

Welcome

Java New Features: Versions 8 to 22

K. Venkata Ramana



Venkata Ramana

Over three decades of IT experience in Corporate Training, Software development and architectural designs involving web technologies, databases, SOA, Microservices & Cloud.

A Prayer of gratitude to Mother Nature:

All actions are being performed by the Mother Nature.

But in my ignorance, I may sometimes think that :

"I am the doer"

No.. Never

"Mother Nature is the doer"

Specification

Provides API , standards, recommended practices, codes
And technical publications, reports and studies.

JCP - Java Community Process

JEP - JDK Enhancement Proposal

JSR - Java Specification Request

JEPs (JDK Enhancement-Proposals) focus on proposing and implementing enhancements within the Java Development Kit (JDK), addressing specific features or optimizations.

JSRs (Java Specification Requests) define broader specifications and APIs, standardizing technologies beyond the core JDK.

JEPs are more tightly integrated into the JDK development process, while JSRs cover a wider range of Java ecosystem components through the Java Community Process (JCP).

JEP 126: Lambda Expressions

JEP 261: Module System

JEP 286: Local-Variable Type Inference (var)

JEP 325: Switch Expressions

JEP 355: Text Blocks

JEP 360: Sealed Classes

JSR 168,286,301 - Portlet Applications

JSR 127,254,314 - JSF

JSR 220 - Ejb3.0 & Jpa JSR 318 – EJB 3.1

JSR 340: Java Servlet 3.1 Specification

JSR 250 - Common Annotations for java

JSR 303 - Java Bean Validations

JSR 224- Jax-ws

JSR 311,370 - Jax-RS

JSR 299 - Context & DI

JSR 330 – DI

JSR-303 - Bean Validations

JSR208,312 – JBI (Java Business Integration)

Java 8 (March 2014):

1. **Lambda Expressions:** Introduces a new syntax for writing anonymous methods (closures) more concisely.
2. **Streams API:** Provides a new abstraction for working with sequences of elements and performing functional-style operations on them.
3. **Default Methods:** Allows interfaces to have method implementations.
4. **Functional Interfaces:** Introduces annotations like `@FunctionalInterface` to indicate that an interface is intended to be a functional interface.
5. **Method References:** Provides a shorthand notation for writing lambda expressions.
6. **Optional:** Introduces the `Optional` class to represent an optional value.

Java 9 (September 2017):

- 1. Module System (Project Jigsaw):** Introduces the module system to improve modularization of code.
- 2. JShell:** Adds a REPL (Read-Eval-Print Loop) tool for interactive development.
- 3. Private Methods in Interfaces:** Allows interfaces to have private methods.

Java 10 (March 2018):

- 1. Local Variable Type Inference (var):** Introduces the ability to use `var` for local variable type inference.
- 2. Optional.orElseThrow():** Adds the `orElseThrow` method to `Optional` for throwing a specific exception if the value is not present.

Java 13 (September 2019):

- 1. Text Blocks:** Introduces multi-line string literals (text blocks) to make it easier to write and read strings.
- 2. Switch Expression (Standard):** Promotes the switch expression feature from preview to standard.

Java 14 (March 2020):

- 1. Records (Preview):** Introduces record classes for a more concise way to define immutable data-carrying classes.
- 2. Pattern Matching for `instanceof`:** Introduces a more concise syntax for conditional casting and type checking.

Java 15 (September 2020):

- 1. Sealed Classes (Second Preview):** Enhances sealed classes and interfaces, allowing more control over subclasses.
- 2. Text Blocks (Second Preview):** Further refines the text blocks feature.

Java 17 (September 2021):

- 1. Sealed Classes (Standard):** Promotes the sealed classes feature from preview to standard.
- 2. Pattern Matching for Switch (Standard):** Promotes the pattern matching for `switch` feature from preview to standard.

Java 21 (September 2023):

(some of the features)

Virtual Threads (Finalized):

- Virtual threads are lightweight threads that reduce the effort of writing concurrent applications.
- Finalized in JDK 21, now supporting thread-local variables all of the time.

Sequenced Collections Proposal:

- Introduces interfaces to represent collections with a defined encounter order.
- Defines interfaces for sequencing collections, sets, and maps, retrofitting them into the existing collections type hierarchy.

String Templates (Preview):

- A preview of string templates complementing Java's existing string literals and text blocks.
- Couples literal text with embedded expressions and processors to produce specialized results.
- Simplifies the writing of Java programs by expressing strings with values computed at runtime.

Java 22 Features: Scoped Values and Stream Gathering

1. Scoped Values (Preview Feature)

Scoped Values in Java 22 provide an alternative to `ThreadLocal` for sharing data across methods in a structured manner. They allow safe and efficient data sharing within the scope of a single thread, particularly useful for concurrent applications.

2. Stream Gathering (Enhancement to Streams API)

Java 22 introduces the `Stream.gather()` method, which enables flexible and efficient grouping of elements during stream processing. It allows custom grouping strategies beyond built-in collectors.

Java Lambda Expressions and Functional Programming

Introduction

- Title: Understanding Java Lambda Expressions and Performance
- Brief Overview:
- Lambda expressions were introduced in Java 8 as a new language feature to bring functional programming capabilities to the Java programming language. A lambda expression is a concise way to represent an anonymous function (a function without a name) that can be passed around as a method parameter or stored in a variable.
- Performance-wise, lambda expressions are more efficient than anonymous inner classes.

Anonymous Classes:

```
// Syntax:  
new InterfaceName() {  
    // Class body  
    // Implementation of interface methods  
}
```

Anonymous classes existed in Java before lambda expressions and are used to create an instance of an interface or an abstract class without explicitly defining a named class. Here's an example:

```
// Functional Interface  
interface MyFunctionalInterface {  
    void myMethod(int x);  
}  
  
public class AnonymousClassExample {  
    public static void main(String[] args) {  
        // Using Anonymous Class  
        MyFunctionalInterface myAnonymousClass = new MyFunctionalInterface() {  
            @Override  
            public void myMethod(int x) {  
                System.out.println(x * 2);  
            }  
        };  
  
        // Calling the method using the anonymous class  
        myAnonymousClass.myMethod(5);  
    }  
}
```

Lambda Expressions:

```
java
```

 Copy code

```
// Syntax:  
(parameters) -> expression
```

Lambda expressions were introduced in Java 8 and are a concise way to express instances of single-method interfaces (functional interfaces). Here's an example:

```
java
```

 Copy code

```
// Functional Interface  
interface MyFunctionalInterface {  
    void myMethod(int x);  
}  
  
public class LambdaExample {  
    public static void main(String[] args) {  
        // Using Lambda Expression  
        MyFunctionalInterface myLambda = (x) -> System.out.println(x * 2);  
  
        // Calling the method using the lambda expression  
        myLambda.myMethod(5);  
    }  
}
```

In this example, `MyFunctionalInterface` is a functional interface with a single abstract method `myMethod`. The lambda expression `(x) -> System.out.println(x * 2)` implements this method.

Comparison:

1. **Syntax:** Lambda expressions have a more concise syntax compared to anonymous classes.
2. **Use Cases:** Lambda expressions are mainly used for functional interfaces (single-method interfaces), while anonymous classes can be used for both interfaces and abstract classes with multiple methods.
3. **Readability:** Lambda expressions are often more readable for simple cases, making the code cleaner and more expressive.

Choose between lambda expressions and anonymous classes based on the specific requirements and the complexity of your code. Lambda expressions are generally preferred for simpler cases, while anonymous classes might be more suitable for more complex scenarios.

An example where anonymous classes might be preferred over lambda expressions is when you need to implement an interface with multiple methods. Anonymous classes allow you to implement all the methods of the interface in a single instance, while lambda expressions are only suitable for single-method interfaces (functional interfaces). Here's an example:

```
// An interface with multiple methods
interface MyInterface {
    void method1();
    void method2();
}

public class AnonymousClassExample {
    public static void main(String[] args) {
        // Using Anonymous Class to implement the interface
        MyInterface myAnonymousClass = new MyInterface() {
            @Override
            public void method1() {
                System.out.println("Implementation of method1");
            }

            @Override
            public void method2() {
                System.out.println("Implementation of method2");
            }
        };

        // Calling the methods using the anonymous class
        myAnonymousClass.method1();
        myAnonymousClass.method2();
    }
}
```

In this example, `MyInterface` is an interface with two methods (`method1` and `method2`). The anonymous class provides an implementation for both methods. If you were to use lambda expressions, you would need a separate lambda expression for each method, which might lead to less readable and more verbose code:

java

 Copy code

```
// Using Lambda Expressions (not suitable for interfaces with multiple methods)
MyInterface myLambda = () -> System.out.println("Implementation of method1"); // Comp

// You would need a separate lambda expression for each method
MyInterface myLambdaMethod1 = () -> System.out.println("Implementation of method1");
MyInterface myLambdaMethod2 = () -> System.out.println("Implementation of method2");
```

In this case, the use of anonymous classes provides a cleaner and more concise solution when dealing with interfaces that have multiple methods. Lambda expressions are still more suitable for single-method interfaces.

A functional interface in Java is an interface that contains only one abstract method. The concept of functional interfaces is closely tied to the introduction of lambda expressions in Java 8. Lambda expressions provide a concise way to express instances of single-method interfaces (functional interfaces), making the code more readable and expressive.

A functional interface can have multiple default or static methods, but it must have exactly one abstract method. The presence of this single abstract method is what makes the interface "functional" because it can be implemented by a lambda expression.

Here's an example:

```
@FunctionalInterface
interface MyFunctionalInterface {
    void myMethod(); // This is the single abstract method

    // Default method (allowed in functional interfaces)
    default void anotherMethod() {
        // Implementation
    }
}

// Valid usage of a functional interface
MyFunctionalInterface functionalObject = () -> {
    // Implementation of the single abstract method
};

// Invalid usage, as it would result in a compilation error
@FunctionalInterface
interface InvalidFunctionalInterface {
    void method1();
    void method2(); // Error: Multiple non-overriding abstract methods found in interface
}
```

In the second example, the `InvalidFunctionalInterface` is marked with `@FunctionalInterface`, but it has more than one abstract method, leading to a compilation error. The annotation helps catch such errors early in the development process.

The `'java.util.function'` package in Java was introduced with Java 8 as part of the Java Functional Interfaces and the lambda expressions feature. This package provides a set of functional interfaces that represent common function types, making it easier to work with functional programming concepts in Java. These interfaces are designed to be used with lambda expressions and method references, providing a more concise and expressive way to write code.

Here are some key purposes and benefits of introducing `'java.util.function'`:

1. **Functional Programming Support:** Java 8 introduced lambda expressions, allowing developers to write more concise and expressive code by treating functions as first-class citizens. The functional interfaces in `'java.util.function'` provide a foundation for working with functional programming concepts in Java.
2. **Standardized Interfaces:** The package defines a set of standard functional interfaces that represent common function types, such as predicates, consumers, suppliers, and functions. This standardization makes the API more consistent and helps developers understand and use these interfaces more easily.

3. **Method References:** The functional interfaces in `java.util.function` can be used in conjunction with method references, another feature introduced in Java 8. Method references provide a shorthand notation for lambda expressions, making the code more readable.
4. **Interoperability:** The functional interfaces in `java.util.function` are often used in the context of the Stream API, which allows developers to process sequences of elements in a functional style. This promotes a more declarative and expressive coding style for operations on collections.
5. **Common Use Cases:** The package provides interfaces like `Predicate`, `Consumer`, `Function`, `Supplier`, etc., which cover common use cases when working with functional programming. For example, `Predicate` is used for boolean-valued functions, `Consumer` is used for operations that take an argument but return no result, and `Function` is used for functions that transform one value into another.

```

@FunctionalInterface
public interface Predicate<T> {

    /**
     * Evaluates this predicate on the given argument.
     *
     * @param t the input argument
     * @return {@code true} if the input argument matches the predicate,
     *         otherwise {@code false}
     */
    boolean test(T t);

    /**
     * Returns a composed predicate that represents a short-circuiting logical
     * AND of this predicate and another. When evaluating the composed
     * predicate, if this predicate is {@code false}, then the {@code other}
     * predicate is not evaluated.
     *
     * @param other a predicate that will be logically-ANDed with this
     *             predicate
     * @return a composed predicate that represents the short-circuiting logical
     *         AND of this predicate and the {@code other} predicate
     * @throws NullPointerException if other is null
     */
    default Predicate<T> and(Predicate<? super T> other) {
        Objects.requireNonNull(other);
        return (t) -> test(t) && other.test(t);
    }
}


```

```

    /**
     * Returns a predicate that represents the logical negation of this
     * predicate.
     *
     * @return a predicate that represents the logical negation of this
     *         predicate
     */
    default Predicate<T> negate() {
        return (t) -> !test(t);
    }

    /**
     * Returns a composed predicate that represents a short-circuiting logical
     * OR of this predicate and another. When evaluating the composed
     * predicate, if this predicate is {@code true}, then the {@code other}
     * predicate is not evaluated.
     *
     * @param other a predicate that will be logically-ORed with this
     *             predicate
     * @return a composed predicate that represents the short-circuiting logical
     *         OR of this predicate and the {@code other} predicate
     * @throws NullPointerException if other is null
     */
    default Predicate<T> or(Predicate<? super T> other) {
        Objects.requireNonNull(other);
        return (t) -> test(t) || other.test(t);
    }

    /**
     * Returns a predicate that tests if two arguments are equal according
     * to {@link Objects#equals(Object, Object)}.
     *
     * @param <T> the type of arguments to the predicate
     * @param targetRef the object reference with which to compare for equality,
     *                  which may be {@code null}
     * @return a predicate that tests if two arguments are equal according
     *         to {@link Objects#equals(Object, Object)}
     */
    static <T> Predicate<T> isEqual(Object targetRef) {
        return (null == targetRef)
            ? Objects::isNull
            : object -> targetRef.equals(object);
    }
}


```

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

public class PredicateExample {
    public static void main(String[] args) {
        // Example 1: Basic Predicate
        Predicate<Integer> isEven = n -> n % 2 == 0;
        System.out.println("Is 4 even? " + isEven.test(4)); // Output: true
        System.out.println("Is 5 even? " + isEven.test(5)); // Output: false

        // Example 2: Combining Predicates with "and"
        Predicate<Integer> isGreaterThanZero = n -> n > 0;
        Predicate<Integer> isPositiveEven = isEven.and(isGreaterThanZero);
        System.out.println("Is 4 positive and even? " + isPositiveEven.test(4)); // Output: true
        System.out.println("Is -2 positive and even? " + isPositiveEven.test(-2)); // Output: false
```

```
// Example 3: Negating a Predicate
Predicate<String> isEmpty = String::isEmpty;
Predicate<String> isNotEmpty = isEmpty.negate();
System.out.println("Is empty string empty? " + isEmpty.test(""));      // Output: true
System.out.println("Is empty string not empty? " + isNotEmpty.test("")); // Output: false

// Example 4: Combining Predicates with "or"
Predicate<String> containsHello = s -> s.contains("Hello");
Predicate<String> containsWorld = s -> s.contains("World");
Predicate<String> containsHelloOrWorld = containsHello.or(containsWorld);
System.out.println("Does the string contain Hello or World? " + containsHelloOrWorld.test("Hello, GPT!")); // Output: true
System.out.println("Does the string contain Hello or World? " + containsHelloOrWorld.test("Hi there!")); // Output: false

// Example 5: Using isEqual static method
Predicate<String> isEqualToHello = Predicate.isEqual("Hello");
System.out.println("Is the string equal to Hello? " + isEqualToHello.test("Hello"));    // Output: true
System.out.println("Is the string equal to Hello? " + isEqualToHello.test("Hi there!")); // Output: false
}
```

Performance Comparison

- Title: Lambda vs. Anonymous Inner Classes
- Key Points:
 - Lambda expressions incur pure compile-time activity, while anonymous inner classes involve class loading, memory allocation, and runtime object initialization.
 - Lambda expressions have better runtime performance compared to anonymous inner classes.

Lambda Expressions Basics

- Title: Lambda Expressions Syntax
- Key Elements:
 - Lambda Syntax: `lambda operator -> body`
 - Examples:
 - Zero Parameter: `() -> System.out.println("Zero parameter lambda");`
 - One Parameter: `(p) -> System.out.println("One parameter: " + p);`
 - Multiple Parameters: `(p1, p2) -> System.out.println("Multiple parameters: " + p1 + ", " + p2);`

Lambda vs. Anonymous Class

- Title: **Key Differences**
- Differences:
 - Usage of 'this' keyword varies between lambda expressions and anonymous classes.
 - Compilation differences: Lambda expressions are compiled into private methods using invokedynamic instruction.

Java anonymous classes are a powerful feature that allows you to create classes without explicitly defining them. They are often used when you need to make a one-time use of a class, especially when dealing with interfaces or abstract classes. Here are some common use cases for anonymous classes:

Event Handling in GUI Applications

In GUI frameworks like Swing, you often need to define event listeners. Anonymous classes are handy for implementing interfaces like ActionListener on the fly.

```
// Using an anonymous class to handle button click
button.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        System.out.println("Button clicked!");
    }
});
```

Thread Creation

Instead of creating a separate class for a thread, you can use an anonymous class to implement the Runnable interface directly.

```
Thread thread = new Thread(new Runnable() {  
    @Override  
    public void run() {  
        System.out.println("Thread is running!");  
    }  
});
```

Simple Callback Implementations

When you need a simple callback mechanism, anonymous classes can be a concise way to implement it.

```
Example example = new Example();
example.doSomethingAsync(new Callback() {
    @Override
    public void onComplete(String result) {
        System.out.println(result);
    }
});
```

Functional Interfaces

- Title: Functional Interfaces and Built-in Functional Interfaces
- Definitions:
 - Functional Interface: Interface with a single abstract method.
 - Java 8 introduced built-in functional interfaces in the '`java.util.function`' package.

java.util.function Package

- Title: Common Functional Interfaces
- Examples:
 - `Predicate`: Tests a specified argument.
 - `BinaryOperator`: Applies a specified operation to two arguments.
 - `Function`: Applies a function to a specified argument.

Stream API: The Stream API was introduced to facilitate functional-style operations on sequences of elements. Streams make extensive use of lambda expressions and functional interfaces. The interfaces in the '`java.util.function`' package play a crucial role in defining the behavior of operations performed on streams, such as '`map`', '`filter`', and '`reduce`'.

- **Code Readability and Conciseness:** The functional interfaces in '`java.util.function`' are designed to cover common use cases, making code more readable and concise. Developers can use these interfaces directly or create their own functional interfaces when needed.

The `java.util.function` package in Java provides a set of functional interfaces that represent common function types like predicates, consumers, functions, and suppliers. These interfaces are designed to be used with lambda expressions and method references, enabling functional programming features in Java. Here are some important functional interfaces from the `java.util.function` package:

1. Predicate<T>:

- Syntax:

```
@FunctionalInterface  
public interface Predicate<T> {  
    boolean test(T t);  
}
```

- Description: Represents a predicate (boolean-valued function) of one argument. It is commonly used for filtering elements.

1. Predicate<T> Example:

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

public class PredicateExample {
    public static void main(String[] args) {
        // Predicate to check if a number is positive
        Predicate<Integer> isPositive = num -> num > 0;

        System.out.println(isPositive.test(5)); // true
        System.out.println(isPositive.test(-2)); // false
    }
}
```

2. Consumer<T>:

- **Syntax:**

```
@FunctionalInterface  
public interface Consumer<T> {  
    void accept(T t);  
}
```

- **Description:** Represents an operation that takes a single input argument and returns no result. It is commonly used for side-effect operations.

2. Consumer<T> Example:

```
import java.util.function.Consumer;

public class ConsumerExample {
    public static void main(String[] args) {
        // Consumer to print the length of a string
        Consumer<String> printLength = str -> System.out.println("Length: " + str.length());

        printLength.accept("Java"); // Length: 4
        printLength.accept("Lambda"); // Length: 6
    }
}
```

3. Function<T, R>:

- **Syntax:**

```
@FunctionalInterface  
public interface Function<T, R> {  
    R apply(T t);  
}
```

- **Description:** Represents a function that takes one argument and produces a result. It is commonly used for data transformation and mapping.

3. Function<T, R> Example:

```
import java.util.function.Function;

public class FunctionExample {
    public static void main(String[] args) {
        // Function to square a number
        Function<Integer, Integer> square = num -> num * num;

        System.out.println(square.apply(5)); // 25
        System.out.println(square.apply(8)); // 64
    }
}
```

4. Supplier<T>:

- Syntax:

```
@FunctionalInterface  
public interface Supplier<T> {  
    T get();  
}
```

- Description: Represents a supplier of results. It has no input arguments and provides a value when invoked. It is often used for lazy initialization.

4. Supplier<T> Example:

```
import java.util.function.Supplier;

public class SupplierExample {
    public static void main(String[] args) {
        // Supplier to generate a random number
        Supplier<Integer> randomNumber = () -> (int) (Math.random() * 100);

        System.out.println("Random Number: " + randomNumber.get());
    }
}
```

5. UnaryOperator<T>:

- Syntax:

```
@FunctionalInterface  
public interface UnaryOperator<T> extends Function<T, T> {  
    // Inherits apply(T t) from Function<T, T>  
}
```

- Description: Represents an operation on a single operand that produces a result of the same type as its operand. It is commonly used for operations that transform the input into the same type.

5. UnaryOperator<T> Example:

```
import java.util.function.UnaryOperator;

public class UnaryOperatorExample {
    public static void main(String[] args) {
        // UnaryOperator to double a number
        UnaryOperator<Integer> doubleNumber = num -> num * 2;

        System.out.println(doubleNumber.apply(7)); // 14
        System.out.println(doubleNumber.apply(10)); // 20
    }
}
```

6. BinaryOperator<T>:

- Syntax:

```
java
```

 Copy code

```
@FunctionalInterface  
public interface BinaryOperator<T> extends BiFunction<T, T, T> {  
    // Inherits apply(T t, T u) from BiFunction<T, T, T>  
}
```

- Description: Represents an operation upon two operands of the same type, producing a result of the same type. It is commonly used for operations that combine two values into one.

6. BinaryOperator<T> Example:

```
import java.util.function.BinaryOperator;

public class BinaryOperatorExample {
    public static void main(String[] args) {
        // BinaryOperator to find the maximum of two numbers
        BinaryOperator<Integer> max = Integer::max;

        System.out.println(max.apply(15, 8));    // 15
        System.out.println(max.apply(20, 25));   // 25
    }
}
```

- Description: Represents a predicate.

7. BiPredicate<T, U>:

- Syntax:

```
@FunctionalInterface  
public interface BiPredicate<T, U> {  
    boolean test(T t, U u);  
}
```

- Description: Represents a predicate (boolean-valued function) of two arguments. It is commonly used for filtering elements with two conditions.

7. BiPredicate<T, U>:

```
import java.util.function.BiPredicate;

public class BiPredicateExample {
    public static void main(String[] args) {
        // BiPredicate to check if two strings have the same length
        BiPredicate<String, String> sameLength =
            (str1, str2) -> str1.length() == str2.length();

        System.out.println(sameLength.test("Java", "Python")); // false
        System.out.println(sameLength.test("Hello", "World")); // true
        System.out.println(sameLength.test("Lambda", "Java")); // false
    }
}
```

8. BiConsumer<T, U>:

- Syntax:

```
@FunctionalInterface  
public interface BiConsumer<T, U> {  
    void accept(T t, U u);  
}
```

- Description: Represents an operation that accepts two input arguments and returns no result. It is commonly used for side-effect operations with two inputs.

9. BiFunction<T, U, R>:

- Syntax:

```
@FunctionalInterface  
public interface BiFunction<T, U, R> {  
    R apply(T t, U u);  
}
```

- Description: Represents a function that takes two arguments and produces a result. It is similar to `Function` but with two input parameters.

9. BiFunction<T, U, R>:

```
import java.util.function.BiConsumer;

public class BiConsumerExample {
    public static void main(String[] args) {
        // BiConsumer to print the sum of two numbers
        BiConsumer<Integer, Integer> printSum = (num1, num2) ->
            System.out.println("Sum: " + (num1 + num2));

        // BiConsumer to print the product of two numbers
        BiConsumer<Integer, Integer> printProduct = (num1, num2) ->
            System.out.println("Product: " + (num1 * num2));

        // Example usage
        printSum.accept(10, 5);           // Sum: 15
        printProduct.accept(7, 3);        // Product: 21

        // Combining two BiConsumers using andThen
        BiConsumer<Integer, Integer> combined = printSum.andThen(printProduct);
        combined.accept(8, 4);
        // Output:
        // Sum: 12
        // Product: 32
    }
}
```

Functional Programming

- Title: Introduction to Functional Programming
- Characteristics:
 - Focuses on results, not the process.
 - Emphasizes what is to be computed.
 - Data is immutable.
 - Decomposes problems into functions.

Pure vs. Impure Functions

- Title: Pure and Impure Functions
- Definitions:
 - Pure Functions: Act only on their parameters, providing the same output for given parameters.
 - Impure Functions: Have hidden inputs or outputs, making them dependent and not suitable for isolation.

Method References

- Title: Java Method References
- Definition:
 - Introduced in Java 8, allows referencing methods or constructors without invoking them.
- Example:
 - `System.out::println` vs. `System.out.println`

Method References in Action

- Title: Implementing 'Calculator' Interface
- Example Code:
 - Demonstrates various ways to implement the 'add' method using anonymous classes, lambda expressions, and method references.

Example 1: Functional Interface with Lambda

```
// Functional Interface
@FunctionalInterface
interface MyFunction {
    void myMethod();
}

public class LambdaExample1 {
    public static void main(String[] args) {
        // Lambda expression implementing the functional interface
        MyFunction myFunction = () -> System.out.println("Hello, Lambda!");

        // Calling the method using the lambda expression
        myFunction.myMethod();
    }
}
```

Example 2: Lambda with Parameters

```
// Functional Interface with parameters
@FunctionalInterface
interface AddOperation {
    int add(int a, int b);
}

public class LambdaExample2 {
    public static void main(String[] args) {
        // Lambda expression for addition
        AddOperation addOperation = (a, b) -> a + b;

        // Calling the method using the lambda expression
        System.out.println("Sum: " + addOperation.add(5, 3));
    }
}
```

Example 3: Using forEach with Lambda for List

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

public class LambdaExample3 {
    public static void main(String[] args) {
        List<String> fruits = Arrays.asList("Apple", "Banana", "Orange", "Mango");

        // Lambda expression with forEach for printing each fruit
        fruits.forEach(fruit -> System.out.println(fruit));
    }
}
```

Example 4: Thread using Lambda

```
public class LambdaExample4 {  
    public static void main(String[] args) {  
        // Lambda expression for a new thread  
        Thread myThread = new Thread(() -> System.out.println("Thread using Lambda"))  
  
        // Starting the thread  
        myThread.start();  
    }  
}
```

Example 5: Comparator with Lambda for Sorting

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class LambdaExample5 {
    public static void main(String[] args) {
        List<String> languages = new ArrayList<>(List.of("Java", "Python", "C++",
                "JavaScript"));
        // Lambda expression for sorting in ascending order
        Collections.sort(languages, (s1, s2) -> s1.compareTo(s2));

        // Printing the sorted list
        System.out.println("Sorted Languages: " + languages);
    }
}
```

Static Method Reference:

```
import java.util.function.Function;

public class StaticMethodReferenceExample {

    static class StaticMethodReference {
        public static int parseInt(String s) {
            return Integer.parseInt(s);
        }
    }

    public static void main(String[] args) {
        // Static Method Reference
        Function<String, Integer> parseIntReference = StaticMethodReference::parseInt;
        System.out.println("Static Method Reference: " + parseIntReference.apply("123"));
    }
}
```

Instance Method Reference:

```
import java.util.function.Function;

class Person {
    private String name;

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

    public String getName() {
        return name;
    }

    public String greet() {
        return "Hello!";
    }
}
```

```
public class MethodReferenceExample {

    public static void main(String[] args) {
        // Creating an instance of Person
        Person person = new Person("John");

        // Using an instance method reference to refer to the getName method
        Function<Person, String> getNameFunction = Person::getName;

        // Calling the method using the instance method reference
        String personName = getNameFunction.apply(person);
        System.out.println("Person's name: " + personName);

        // Using an instance method reference to refer to the greet method
        Function<Person, String> greetFunction = Person::greet;

        // Calling the method using the instance method reference
        String greeting = greetFunction.apply(person);
        System.out.println("Person says: " + greeting);
    }
}
```

Constructor Method Reference:

```
import java.util.ArrayList;
import java.util.List;
import java.util.function.Supplier;

public class ConstructorMethodReferenceExample {

    public static void main(String[] args) {
        // Constructor Method Reference
        Supplier<List<String>> listSupplierReference = ArrayList::new;
        List<String> newList = listSupplierReference.get();
        newList.add("Java");
        newList.add("Method Reference");
        System.out.println("New List: " + newList);
    }
}
```

Key Differences Between Lambda Expressions and Method References

Feature	Lambda Expression	Method Reference
Syntax	Explicit function body	Shorter, references existing method
Readability	Slightly verbose	More concise
Flexibility	Can have multiple statements	Only works if method signature matches
Code Complexity	Allows inline operations	Only calls a method directly
Performance	Slightly more overhead (captures context)	Potentially optimized by JVM

Different Types of Method References

Type	Lambda Example	Equivalent Method Reference
Static Method Reference	<code>(s) -> Integer.parseInt(s)</code>	<code>Integer::parseInt</code>
Instance Method Reference	<code>(s) -> s.toUpperCase()</code>	<code>String::toUpperCase</code>
Instance Method of a Specific Object	<code>(str) -> obj.someMethod(str)</code>	<code>obj::someMethod</code>
Constructor Reference	<code>(s) -> new String(s)</code>	<code>String::new</code>

- ✓ Use Lambda Expressions when you need **more flexibility** (multiple statements, logic, conditions).
- ✓ Use Method References when the lambda **only calls an existing method** to improve readability.

Example: Lambda Expression for More Flexibility

Let's say you want to process a list of names by trimming whitespace and converting them to uppercase, but only if the name length is greater than 3.

```
List<String> names = Arrays.asList(" Sam ", "John", " Al ", "Mike");

List<String> processedNames = names.stream()
    .map(name -> {
        String trimmed = name.trim(); // Trim whitespace
        if (trimmed.length() > 3) {
            return trimmed.toUpperCase(); // Convert to uppercase if length > 3
        }
        return trimmed;
    })
    .collect(Collectors.toList());

System.out.println(processedNames);
```

Why a Method Reference Won't Work?

A **method reference** like `String::toUpperCase` only calls an existing method without allowing extra logic, such as trimming spaces or checking length conditions.

If you use:

```
names.stream().map(String::toUpperCase).collect(Collectors.toList());
```

It fails to trim spaces and conditionally convert only longer names.

Key Takeaway

- Use **lambda expressions** when you need **extra processing**, conditions, or multiple steps.
- Use **method references** when you only need to call a single existing method directly, without modifications.

Java Streams

Introduction

- Title: **Unlocking the Power of Java Streams**
- Brief Overview:
 - Java Streams: A powerful feature introduced in Java 8.
 - Streamlining data processing operations on collections.
 - Enables functional-style programming.

What are Java Streams?

- Title: Understanding Java Streams
- Explanation:
 - Sequence of Elements: Represents a sequence of elements.
 - Functional Programming: Enables functional-style operations on data.
 - Lazy Evaluation: Stream operations do not evaluate until required.

Basic Stream Operations

- Title: Essential Stream Operations
- Key Operations:
 - Filtering
 - Mapping
 - Sorting
 - Collecting
 - Reducing

Code Example

- Title: Exploring Basic Stream Operations
- Code Snippet:

```
List<String> languages = Arrays.asList("Java", "Python", "JavaScript", "Ruby");

languages.stream()
    .filter(lang -> lang.length() > 4)
    .map(String::toUpperCase)
    .sorted()
    .forEach(System.out::println);
```

- Explanation:
 - **Filtering:** Selects elements based on a condition.
 - **Mapping:** Applies a function to each element.
 - **Sorting:** Orders elements.
 - **ForEach:** Performs an action for each element.

The ` `.filter(lang -> lang.length() > 4)` operation filters the languages based on the condition that their length is greater than 4. However, using a method reference for this specific condition is not straightforward because the ` `length()`` method is an instance method of the ` `String` ` class, and method references work best with static methods or instance methods of the stream elements.

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

public class LambdaExample {
    public static void main(String[] args) {
        List<String> languages = Arrays.asList("java", "python", "Ruby");

        languages.stream()
            .filter(LambdaExample::isLengthGreaterThan4)
            .map(String::toUpperCase) // Method reference for filter
            .sorted()
            .forEach(System.out::println);
    }

    private static boolean isLengthGreaterThan4(String lang) {
        return lang.length() > 4;
    }
}
```

modified code with lambda expressions for the 'map' and 'forEach' operations:

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

public class LambdaExample {
    public static void main(String[] args) {
        List<String> languages = Arrays.asList("java", "python", "Ruby");

        languages.stream()
            .filter(lang -> lang.length() > 4)
            .map(lang -> lang.toUpperCase()) // Lambda expression for map
            .sorted()
            .forEach(lang -> System.out.println(lang));
    }
}
```

Intermediate vs. Terminal Operations

- Title: Understanding Operations
- Explanation:
 - Intermediate Operations: Transform or filter the data.
 - Terminal Operations: Trigger the processing and produce a result.

Intermediate Operations:

1. Filter:

```
stream.filter(x -> x > 5)
```

Filter: Select elements greater than 5.

2. Map:

```
stream.map(x -> x * 2)
```

Map: Multiply each element by 2.

3. FlatMap:

```
stream.flatMap(line -> Arrays.stream(line.split(" ")))
```

FlatMap: Split each line into words and flatten the result.

4. Distinct:

```
stream.distinct()
```

Distinct: Remove duplicate elements.

5. Sorted:

```
stream.sorted()
```

Sorted: Sort elements.

6. Peek:

```
stream.peek(x -> System.out.println(x))
```

Peek: Perform an action for each element without consuming the stream.

7. Limit:

```
stream.limit(10)
```

Limit: Truncate the stream to the first 10 elements.

8. Skip:

```
stream.skip(5)
```

Skip: Skip the first 5 elements.

Terminal Operations:

1. AllMatch:

```
stream.allMatch(x -> x > 0)
```

AllMatch: Check if all elements are greater than 0.

2. AnyMatch:

```
stream.anyMatch(x -> x == 0)
```

AnyMatch: Check if any element is equal to 0.

3. NoneMatch:

```
stream.noneMatch(x -> x < 0)
```

NoneMatch: Check if no elements are less than 0.

4. FindFirst:

```
stream.findFirst()
```

FindFirst: Return the first element.

5. FindAny:

```
stream.findAny()
```

FindAny: Return any element.

6. Count:

```
stream.count()
```

Count: Count the number of elements.

7. Reduce:

```
stream.reduce(0, (x, y) -> x + y)
```

Reduce: Sum all elements starting with an initial value of 0.

8. Collect:

```
stream.collect(Collectors.toList())
```

Collect: Transform elements into a list.

9. ForEach:

```
stream.forEach(x -> System.out.println(x))
```

ForEach: Print each element.

Java Streams Resemble a Graph Topology:

1. Nodes and Edges:

- Each operation (intermediate or terminal) can be viewed as a node in the graph.
- The flow of data from one operation to the next forms the edges connecting these nodes.

2. Intermediate Operations:

- Intermediate operations (like `filter`, `map`, etc.) create a sequence of transformations. Each operation is a node that takes input from the previous node and passes output to the next.
- Since these operations are lazy, they don't execute immediately but instead define how data will flow through the graph.

3. Terminal Operations:

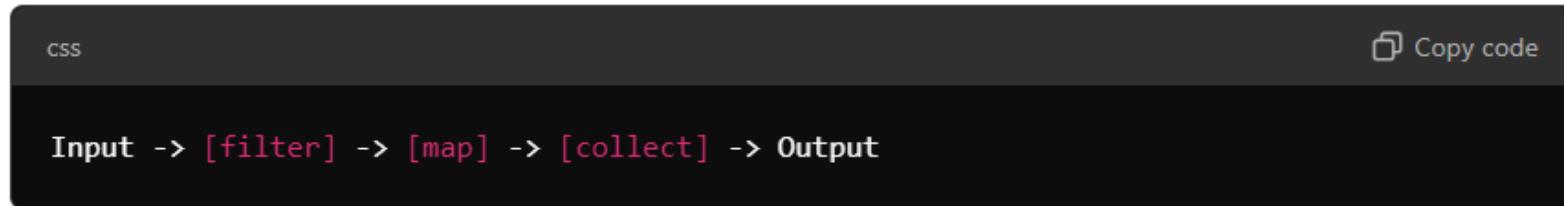
- Terminal operations (like `collect`, `forEach`, `findFirst`, etc.) represent the endpoints of the graph. They trigger the execution of the entire pipeline, processing the data according to the defined flow.

4. Pipelining:

- The overall structure can be visualized as a pipeline where data flows through the nodes (operations) and gets transformed at each step until it reaches the terminal operation.

Visualization

If you visualize a stream pipeline with operations like this:



You can imagine:

- **Input:** The starting point or the source of the data.
- **Nodes:** Each operation (`filter`, `map`, etc.) represents a transformation applied to the data.
- **Edges:** The flow of data from one operation to the next.

Advantages of This Topology

- **Optimization:** The stream framework can analyze the entire structure before executing it, allowing for optimizations, such as combining operations or short-circuiting.
- **Clear Flow:** The graph-like structure makes it clear how data is being transformed at each step.

Lazy Evaluation

- Title: Lazy Evaluation in Streams
- Explanation:
 - Deferred Execution: Operations do not execute until a terminal operation is invoked.
 - Efficient Processing: Only processes the elements needed for the final result.

: Parallel Streams

- Title: Harnessing Parallelism
- Explanation:
 - Parallel Processing: Streams can be processed concurrently.
 - Parallel Stream Creation: `stream.parallel()`

Stream API Methods

- Title: Common Stream API Methods
- Overview:
 - `filter(Predicate<T>)`
 - `map(Function<T, R>)`
 - `sorted()`
 - `collect(Collectors)`
 - `reduce(BinaryOperator)`

: Benefits of Java Streams

- Title: Advantages of Using Streams
- Key Benefits:
 - Conciseness
 - Readability
 - Reusability
 - Parallel Processing

Example 1: Filtering and Printing

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

public class StreamExample1 {
    public static void main(String[] args) {
        List<String> fruits = Arrays.asList("Apple", "Banana", "Orange", "Grapes", "Kiwi");

        // Using stream to filter and print fruits starting with 'A'
        fruits.stream()
            .filter(fruit -> fruit.startsWith("A"))
            .forEach(System.out::println);
    }
}
```

This example uses the `filter` operation to select fruits that start with the letter 'A' and then prints each of them.

Example 2: Mapping and Collecting

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

public class StreamExample2 {
    public static void main(String[] args) {
        List<String> words = Arrays.asList("hello", "world", "java", "stream");

        // Using stream to map words to uppercase and collect them into a new list
        List<String> upperCaseWords = words.stream()
            .map(String::toUpperCase)
            .collect(Collectors.toList());

        // Printing the resulting list
        System.out.println(upperCaseWords);
    }
}
```

In this example, the `map` operation is used to convert each word to uppercase, and then the `collect` operation gathers them into a new list.

Example 3: Summing Numbers

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

public class StreamExample3 {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

        // Using stream to sum the numbers
        int sum = numbers.stream()
            .mapToInt(Integer::intValue)
            .sum();

        // Printing the sum
        System.out.println("Sum of numbers: " + sum);
    }
}
```

Here, the `mapToInt` operation is used to convert each `Integer` to an `int`, and then the `sum` operation calculates the sum of the numbers.

Here's how lazy evaluation works in the context of Java Streams:

1. **Intermediate Operations are Lazy:** Operations like `filter`, `map`, and `flatMap` are intermediate operations. When you call them on a stream, they don't execute immediately. Instead, they modify the stream pipeline and return a new stream.

java

 Copy code

```
Stream<Person> filteredStream = people.stream()
    .filter(person -> person.getAge() > 50 && "male".equals(person.getGender()))
```

At this point, the `filter` operation has modified the stream, but nothing has been processed yet.

2. **Terminal Operation Triggers Processing:** The processing of the stream occurs when a terminal operation is invoked. Examples of terminal operations include `forEach`, `collect`, `reduce`, and `count`.

java

 Copy code

```
filteredStream.forEach(person -> System.out.println("Result: " + person))
```

The `forEach` is a terminal operation, and it triggers the processing of the stream. At this point, the elements of the stream are processed through the pipeline.

3. Optimizations with Short-Circuiting: Some stream operations are designed to short-circuit, meaning they can stop processing elements once a certain condition is met. For example, '`findFirst`' on a stream will stop as soon as it finds the first element that matches the condition.

java

 Copy code

```
Optional<Person> firstPerson = people.stream()
    .filter(person -> person.getAge() > 50 && "male".equals(person.getGender()))
    .findFirst();
```

If a matching element is found early in the stream, the subsequent elements won't be processed.

Lazy evaluation allows for more efficient processing of large datasets, as only the elements necessary to compute the final result are processed. It enables optimizations like short-circuiting and avoids unnecessary computations.

For Java Streams, short-circuiting operations are those that can terminate the processing of elements in the stream before reaching the end of the data source.

Short-circuiting operations are often associated with boolean conditions or the need to find a specific element in the stream.

Here are some common short-circuiting operations in Java Streams:

- . **findFirst()**: Returns the first element of the stream. The stream processing stops once the first element is found.

```
Optional<Person> firstPerson = people.stream()
    .filter(person -> person.getAge() > 50
        .findFirst();                                && "male".equals(person.getGender()))
```

.findAny(): Returns any element of the stream. It's useful in parallel streams where the order is not guaranteed. Like '**findFirst()**', the stream processing stops once any matching element is found.

```
Optional<Person> firstPerson = people.stream()
    .filter(person -> person.getAge() > 50
    .findAny();                      && "male".equals(person.getGender()))
```

limit(): Limits the number of elements in the stream. If you only need a certain number of elements, processing stops once that limit is reached.

```
Optional<Person> firstPerson = people.stream()
    .filter(person -> person.getAge() > 50
    .limit(5)                      && "male".equals(person.getGender()))
    .collect(Collectors.toList());
```

Error in the code:

Streams in Java are designed to be consumed only once. Once a terminal operation like `forEach` or `collect` is performed on a stream, it is considered "consumed" and cannot be reused.

```
Stream<Person> filteredStream = people.stream()
    .filter(person -> person.getAge() > 50 && "male".equals(person.getGender()));

filteredStream.forEach(person -> System.out.println("Result: " + person.getName()));

// Additional conditions: Convert names to uppercase
List<String> filteredNamesUppercase = filteredStream
    .map(person -> person.getName().toUpperCase())
    .collect(Collectors.toList());
```

After you call `filteredStream.forEach(...)`, the stream is consumed, and you cannot use it again. Therefore, when you try to use `filteredStream.map(...)`, it will result in an error.

To fix this issue, you have a couple of options:

1. Create a new stream for the additional operations:

```
Stream<Person> filteredStream = people.stream()
    .filter(person -> person.getAge() > 50 && "male".equals(person.getGender()));
```

```
filteredStream.forEach(person -> System.out.println("Result: " + person.getName()));
```

```
// Create a new stream for additional conditions
List<String> filteredNamesUppercase = people.stream()
    .filter(person -> person.getAge() > 50 && "male".equals(person.getGender()))
    .map(person -> person.getName().toUpperCase())
    .collect(Collectors.toList());
```

2. Collect the results into a list and then perform additional operations on the list:

```
Stream<Person> filteredStream = people.stream()
    .filter(person -> person.getAge() > 50 &&
"male".equals(person.getGender()));

filteredStream.forEach(person -> System.out.println("Result: " +
person.getName()));

// Collect the results into a list
List<Person> filteredList = people.stream()
    .filter(person -> person.getAge() > 50 &&
"male".equals(person.getGender()))
    .collect(Collectors.toList());

// Additional conditions: Convert names to uppercase
List<String> filteredNamesUppercase = filteredList.stream()
    .map(person -> person.getName().toUpperCase())
    .collect(Collectors.toList());
```

Here's how lazy evaluation works in the context of Java Streams:

Intermediate Operations are Lazy: Operations like filter, map, and flatMap are intermediate operations. When you call them on a stream, they don't execute immediately. Instead, they modify the stream pipeline and return a new stream.

```
Stream<Person> filteredStream = people.stream()
    .filter(person -> person.getAge() > 50 && "male".equals(person.getGender()));
```

At this point, the filter operation has modified the stream, but nothing has been processed yet.

Terminal Operation Triggers Processing: The processing of the stream occurs when a terminal operation is invoked. Examples of terminal operations include forEach, collect, reduce, and count.

```
filteredStream.forEach(person -> System.out.println("Result: " + person.getName()));
```

The forEach is a terminal operation, and it triggers the processing of the stream. At this point, the elements of the stream are processed through the pipeline.

Optimizations with Short-Circuiting: Some stream operations are designed to short-circuit, meaning they can stop processing elements once a certain condition is met. For example, `findFirst` on a stream will stop as soon as it finds the first element that matches the condition.

```
Optional<Person> firstPerson = people.stream()
    .filter(person -> person.getAge() > 50 && "male".equals(person.getGender()))
    .findFirst();
```

If a matching element is found early in the stream, the subsequent elements won't be processed.

Lazy evaluation allows for more efficient processing of large datasets, as only the elements necessary to compute the final result are processed. It enables optimizations like short-circuiting and avoids unnecessary computations.

Concept of lazy evaluation is fundamental to Java Streams, it's important to note that the lazy evaluation itself is not unique to streams. Lazy evaluation can be implemented in various ways, and it's a broader concept in computer science and programming languages.

For example, consider the following non-stream code:

```
List<Person> filteredPeople = new ArrayList<>();
for (Person person : people) {
    if (person.getAge() > 50 && "male".equals(person.getGender())) {
        filteredPeople.add(person);
    }
}

for (Person person : filteredPeople) {
    System.out.println("Result: " + person.getName());
}
```

In this non-stream code, lazy evaluation occurs implicitly because the elements are processed one by one in a loop. The second loop, which prints the results, only executes after the filtering is complete.

Java22

Stream Gatherers: A New Way to Enhance Your Java Streams

Stream Gatherers: A New Way to Enhance Your Java Streams

Stream gatherers enable you to create custom intermediate operations, which enables stream pipelines to transform data in ways that aren't easily achievable with existing built-in intermediate operations.

A gatherer is an intermediate operation that transforms a stream of input elements into a stream of output elements, optionally applying a final action when it reaches the end of the stream of input elements.

Gatherers can do the following:

- ✓ Transform elements in a one-to-one, one-to-many, many-to-one, or many-to-many fashion
- ✓ Track previously seen elements to influence the transformation of later elements
- ✓ Short-circuit, or stop processing input elements to transform infinite streams to finite ones
- ✓ Process a stream in parallel

Examples of gathering operations include the following:

- ✓ Grouping elements into batches
- ✓ Deduplicating consecutively similar elements
- ✓ Incremental accumulation functions
- ✓ Incremental reordering functions

How It Works

Gatherers work by consuming elements from an input stream and producing elements for an output stream. They have four key functions:

- ✓ **Initializer:** Creates a state object to track information during processing.
- ✓ **Integrator:** Consumes each input element, updates the state, and optionally pushes elements to the output stream.
- ✓ **Combiner:** Merges state objects from parallel streams, if applicable.
- ✓ **Finisher:** Performs any final actions and optionally pushes more elements to the output stream.

Built-in Gatherers

To get you started, Java 22 provides several useful built-in stream gatherers:

fold: Creates an aggregate incrementally and emits it at the end.

mapConcurrent: Executes a function concurrently for each input element.

scan: Produces a new element based on the current state and input element.

windowFixed: Groups elements into fixed-size lists.

windowSliding: Creates sliding windows over the input stream.

fold(Supplier initial, BiFunction folder): This is an many-to-one gatherer that constructs an aggregate incrementally until no more input elements exist. It has two parameters:

initial: This is the identity value or the value that the gatherer emits if the input stream contains no elements.

folder: This is a lambda expression that contains two parameters: the first is the aggregate the gatherer is constructing and the second is the element that's currently being processed.

```
var semicolonSeparated =  
    Stream.of(1,2,3,4,5,6,7,8,9)  
        .gather(  
            Gatherers.fold(  
                () -> "",  
                (result, element) -> {  
                    if (result.equals("")) return element.toString();  
                    return result + ";" + element;  
                }  
            )  
        )  
.findFirst()  
.get();  
  
System.out.printlnsemicolonSeparated);
```

Output: 1;2;3;4;5;6;7;8;9

mapConcurrent(int maxConcurrency, Function mapper): This is a one-to-one gatherer that invokes mapper for each input element in the stream concurrently, up to the limit specified by maxConcurrency. You can use this limit for the following:

As a rate-limiting construct to prevent the gatherer from issuing too many concurrent requests to things like an external service or a database.

As a performance-enhancer to enable multiple, separate operations to be performed concurrently while avoiding converting the entire stream into a parallel stream.

This gatherer preserves the ordering of the stream.

scan(Supplier initial, BiFunction scanner):

This is a one-to-one gatherer that performs a prefix scan, which is an incremental accumulation. Starting with an initial value obtained from the parameter initial, it obtains subsequent values by applying scanner to the current value and the next input element. The gatherer then emits the value downstream.

The following example demonstrates this gatherer:

```
Stream.of(1,2,3,4,5,6,7,8,9)
    .gather(Gatherers.scan(() -> 100,
                           (current, next) -> current + next))
    .forEach(System.out::println);
```

Output:

```
101
103
106
110
115
121
128
136
145
```

windowFixed(int windowSize):

This is a many-to-many gatherer that gathers elements in windows, which are encounter-ordered groups of elements.

The parameter `windowSize` specifies the size of the windows.

The following example demonstrates this gatherer:

```
List<List<Integer>> windows =  
    Stream.of(1,2,3,4,5,6,7,8).gather(Gatherers.windowFixed(3)).toList();  
    windows.forEach(System.out::println);
```

Output:

```
[1, 2, 3]  
[4, 5, 6]  
[7, 8]
```

windowSliding(int windowSize):

Similar to windowFixed, this is a many-to-many gatherer that gathers elements in windows. However, each subsequent window includes all elements of the previous window except for its first element, and adds the next element in the stream.

The following example demonstrates this gatherer:

```
List<List<Integer>> moreWindows =  
    Stream.of(1,2,3,4,5,6,7,8).gather(Gatherers.windowSliding(3)).toList();  
    moreWindows.forEach(System.out::println);
```

Output:

```
[1, 2, 3]  
[2, 3, 4]  
[3, 4, 5]  
[4, 5, 6]  
[5, 6, 7]  
[6, 7, 8]
```

However, there are differences between the stream and non-stream approaches:

1. **Expressiveness:** Stream API provides a more concise and expressive way to write operations on collections, making the code more readable and maintainable.
2. **Parallelism:** Streams make it easier to parallelize operations on data, allowing for more efficient processing on multicore machines. The parallel processing is handled by the Stream API, and you don't need to explicitly manage threads.
3. **Functional Style:** Streams encourage a functional programming style, where you express operations as a series of transformations on data. This style can lead to more modular and composable code.

In summary, lazy evaluation is a general concept, and both streams and non-stream code can exhibit lazy behavior. However, the Stream API in Java provides a convenient and expressive way to work with data, leveraging lazy evaluation and offering additional benefits in terms of expressiveness and parallelism.

Conclusion

- Title: Empowering Java Development with Streams
- Summary:
 - Simplify Data Processing
 - Enable Functional-Style Programming
 - Harness Parallelism for Efficiency

Introduction to Java Date and Time API

Introduction to Java Date and Time API

- Title: Modernizing Date and Time Handling with Java 8
- Content:
 - Java 8 introduced the `'java.time'` package for a robust Date and Time API.
 - Addresses shortcomings and complexities of `'java.util.Date'` and `'java.util.Calendar'`.
 - Key classes include:
 - `'LocalDate'`
 - `'LocalTime'`
 - `'LocalDateTime'`
 - `'ZonedDateTime'`
 - `'Instant'`
 - `'Duration'`
 - `'Period'`
 - `'DateTimeFormatter'`
 - Designed for immutability and thread-safety.

Key Date and Time Classes

- Title: Essential Classes in `java.time`
- Content:
 - `'LocalDate'`: Represents date (year, month, day) without time information.

java

 Copy code

```
LocalDate date = LocalDate.now(); // Current date
```

- `'LocalTime'`: Represents time without date information.

java

 Copy code

```
LocalTime time = LocalTime.now(); // Current time
```

- `LocalDateTime`: Represents date and time without a time zone.

java

 Copy code

```
LocalDateTime dateTime = LocalDateTime.now(); // Current date and time
```

- `ZonedDateTime`: Represents date and time with a time zone.

java

 Copy code

```
ZoneId zoneId = ZoneId.of("America/New_York");
ZonedDateTime zonedDateTime = ZonedDateTime.now(zoneId); // Current date and time
```

- `Instant`: Represents a point in time, typically used for timestamps.

java

 Copy code

```
Instant instant = Instant.now(); // Current timestamp
```

More Date and Time Classes

- Title: Additional Classes in `java.time`

- Content:

- `Duration`: Represents a length of time.

```
LocalDateTime start = LocalDateTime.now();
LocalDateTime end = start.plusHours(2).plusMinutes(30);
Duration duration = Duration.between(start, end);
```

- `Period`: Represents a date-based amount of time.

```
LocalDate birthday = LocalDate.of(1990, 5, 15);
LocalDate today = LocalDate.now();
Period period = Period.between(birthday, today);
```

- `DateTimeFormatter`: Allows formatting and parsing of dates and times.

```
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss")
String formattedDateTime = LocalDateTime.now().format(formatter);
```

Stream API Integration

- Title: **Using Stream API with Date and Time**
- Content:
 - Stream API in Java 8 is designed for processing collections.
 - While not directly related to date and time, Stream API complements date/time operations.
 - Example 1:
 - Filtering LocalDateTime objects by month using Stream API.
 - Example 2:
 - Generating a stream of dates for the next 7 days.

Example 1 - Filtering DateTimes

- Title: **Example 1: Filtering DateTimes**
- Content:
 - Code Example:

```
java Copy code  
  
List<LocalDateTime> dateTimes = /* Populate the list */;  
dateTimes.stream()  
    .filter(dateTime -> dateTime.getMonthValue() == 2) // Selecting February  
    .forEach(System.out::println);
```

- Explanation:
 - Creates a list of LocalDateTime objects.
 - Uses Stream API to filter date times for February.
 - Prints the filtered date times.

Example 2 - Generating Dates

- Title: Example 2: Generating Dates with Stream

- Content:

- Code Example:

```
java Copy code  
  
LocalDate startDate = LocalDate.now();  
List<LocalDate> next7Days = Stream.iterate(startDate, date -> date.plusDays(1))  
    .limit(7)  
    .collect(Collectors.toList());  
next7Days.forEach(System.out::println);
```

- Explanation:

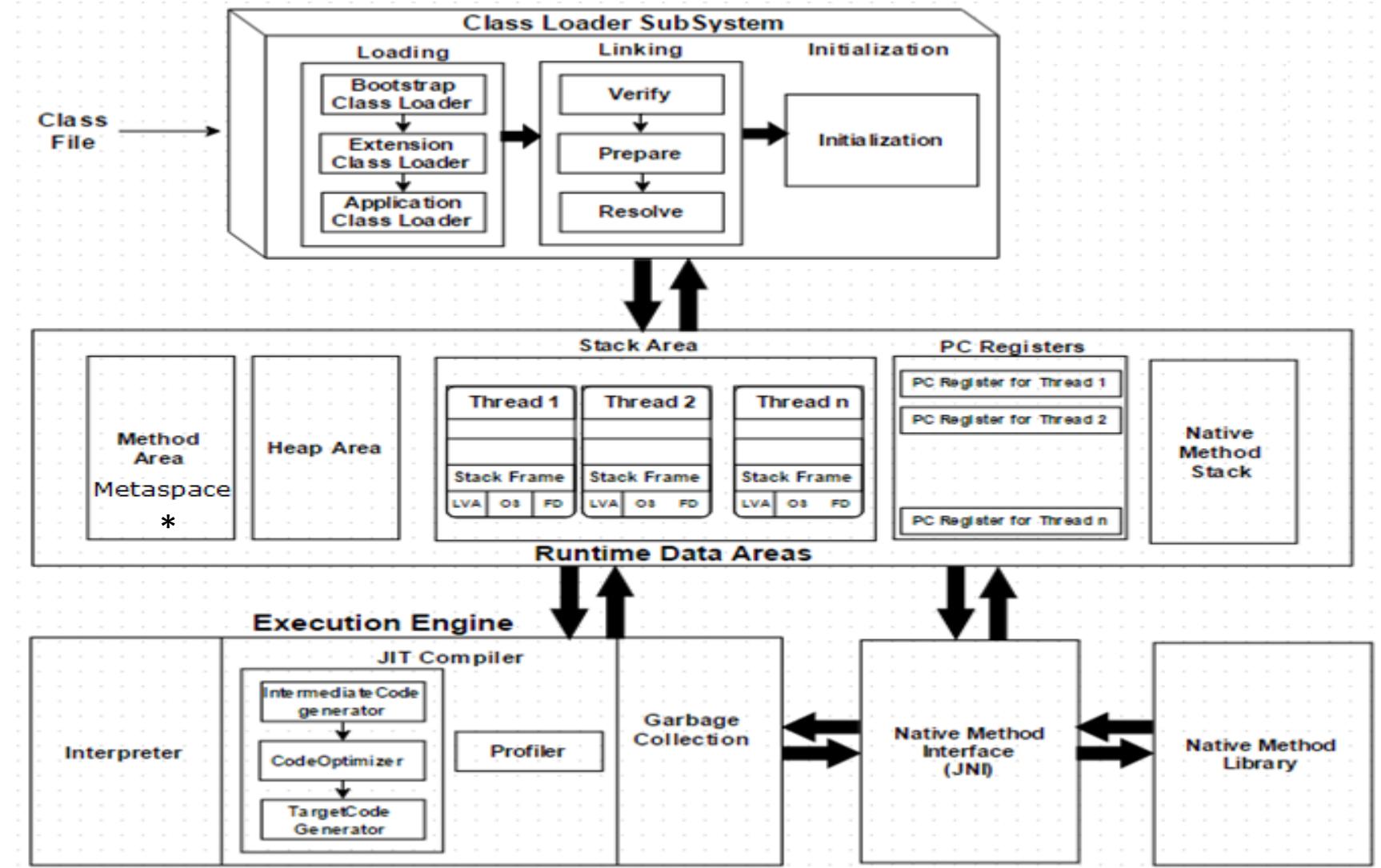
- Starts with the current date.
 - Uses Stream.iterate to generate a stream of dates for the next 7 days.
 - Collects the stream elements into a List.
 - Prints the generated dates.

Conclusion

- Title: **Conclusion: Java Date and Time Mastery**
- Content:
 - Java 8's `java.time` package revolutionizes date and time handling.
 - Key classes provide flexibility and precision.
 - Integration with Stream API enhances data processing capabilities.
 - Say goodbye to the complexities of old Date and Calendar classes.

Java modules

JVM Architecture Diagram



* Compressed Class Space

Class Loader Subsystem

Java's dynamic class loading functionality is handled by the class loader subsystem.

It loads, links, and initializes the class file when it refers to a class for the first time at runtime, not compile time.

Class Loading

Classes will be loaded by this component. Boot Strap class Loader, Extension class Loader, and Application class Loader are the three class loader which will help in achieving it.

Boot Strap ClassLoader – Responsible for loading classes from the bootstrap classpath, nothing but rt.jar. Highest priority will be given to this loader.

Extension ClassLoader – Responsible for loading classes which are inside ext folder (jre\lib\ext).

-Djava.ext.dirs.=<directory-name>

* jdk 9 onwards, the extension class loader is removed

System ClassLoader – Responsible for loading Application Level Classpath, path mentioned Environment Variable etc.

The above Class Loaders will follow Delegation Hierarchy Algorithm.

Note : Use OSGI/ Java Dynamic Modules

Linking

Verify – Bytecode verifier will verify whether the generated bytecode is proper or not if verification fails we will get the verification error.

Prepare – For all static variables memory will be allocated (Method area) and assigned with default values.

Resolve – Java class is compiled, all references to variables and methods are stored in the class's constant pool as a symbolic reference. All symbolic memory references are replaced with the original references from Method Area.

A *Java module* is a packaging mechanism that enables you to package a Java application or Java API as a separate Java module.

A Java module is packaged as a *modular JAR file*.

A Java module can specify which of the Java packages it contains that should be visible to other Java modules which uses this module.

Before Java 9 and the Java Platform Module System you would have had to package all of the Java Platform APIs with your Java application.

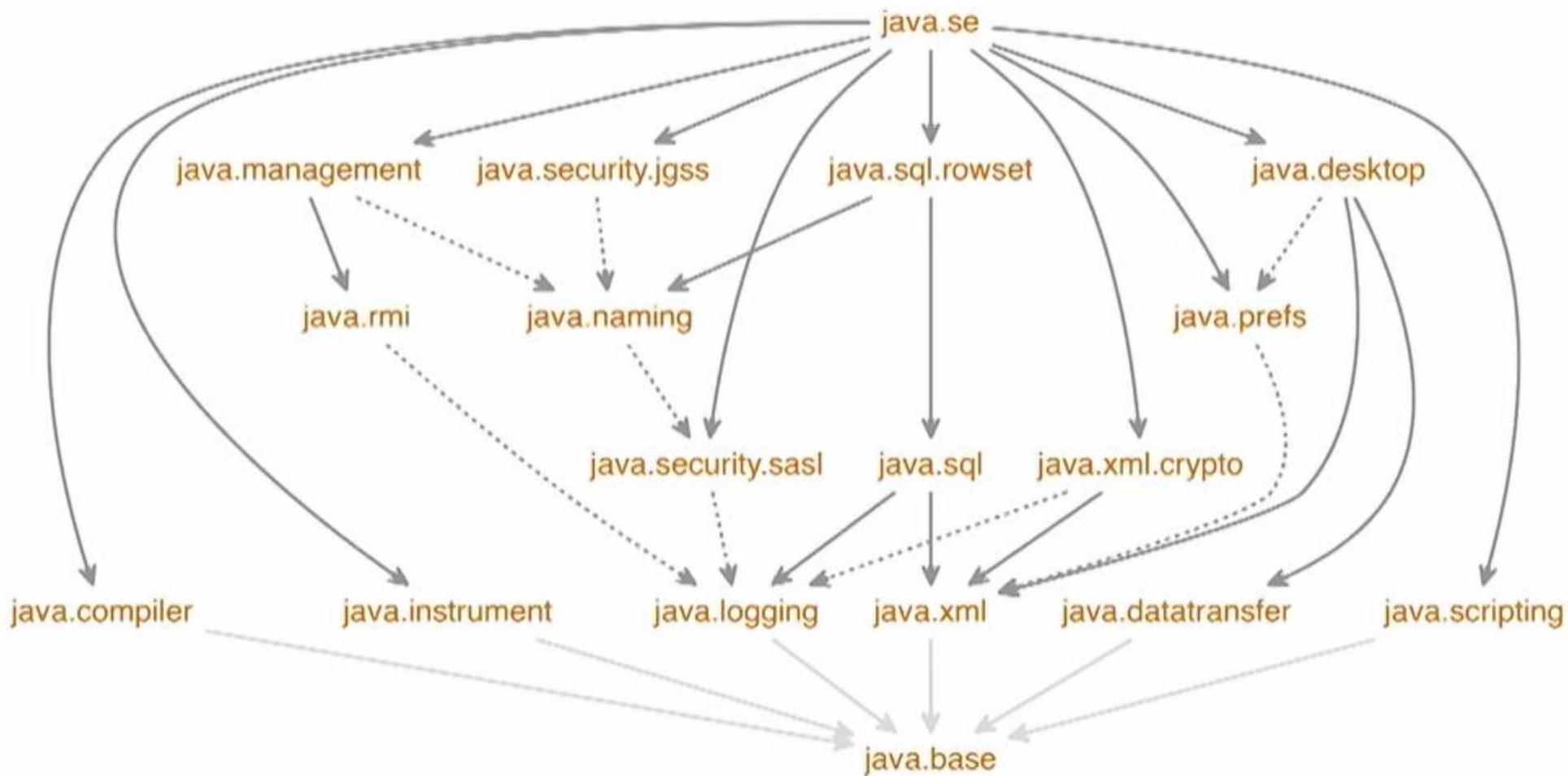
The unused classes makes your application distributable bigger than it needs to be.

Java Module System is a major change in Java 9 version. Java added this feature to collect Java packages and code into a single unit called module.

In earlier versions of Java, there was no concept of module to create modular Java applications, that why size of application increased and difficult to move around. Even JDK itself was too heavy in size, in Java 8, rt.jar file size is around 64MB.

Java 9 restructured JDK into set of modules so that we can use only required module for our project.

The Modular JDK



Default Modules

When we install Java 9, we can see that the JDK now has a new structure. They have taken all the original packages and moved them into the new module system.

We can see what these modules are by typing into the command line:

java --list-modules

These modules are split into four major groups: java, javafx, jdk, and Oracle.

java modules are the implementation classes for the core SE Language Specification.

javafx modules are the FX UI libraries.

Anything needed by the JDK itself is kept in the jdk modules.

And finally, anything that is Oracle-specific is in the oracle module

Encapsulation of Internal Packages

A Java module must explicitly tell which Java packages inside the module are to be exported (visible) to other Java modules using the module.

A Java module can contain Java packages which are not exported. Classes in unexported packages cannot be used by other Java modules. Such packages can only be used internally in the Java module that contains them.

Packages that are not exported are also referred to as hidden packages, or encapsulated packages.

Modules Contain One or More Packages

A Java module is one or more Java packages that belong together. A module could be either a full Java application, a Java Platform API, or a third party API.

Module Naming

A Java module must be given a unique name.

com.examples.samplemodule

Module Root Directory

From Java 9 the Java Platform Module System offers an alternative directory structure which can make it easier to compile Java sources. From Java 9 a module can be nested under a root directory with the same name as the module.

com.examples.samplemodule/com/examples/samplemodule

Module Descriptor (module-info.java)

Each Java module needs a Java module descriptor named module-info.java which has to be located in the corresponding module root directory.

com.examples.samplemodule/com/examples/samplemodule/module-info.java

Module Exports

A Java module must explicitly export all packages in the module that are to be accessible for other modules using the module.

```
module com.examples.samplemodule {  
    exports com.examples.samplemodule;  
}
```

Module Requires

If a Java module requires another module to do its work, that other module must be specified in the module descriptor too. Here is an example of a Java module requires declaration:

```
module com.examples.samplemodule {  
    requires javafx.graphics;  
}
```

Introduction

- Title: Importance of Modules in Java
- Brief overview of Java modules and their significance in application development.

Encapsulation

Traditional Approach

- All classes in a single unnamed module.
- Potential encapsulation issues.
- Any class can access any other class.

With Modules

- Explicitly declare accessible packages.
- Promotes encapsulation.
- Reduces unintended dependencies.

Dependency Management

Traditional Approach

- Challenges in managing dependencies.
- Classpath conflicts and version mismatches.

With Modules

- Explicit declaration of dependencies.
- Easier understanding and management.
- Express transitive dependencies.

Service Provider Mechanism

Traditional Approach

- Service provider patterns using external mechanisms or frameworks.

With Modules

- Use of provides and uses directives.
- Simplifies implementation of service provider patterns.
- Enhances modularity and extensibility.

Improved Readability and Maintainability

Traditional Approach

- Harder to understand the structure as codebases grow.

With Modules

- Clear module structure.
- `module-info.java` as a high-level overview.
- Enhances code readability and maintainability.

Reduced Classpath Hell

Traditional Approach

- Managing a large classpath leads to conflicts and errors.

With Modules

- Clear boundary between application parts.
- Reduces naming conflicts.
- Easier reasoning about dependencies.

Security

Traditional Approach

- Non-modular projects expose sensitive functionality.

With Modules

- Restrict access to certain packages.
- Limits scope for other modules.
- Enhances security.

Summary

- Java modules provide a structured and modular approach.
- Address challenges of the traditional classpath-based approach.
- Enhance encapsulation, improve dependency management, and provide a foundation for more maintainable and secure software.

Example Overview - Common Module com.example

Purpose

- Defines MyService interface.

module-info.java

- exports com.example: Makes the package accessible to other modules.

Example Overview - Module `com.module.simple`

Purpose

- Provides `Simple` class.
- Implements `MyService` interface.

`module-info.java`

- `exports com.module.simple`: Makes the package accessible.
- `requires transitive java.logging`: Indicates a transitive dependency.
- `provides com.example.MyService with com.module.simple.Simple`: Specifies the implementation.

Example Overview - Module com.module.client

Purpose

- Contains SimpleClient class.
- Interacts with Simple class and MyService interface.

module-info.java

- requires com.module.simple: Declares a dependency.
- uses com.example.MyService: Indicates interface usage.

Conclusion

- Java modules offer a more organized development approach.
- Address encapsulation, dependency management, service provider patterns, readability, classpath issues, and security.
- Encourage modular and extensible software design.

What is JShell?

What is JShell?

Definition

- Java Shell (JShell):
 - Interactive command-line tool.
 - Introduced in Java 9.
 - Execute Java code snippets and commands directly.

Purpose

- Provides a lightweight and interactive environment.
- Allows experimentation with Java code.
- Facilitates testing ideas and learning language features.

Features of JShell

- **Interactive Execution:**
 - No need for a complete Java program.
 - Directly execute code snippets and commands.
- **Experimentation:**
 - Test and iterate code quickly.
 - Instant feedback on language features.
- **Learning Tool:**
 - Ideal for learning Java syntax and features.
 - Explore and understand language constructs.

Examples in JShell

1. Basic Arithmetic:

```
java
```

 Copy code

```
int result = 5 + 3
```

2. Variable Declaration and Assignment:

```
java
```

 Copy code

```
String message = "Hello, JShell!"
```

3. Defining a Method:

```
java
```

 Copy code

```
void greet(String name) {  
    System.out.println("Hello, " + name + "!");  
}
```

4. Invoking a Method:

```
java
```

 Copy code

```
greet("Ramana")
```

5. Exploring APIs:

```
java
```

 Copy code

```
import java.time.LocalDate  
LocalDate.now()
```

6. Lambda Expression:

```
java
```

 Copy code

```
IntBinaryOperator sumFunction = (int x, int y) -> x + y  
sumFunction.applyAsInt(3, 7)
```

JShell Benefits

- **Rapid Prototyping:**
 - Quickly try out code snippets.
- **Learning Aid:**
 - Experiment with language features.
- **Debugging Assistance:**
 - Test code fragments for correctness.
- **API Exploration:**
 - Interactively explore and test APIs.

Conclusion

- **JShell:**
 - Empowers developers with an interactive Java environment.
 - Enables rapid experimentation and learning.
 - Valuable tool for testing and exploring Java code.

Enhanced Streams

`takeWhile` and `dropWhile`

`takeWhile`

- Takes elements from the stream until the given predicate becomes false.

Example:

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

List<Integer> result = numbers.stream()
    .takeWhile(n -> n < 4)
    .collect(Collectors.toList());
```

`'dropWhile'`

- Skips elements from the stream until the given predicate becomes false.

Example:

java

 Copy code

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

List<Integer> result = numbers.stream()
    .dropWhile(n -> n < 4)
    .collect(Collectors.toList());
```

``ofNullable` for Streams`

- `Stream.ofNullable` creates a stream containing a single non-null element.
- It returns an empty stream if the element is null, avoiding `NullPointerExceptions`.

Example:

java

 Copy code

```
String value = "example";  
  
Stream<String> stream = Stream.ofNullable(value);
```

`'iterate' with a Predicate`

- `'Stream.iterate'` allows iteration based on a predicate.
- In this example, we start with 1 and iterate, doubling the value until the predicate ($n < 10$) becomes false.

Example:

java

 Copy code

```
Stream<Integer> stream = Stream.iterate(1, n -> n < 10, n -> n * 2);
```

Benefits and Use Cases

`'takeWhile'` and `'dropWhile'`

- Benefits:
 - Precise control over stream elements.
 - Enhanced readability for conditional stream processing.
- Use Cases:
 - Filtering elements based on specific conditions.
 - Dynamically adjusting stream content based on changing criteria.

`'ofNullable'` for Streams

- Benefits:
 - Avoids NullPointerExceptions in stream operations.
 - Provides a concise way to handle null values.
- Use Cases:
 - Safely incorporating potentially null values into streams.
 - Enhancing robustness in stream processing.

`'iterate'` with a Predicate

- Benefits:
 - Allows dynamic iteration based on a condition.
 - Enhances flexibility in stream generation.
- Use Cases:
 - Iterative operations until a specific condition is met.
 - Dynamic stream generation based on changing criteria.

Conclusion

- Stream API Enhancements:
 - `takeWhile` and `dropWhile` for precise control.
 - `ofNullable` for safer stream processing.
 - `iterate` with a Predicate for dynamic iteration.
- Powerful Tools:
 - Empower developers with more expressive and robust stream operations.
 - Enhance code readability and reduce the risk of errors.

Reactive Streams in Java

Introduction to Reactive Streams

- **Definition:**
 - Specification for asynchronous stream processing.
 - Addresses challenges of handling asynchronous and potentially unbounded data streams efficiently.
- **Purpose:**
 - Support the development of reactive systems.
 - Handle streams of data asynchronously and responsively.

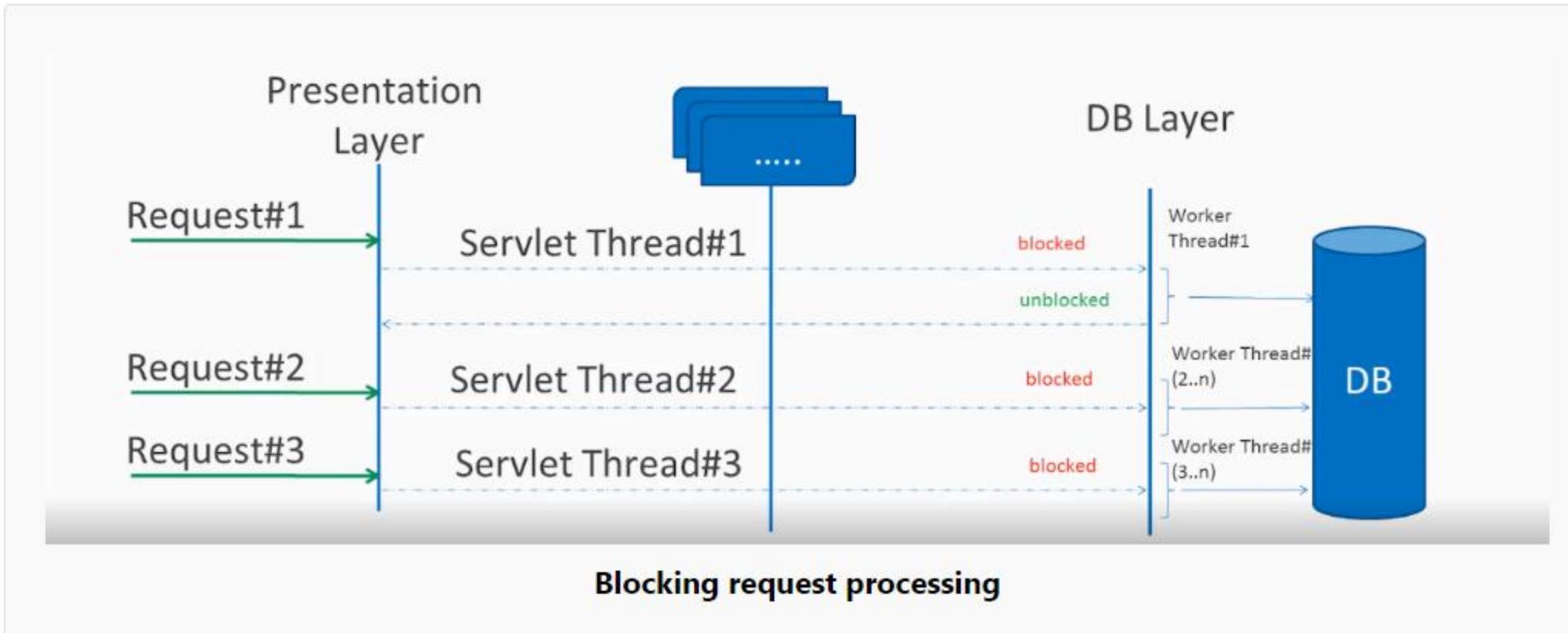
Use cases:

1. Capital One redesigned its auto loan application around Reactive principles to simplify online car shopping and financing. Customers can browse more than four million cars from over 12,000 dealers and pre-qualify for financing in seconds, without impacting credit scores.
2. LinkedIn turned to Reactive principles to build real-time presence indicators (online indicators) for the half billion users on its social network.
3. Verizon Wireless, operators of the largest 4G LTE network in the United States, slashed response times in half using Reactive principles in the upgrade of its e-commerce website that supports 146 million subscribers handling 2.5 billion transactions a year.
4. Walmart Canada rebuilt its entire Web application and mobile stack as a Reactive system and saw a 20 percent increase in conversion to sales from web traffic and a 98 percent increase in mobile orders while cutting page load times by more than a third.

Blocking request processing

In traditional MVC applications, when a request come to server, a servlet thread is created. It delegates the request to worker threads for I/O operations such as database access etc.

During the time worker threads are busy, servlet thread (request thread) remain in waiting status and thus it is blocked. It is also called synchronous request processing.

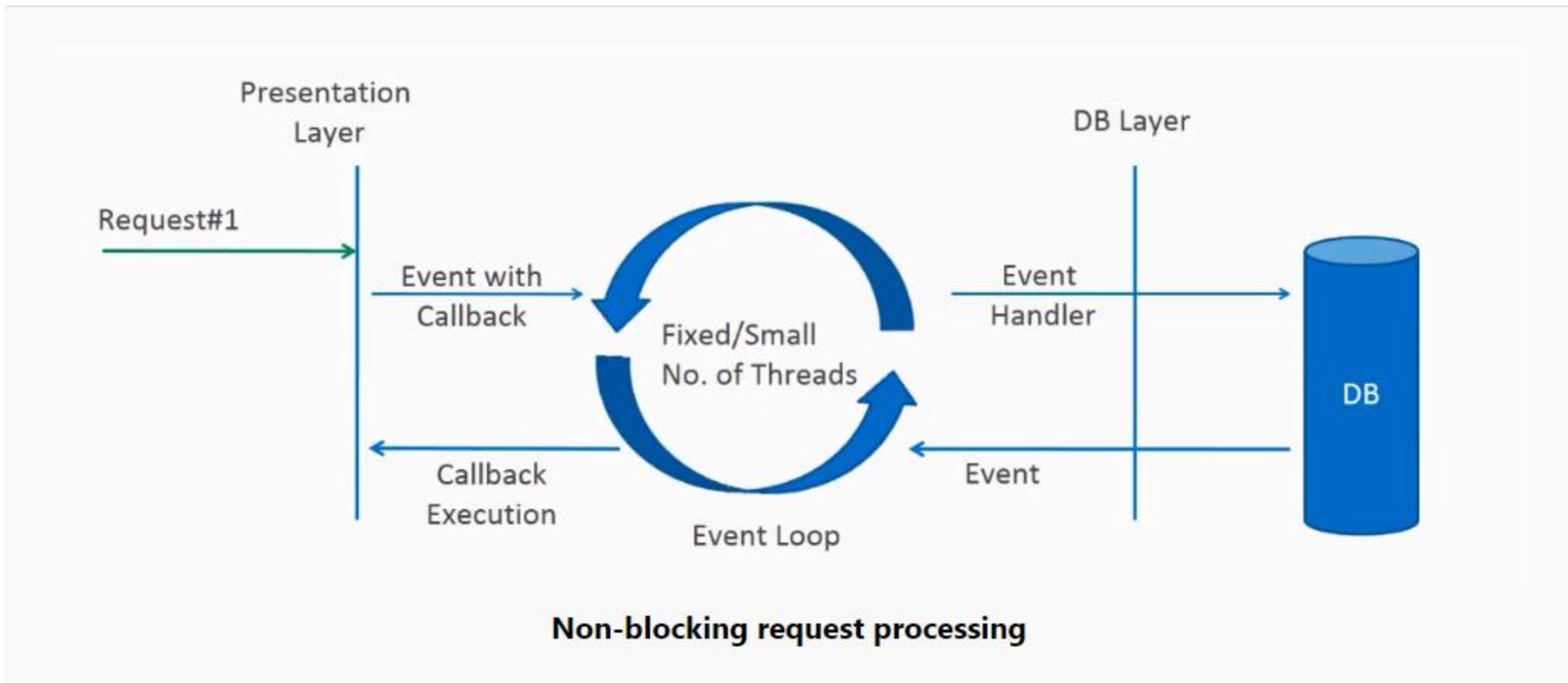


Non-blocking request processing

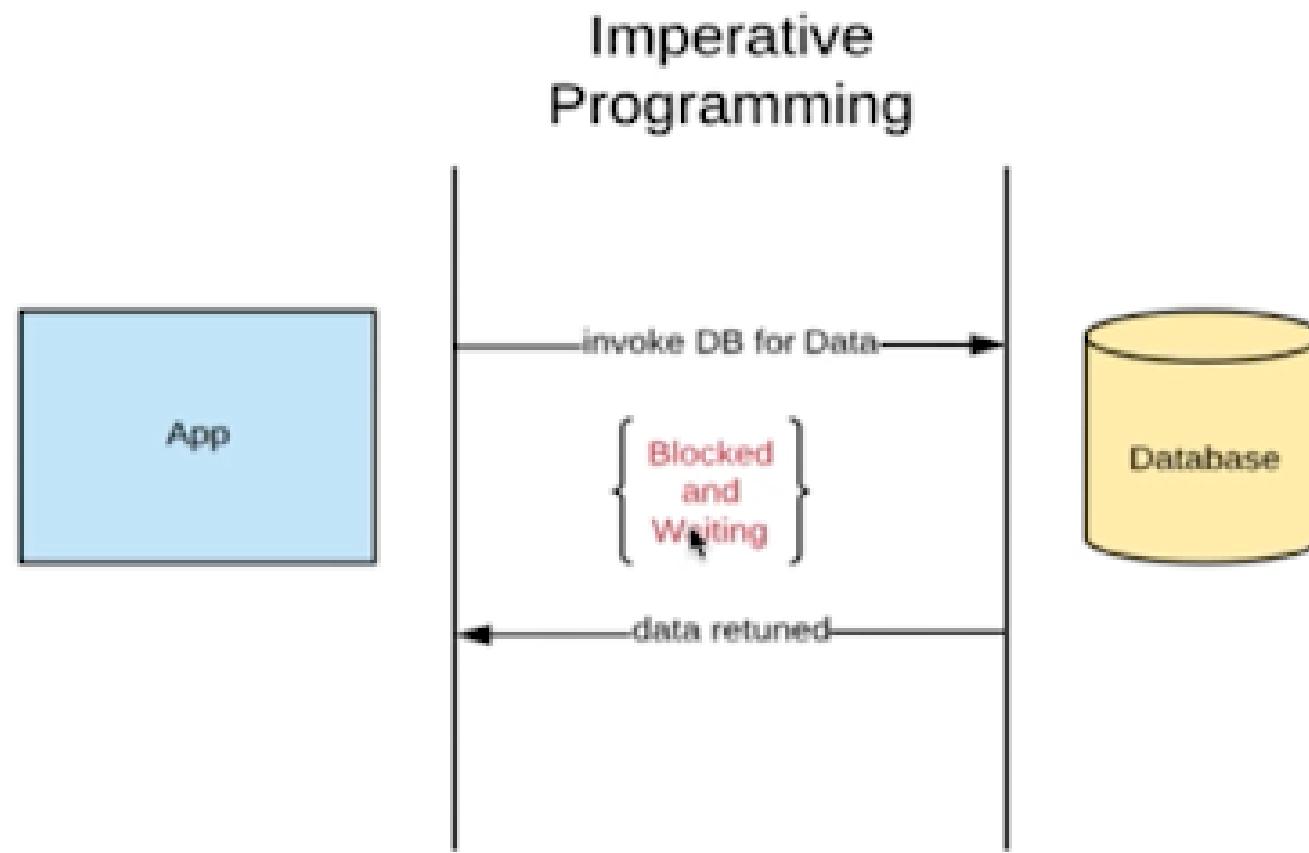
In non-blocking or asynchronous request processing, no thread is in waiting state. There is generally only one request thread receiving the request.

All incoming requests come with a event handler and call back information. Request thread delegates the incoming requests to a thread pool (generally small number of threads) which delegate the request to it's handler function and immediately start processing other incoming requests from request thread.

When the handler function is complete, one of thread from pool collect the response and pass it to the call back function.



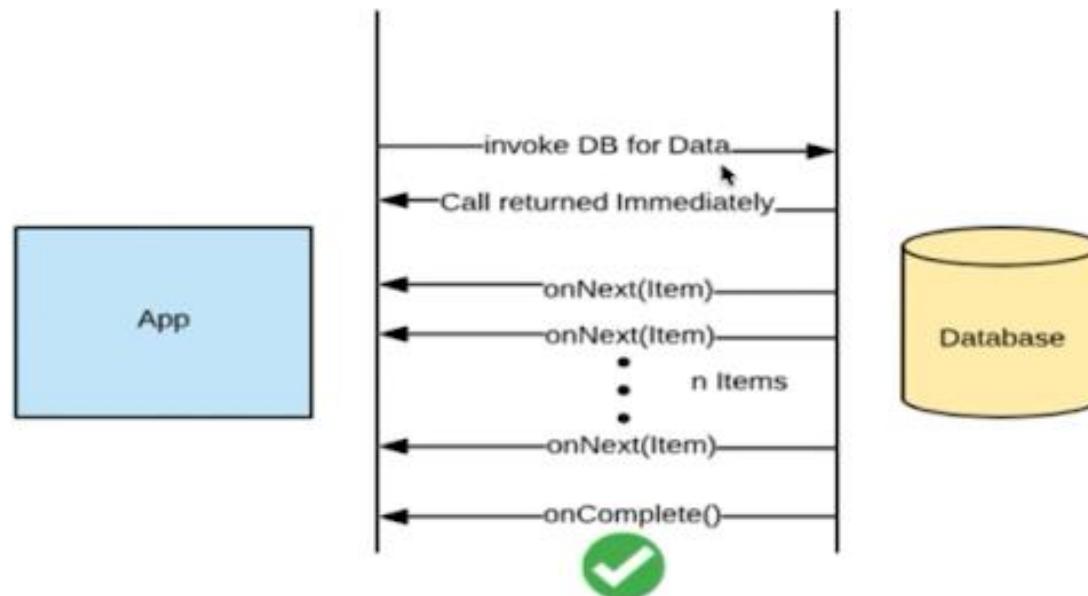
Imperative Programming, Blocking & synchronous:



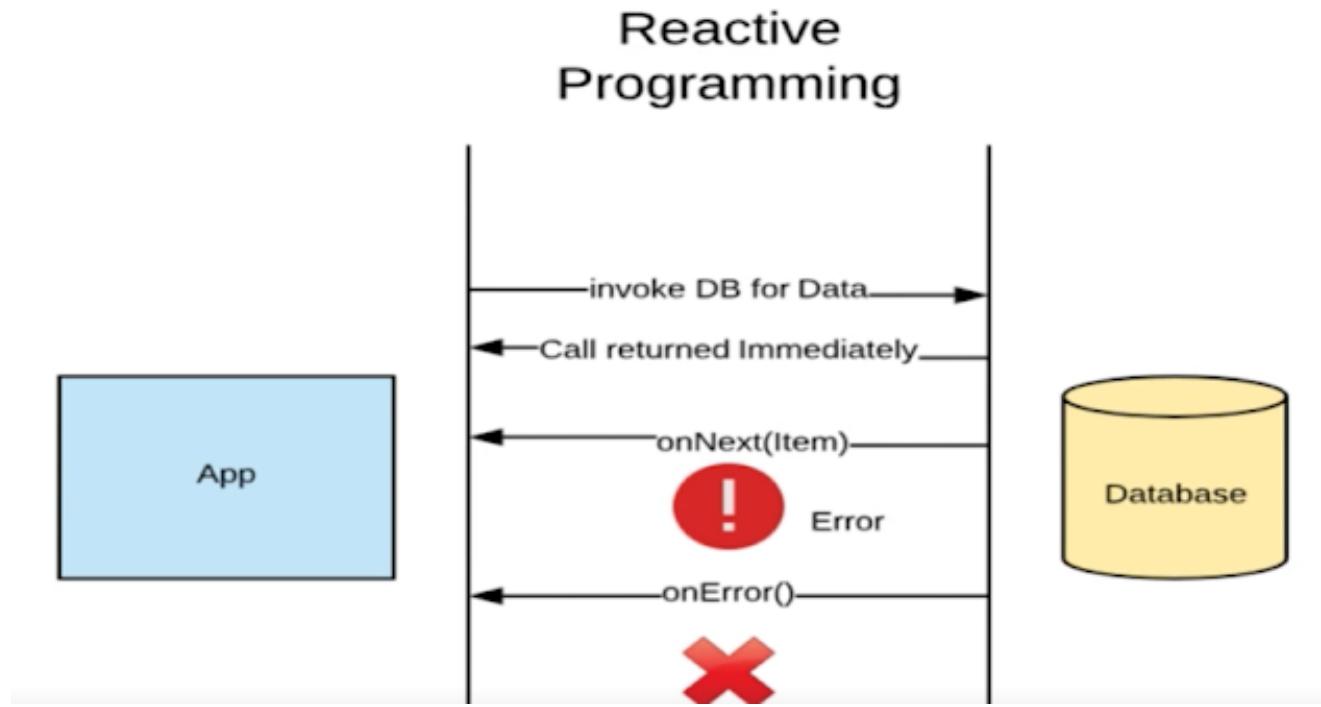
Happy path - where there is no Error:

```
List<Item> items = itemRepository.getAllItems();
```

Reactive Programming

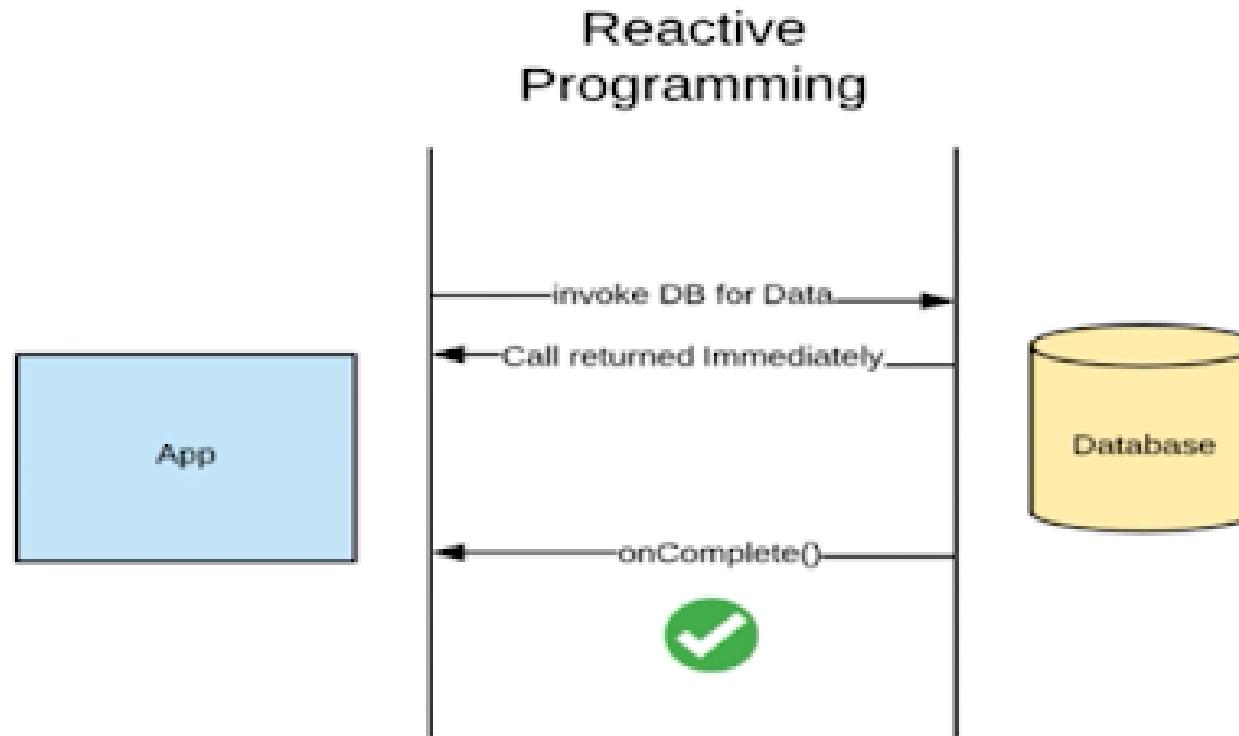


Error case:



Incase of any error during while passing the data to the client in **onNext(Item)** method call, the **onError()** method will be triggered and will communicate the client about the error.

No-data flow:



Incase of no data found, then the transaction will be made complete with onComplete() call.

Key Interfaces in Reactive Streams API

1. Publisher

- Represents a source of data or events.
- Produces items and can be subscribed to by one or more subscribers.

Example:

```
Publisher<String> dataPublisher = // Create a data publisher
```

1. Subscriber

- Consumes items from a Publisher.
- Subscribes to a Publisher and receives items through the `onNext` method.
- Can handle errors (`onError`) and signal completion (`onComplete`).

Example:

```
Subscriber<String> dataSubscriber = // Create a data subscriber
```

1. Subscription

- Represents the link between a Publisher and a Subscriber.
- Allows a Subscriber to request a certain number of items and handle cancellation.

Example:

```
Subscription dataSubscription = // Create a data subscription
```

1. Processor

- Represents a processing stage that is both a Subscriber and a Publisher.
- Consumes items from a source Publisher, applies processing, and publishes results to downstream Subscribers.

Example:

```
Processor<String, Integer> dataProcessor = // Create a data processor
```

Reactive Streams in Action

1. Creating a Simple Reactive Pipeline:

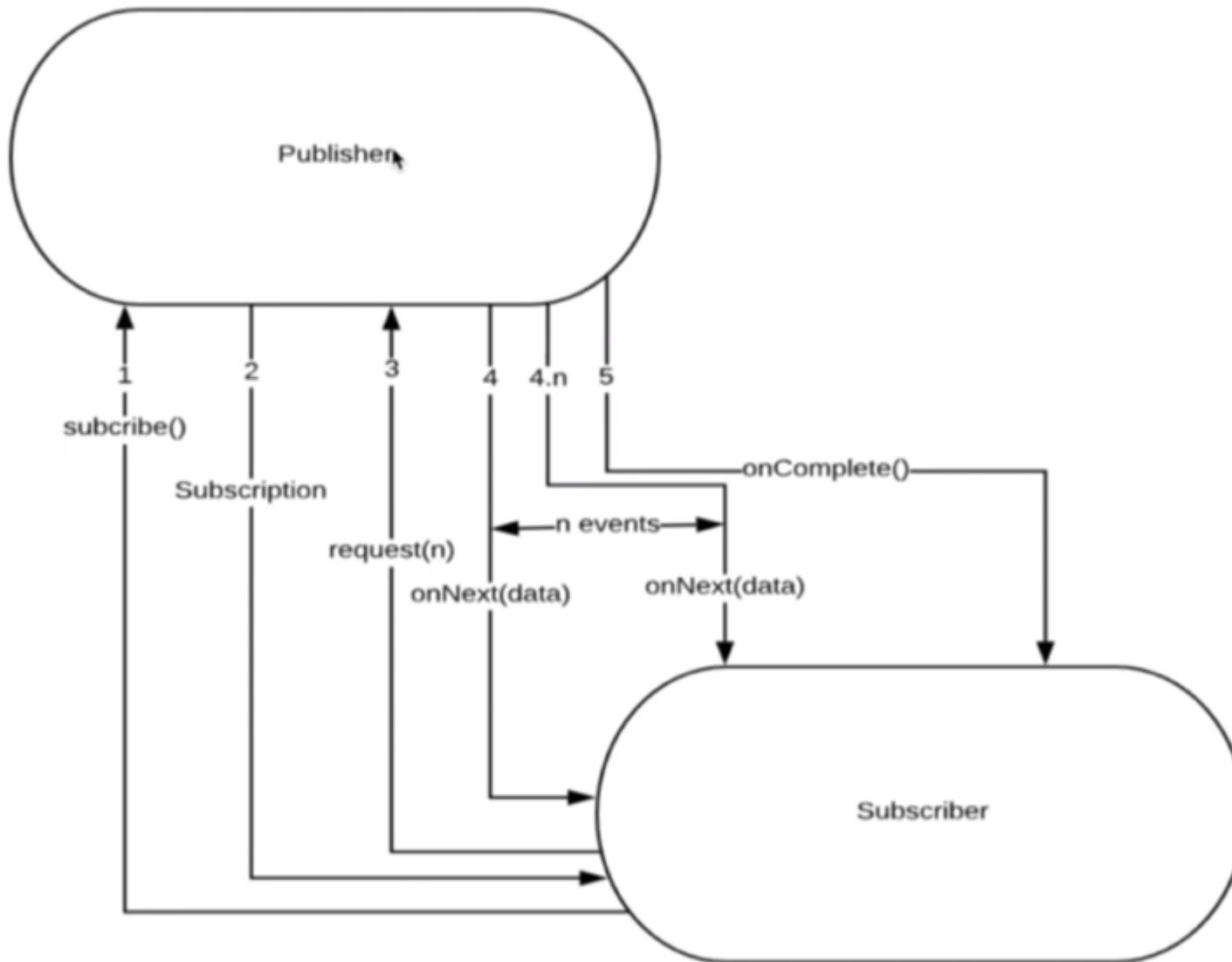
- **Publisher:** Produces data.
- **Processor:** Applies transformation.
- **Subscriber:** Consumes processed data.

Example:

```
Publisher<String> source = // Create a data source
Processor<String, Integer> processor = // Create a processor
Subscriber<Integer> sink = // Create a data sink

source.subscribe(processor);
processor.subscribe(sink);
```

Publisher/Subscriber Event Flow



1. Handling Backpressure:

- Reactive Streams handle backpressure to avoid overwhelming subscribers.

Example:

```
Subscription subscription = // Obtain a subscription
subscription.request(10); // Request a specific number of items
```

Java itself provides support for reactive programming through the introduction of Reactive Streams in Java 9. The Reactive Streams API defines a standard for asynchronous stream processing with non-blocking backpressure. This API is part of the `'java.util.concurrent.Flow'` package and includes interfaces like `'Publisher'`, `'Subscriber'`, and `'Subscription'`.

However, for more advanced and convenient reactive programming features, developers often use third-party libraries that build on top of the Reactive Streams API. Two popular libraries for reactive programming in Java are Project Reactor and RxJava.

1. Project Reactor:

- **Features:** Project Reactor is a powerful reactive programming library that provides additional abstractions beyond the Reactive Streams API. It introduces the concepts of `Flux` for handling multiple items and `Mono` for zero or one item.
- **Use Cases:** Project Reactor is commonly used in Spring applications for building reactive microservices, but it can be used in various scenarios where reactive programming is beneficial.

2. RxJava:

- **Features:** RxJava is the Java implementation of the ReactiveX (Rx) pattern. It offers a rich set of operators for composing asynchronous and event-driven programs. RxJava is known for its expressive API and comprehensive set of operators.
- **Use Cases:** RxJava is widely used in various domains, including Android development, server-side applications, and systems dealing with asynchronous and event-driven scenarios.

Conclusion

- **Reactive Streams:**
 - Efficiently handle asynchronous and potentially unbounded data.
 - Provide a set of interfaces for building reactive systems.
- **Key Interfaces:**
 - Publisher, Subscriber, Subscription, Processor.
- **Implementation Examples:**
 - Create a simple reactive pipeline.
 - Handle backpressure effectively.

Advantages of Var Keyword

Examples:

```
var id = 0; // Compiler interprets id as an integer.  
id = "34"; // Compilation error due to incompatible types.
```

```
var p = new Doctor(); // Compiler infers the type.
```

```
var list = new ArrayList<String>(); // Type inferred as ArrayList<String>.
```

```
var list = Arrays.asList("One", "Two", "Three", "Four", "Five");  
                    // Inferred as List<String>.
```

Limitations of `var`

- **Cannot declare local variables inside methods:**
 - `var` cannot be used to declare local variables within methods.
- **Cannot declare method parameters or instance fields:**
 - `var` is not allowed for method parameters or instance fields.
- **Cannot specify the return type of a method:**
 - The return type of a method cannot be declared using `var`.

Examples of `var` Limitations

```
// Cannot use var for local variables inside methods
public void exampleMethod() {
    var count = 10; // Compilation error
}
```

```
// Cannot use var for method parameters or instance fields
public class ExampleClass {
    private var fieldName; // Compilation error

    public void exampleMethod(var parameter) { // Compilation error
        // Method implementation
    }
}
```

```
// Cannot specify the return type of a method using var
public var exampleMethod() { // Compilation error
    // Method implementation
}
```

Conclusion

- **Advantages:**
 - Reduces verbosity in variable declarations.
 - Enhances code readability.
- **Limitations:**
 - Cannot be used for local variables inside methods.
 - Not allowed for method parameters or instance fields.
 - Cannot specify the return type of a method.
- **Use `var` judiciously:**
 - Consider readability and context when deciding to use or not use `var`.

Class Data Sharing (CDS) in Java

Class Data Sharing (CDS) in Java

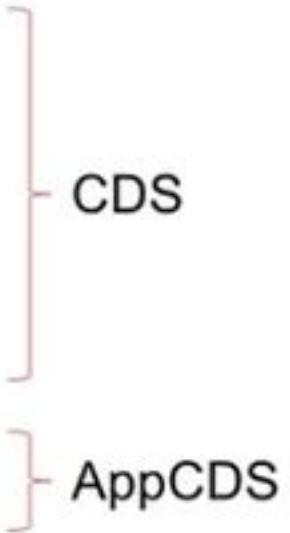
Overview:

- Feature allowing multiple JVMs to share a common read-only memory space for classes and methods.
- Improves startup time and reduces memory footprint.

How CDS Works:

- **Dynamic Class Loading:**
 - In Java, classes are dynamically loaded by the JVM as needed.
- **Class Data Sharing:**
 - Precomputes and stores internal representations of classes in a shared archive file.
 - Contains class metadata, bytecode, and internal representations.
 - Shared among multiple JVM instances.
- **Benefits:**
 - Reduces startup time.
 - Decreases memory usage.

The essence of CDS and AppCDS

- Different JVM instances on **the same** host often load **the same** JDK classes
 - It **consumes** CPU time and RAM space
 - Idea #1: store those classes into an archive and **share*** it between JVM instances
 - Idea #2: do the same with **application** classes and 3rd party **library** classes
- 
- CDS
- AppCDS

* *memory-mapping for read-only data*

Creating the Shared Archive:

- Use the ``-Xshare:dump`` option when running the Java application.
- Example:

```
java -Xshare:dump -cp <classpath> <MainClass>
```

Using the Shared Archive:

- After creating the shared archive, use the `'-Xshare:on` option when starting other Java applications.
- Example:

```
php
```

 Copy code

```
java -Xshare:on -cp <classpath> <MainClass>
```

Benefits of CDS:

1. Reduced Startup Time:

- Loading classes from the shared archive is faster than loading from individual class files.

2. Memory Footprint Reduction:

- Shared classes among JVM instances reduce overall memory consumption.

Example:

- To create the shared archive:

```
bash
```

 Copy code

```
java -Xshare:dump -cp . HelloWorld
```

- To use the shared archive (compare time with on/off):

```
bash
```

 Copy code

```
java -Xshare:off -cp . HelloWorld
```

```
java -Xshare:on -cp . HelloWorld
```

- Note: To remove the shared archive, delete `jdk/bin/server/classes.jsa`.

Conclusion:

- **Class Data Sharing (CDS):**
 - Improves Java application startup time.
 - Reduces memory usage by sharing class data among JVM instances.
- **Usage:**
 - Create shared archive with `'-Xshare:dump'`.
 - Use shared archive with `'-Xshare:on'`.
- **Considerations:**
 - Monitor and assess the impact on startup time and memory usage.
 - Removal of shared archive is possible by deleting the appropriate file.

HttpClient

HttpClient in Java

Introduction:

- Class in `java.net.http` package.
- Introduced in Java 11 for HTTP requests and responses.
- Replaces older `HttpURLConnection` API.

Key Features of HttpClient:

1. Immutability:

- Instances of `HttpClient` are immutable.
- Configuration cannot be changed after creation.

2. Builder Pattern:

- Follows the builder pattern for configuration.
- Use `HttpClient.Builder` for setting options.

3. Asynchronous API:

- Supports asynchronous programming using `CompletableFuture`.
- Enables non-blocking execution of HTTP requests.

4. Fluent API:

- Designed with a fluent API.
- Easy chaining of method calls for request configuration.

Comparison: HttpClient vs. WebClient

HttpClient:

- **Library:** Standard Java library (`java.net` package).
- **Programming Model:** Traditional blocking I/O.
- **Reactivity:** Not inherently reactive; provides synchronous API.
- **Integration:** Can be used in any Java application; may not integrate seamlessly with some reactive frameworks.
- **Flexibility:** Lower-level API for detailed control over HTTP request and response.

WebClient:

- **Library:** Part of the Spring Framework (`Spring WebFlux` module).
- **Programming Model:** Reactive and non-blocking.
- **Reactivity:** Built on Spring's reactive foundation; supports reactive programming.
- **Integration:** Designed to work well with Spring components; integrates smoothly with Spring's reactive stack.
- **Flexibility:** High-level and declarative API; built-in support for common use cases.

`HttpClient` Example:

```
import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;

public class HttpClientExample {
    public static void main(String[] args) {
        // Create an instance of HttpClient
        HttpClient httpClient = HttpClient.newHttpClient();

        // Define the request
        HttpRequest request = HttpRequest.newBuilder()
            .uri(URI.create("https://jsonplaceholder.typicode.com/posts/1"))
            .build();

        // Send the request and handle the response asynchronously
        httpClient.sendAsync(request, HttpResponse.BodyHandlers.ofString())
            .thenApply(HttpResponse::body)
            .thenAccept(System.out::println)
            .join(); // Ensure the program waits for the asynchronous operation to complete
    }
}
```

`'WebClient' Example:`

```
import org.springframework.web.reactive.function.client.WebClient;

public class WebClientExample {
    public static void main(String[] args) {
        // Create an instance of WebClient
        WebClient webClient = WebClient.create("https://jsonplaceholder.typicode.com"

        // Define the request and retrieve the response
        String responseBody = webClient.get()
            .uri("/posts/1")
            .retrieve()
            .bodyToMono(String.class)
            .block(); // Block to get the response synchronously

        // Print the response
        System.out.println(responseBody);
    }
}
```

Usage in Microservices:

- **WebClient:**
 - Commonly used in microservices architectures.
 - Well-suited for handling a large number of concurrent requests efficiently.
 - Ideal for projects within the Spring ecosystem.
- **HttpClient:**
 - Can be used in microservices, especially if not built around reactive principles.
 - Suitable for projects outside of the Spring framework.

Conclusion:

- **Choice Depends on Use Case:**
 - **WebClient:** Reactive programming, Spring ecosystem.
 - **HttpClient:** Traditional blocking I/O, standalone projects.
- **Considerations:**
 - **Flexibility:** HttpClient for detailed control; WebClient for simplicity.
 - **Integration:** Choose based on the existing framework and architecture.
- **In Summary:**
 - **WebClient:** Reactive, non-blocking, Spring.
 - **HttpClient:** Traditional, lower-level, standalone.

Java Language Enhancements

Improved Lambda Parameter Syntax (Java 11)

- **Enhancement:**
 - Use underscores (_) as identifiers for lambda parameters when types can be inferred.
- **Use Cases:**
 - Particularly useful in single-method interfaces.
 - Improves readability in cases where parameter types are clear from context.
- **Example:**

java

 Copy code

```
// Before Java 11
Function<Integer, String> func1 = (Integer x) -> String.valueOf(x);

// Java 11 and later
Function<Integer, String> func2 = _ -> String.valueOf(_);
```

Switch Expressions (Java 14)

- **Enhancement:**

- Switch statement enhanced to support switch expressions.
- Allows using the switch statement as an expression.

- **Benefits:**

- Expressive and concise syntax.
- Direct assignment of switch result to a variable.

- **Example:**

```
int day = 3;
String dayType = switch (day) {
    case 1, 2, 3, 4, 5 -> "Weekday";
    case 6, 7 -> "Weekend";
    default -> throw new IllegalArgumentException("Invalid day: " + day);
};
```

NumberFormatter (Java 12 and later)

- **Enhancement:**
 - Powerful and flexible formatting options for numbers.
 - Includes compact number formatting.
- **Example:**

```
NumberFormat numberFormatter = NumberFormat.getCompactNumberInstance(Locale.US, Style.SHORT);  
String formattedNumber = numberFormatter.format(10000);
```

String Enhancements (Java 11 and later)

1. `isBlank()` (Java 11):

- Checks if a string is empty or contains only white spaces.
- Example:

```
String str = "  ";
boolean isBlank = str.isBlank(); // true
```

2. `strip()`, `stripLeading()`, `stripTrailing()` (Java 11):

- Removes leading and trailing whitespaces from a string.
- Example:

```
String str = "  Hello, World!  ";
String stripped = str.strip(); // "Hello, World!"
```

3. `repeat(int count)` (Java 11):

- Repeats the string a specified number of times.
- Example:

```
String repeated = "abc".repeat(3); // "abcabcabc"
```

4. `lines()` (Java 11):

- Returns a stream of lines extracted from the string.
- Example:

```
String multilineString = "Line 1\nLine 2\nLine 3";
List<String> lines = multilineString.lines().collect(Collectors.toList());
```

Text Blocks

- **Introduction:**

- Provide a readable and natural way to declare multi-line strings.
- Preserve formatting without escape sequences or concatenation.

- **Syntax:**

- Triple double-quote (`"""`) syntax.

- **Example:**

```
String message = """  
    Hello, there!  
    How are you doing?  
    Have a great day!  
""";
```

- **Benefits:**

- Preserve formatting, including whitespaces, line breaks, and indentation.
- Improve readability for multi-line strings.

Conclusion

- **Java Language Enhancements:**
 - Improved Lambda Parameter Syntax.
 - Switch Expressions.
 - NumberFormatter for powerful number formatting.
 - String Enhancements for better string manipulation.
 - Text Blocks for improved readability.
- **Adopting Modern Java Features:**
 - Enhances code readability and maintainability.
 - Facilitates more expressive and concise code.
 - Choose features based on compatibility and project requirements.

Records in Java 14

Introduction:

- **Definition:**
 - New kind of class introduced in Java 14.
 - Concise way to declare classes for storing data.
- **Main Purpose:**
 - Simplified syntax for data storage.
 - Automatically generates common methods.

Introduction:

- **Definition:**
 - New kind of class introduced in Java 14.
 - Concise way to declare classes for storing data.
- **Main Purpose:**
 - Simplified syntax for data storage.
 - Automatically generates common methods.

Record Declaration Example:

```
public record Person(String name, int age) {  
    // Record body (optional)  
}
```

- **Fields:**
 - `name` of type `String`.
 - `age` of type `int`.
- **Generated Methods:**
 - `toString()`
 - `equals(Object)`
 - `hashCode()`
 - `name()`
 - `age()`

Generated Methods:

1. `toString()`:

- Returns a string representation of the record.
- Includes values of its fields.

2. `equals(Object)`:

- Compares the record with another object for equality.
- Based on the values of its fields.

3. `hashCode()`:

- Generates a hash code for the record.
- Based on the values of its fields.

4. `name()`:

- Returns the value of the `name` field.

5. `age()`:

- Returns the value of the `age` field.

Immutable by Default:

- Records are immutable by default.
- Fields cannot be modified once the record is created.
- Example:

```
Person person = new Person("John", 25);
// person.age = 26; // Compilation error
```

Customization of Records:

- Customize behavior by adding methods and constructors in the record body.
- Example:

```
public record Person(String name, int age) {  
    public Person {  
        if (age < 0) {  
            throw new IllegalArgumentException("Age must be non-negative");  
        }  
    }  
}
```

Advantages of Records:

1. Conciseness:

- Provide a more concise syntax for simple data-holding classes.

2. Reduced Boilerplate:

- Automatically generate common methods, reducing boilerplate code.

3. Functional Programming Style:

- Promote a more functional programming style.

4. Use Cases:

- Ideal for classes solely for storing and transporting data.

Conclusion:

- **Records:**
 - Introduced in Java 14 for concise data storage.
 - Automatic generation of common methods.
 - Immutable by default; customization is possible.
 - Promote a functional programming style.
- **Usage:**
 - Best suited for simple data-holding classes.
 - Reduces boilerplate code and enhances readability.

Sealed Classes (JEP 360)

Introduction:

- **Feature:**
 - Enhances Java language to allow "sealed" classes and interfaces.
 - Specifies which classes or interfaces can be subclasses or implementers.
- **Purpose:**
 - Control the inheritance hierarchy.
 - Explicitly specify permitted subclasses.

Example: Sealed Classes

```
public sealed class Shape permits Circle, Rectangle {  
    // Shape class definition  
}  
  
public final class Circle extends Shape {  
    // Circle class definition  
}  
  
public final class Rectangle extends Shape {  
    // Rectangle class definition  
}
```

- **Explanation:**
 - `Shape` is declared as sealed.
 - `Circle` and `Rectangle` explicitly permitted using `permits` keyword.
 - `Circle` and `Rectangle` are declared as `final`.
- **Benefits:**
 - Control over the design and usage of subclasses.
 - Explicitly define the permissible subclasses.

Introduction:

- **Introduced in Java 11:**
 - Part of nestmates concept.
 - Provide access to private members without reflection.
- **Hidden Classes:**
 - A specific type of nestmate.
 - Enables access to private members efficiently and securely.

Example: Java Hidden Classes

```
public class Outer {  
    private void privateMethod() {  
        // Private method implementation  
    }  
  
    public static class HiddenClass {  
        private void hiddenMethod() {  
            // Access private method of Outer  
            Outer outer = new Outer();  
            outer.privateMethod();  
        }  
  
        public void invokePrivateMethod(Outer outer) {  
            // Invoke hidden method  
            hiddenMethod();  
        }  
    }  
  
    public static void main(String[] args) {  
        // Example usage  
        Outer outer = new Outer();  
        HiddenClass hiddenClass = new HiddenClass();  
        hiddenClass.invokePrivateMethod(outer);  
    }  
}
```

- **Usage:**
 - `HiddenClass` is a nestmate of `Outer`.
 - `HiddenClass` can access private method `privateMethod()` of `Outer`.
 - No need for reflection.
- **Advantages:**
 - Efficient and secure access to private members.
 - Reduces the need for reflection.

Considerations:

- **Advanced Feature:**
 - Usage of hidden classes is relatively advanced.
 - May not be necessary in many typical Java applications.
- **Efficiency and Security:**
 - Hidden classes provide a more efficient and secure way to access private members.
- **Use Cases:**
 - Situations where access to private members without reflection is crucial.

Conclusion:

- **Sealed Classes:**
 - Control over inheritance hierarchy.
 - Explicit specification of permissible subclasses.
- **Hidden Classes:**
 - Introduced in Java 11 as nestmates.
 - Efficient and secure access to private members.
 - Advanced feature; consider use cases carefully.

String Templates (JEP 430)

Introduction:

- **Feature:**

- Promoted from Proposed to Targeted status for JDK 21.
- Enhances Java language with string templates.
- Similar to string literals with embedded expressions.

- **Purpose:**

- Simplify writing Java programs.
- Improve readability of text and expressions.
- Enhance security when composing strings.

Example Usage:

```
java
```

 Copy code

```
// Run from command-line  
java --enable-preview --source 21 TestMain.java
```

```
// Note: May not work from STS, shows error
```

- **Execution:**
 - Enable preview features.
 - Specify source version as 21.
- **Note:**
 - Not supported in STS.

Sequenced Collections (JEP 431)

Introduction:

- **Added Since Java 21:**
 - New feature for existing Collection classes/interfaces.
 - Allows access to the first and last elements.
 - Provides a reversed view of the collection.
- **New Interfaces:**
 - `SequencedCollection`
 - `SequencedSet`
 - `SequencedMap`

Motivation:

- **Demand:**
 - Long-pending demand for simple methods.
 - Fetching the first and last elements.
- **Cleaner Code:**
 - Simplifies code compared to pre-Java 21.

Example: Fetching Elements

java

 Copy code

```
// Before Java 21
var firstItem = arrayList.iterator().next();
var lastItem = arrayList.get(arrayList.size() - 1);

// With Sequenced Collections
var firstItem = arrayList.getFirst();
var lastItem = arrayList.getLast();
```

- **Cleaner Code:**

- Methods like `getFirst()` and `getLast()`.

SequencedCollection Interface:

java

 Copy code

```
interface SequencedCollection<E> extends Collection<E> {

    // New Method
    SequencedCollection<E> reversed();

    // Promoted methods from Deque<E>
    void addFirst(E);
    void addLast(E);

    E getFirst();
    E getLast();

    E removeFirst();
    E removeLast();
}
```

- Usage Example:
 - Creation of ArrayList and sequenced operations.

Platform Thread vs. Virtual Thread

Platform Thread:

- **Implementation:**
 - Thin wrapper around an OS thread.
 - Runs Java code on its underlying OS thread.
- **Resources:**
 - Large thread stack.
 - Maintained by the operating system.
 - Limited resource.

Virtual Thread:

- **Implementation:**
 - Instance of `java.lang.Thread`.
 - Not tied to a specific OS thread.
- **I/O Operations:**
 - Suspends on blocking I/O.
 - OS thread becomes available for other tasks.

Virtual Thread Example:

java

 Copy code

```
Thread.Builder builder = Thread.ofVirtual().name("MyThread");
Runnable task = () -> {
    System.out.println("Running thread");
};
Thread t = builder.start(task);
System.out.println("Thread t name: " + t.getName());
t.join();
```

- Creation:
 - Creation of a virtual thread with a name.
 - Running a simple task.

Virtual Thread Example (contd.):

```
java Copy code  
  
Thread.Builder builder = Thread.ofVirtual().name("worker-", 0);  
Runnable task = () -> {  
    System.out.println("Thread ID: " + Thread.currentThread().threadId());  
};  
  
Thread t1 = builder.start(task);  
t1.join();  
System.out.println(t1.getName() + " terminated");  
  
Thread t2 = builder.start(task);  
t2.join();  
System.out.println(t2.getName() + " terminated");
```

- **Multiple Virtual Threads:**
 - Creation and execution of multiple virtual threads.

Virtual Thread with Executors.newVirtualThreadPerTaskExecutor():

```
try (ExecutorService myExecutor = Executors.newVirtualThreadPerTaskExecutor()) {  
    Future<?> future = myExecutor.submit(() -> System.out.println("Running thread"));  
    future.get();  
    System.out.println("Task completed");  
    // ...  
}
```

- **Executor Usage:**
 - Creation and usage of virtual threads with an executor.
 - Task completion check.

Conclusion:

- **String Templates (JEP 430):**
 - Enhances Java language with string templates.
 - Improves program writing, readability, and security.
- **Sequenced Collections (JEP 431):**
 - Adds new interfaces for ordered collections.
 - Fetching first and last elements with cleaner code.
- **Platform vs. Virtual Threads:**
 - Platform threads tied to OS threads.
 - Virtual threads provide flexibility for I/O operations.
- **Usage Considerations:**
 - Adopt based on specific use cases and compatibility.

JDK 22
|_ JEP 464: Scoped Values (Second Preview)

Scoped values are a new way to share data between threads without using method parameters.

They also use less memory compared to Thread local variables.

Scoped values allow you to share data between threads in a memory-efficient way. They are memory efficient because the data inside a scoped value is immutable. This means that many threads can use the data without every thread needing a copy of the data. This is especially useful in combination with virtual threads. When running lots of virtual threads you will only need a single copy of the data for any number of virtual threads.

Types Of References In Java

Depending upon how objects are garbage collected, references to those objects in java are grouped into 4 types.

They are:

- 1) Strong References
- 2) Soft References
- 3) Weak References
- 4) Phantom References

Strong References

Any object in the memory which has active **strong reference** is not eligible for garbage collection.

```
A a = new A();
```

Soft References

The objects which are softly referenced will not be garbage collected (even though they are available for garbage collection) until JVM badly needs memory.

These objects will be cleared from the memory only if JVM runs out of memory. You can create a soft reference to an existing object by using **java.lang.ref.SoftReference** class.

```
A a = new A(); //Strong Reference
```

```
//Creating Soft Reference to A-type object to which 'a' is also pointing
```

```
SoftReference<A> softA = new SoftReference<A>(a);
```

a = null; //Now, A-type object to which 'a' is pointing earlier is eligible for garbage collection. But, it will be garbage collected only when JVM needs memory.

```
a = softA.get(); //You can retrieve back the object which has been softly referenced
```

SOFT REFERENCES

Soft references are good for providing caches of objects that can be garbage collected when the Java Virtual Machine (JVM) is running low on memory.

In particular, the JVM guarantees a couple of useful things:

- It will not garbage collect an object as long as there are normal, strong references to it.
- It will not throw an `OutOfMemoryError` until it has "cleaned up" all softly reachable objects (objects not reachable by strong references but reachable through one or more soft references).
- This allows us to use soft references to refer to objects that we could afford to have garbage collected, but that are convenient to have around until memory becomes tight.

In Java, a soft reference is represented by the `java.lang.ref.SoftReference` class.

We have two options to initialize it.

The first way is to pass a referent only:

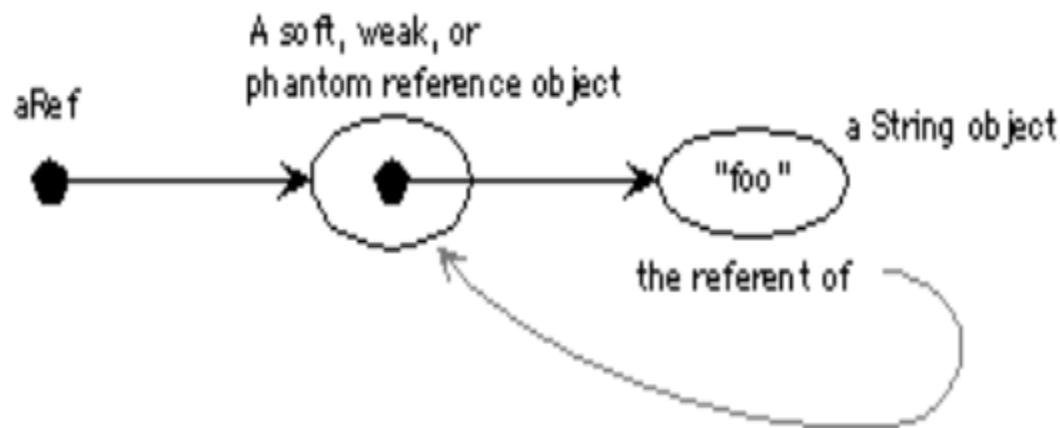
```
StringBuilder builder = new StringBuilder();
SoftReference<StringBuilder> reference1 = new SoftReference<>(builder);
```

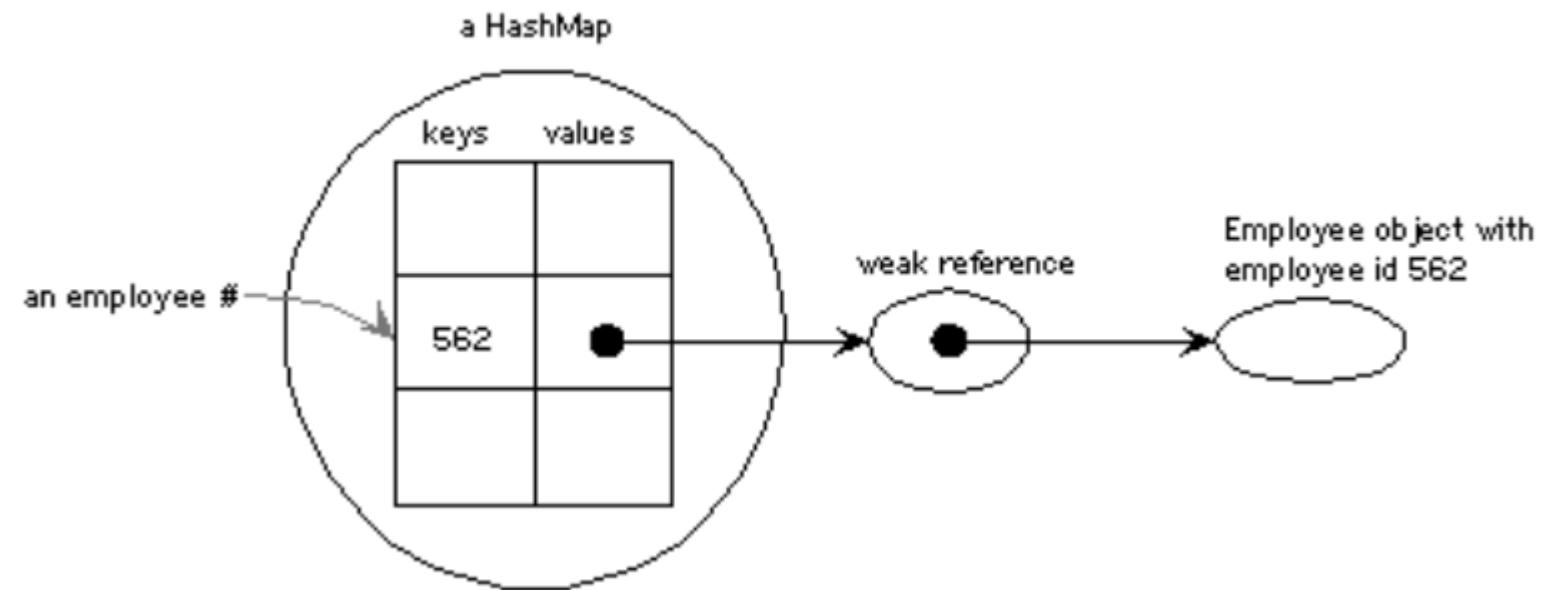
The second option implies passing a reference to a `java.lang.ref.ReferenceQueue` as well as a reference to a referent. Reference queues are designed for making us aware of actions performed by the Garbage Collector. It appends a reference object to a reference queue as it decides to remove the referent of this reference.

Here's how to initialize a `SoftReference` with a `ReferenceQueue`:

```
ReferenceQueue<StringBuilder> referenceQueue = new ReferenceQueue<>();
SoftReference<StringBuilder> reference2
= new SoftReference<>(builder, referenceQueue);
```

```
Reference aRef = new SoftReference( new String( 'foo' ) );
```





org.apache.commons.pool2.impl

Class SoftReferenceObjectPool<T>

java.lang.Object

 org.apache.commons.pool2.BaseObject

 org.apache.commons.pool2.BaseObjectPool<T>

 org.apache.commons.pool2.impl.SoftReferenceObjectPool<T>

Hibernate cache example:

<https://github.com/manuelbernhardt/hibernate-core/blob/master/hibernate-core/src/main/java/org/hibernate/util/SoftLimitMRUCache.java>

Though, no guarantees are placed upon the time when a soft reference gets cleared or the order in which a set of such references to different objects get cleared.

As a rule, JVM implementations choose between cleaning of either recently-created or recently-used references.

Softly reachable objects will remain alive for some time after the last time they are referenced. The default value is a one second of lifetime per free megabyte in the heap. This value can be adjusted using the `-XX:SoftRefLRUPolicyMSPerMB` flag.

For example, to change the value to 2.5 seconds (2500 milliseconds), we can use:

`-XX:SoftRefLRUPolicyMSPerMB=2500`

Weak References

JVM ignores the **weak references**. That means objects which has only week references are eligible for garbage collection. They are likely to be garbage collected when JVM runs garbage collector thread. JVM doesn't show any regard for weak references.

```
A a = new A(); //Strong Reference
```

```
//Creating Weak Reference to A-type object to which 'a' is also pointing.
```

```
WeakReference<A> weakA = new WeakReference<A>(a);
```

```
a = null; //Now, A-type object to which 'a' is pointing earlier is available  
for garbage collection.
```

```
a = weakA.get(); //You can retrieve back the object which has been  
weakly referenced.
```

Phantom References

The objects which are being referenced by **phantom references** are eligible for garbage collection. But, before removing them from the memory, JVM puts them in a queue called '**reference queue**' . You can't retrieve back the objects which are being phantom referenced.

```
A a = new A();    //Strong Reference  
                  //Creating ReferenceQueue
```

```
ReferenceQueue<A> refQueue = new ReferenceQueue<A>();
```

```
//Creating Phantom Reference to A-type object to which 'a' is also pointing  
PhantomReference<A> phantomA = new PhantomReference<A>(a, refQueue);
```

a = null; //Now, A-type object to which 'a' is pointing earlier is available for garbage collection. But, this object is kept in 'refQueue' before removing it from the memory.

```
a = phantomA.get(); //it always returns null
```

```
refQueue.remove();      // This will block till it is GCd
```

Soft vs Weak vs Phantom References				
Type	Purpose	Use	When GCed	Implementing Class
Strong Reference	An ordinary reference. Keeps objects alive as long as they are referenced.	normal reference.	Any object not pointed to can be reclaimed.	default
Soft Reference	Keeps objects alive provided there's enough memory.	to keep objects alive even after clients have removed their references (memory-sensitive caches), in case clients start asking for them again by key.	After a first gc pass, the JVM decides it still needs to reclaim more space.	java.lang.ref.SoftReference
Weak Reference	Keeps objects alive only while they're in use (reachable) by clients.	Containers that automatically delete objects no longer in use.	After gc determines the object is only weakly reachable	java.lang.ref.WeakReference java.util.WeakHashMap
Phantom Reference	Lets you clean up after finalization but before the space is reclaimed (replaces or augments the use of <code>finalize()</code>)	Special clean up processing		

Phantom references are safe way to know an object has been removed from memory. For instance, consider an application that deals with large images. Suppose that we want to load a big image in to memory when large image is already in memory which is ready for garbage collected.

In such case, we want to wait until the old image is collected before loading a new one. Here, the phantom reference is flexible and safely option to choose.

The reference of the old image will be enqueued in the ReferenceQueue once the old image object is finalized. After receiving that reference, we can load the new image in to memory.

Phantom reference are the weakest level of reference in Java; in order from strongest to weakest, they are: strong, soft, weak, phantom.

Java Lock vs synchronized

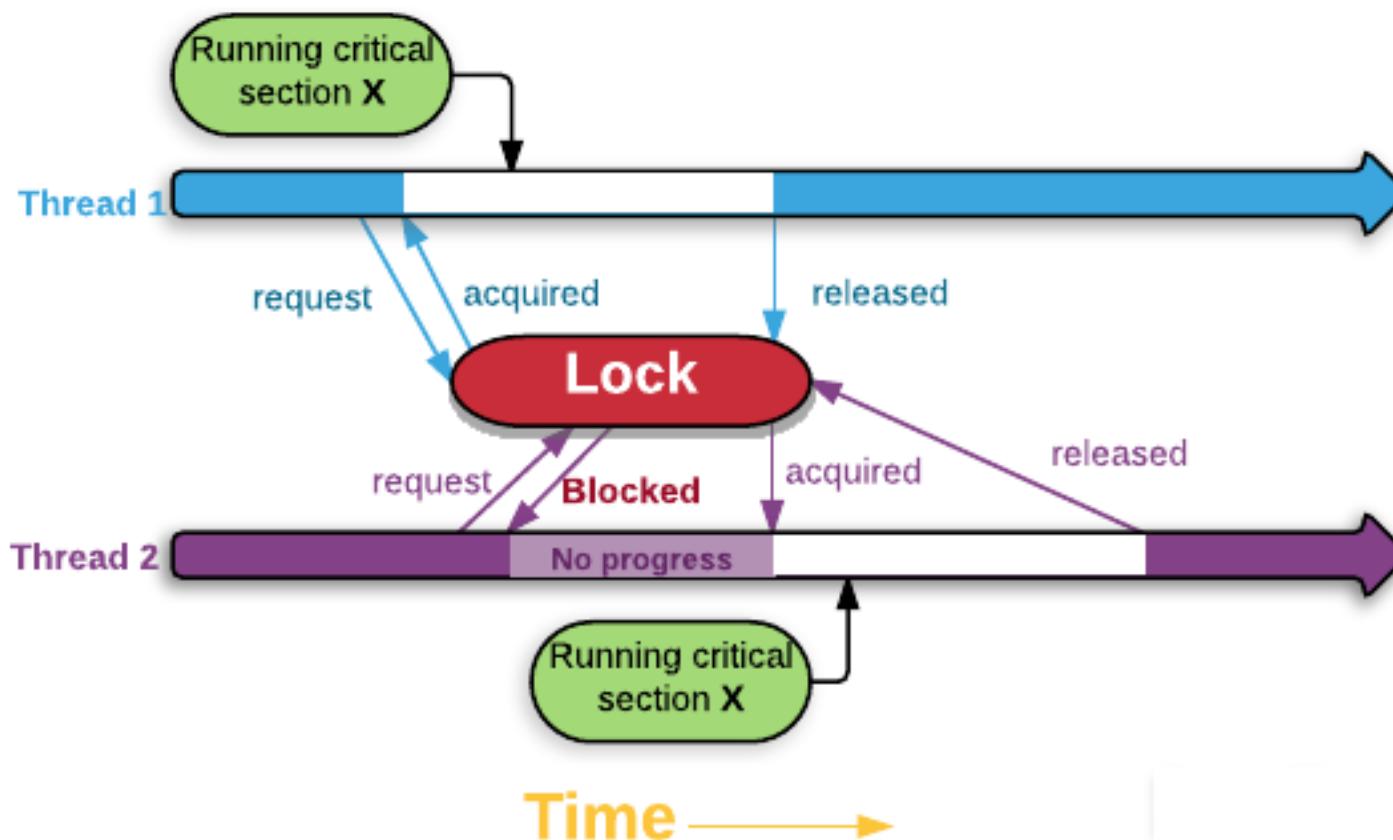
Synchronization in Java

Synchronization in java is the capability to control the access of multiple threads to any shared resource.

Java Synchronization is better option where we want to allow only one thread to access the shared resource.

In the Java virtual machine, every object and class is logically associated with a monitor. To implement the mutual exclusion capability of monitors, a lock is associated with each object and class.

Mutual Exclusion of Critical Section



Why use Synchronization?

The synchronization is mainly used to

- To prevent thread interference.
- To prevent consistency problem.

Ex : The classic example : Joint account holders are trying to withdraw the amount at the same time [money transfer Online , withdraw cash at ATM]

There are two kinds of locks

Those with synchronized blocks, and those which use `java.util.concurrent.Lock`.

Java Lock vs synchronized

Java Lock API provides more visibility and options for locking, unlike synchronized where a thread might end up waiting indefinitely for the lock, we can use tryLock() to make sure thread waits for specific time only.

Synchronization code is much cleaner and easy to maintain whereas with Lock we are forced to have try-finally block to make sure Lock is released even if some exception is thrown between lock() and unlock() method calls.

synchronization blocks or methods can cover only one method whereas we can acquire the lock in one method and release it in another method with Lock API.

synchronized keyword doesn't provide fairness whereas we can set fairness to true while creating ReentrantLock object so that longest waiting thread gets the lock first. We can create different conditions for Lock and different thread can await() for different conditions.

ReentrantLock

The class ReentrantLock is a mutual exclusion lock with the same basic behavior as the implicit monitors accessed via the synchronized keyword but with extended capabilities (like the longest waiting thread gets the lock first)

```
ReentrantLock lock = new ReentrantLock();
int count = 0;

void increment() {
    lock.lock();
    try {
        count++;
    } finally {
        lock.unlock();
    }
}
```

ReadWriteLock

The interface `ReadWriteLock` specifies another type of lock maintaining a pair of locks for read and write access.

The idea behind read-write locks is that it's usually safe to read mutable variables concurrently as long as nobody is writing to this variable. So the read-lock can be held simultaneously by multiple threads as long as no threads hold the write-lock.

This can improve performance and throughput in case that reads are more frequent than writes.

In the below code, both read tasks are executed in parallel and print the result simultaneously to the console

```
Runnable readTask = () -> {
    lock.readLock().lock();
    try {
        System.out.println(map.get("foo"));
        sleep(1);
    } finally {
        lock.readLock().unlock();
    }
};

executor.submit(readTask);
executor.submit(readTask);
```

The below example first acquires a write-lock in order to put a new value to the map after sleeping for one second.

```
ExecutorService executor = Executors.newFixedThreadPool(2);
Map<String, String> map = new HashMap<>();
ReadWriteLock lock = new ReentrantReadWriteLock();

executor.submit(() -> {
    lock.writeLock().lock();
    try {
        sleep(1);
        map.put("foo", "bar");
    } finally {
        lock.writeLock().unlock();
    }
});
```

Note: read tasks have to wait the whole second until the write task has finished. After the write lock has been released both read tasks are executed in parallel

ReadWriteLock Locking Rules

The rules by which a thread is allowed to lock the ReadWriteLock either for reading or writing the guarded resource, are as follows:

Read Lock

If no threads have locked the ReadWriteLock for writing, and no thread have requested a write lock (but not yet obtained it). Thus, multiple threads can the lock for reading.

Write Lock

If no threads are reading or writing. Thus, only one thread at a time can lock the lock for writing.

StampedLock

StampedLock which also support read and write locks.

In contrast to ReadWriteLock the locking methods of a StampedLock return a stamp represented by a long value.

We can use these stamps to either release a lock or to check if the lock is still valid.

Additionally stamped locks support another lock mode called optimistic locking.

```
ExecutorService executor = Executors.newFixedThreadPool(2);
Map<String, String> map = new HashMap<>();
StampedLock lock = new StampedLock();

executor.submit(() -> {
    long stamp = lock.writeLock();
    try {
        sleep(1);
        map.put("foo", "bar");
    } finally {
        lock.unlockWrite(stamp);    } });

Runnable readTask = () -> {
    long stamp = lock.readLock();
    try {
        System.out.println(map.get("foo"));
        sleep(1);
    } finally {
        lock.unlockRead(stamp);    } };

executor.submit(readTask);
executor.submit(readTask);
```

Optimistic Locking:

An optimistic read lock is acquired by calling `tryOptimisticRead()` which always returns a stamp without blocking the current thread, no matter if the lock is actually available.

If there's already a write lock active the returned stamp equals zero. You can always check if a stamp is valid by calling `lock.validate(stamp)`.

```
ExecutorService executor = Executors.newFixedThreadPool(2);
StampedLock lock = new StampedLock();

executor.submit(() -> {
    long stamp = lock.tryOptimisticRead();
    try {
        System.out.println("Optimistic Lock Valid: " + lock.validate(stamp));
        sleep(1);
        System.out.println("Optimistic Lock Valid: " + lock.validate(stamp));
        sleep(2);
        System.out.println("Optimistic Lock Valid: " + lock.validate(stamp));
    } finally {
        lock.unlock(stamp);    } });

executor.submit(() -> {
    long stamp = lock.writeLock();
    try {
        System.out.println("Write Lock acquired");
        sleep(2);
    } finally {
        lock.unlock(stamp);
        System.out.println("Write done");    }});
```

Semaphores

Concurrency API also supports counting semaphores. Whereas locks usually grant exclusive access to variables or resources, a semaphore is capable of maintaining whole sets of permits.

This is useful in different scenarios where you have to limit the amount concurrent access to certain parts of your application.

Semaphores – Restrict the number of threads that can access a resource. Example, limit max 10 connections to access a file simultaneously.

Mutex – Only one thread to access a resource at once. Example, when a client is accessing a file, no one else should have access the same file at the same time.

```
ExecutorService executor = Executors.newFixedThreadPool(10);
```

```
Semaphore semaphore = new Semaphore(5);
```

```
Runnable longRunningTask = () -> {
    boolean permit = false;
    try {
        permit = semaphore.tryAcquire(1, TimeUnit. MILLISECONDS);
        if (permit) {
            System.out.println("Semaphore acquired");
            sleep(5);
        } else {
            System.out.println("Could not acquire semaphore");
        }
    } catch (InterruptedException e) {
        throw new IllegalStateException(e);
    } finally {
        if (permit) {
            semaphore.release();
        }
    }
}
```

```
IntStream.range(0, 10)
    .forEach(i -> executor.submit(longRunningTask));
```

CPU | Threads | Deadlocks | Memory | Monitor Usage | Exceptions | Performance Charts | Events

Group by: C Monitor class then group by: G Waiting/blocked thread Show blocked threads only

Name
Monitor of class demo.Table
was waited by thread Thread-2 native ID: 0x2B8 group: 'main'
that was blocked by thread Thread-1 native ID: 0x2BC group: 'main'

Reverse Call Tree

demo.Table.printTable(int) TestSynchronizedBlock1.java:7
demo.MyThread2.run() TestSynchronizedBlock1.java:33

Reducing Locks

Try to use the concurrent.locks package which provide extended capabilities compared to synchronized regions or method calls for timed lock acquisition, fairness among threads etc.

Avoid synchronized methods. Go with smaller synchronized regions whenever possible. Try to use synchronizing on a specific object.

Increase the number of resources, where access to them leads to locking.

Reducing Locks (contd)

Try to cache resources if each call to create the resource requires synchronized calls.

Try to avoid the synchronized call entirely if possible by changing the logic of execution.

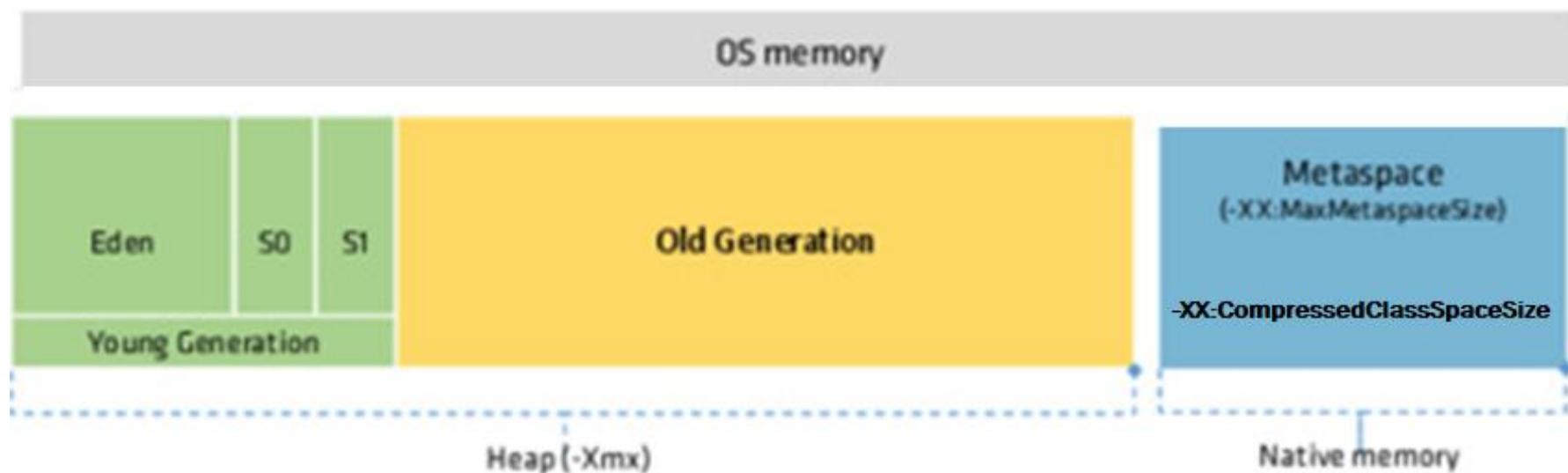
Try to control the order of locking in cases of deadlocks. For example, every thread has to obtain LockA before obtaining LockB and not mix up the order of obtaining locks.

If the owner of the lock has to wait for an event, then do a synchronization wait on the lock which would release the lock and put the owner itself on the blocked list for the lock automatically so other threads can obtain the lock and proceed.

The G1 Garbage Collector

JVM memory

JDK8



GC for the Old Generation

The old generation basically performs a GC when the data is full. The execution procedure varies by the GC type.

According to JDK 7, there are 5 GC types.

Serial GC

Parallel GC

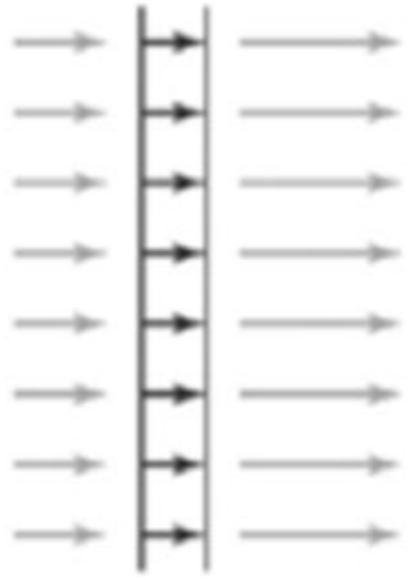
Parallel Old GC (Parallel Compacting GC)

Concurrent Mark & Sweep GC (or "CMS")

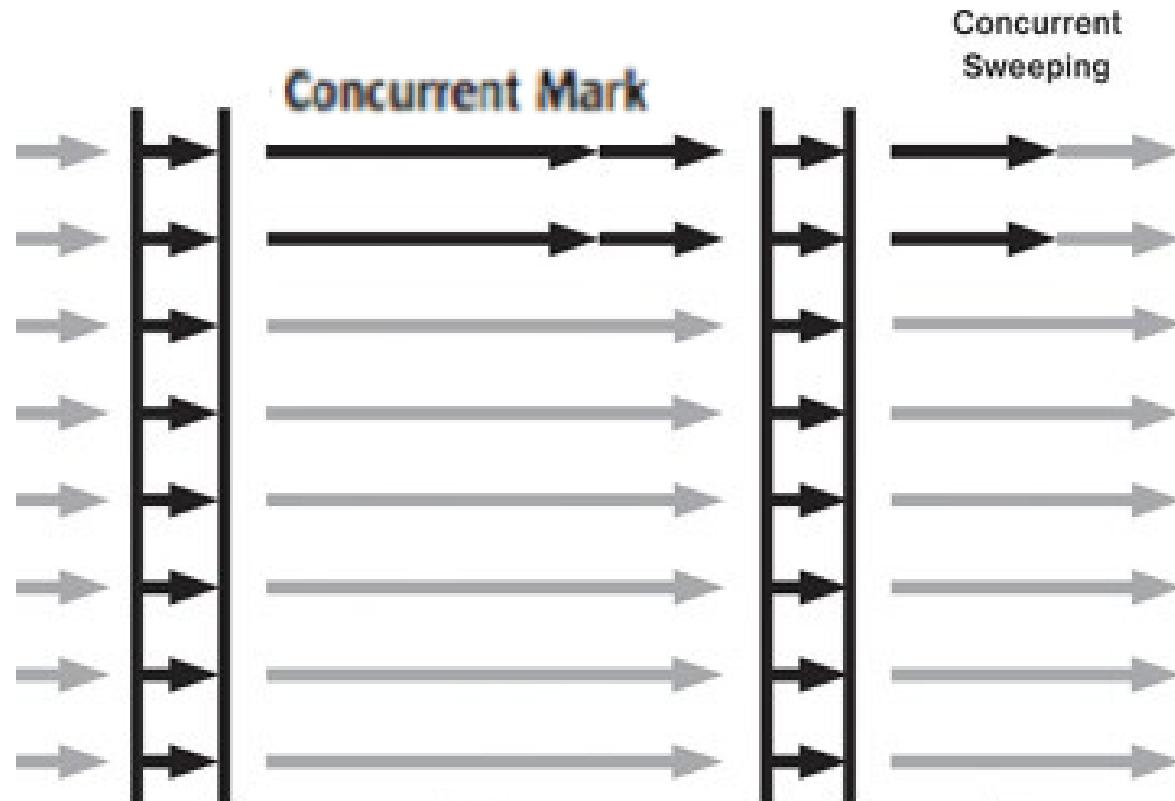
Garbage First (G1) GC

Characteristics of GC's

- ✓ Concurrent
- ✓ Parallel
- ✓ Compacting



Young Generation



Tenured Generation

The G1 Garbage Collector

The Garbage-First (G1) collector is a server-style garbage collector, targeted for multi-processor machines with large memories.

It meets garbage collection (GC) pause time goals(defined through -XX:MaxGCPauseMillis) with a high probability, while achieving high throughput.

G1 works to accomplish those goals in a few different ways.

First, being true to its name, G1 collects regions with the least amount of live data (Garbage First!) and compacts/evacuates live data into new regions.

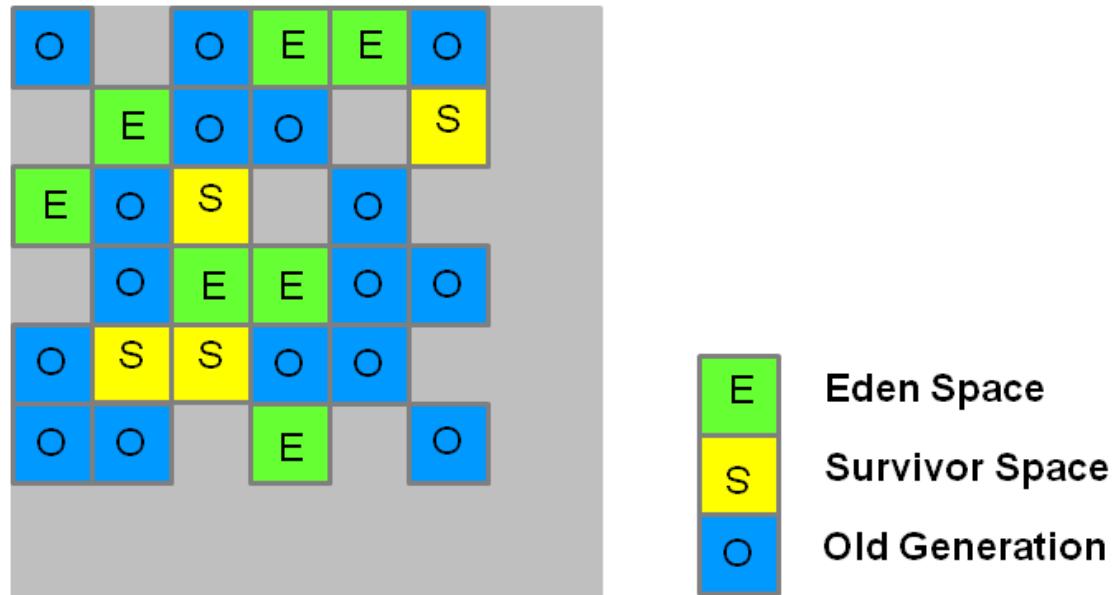
Secondly, it uses a series of incremental, parallel and multi-phased cycles to achieve its soft pause time target.

Regions

A region represents a block of allocated space that can hold objects of any generation without the need to maintain contiguity with other regions of the same generation.

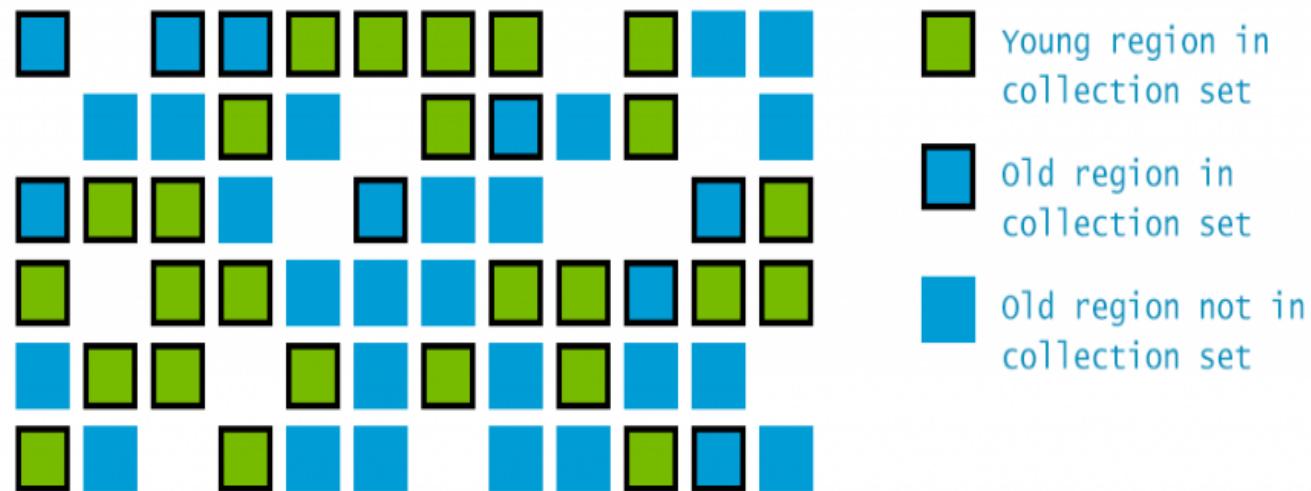
The heap is one memory area split into many fixed sized regions.

G1 Heap Allocation



Collection set

To avoid collecting the entire heap at once, only a subset of the regions, called the collection set will be considered at a time.



Collection Set (continued)

During the concurrent phase it estimates the amount of live data that each region contains. This is used in building the collection set: the regions that contain the most garbage are collected first.

Hence the name: *garbage-first* collection.

Heap Memory allocation:

Allocation is done on a per-thread basis.

Thread Local Area (TLA), is a dedicated partition that a thread allocates freely within, without having to claim a full heap lock. Once the area is full, the thread is assigned a new area until the heap runs out of areas to dedicate.

Note : TLAs are part of the heap. Thread Stacks are not on the heap.

String Deduplication

String Deduplication:

String class keeps string value in char[] array.

The char array is not accessible and modified from outside, this means that char array can be used safely by multiple instances of String at the same time.

Deduplication may happen during minor GC.

When GC detects another String object with the same hash code, it compares two strings char by char. When they match, one String's char array will be re-assigned to the char array of the second string and the unreferenced char array of the first string becomes available for GC.

Enable String Deduplication:

`-XX:+UseG1GC -XX:+UseStringDeduplication`

Important Points:

- This option is only available from Java 8 Update 20 JDK release.
- This feature will only work along with the G1 garbage collector.
- You need to provide both `-XX:+UseG1GC` and `-XX:+StringDeduplication` JVM options to enable this feature.
- To check if it happens in your system you can use ---
`XX:+PrintStringDeduplicationStatistics` parameter.
- You can control this by using `-XX:StringDeduplicationAgeThreshold=3` option to change when Strings become eligible for deduplication.

[GC concurrent-string-deduplication, 2893.3K->2672.0B(2890.7K), avg 97.3%, 0.0175148 secs]

[Last Exec: 0.0175148 secs, Idle: 3.2029081 secs, Blocked: 0/0.0000000 secs]

[Inspected: 96613]

[Skipped: 0(0.0%)]

[Hashed: 96598(100.0%)]

[Known: 2(0.0%)]

[New: 96611(100.0%) 2893.3K]

[Deduplicated: 96536(99.9%) 2890.7K(99.9%)]

[Young: 0(0.0%) 0.0B(0.0%)]

[Old: 96536(100.0%) 2890.7K(100.0%)]

[Total Exec: 452/7.6109490 secs, Idle: 452/776.3032184 secs, Blocked: 11/0.0258406 secs]

[Inspected: 27108398]

[Skipped: 0(0.0%)]

[Hashed: 26828486(99.0%)]

[Known: 19025(0.1%)]

[New: 27089373(99.9%) 823.9M]

[Deduplicated: 26853964(99.1%) 801.6M(97.3%)]

[Young: 4732(0.0%) 171.3K(0.0%)]

[Old: 26849232(100.0%) 801.4M(100.0%)]

[Table]

[Memory Usage: 2834.7K]

[Size: 65536, Min: 1024, Max: 16777216]

[Entries: 98687, Load: 150.6%, Cached: 415, Added: 252375, Removed: 153688]

[Resize Count: 6, Shrink Threshold: 43690(66.7%), Grow Threshold: 131072(200.0%)]

[Rehash Count: 0, Rehash Threshold: 120, Hash Seed: 0x0]

[Age Threshold: 3]

[Queue]

[Dropped: 0]

These are the results after running the app for 10 minutes. As we can see String Deduplication was executed 452 times and "deduplicated" 801.6 MB Strings. String Deduplication inspected 27 000 000 Strings.

When we compare memory consumption from Java 7 with the standard Parallel GC to Java 8u20 with the G1 GC and enabled String Deduplication the heap dropped approximatley 50%:

Basic OO Principles

- ✓ **Abstraction**
- ✓ **Encapsulation**
- ✓ **Inheritance**
- ✓ **Composition**

Abstraction

Abstraction is the process of hiding the implementation details of an object so that it can be used without understanding how it works. This allows you to create code that is easy to use and maintain.

For example, we could have a class called Vehicle with methods such as drive() and stop(). The details of how these methods work are hidden from the user, so they can simply call the methods and trust that they will work as expected.

Abstraction is important because it allows you to create code that is easy to use and understand. Abstraction allows the user to use the code without needing to know the details of how it works.

Encapsulation

Encapsulation is the process of hiding information within an object so that it cannot be accessed directly from outside the object. This allows you to control how data is used and prevents accidental modification of data.

Ex1: We could have a class called Person with attributes such as name and age. We could then create methods to get and set these attributes. This would allow you to control how the data is used, and you could add validation to ensure that the data is valid before it is set.

Ex2: The module descriptor (`module-info.java`)

Encapsulation is important because it helps to keep the data safe and secure. It also allows us to change the implementation of the code without affecting the rest of your codebase.

```
public abstract class Vehicle {  
    private String fuelType; // Encapsulated attribute  
  
    public Vehicle(String fuelType) {  
        this.fuelType = fuelType;  
    }  
  
    // Abstract method representing the abstraction of driving  
    public abstract void drive();  
  
    // Concrete method representing a common behavior  
    public void stop() { System.out.println("Vehicle stopped"); }  
  
    // Getter method providing controlled access to the encapsulated attribute  
    public String getFuelType() { return fuelType; }  
  
    // Setter method allowing controlled modification of the encapsulated attribute  
    public void setFuelType(String fuelType) { this.fuelType = fuelType; }  
}
```

Encapsulation:

- The `fuelType` attribute is encapsulated within the `Vehicle` class. It is marked as `private`, restricting direct access from outside the class.
- The constructor `public Vehicle(String fuelType)` is responsible for initializing the encapsulated attribute during object creation.
- Getter (`getFuelType`) and setter (`setFuelType`) methods are provided to allow controlled access and modification of the encapsulated attribute. This encapsulation helps in protecting the internal state of the class.

Abstraction:

- The `Vehicle` class contains an abstract method `public abstract void drive()`. This method represents the abstraction of the concept of driving.
- By declaring the `drive` method as abstract, the `Vehicle` class defines a contract that all concrete subclasses must adhere to. Every subclass must provide its own implementation of the `drive` method.
- The `stop` method provides a concrete (non-abstract) method representing a common behavior shared by all vehicles. This is another form of abstraction, where common functionalities are implemented in the abstract class.

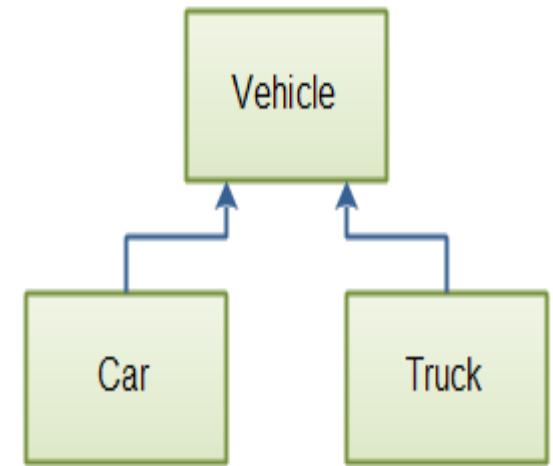
Inheritance

Inheritance is the ability of one class to inherit the attributes and methods of another class. This is useful because it allows you to create subclasses that are specialized versions of a parent class.

For example, we could have a parent class called Vehicle, with subclasses Car and Truck.

The Vehicle class would contain general attributes and methods that are common to all vehicles, such as the number of wheels and the color.

The Car and Truck subclasses would then each have their own unique attributes and methods, such as the number of doors and the size of the engine.

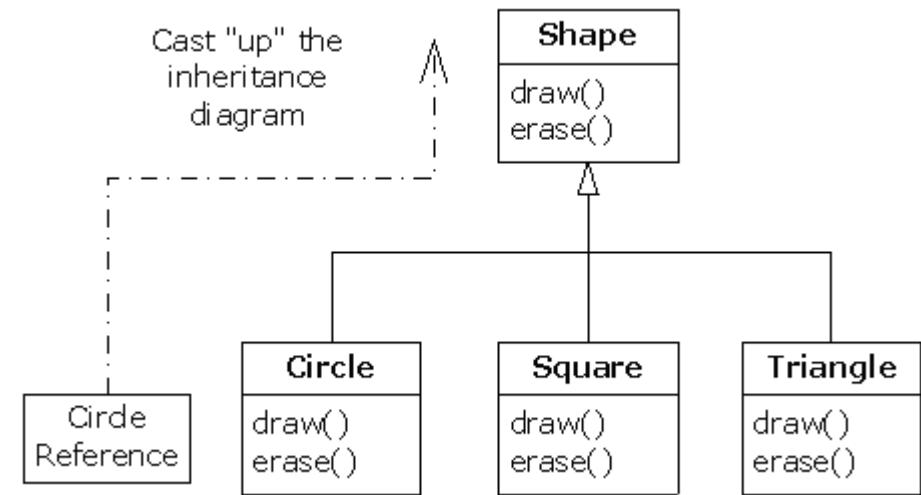


Inheritance is important in Object-Oriented Programming (OOP) because it allows for code reuse.

This means that we can write code once and then use it in multiple places, which makes your code more efficient and maintainable.

Polymorphism

Polymorphism is the ability of an object to take on multiple forms. This is useful because it allows you to create code that is more flexible and adaptable. For example, we could have a class called Shape with subclasses Circle and Rectangle. The Shape class would contain general methods such as getArea and getPerimeter. The Circle and Rectangle subclasses would then each have their own unique implementation of these methods.



Polymorphism is important because it allows you to write code that is more flexible and adaptable. Polymorphism allows you to write code that can be used with multiple types of objects.

Association in Java

Association in java, let us briefly explore the types of object relationships that can exist in OOPs. There can be two types of relationships in OOPs:

IS-A

HAS-A

IS-A (Inheritance)

The IS-A relationship is nothing but Inheritance. The relationships that can be established between classes using the concept of inheritance are called IS-A relations.

Ex: A parrot is-a Bird. Here Bird is a base class, and Parrot is a derived class, Parrot class inherits all the properties and attributes & methods (other than those that are private) of base class Bird, thus establishing inheritance(IS-A) relation between the two classes.

HAS-a (Association)

The HAS-A association on the other hand is where the Instance variables of a class refer to objects of another class. In other words, one class stores the objects of another class as its instance variables thereby establishing a HAS-A association between the two classes.

UML Diagram

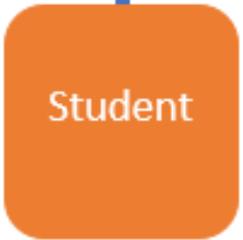
Super Class (Parent)



Person

"Is a" Relationship

[Student is a kind of Person]



Sub Class (Child)

Inheritance

"Has a"
Relationship

Composition



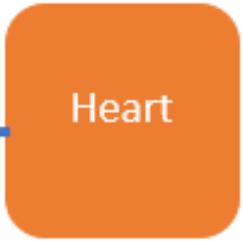
Person



Heart

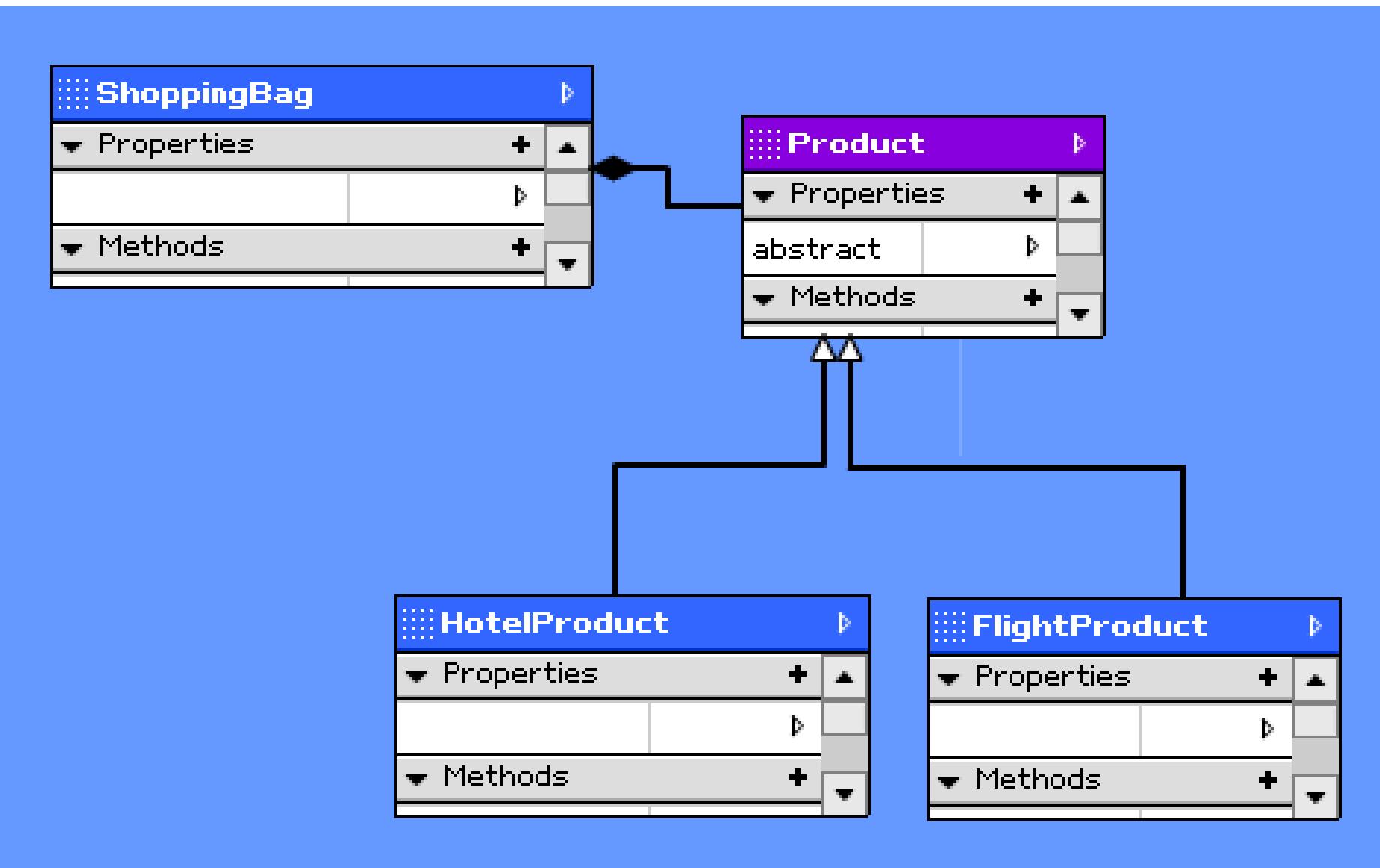


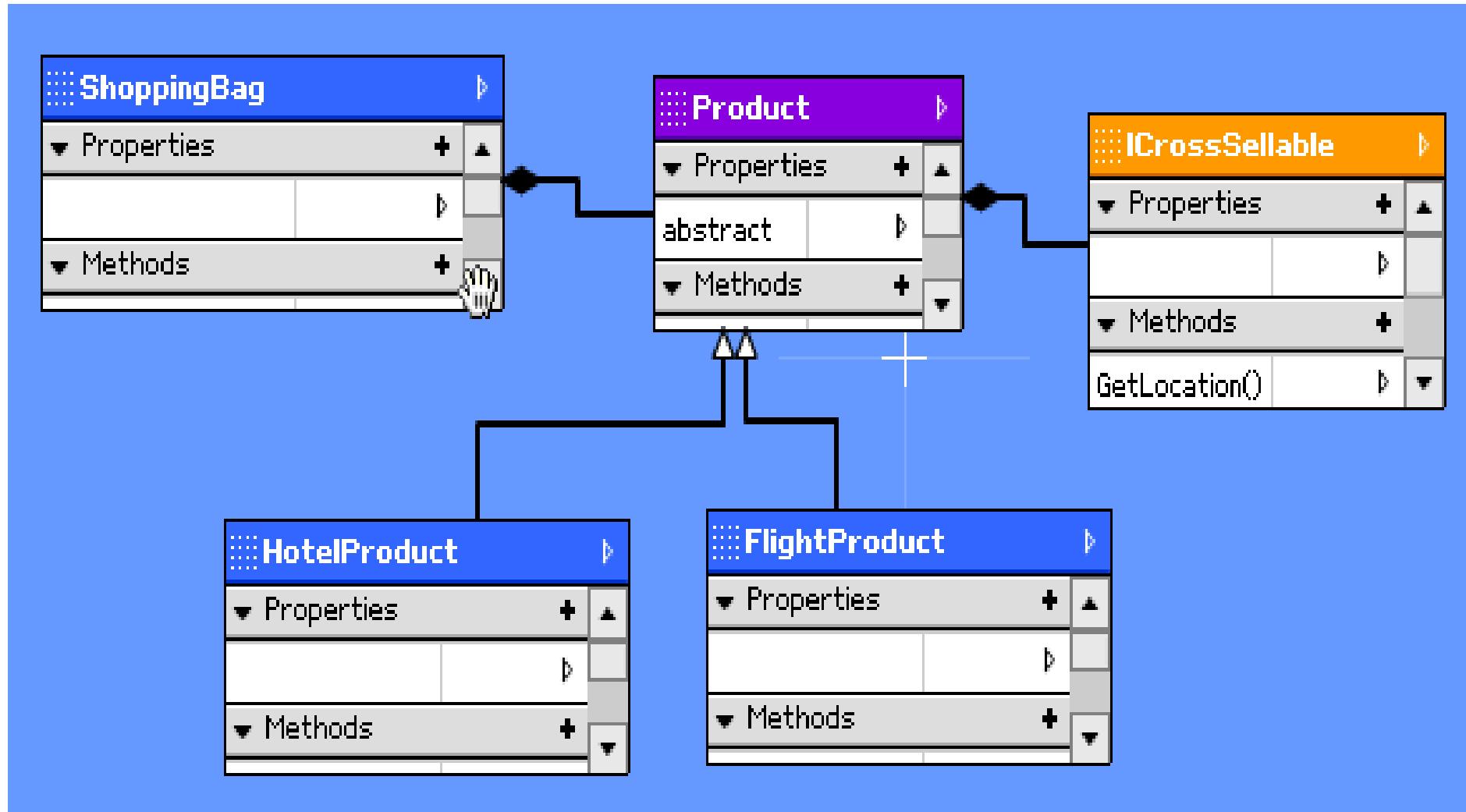
Person



Heart

Aggregation





There are two forms of Association that are possible in Java:

- a) Aggregation
- b) Composition

Aggregation:

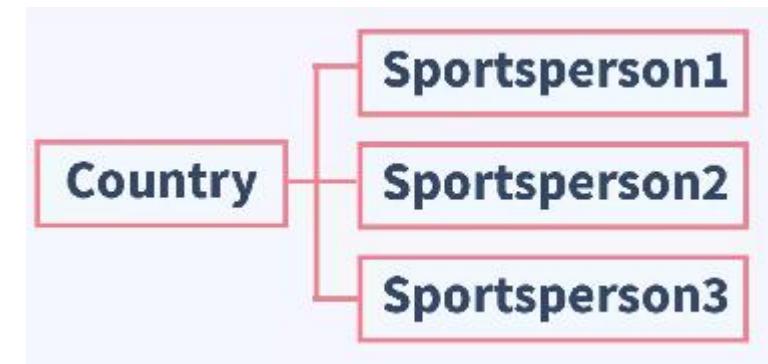
Aggregation in java is a form of HAS-A relationship between two classes. It is a relatively more loosely coupled relation than composition in that, although both classes are associated with each other, one can exist without the other independently. So Aggregation in java is also called a weak association. Let us look at a simple aggregation example to understand this better.

Example: Consider the association between a Country class and a Sportsperson class. Here's how it is defined

Country class is defined with a name and other attributes like size, population, capital, etc, and a list of all the Sportspersons that come from it.

A Sportsperson class is defined with a name and other attributes like age, height, weight, etc.

In a real-world context, we can infer an association between a country and a sports person that hails from that country. Modeling this relation to OOPs, a Country object has-a list of Sportsperson objects that are related to it. Note that a sportsperson object can exist with his own attributes and methods, alone without the association with the country object. Similarly, a country object can exist independently without any association to a sportsperson object. In, other words both Country and Sportsperson classes are independent although there is an association between them. Hence Aggregation is also known as a weak association.



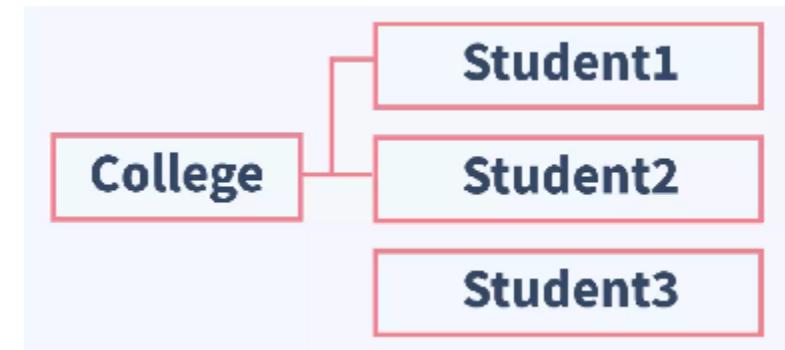
Composition:

Composition in java is a form of relation that is more tightly coupled. Composition in java is also called Strong association. This association is also known as Belongs-To association as one class, for all intents and purpose belongs to another class, and exists because of it. In a Composition association, the classes cannot exist independent of each other. If the larger class which holds the objects of the smaller class is removed, it also means logically the smaller class cannot exist. Let us explore this association clearly with an example

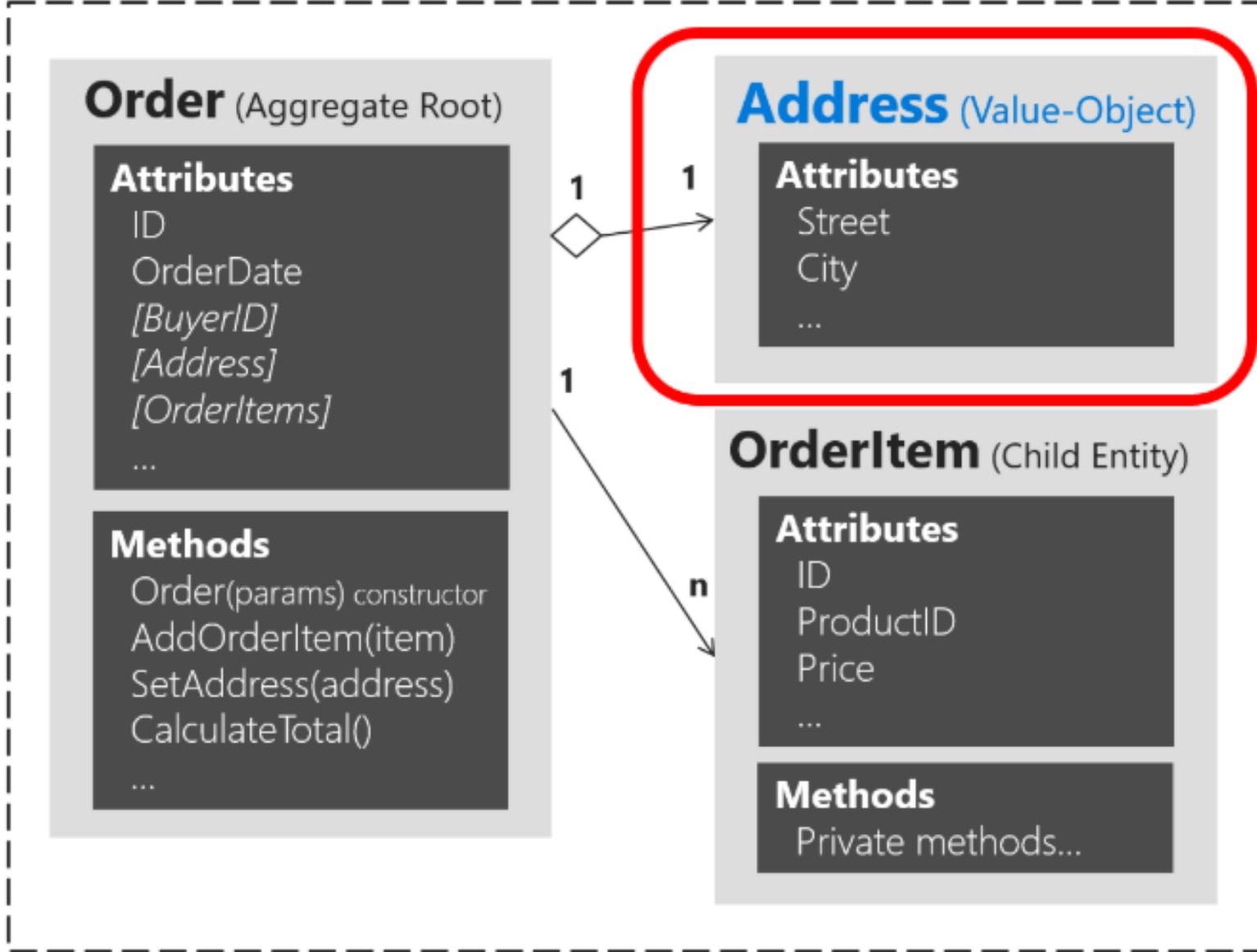
Example: The association between College and Student.
Below is how it is defined.

College class is defined with name and the list of students that are studying in it

A Student class is defined with name and the college he is studying at. Here a student must be studying in at least one college if he is to be called Student. If the college class is removed, Student class cannot exist alone logically, because if a person is not studying in any college then he is not a student.



Order Aggregate (Multiple entities and Value-Object)



Single Responsibility

Each software module or a class should have one and only one reason to change

Liskov Substitution

You should be able to use any derived class instead of a base class without modification

Dependency Inversion

High level classes should not depend on low level classes instead both should depend upon abstraction

S O L I D

Open/Closed

A Software Class or module should be open for extension but closed for modification

Design Principles

Interface Segregation

Client should not be forced to use an interface which is not relevant to it

The SOLID principles and design patterns are both fundamental concepts in software engineering, but they serve different purposes and operate at different levels of abstraction. Here are the key differences between the two:

Purpose:

- ✓ **SOLID Principles:** These are a set of five principles aimed at guiding software design to make software more understandable, flexible, and maintainable. They focus on the design of individual classes and the relationships between them.
- ✓ **Design Patterns:** Design patterns are reusable solutions to commonly occurring problems in software design. They provide general, high-level templates for organizing code and solving specific design problems.

Granularity:

- ✓ **SOLID Principles:** These principles operate at a lower level of abstraction, focusing on the design of individual classes and their responsibilities, relationships, and interactions.
- ✓ **Design Patterns:** Design patterns operate at a higher level of abstraction, providing solutions that can be applied to entire components, subsystems, or even entire applications.

Scope:

- ✓ **SOLID Principles:** These principles are more universal and apply across various programming languages and paradigms. They are fundamental guidelines for designing object-oriented software.
- ✓ **Design Patterns:** Design patterns are more specific and often tied to particular programming paradigms or languages. While many design patterns are applicable across different languages, some are more closely associated with specific environments or technologies.

Number:

- ✓ **SOLID Principles:** There are five SOLID principles: Single Responsibility Principle (SRP), Open/Closed Principle (OCP), Liskov Substitution Principle (LSP), Interface Segregation Principle (ISP), and Dependency Inversion Principle (DIP).
- ✓ **Design Patterns:** There are numerous design patterns, categorized into three main types: creational, structural, and behavioral patterns. Examples include Singleton, Factory Method, Observer, Strategy, etc.

Usage:

- SOLID Principles:** SOLID principles guide developers in creating well-structured, maintainable, and extensible code. They help in designing classes and their relationships in a way that makes the code easier to understand, maintain, and extend.

- Design Patterns:** Design patterns provide specific solutions to recurring design problems. They help developers create more modular, reusable, and maintainable code by following proven best practices.

Single Responsibility Principle (SRP):

A class should have only one reason to change. In other words, each class should have only one responsibility or should only do one thing.

Single Responsibility Principle (SRP):

Use case: E-commerce Platform

Microservice: Order Management Service

Responsibility: Handles order creation, updating, and processing

Justification: This microservice has a single responsibility: managing orders. It doesn't handle user authentication or inventory management. This separation ensures that changes to the order management process won't affect other parts of the system.

Open/Closed Principle (OCP):

Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification. This means that you should be able to extend the behavior of a module without modifying its source code

Open/Closed Principle (OCP):

Use case: E-commerce Platform

Microservice: Payment Gateway Service

Requirement: Support for new payment methods

Justification: The Payment Gateway Service is designed to be open for extension. When a new payment method needs to be added (e.g., cryptocurrency), a new module can be created to handle this without modifying the existing codebase. This ensures that existing payment methods continue to work without disruption.

Liskov Substitution Principle (LSP):

Subtypes must be substitutable for their base types without affecting the correctness of the program. In simpler terms, objects of a superclass should be replaceable with objects of its subclasses without altering the correctness of the program.

Liskov Substitution Principle (LSP):

Use case: E-commerce Platform

Microservice: Shipping Service

Requirement: Support for different shipping carriers (e.g., UPS, FedEx, DHL)

Justification: Each shipping carrier implementation should be substitutable without affecting the correctness of the system. For example, if the system expects to receive tracking information from any carrier, the implementation for UPS should behave similarly to the implementation for FedEx. This allows for seamless switching between carriers.

Interface Segregation Principle (ISP):

A client should not be forced to implement interfaces they don't use. This principle encourages the creation of small, cohesive interfaces rather than large, monolithic ones.

Interface Segregation Principle (ISP):

Use case: E-commerce Platform

Microservice: Product Catalog Service

Interfaces: ProductInformation, ProductSearch

Justification: Clients consuming the Product Catalog Service might only need to retrieve product information or search for products. By segregating the interfaces into smaller, focused ones, clients can implement only the interfaces they require, reducing unnecessary dependencies and potential coupling.

DIP: Microservices should depend on abstractions rather than concrete implementations. This allows for flexibility in choosing different implementations or technologies without impacting the overall architecture.

* Dependency Injection is a technique used to implement the Dependency Inversion Principle.

Dependency Inversion Principle (DIP):

Use case: E-commerce Platform

Microservice: Notification Service

Dependency: Email Service

Justification: Rather than directly depending on a specific email service implementation, the Notification Service depends on an abstraction (e.g., EmailSender interface). This allows for easy switching between different email service providers (e.g., SendGrid, AWS SES) without modifying the Notification Service itself. Additionally, it facilitates testing by enabling the use of mock implementations during testing.

The SOLID principles are a set of design principles intended to make software designs more understandable, flexible, and maintainable. When applied to microservices architecture, each SOLID principle can guide the design and implementation of individual microservices as well as the interactions between them. Here's how each SOLID principle relates to microservices:

1. Single Responsibility Principle (SRP):

- ✓ In microservices, each service should have a single responsibility or purpose. This means that each microservice should encapsulate a specific business capability or functionality.
- ✓ By adhering to SRP, microservices become more focused and easier to understand, maintain, and scale.

2. Open/Closed Principle (OCP):

- ✓ Microservices should be open for extension but closed for modification. This means that you should be able to extend the functionality of a microservice without modifying its existing codebase.
- ✓ OCP encourages designing microservices in a way that allows for easy addition of new features or changes in requirements without disrupting existing functionality.

Liskov Substitution Principle (LSP):

- ✓ In microservices, different implementations of the same service interface should be interchangeable without affecting the correctness of the system.
- ✓ LSP ensures that microservices can be replaced or upgraded with alternative implementations without causing unexpected behavior or breaking other parts of the system.

Interface Segregation Principle (ISP):

- ✓ Microservices should expose well-defined and focused interfaces that are tailored to the specific needs of clients.

- ✓ ISP encourages designing microservice interfaces that are cohesive and only expose functionality relevant to the clients that consume them. This helps reduce unnecessary dependencies and potential coupling between microservices.

5. Dependency Inversion Principle (DIP):

- ✓ Microservices should depend on abstractions rather than concrete implementations. This allows for flexibility in choosing and changing dependencies.
- ✓ DIP promotes designing microservices with loosely coupled dependencies, enabling easier management of dependencies and facilitating the replacement of dependencies with alternative implementations.

By applying the SOLID principles to microservices architecture, you can create a system that is more modular, maintainable, and scalable. Each microservice becomes a cohesive unit with clear boundaries and responsibilities, making it easier to develop, test, deploy, and evolve independently.