Exercício 4 - Funções Lambda e Threads em Java

Índice

---

1. Funções Lambda (Lambda Expressions)

1.1. Conceito e Definição

O que é uma Função Lambda?

Uma **Função Lambda** (ou **Expressão Lambda**) é uma função anônima que pode ser criada sem pertencer a nenhuma classe. Introduzida no Java 8 (2014), as expressões lambda representam uma forma concisa de implementar **interfaces funcionais** (interfaces com apenas um método abstrato).

Características Principais

1. **Anônima**: Não possui nome identificador
2. **Concisa**: Reduz significativamente a quantidade de código
3. **Funcional**: Trata funções como cidadãos de primeira classe
4. **Imutável**: Incentiva programação funcional e imutabilidade
5. **Type Inference**: O compilador infere o tipo automaticamente

Interface Funcional

Para usar lambda, é necessária uma **interface funcional**:

```java
@FunctionalInterface
interface Calculadora {
    int calcular(int a, int b);
}
```

A anotação `@FunctionalInterface` é opcional, mas recomendada para documentação e validação em tempo de compilação.

1.2. Tradução Literal

Português → Inglês

Português	Inglês	Observações
Função Lambda	Lambda Function	Termo mais comum
Expressão Lambda	Lambda Expression	Termo técnico oficial em Java
Função Anônima	Anonymous Function	Sinônimo
Closure	Closure	Conceito relacionado
Interface Funcional	Functional Interface	Requisito para lambda

Etimologia

O termo **"Lambda" (λ)** vem da letra grega usada no **Cálculo Lambda** (λ-calculus), desenvolvido por **Alonzo Church** na década de 1930, um sistema formal em lógica matemática e ciência da computação para expressar computação baseada em abstração e aplicação de funções.

1.3. Sintaxe e Estrutura

Sintaxe Geral

```
(parametros) -> expressao
```

ou

```
(parametros) -> {
    // bloco de codigo
    return resultado;
}
```

Componentes

1. **Lista de Parâmetros**: Entre parênteses, separados por vírgula
2. **Operador Arrow**: `->` (token lambda)
3. **Corpo**: Expressão única ou bloco de código

Evolução Sintática

Antes do Java 8 (Classe Anônima)

```
Calculadora soma = new Calculadora() {
    @Override
    public int calcular(int a, int b) {
        return a + b;
    }
};
```

Linhas de código: 6

Com Java 8 (Lambda)

```
Calculadora soma = (a, b) -> a + b;
```

Linhas de código: 1

Redução: 83% menos código!

1.4. Exemplos Práticos

Exemplo 1: Lambda Sem Parâmetros

```
// Interface funcional
@FunctionalInterface
interface Saudacao {
    void dizerOla();
}

// Uso com lambda
Saudacao saudacao = () -> System.out.println("Ola, mundo!");
saudacao.dizerOla();

// Saída: Ola, mundo!
```

Análise:

- `()`: Sem parâmetros
- `->`: Operador lambda

- `System.out.println(...)`: Corpo da expressão

Exemplo 2: Lambda com Um Parâmetro

```
@FunctionalInterface
interface Quadrado {
    int calcular(int numero);
}

// Parênteses são opcionais com 1 parâmetro
Quadrado quad = n -> n * n;

System.out.println(quad.calcular(5));  // Saída: 25
System.out.println(quad.calcular(10)); // Saída: 100
```

Análise:

- `n`: Parâmetro único (parênteses opcionais)
- `n * n`: Expressão única (return implícito)

Exemplo 3: Lambda com Múltiplos Parâmetros

```
@FunctionalInterface
interface Operacao {
    int executar(int a, int b);
}

// Soma
Operacao soma = (a, b) -> a + b;
System.out.println(soma.executar(10, 5)); // 15

// Subtração
Operacao subtracao = (a, b) -> a - b;
System.out.println(subtracao.executar(10, 5)); // 5

// Multiplicação
Operacao multiplicacao = (a, b) -> a * b;
System.out.println(multiplicacao.executar(10, 5)); // 50

// Divisão
Operacao divisao = (a, b) -> a / b;
System.out.println(divisao.executar(10, 5)); // 2
```

Exemplo 4: Lambda com Bloco de Código

```java
@FunctionalInterface
interface Validador {
    boolean validar(String texto);
}

Validador validarEmail = (email) -> {
    if (email == null || email.isEmpty()) {
        return false;
    }
    return email.contains("@") && email.contains(".");
};

System.out.println(validarEmail.validar("usuario@email.com")); // true
System.out.println(validarEmail.validar("invalido"));          // false
System.out.println(validarEmail.validar(null));                // false
```

Análise:

- Bloco de código requer `{ }`
- `return` explícito é necessário
- Múltiplas instruções permitidas

Exemplo 5: Lambda com Collections (forEach)

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

public class LambdaCollections {
    public static void main(String[] args) {
        List<String> nomes = Arrays.asList("Ana", "Bruno", "Carlos", "Diana");

        // Antes do Java 8
        for (String nome : nomes) {
            System.out.println(nome);
        }

        // Com Lambda (Java 8+)
        nomes.forEach(nome -> System.out.println(nome));

        // Method Reference (ainda mais conciso)
        nomes.forEach(System.out::println);
    }
}
```

Saída:

```
Ana
Bruno
Carlos
Diana
```

Exemplo 6: Lambda com Streams

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

public class LambdaStreams {
    public static void main(String[] args) {
        List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

        // Filtrar números pares
        List<Integer> pares = numeros.stream()
            .filter(n -> n % 2 == 0)
            .collect(Collectors.toList());

        System.out.println("Pares: " + pares);
        // Saída: Pares: [2, 4, 6, 8, 10]

        // Multiplicar por 2
        List<Integer> dobrados = numeros.stream()
            .map(n -> n * 2)
            .collect(Collectors.toList());

        System.out.println("Dobrados: " + dobrados);
        // Saída: Dobrados: [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]

        // Somar todos os números
        int soma = numeros.stream()
            .reduce(0, (a, b) -> a + b);

        System.out.println("Soma: " + soma);
        // Saída: Soma: 55
    }
}
```

Exemplo 7: Lambda com Comparator

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

```
public class LambdaComparator {
    public static void main(String[] args) {
        List<String> frutas = Arrays.asList("Banana", "Maçã", "Uva", "Abacaxi");

        // Ordenar por ordem alfabética
        frutas.sort((f1, f2) -> f1.compareTo(f2));
        System.out.println("Alfabética: " + frutas);
        // Saída: [Abacaxi, Banana, Maçã, Uva]

        // Ordenar por tamanho
        frutas.sort((f1, f2) -> Integer.compare(f1.length(), f2.length()));
        System.out.println("Por tamanho: " + frutas);
        // Saída: [Uva, Maçã, Banana, Abacaxi]

        // Usando Comparator.comparing
        frutas.sort(Comparator.comparing(String::length));
        System.out.println("Por tamanho (method ref): " + frutas);
    }
}
```

Exemplo 8: Interfaces Funcionais Pré-definidas

Java fornece várias interfaces funcionais no pacote `java.util.function`:

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

public class InterfacesFuncionais {
    public static void main(String[] args) {
        // Predicate<T>: T -> boolean
        Predicate<Integer> ehPar = n -> n % 2 == 0;
        System.out.println(ehPar.test(4)); // true
        System.out.println(ehPar.test(5)); // false

        // Function<T, R>: T -> R
        Function<String, Integer> tamanho = s -> s.length();
        System.out.println(tamanho.apply("Java")); // 4

        // Consumer<T>: T -> void
        Consumer<String> imprimir = s -> System.out.println("Valor: " + s);
        imprimir.accept("Lambda"); // Valor: Lambda

        // Supplier<T>: () -> T
        Supplier<Double> aleatorio = () -> Math.random();
        System.out.println(aleatorio.get()); // Número aleatório

        // BiFunction<T, U, R>: (T, U) -> R
        BiFunction<Integer, Integer, Integer> soma = (a, b) -> a + b;
        System.out.println(soma.apply(5, 3)); // 8

        // UnaryOperator<T>: T -> T
```

```
        UnaryOperator<Integer> dobrar = n -> n * 2;
        System.out.println(dobrar.apply(5)); // 10

        // BinaryOperator<T>: (T, T) -> T
        BinaryOperator<Integer> multiplicar = (a, b) -> a * b;
        System.out.println(multiplicar.apply(4, 5)); // 20
    }
}
```

1.5. Aplicações

1. **Programação Funcional**

```
// Cadeia de operações funcionais
List<String> resultado = nomes.stream()
    .filter(nome -> nome.startsWith("A"))
    .map(nome -> nome.toUpperCase())
    .sorted()
    .collect(Collectors.toList());
```

2. **Event Handlers (GUI)**

```
// JavaFX Button
Button botao = new Button("Clique");
botao.setOnAction(evento -> System.out.println("Botão clicado!"));

// Swing Button
JButton botaoSwing = new JButton("Clique");
botaoSwing.addActionListener(e -> JOptionPane.showMessageDialog(null, "Clicou!"));
```

3. **Processamento de Coleções**

```
// Filtrar, transformar e coletar
List<Integer> quadradosPares = numeros.stream()
    .filter(n -> n % 2 == 0)
    .map(n -> n * n)
    .collect(Collectors.toList());
```

4. **Threads e Concorrência**

```
// Executar tarefa em thread separada
new Thread(() -> {
    System.out.println("Executando em outra thread");
```

```
}).start();

// ExecutorService
ExecutorService executor = Executors.newFixedThreadPool(4);
executor.submit(() -> processarDados());
```

5. **APIs Reativas**

```
// RxJava
Observable.just(1, 2, 3, 4, 5)
    .filter(n -> n > 2)
    .map(n -> n * 10)
    .subscribe(resultado -> System.out.println(resultado));
```

6. **Ordenação Customizada**

```
// Ordenar objetos complexos
pessoas.sort((p1, p2) -> p1.getIdade() - p2.getIdade());
```

7. **Validação de Dados**

```
Predicate<Pessoa> adulto = p -> p.getIdade() >= 18;
Predicate<Pessoa> nomeValido = p -> p.getNome() != null && !p.getNome().isEmpty();

List<Pessoa> validas = pessoas.stream()
    .filter(adulto.and(nomeValido))
    .collect(Collectors.toList());
```

8. **Optional (Tratamento de Nulos)**

```
Optional<String> opcional = Optional.ofNullable(obterNome());
opcional.ifPresent(nome -> System.out.println("Nome: " + nome));

String resultado = opcional
    .map(String::toUpperCase)
    .orElse("Nome não disponível");
```

---

2. Threads em Java (Java Threads)

2.1. Conceito e Definição

O que é uma Thread?

Uma **Thread** (linha de execução) é a menor unidade de processamento que pode ser agendada por um sistema operacional. Em Java, threads permitem que um programa execute múltiplas tarefas **concorrentemente**, melhorando a performance e a responsividade de aplicações.

Características Principais

1. **Concorrência**: Múltiplas threads executam simultaneamente
2. **Compartilhamento de Memória**: Threads do mesmo processo compartilham heap
3. **Lightweight**: Mais leves que processos completos
4. **Independentes**: Cada thread tem sua própria pilha de execução
5. **Sincronização**: Requerem mecanismos para evitar race conditions

Processo vs Thread

Aspecto	Processo	Thread
Memória	Espaço próprio	Compartilhada
Comunicação	IPC (mais lento)	Variáveis compartilhadas
Criação	Pesada	Leve
Custo	Alto	Baixo
Isolamento	Completo	Parcial

2.2. Tradução Literal

Português → Inglês

Português	Inglês	Observações
Thread	Thread	Sem tradução literal
Linha de Execução	Execution Thread	Tradução descritiva
Processo Leve	Lightweight Process	Termo técnico
Multithreading	Multithreading	Múltiplas threads
Concorrência	Concurrency	Execução simultânea
Paralelismo	Parallelism	Execução verdadeiramente simultânea
Sincronização	Synchronization	Coordenação entre threads
Deadlock	Deadlock	Impasse/travamento
Race Condition	Race Condition	Condição de corrida

Analogia

Imagine um restaurante (processo):

- **Processo**: O restaurante inteiro
- **Threads**: Cada garçom trabalhando
- **Concorrência**: Múltiplos garçons atendendo mesas
- **Sincronização**: Garçons coordenando uso da cozinha
- **Deadlock**: Dois garçons esperando um pelo outro indefinidamente

---

2.3. Criação de Threads

Método 1: Estendendo a Classe Thread

```
class MinhaThread extends Thread {
    @Override
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println(Thread.currentThread().getName() + ": " + i);
            try {
                Thread.sleep(1000); // Pausa de 1 segundo
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

public class ExemploThread1 {
    public static void main(String[] args) {
        MinhaThread t1 = new MinhaThread();
        MinhaThread t2 = new MinhaThread();

        t1.setName("Thread-A");
        t2.setName("Thread-B");

        t1.start(); // Inicia thread 1
        t2.start(); // Inicia thread 2
    }
}
```

Saída (intercalada):

```
Thread-A: 1
Thread-B: 1
Thread-A: 2
Thread-B: 2
Thread-A: 3
Thread-B: 3
...
```

Método 2: Implementando a Interface Runnable

```
class MinhaRunnable implements Runnable {
    @Override
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println(Thread.currentThread().getName() + ": " + i);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

public class ExemploThread2 {
    public static void main(String[] args) {
        Thread t1 = new Thread(new MinhaRunnable(), "Thread-A");
        Thread t2 = new Thread(new MinhaRunnable(), "Thread-B");

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

Vantagem: Java não permite herança múltipla, então usar `Runnable` permite que a classe estenda outra classe.

Método 3: Com Lambda (Java 8+)

```
public class ExemploThread3 {
    public static void main(String[] args) {
        // Thread com lambda
        Thread t1 = new Thread(() -> {
            for (int i = 1; i <= 5; i++) {
                System.out.println(Thread.currentThread().getName() + ": " + i);
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }, "Thread-Lambda");

        t1.start();
```

```
    }
}
```

Análise: Lambda reduz drasticamente o código necessário para criar uma thread!

2.4. Exemplos Práticos

Exemplo 1: Thread com Contador

```
public class ContadorThread extends Thread {
    private String nome;
    private int limite;

    public ContadorThread(String nome, int limite) {
        this.nome = nome;
        this.limite = limite;
    }

    @Override
    public void run() {
        for (int i = 1; i <= limite; i++) {
            System.out.println(nome + " - Contagem: " + i);
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println(nome + " finalizou!");
    }

    public static void main(String[] args) {
        ContadorThread t1 = new ContadorThread("Contador-A", 5);
        ContadorThread t2 = new ContadorThread("Contador-B", 3);
        ContadorThread t3 = new ContadorThread("Contador-C", 4);

        t1.start();
        t2.start();
        t3.start();
    }
}
```

Exemplo 2: Sincronização com synchronized

```
class ContaBancaria {
    private double saldo = 1000.0;
```

```
    // Método sincronizado
    public synchronized void sacar(double valor) {
        if (saldo >= valor) {
            System.out.println(Thread.currentThread().getName() +
                " vai sacar " + valor);

            try {
                Thread.sleep(100); // Simula processamento
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            saldo -= valor;
            System.out.println(Thread.currentThread().getName() +
                " sacou. Saldo: " + saldo);
        } else {
            System.out.println(Thread.currentThread().getName() +
                " - Saldo insuficiente!");
        }
    }

    public double getSaldo() {
        return saldo;
    }
}

public class ExemploSincronizacao {
    public static void main(String[] args) {
        ContaBancaria conta = new ContaBancaria();

        // Várias threads tentando sacar simultaneamente
        Runnable operacao = () -> {
            for (int i = 0; i < 3; i++) {
                conta.sacar(200);
            }
        };

        Thread t1 = new Thread(operacao, "Cliente-1");
        Thread t2 = new Thread(operacao, "Cliente-2");
        Thread t3 = new Thread(operacao, "Cliente-3");

        t1.start();
        t2.start();
        t3.start();
    }
}
```

Análise: Sem `synchronized`, múltiplas threads poderiam sacar ao mesmo tempo, causando saldo negativo (race condition).

Exemplo 3: Producer-Consumer com wait/notify

```
import java.util.LinkedList;
import java.util.Queue;

class Buffer {
    private Queue<Integer> fila = new LinkedList<>();
    private int capacidade = 5;

    public synchronized void produzir(int item) throws InterruptedException {
        while (fila.size() == capacidade) {
            System.out.println("Buffer cheio. Produtor aguardando...");
            wait(); // Aguarda consumidor liberar espaço
        }

        fila.add(item);
        System.out.println("Produzido: " + item + " (Tamanho: " + fila.size() +
")");
        notify(); // Notifica consumidor
        Thread.sleep(500);
    }

    public synchronized int consumir() throws InterruptedException {
        while (fila.isEmpty()) {
            System.out.println("Buffer vazio. Consumidor aguardando...");
            wait(); // Aguarda produtor adicionar item
        }

        int item = fila.poll();
        System.out.println("Consumido: " + item + " (Tamanho: " + fila.size() +
")");
        notify(); // Notifica produtor
        Thread.sleep(1000);
        return item;
    }
}

public class ProducerConsumer {
    public static void main(String[] args) {
        Buffer buffer = new Buffer();

        // Thread Produtora
        Thread produtor = new Thread(() -> {
            for (int i = 1; i <= 10; i++) {
                try {
                    buffer.produzir(i);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }, "Produtor");

        // Thread Consumidora
        Thread consumidor = new Thread(() -> {
```

```
            for (int i = 1; i <= 10; i++) {
                try {
                    buffer.consumir();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }, "Consumidor");

        produtor.start();
        consumidor.start();
    }
}
```

Exemplo 4: ThreadPool com ExecutorService

```
import java.util.concurrent.*;

public class ThreadPoolExemplo {
    public static void main(String[] args) {
        // Cria pool com 3 threads
        ExecutorService executor = Executors.newFixedThreadPool(3);

        // Submete 10 tarefas
        for (int i = 1; i <= 10; i++) {
            final int taskId = i;

            executor.submit(() -> {
                System.out.println("Tarefa " + taskId + " iniciada em " +
                    Thread.currentThread().getName());

                try {
                    Thread.sleep(2000); // Simula processamento
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }

                System.out.println("Tarefa " + taskId + " finalizada");
            });
        }

        executor.shutdown(); // Não aceita novas tarefas

        try {
            // Aguarda todas as tarefas finalizarem (máximo 1 minuto)
            if (!executor.awaitTermination(60, TimeUnit.SECONDS)) {
                executor.shutdownNow(); // Força encerramento
            }
        } catch (InterruptedException e) {
            executor.shutdownNow();
```

```
        }

        System.out.println("Todas as tarefas finalizaram!");
    }
}
```

Vantagem: Reutiliza threads, evitando overhead de criação/destruição.

Exemplo 5: Callable e Future (com Retorno)

```
import java.util.concurrent.*;
import java.util.ArrayList;
import java.util.List;

public class CallableFutureExemplo {
    public static void main(String[] args) throws InterruptedException,
ExecutionException {
        ExecutorService executor = Executors.newFixedThreadPool(3);

        // Lista de tarefas que retornam valores
        List<Callable<Integer>> tarefas = new ArrayList<>();

        for (int i = 1; i <= 5; i++) {
            final int numero = i;
            tarefas.add(() -> {
                Thread.sleep(1000);
                int resultado = numero * numero;
                System.out.println("Calculado: " + numero + "^2 = " + resultado);
                return resultado;
            });
        }

        // Executa todas as tarefas
        List<Future<Integer>> resultados = executor.invokeAll(tarefas);

        // Coleta os resultados
        int soma = 0;
        for (Future<Integer> future : resultados) {
            soma += future.get(); // Bloqueia até resultado estar disponível
        }

        System.out.println("Soma de todos os quadrados: " + soma);
        // Saída: 1 + 4 + 9 + 16 + 25 = 55

        executor.shutdown();
    }
}
```

Exemplo 6: Estados de uma Thread

```
public class EstadosThread {
    public static void main(String[] args) throws InterruptedException {
        Thread thread = new Thread(() -> {
            System.out.println("Estado RUNNABLE: Thread executando");

            try {
                Thread.sleep(2000); // Estado TIMED_WAITING
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            System.out.println("Thread finalizando");
        });

        // Estado NEW
        System.out.println("Estado inicial: " + thread.getState());

        thread.start();

        // Estado RUNNABLE
        Thread.sleep(100);
        System.out.println("Após start(): " + thread.getState());

        // Estado TIMED_WAITING
        Thread.sleep(500);
        System.out.println("Durante sleep(): " + thread.getState());

        // Aguarda finalização
        thread.join();

        // Estado TERMINATED
        System.out.println("Após finalizar: " + thread.getState());
    }
}
```

Estados possíveis:

1. **NEW**: Thread criada, mas não iniciada
2. **RUNNABLE**: Executando ou pronta para executar
3. **BLOCKED**: Bloqueada aguardando monitor lock
4. **WAITING**: Aguardando indefinidamente outra thread
5. **TIMED_WAITING**: Aguardando por tempo determinado
6. **TERMINATED**: Execução finalizada

Exemplo 7: Prioridades de Thread

```
public class PrioridadesThread {
    public static void main(String[] args) {
        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 5; i++) {
                System.out.println("Thread-Alta: " + i);
            }
        });

        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 5; i++) {
                System.out.println("Thread-Baixa: " + i);
            }
        });

        // Define prioridades (1 = mínima, 10 = máxima, 5 = padrão)
        t1.setPriority(Thread.MAX_PRIORITY);  // 10
        t2.setPriority(Thread.MIN_PRIORITY);  // 1

        t2.start();
        t1.start();

        // Nota: Prioridades são apenas sugestões ao scheduler
    }
}
```

Exemplo 8: Daemon Threads

```
public class DaemonThreadExemplo {
    public static void main(String[] args) throws InterruptedException {
        Thread daemonThread = new Thread(() -> {
            while (true) {
                System.out.println("Daemon rodando...");
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    break;
                }
            }
        });

        // Define como daemon (thread de background)
        daemonThread.setDaemon(true);
        daemonThread.start();

        System.out.println("Thread principal dormindo por 3 segundos...");
        Thread.sleep(3000);

        System.out.println("Thread principal finalizando...");
        // Daemon thread será encerrada automaticamente
```

```
    }
}
```

Daemon Thread: Thread de background que não impede o programa de terminar.

2.5. Aplicações

1. **Servidores Web**

```
// Cada conexão de cliente em uma thread separada
while (true) {
    Socket client = serverSocket.accept();
    new Thread(() -> handleClient(client)).start();
}
```

2. **Interface Gráfica (GUI)**

```
// Event Dispatch Thread (EDT) em Swing
SwingUtilities.invokeLater(() -> {
    JFrame frame = new JFrame("Aplicação");
    frame.setVisible(true);
});
```

3. **Download de Arquivos**

```
// Múltiplos downloads simultâneos
for (String url : urls) {
    new Thread(() -> downloadFile(url)).start();
}
```

4. **Processamento de Dados**

```
// Processar grandes volumes em paralelo
List<Data> chunks = splitData(bigData);
ExecutorService executor = Executors.newFixedThreadPool(4);
chunks.forEach(chunk -> executor.submit(() -> process(chunk)));
```

5. **Games e Animações**

```
// Game loop em thread separada
Thread gameLoop = new Thread(() -> {
```

```
    while (running) {
        update();
        render();
        sleep(16); // ~60 FPS
    }
});
```

6. **Monitoramento e Logging**

```
// Thread de monitoramento contínuo
Thread monitor = new Thread(() -> {
    while (true) {
        checkSystemHealth();
        Thread.sleep(5000);
    }
});
monitor.setDaemon(true);
monitor.start();
```

7. **Banco de Dados**

```
// Connection pool gerenciando threads
HikariConfig config = new HikariConfig();
config.setMaximumPoolSize(10);
HikariDataSource ds = new HikariDataSource(config);
```

8. **APIs Assíncronas**

```
// CompletableFuture (Java 8+)
CompletableFuture.supplyAsync(() -> buscarDados())
    .thenApply(dados -> processar(dados))
    .thenAccept(resultado -> salvar(resultado));
```

---

3. Integração: Lambda + Threads

Exemplo Completo: Processamento Paralelo com Lambda

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

public class LambdaThreadsIntegracao {
```

```
    public static void main(String[] args) throws InterruptedException,
ExecutionException {
        List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

        // Cria thread pool
        ExecutorService executor = Executors.newFixedThreadPool(4);

        // Processa cada número em thread separada usando lambda
        List<Future<Integer>> futuros = numeros.stream()
            .map(num -> executor.submit(() -> {
                // Lambda executado em thread do pool
                System.out.println("Processando " + num + " em " +
                    Thread.currentThread().getName());
                Thread.sleep(1000); // Simula processamento pesado
                return num * num;
            }))
            .toList();

        // Coleta resultados
        List<Integer> resultados = futuros.stream()
            .map(future -> {
                try {
                    return future.get();
                } catch (InterruptedException | ExecutionException e) {
                    throw new RuntimeException(e);
                }
            })
            .toList();

        System.out.println("\nResultados: " + resultados);

        // Soma usando lambda e reduce
        int soma = resultados.stream()
            .reduce(0, (a, b) -> a + b);

        System.out.println("Soma total: " + soma);

        executor.shutdown();
    }
}
```

Parallel Streams (Threads + Lambda automático)

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

public class ParallelStreamsExemplo {
    public static void main(String[] args) {
        List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
```

```
        // Stream paralelo (usa ForkJoinPool internamente)
        int soma = numeros.parallelStream()
            .peek(n -> System.out.println("Thread: " +
                Thread.currentThread().getName() + " - Número: " + n))
            .map(n -> n * n)
            .reduce(0, Integer::sum);

        System.out.println("\nSoma dos quadrados: " + soma);
    }
}
```

Vantagem: Java gerencia threads automaticamente usando ForkJoinPool!

---

4. Referências Bibliográficas

Livros

1. **ORACLE**. *The Java™ Tutorials - Lambda Expressions*. Oracle Corporation, 2024.
   Disponível em: https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html

2. **ORACLE**. *The Java™ Tutorials - Concurrency*. Oracle Corporation, 2024.
   Disponível em: https://docs.oracle.com/javase/tutorial/essential/concurrency/

3. **GOETZ, Brian et al.** *Java Concurrency in Practice*. Addison-Wesley, 2006.
   ISBN: 978-0321349606

4. **BLOCH, Joshua**. *Effective Java*. 3rd Edition. Addison-Wesley, 2018.
   ISBN: 978-0134685991

5. **HORSTMANN, Cay S.**. *Core Java Volume I - Fundamentals*. 11th Edition. Prentice Hall, 2018.
   ISBN: 978-0135166307

6. **URMA, Raoul-Gabriel; FUSCO, Mario; MYCROFT, Alan**. *Java 8 in Action: Lambdas, Streams, and Functional-style Programming*. Manning Publications, 2014.
   ISBN: 978-1617291999

Artigos e Documentação

7. **VENNERS, Bill**. *Inside the Java Virtual Machine*. McGraw-Hill, 2000.

8. **JCP (Java Community Process)**. *JSR 335: Lambda Expressions for the Java™ Programming Language*. 2013.

9. **ORACLE**. *Java Platform, Standard Edition 8 API Specification*.
   Disponível em: https://docs.oracle.com/javase/8/docs/api/

10. **BAELDUNG**. *Java Lambda Expressions*.
    Disponível em: https://www.baeldung.com/java-8-lambda-expressions-tips

Recursos Online

11. **GITHUB**. *Java Design Patterns - Functional Programming*.
    Disponível em: https://github.com/iluwatar/java-design-patterns

12. **STACKOVERFLOW**. *Java Lambda Questions*.
    Disponível em: https://stackoverflow.com/questions/tagged/java+lambda

13. **JENKOV, Jakob**. *Java Concurrency and Multithreading Tutorial*.
    Disponível em: http://tutorials.jenkov.com/java-concurrency/

Glossário Técnico

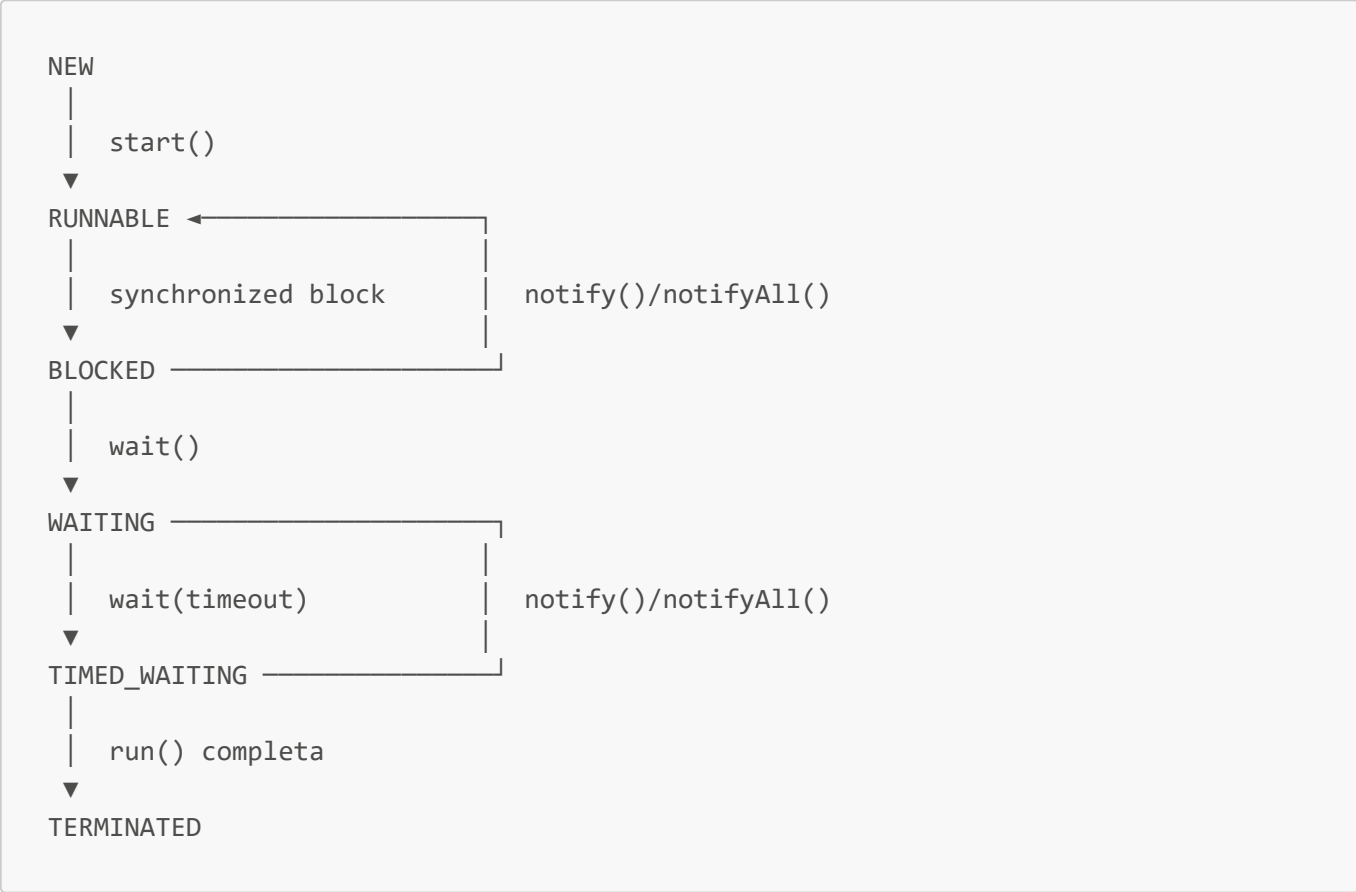
Termo	Definição
Lambda	Função anônima que pode ser passada como argumento
Interface Funcional	Interface com apenas um método abstrato
Closure	Lambda que captura variáveis do escopo externo
Method Reference	Sintaxe abreviada para lambdas (`::`)
Stream	Sequência de elementos que suporta operações funcionais
Thread	Menor unidade de execução em um processo
Concorrência	Múltiplas tarefas progredindo simultaneamente
Paralelismo	Múltiplas tarefas executando ao mesmo tempo
Synchronized	Palavra-chave para sincronização de threads
Race Condition	Comportamento indesejado por acesso concorrente
Deadlock	Threads aguardando mutuamente, indefinidamente
Thread Pool	Conjunto reutilizável de threads
Executor	Framework para gerenciar threads
Future	Resultado assíncrono de uma computação
Callable	Tarefa que retorna resultado

Tabela Comparativa: Lambda vs Classe Anônima

Aspecto	Classe Anônima	Lambda
Sintaxe	Verbosa (6+ linhas)	Concisa (1 linha)
this	Refere à classe anônima	Refere à classe externa

Aspecto	Classe Anônima	Lambda
Escopo	Cria novo escopo	Usa escopo léxico
Variáveis	Podem ser não-final	Devem ser final/effectively final
Bytecode	Classe interna gerada	invokedynamic (Java 7+)
Performance	Mais lenta (criação de objeto)	Mais rápida
Uso	Qualquer interface	Apenas interfaces funcionais

Diagrama: Ciclo de Vida de uma Thread



Autor: Estudo compilado para Disciplina de Linguagens de Programação
Data: 21 de Outubro de 2025
Versão: 1.0
Tecnologias: Java SE 8+, Concurrency API, Functional Programming

---

Nota: Este documento pode ser convertido para PDF usando ferramentas como:

- Pandoc: `pandoc Ex4.md -o Ex4.pdf`
- Markdown to PDF (VS Code Extension)
- Online: https://www.markdowntopdf.com/