and Their Role in Methods

Java defines four access levels:

- **Public**: Accessible from everywhere.

- **Private**: Accessible only within the class.

- **Protected**: Accessible in the same package and subclasses.

- **Package-Private (default)**: Accessible within the same package.

Choosing the right access level protects encapsulation and enforces system design boundaries.

19. Performance Considerations in Method Calls

Java Virtual Machine (JVM) optimizations improve method call performance through:

- **Inlining**: Replacing method calls with method body code.

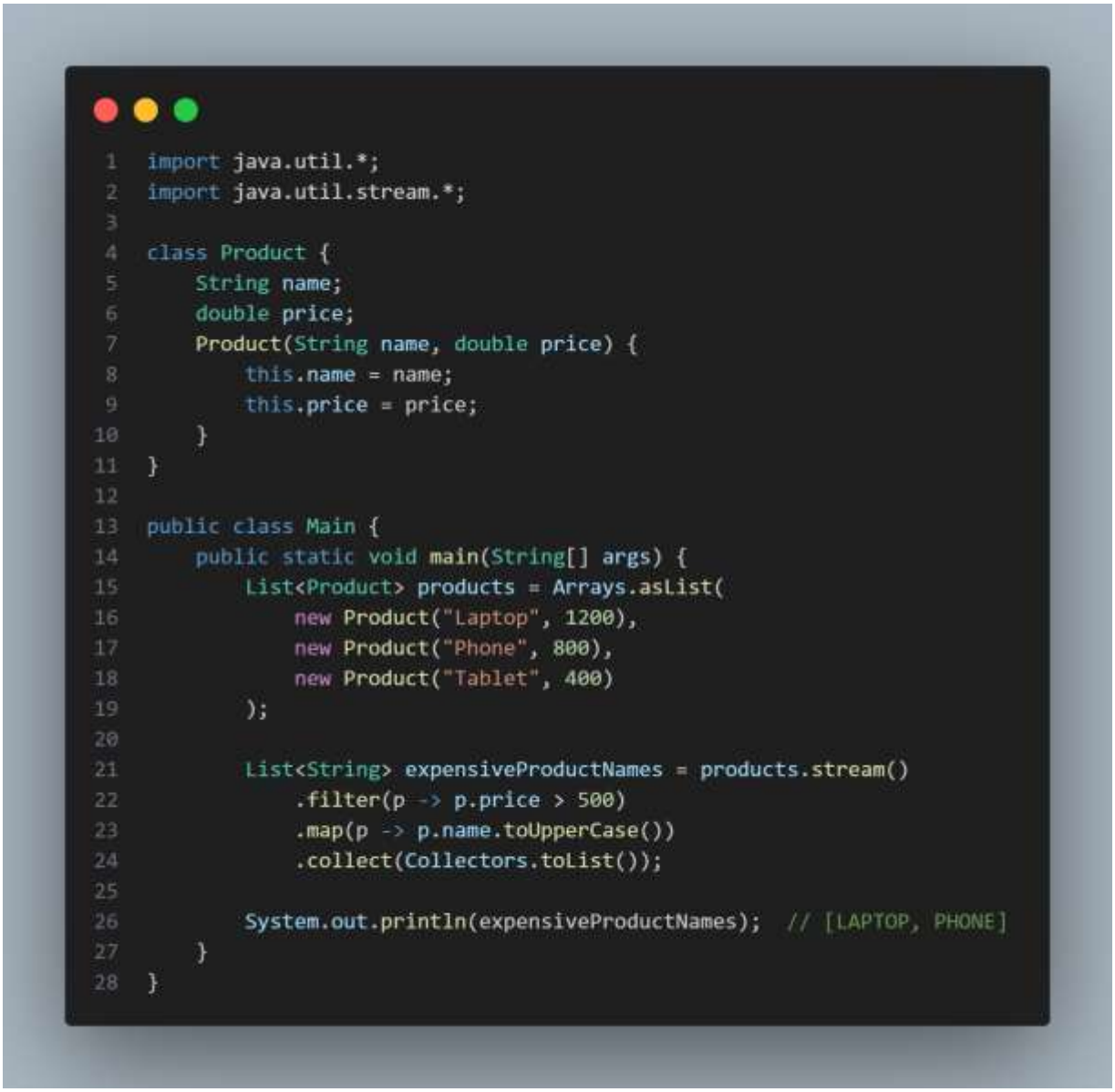
- **Just-In-Time (JIT) Compilation**: Compiling bytecode to machine code at runtime.

- **Dynamic Dispatch**: Efficiently selecting the correct overridden method at runtime.

Efficient method design and avoiding deep call stacks contribute to better application performance.

advanced functional programming code example in Java

- (Using Streams with Complex Filters and Mapping)



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

class Product {
    String name;
    double price;
    Product(String name, double price) {
        this.name = name;
        this.price = price;
    }
}

public class Main {
    public static void main(String[] args) {
        List<Product> products = Arrays.asList(
            new Product("Laptop", 1200),
            new Product("Phone", 800),
            new Product("Tablet", 400)
        );

        List<String> expensiveProductNames = products.stream()
            .filter(p -> p.price > 500)
            .map(p -> p.name.toUpperCase())
            .collect(Collectors.toList());

        System.out.println(expensiveProductNames);  // [LAPTOP, PHONE]
    }
}
```

20. References

1- Java Methods | GeeksforGeeks:
https://www.geeksforgeeks.org/methods-in-java/

2- The Java™ Tutorials:
https://docs.oracle.com/javase/tutorial/java/javaOO/methods.html

3- Oracle Java Documentation:
https://docs.oracle.com/en/java/javase/17/docs/api/index.html

4- Effective Java (Joshua Bloch)