Introdução multithreading em Java

Vagner Fonseca

Definição de Multithreading

Multithreading é a capacidade de um sistema operacional executar múltiplos fluxos de execução (threads) simultaneamente dentro de um único processo. Cada thread possui seu próprio programa contador, pilha e registradores, permitindo a execução paralela de diferentes tarefas.

Importância e aplicabilidade em Java

Melhoria da Eficiência

O multithreading permite que os programas Java aproveitem melhor os recursos de hardware, executando múltiplas tarefas simultaneamente e aumentando a produtividade.

Escalabilidade

Ao dividir uma aplicação em threads, ela pode ser facilmente escalada para lidar com aumentos de carga de trabalho, mantendo o desempenho.

Resposta em Tempo Real

Threads permitem que aplicativos Java respondam de forma rápida e responsiva a eventos e interações do usuário, melhorando a experiência do cliente.

Conceitos Básicos de Threads em Java

1

O que são Threads?

Threads são unidades independentes de execução dentro de um mesmo processo. Elas permitem que um programa execute múltiplas tarefas simultaneamente, aumentando a eficiência e o desempenho.

2

Ciclo de Vida de uma Thread

Uma thread pode passar por diferentes estados, como Criada, Pronta, Em Execução, Bloqueada e Terminada. Esses estados determinam o comportamento e o fluxo de execução da thread.

3

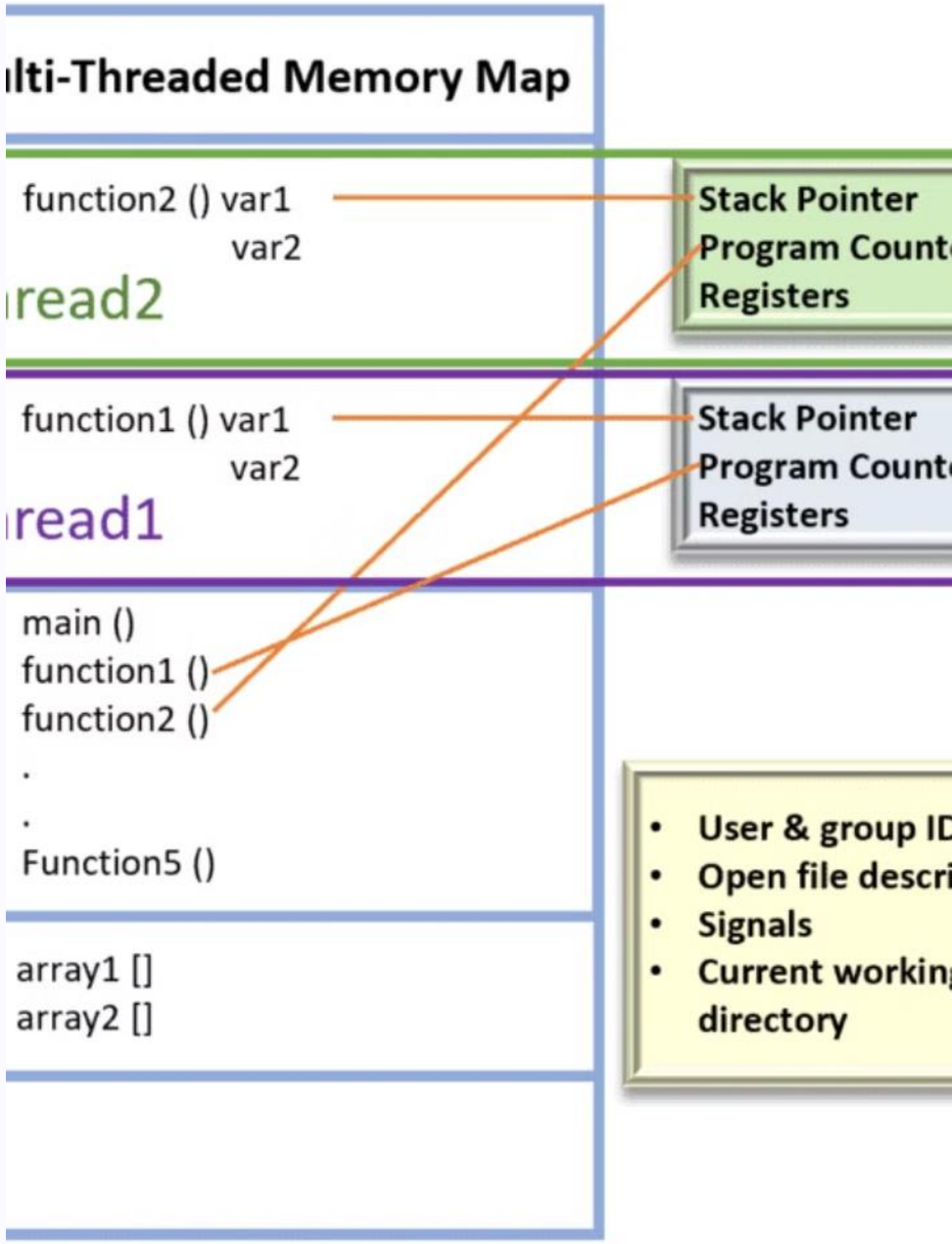
Criação e Inicialização

Para criar uma thread em Java, é necessário implementar a interface Runnable ou estender a classe Thread. Após a criação, a thread deve ser inicializada chamando o método start().

O que são Threads

Threads são unidades independentes de execução dentro de um mesmo processo de software. Elas permitem que um programa execute múltiplas tarefas concorrentemente, aumentando a eficiência e responsividade da aplicação.

Cada thread possui seu próprio caminho de execução, pilha de execução e contador de programa, podendo operar em paralelo com outras threads do mesmo processo.



Ciclo de vida de uma Thread

1. O ciclo de vida de uma **Thread** no Java é composto por diferentes estados:
2. Iniciada (New): Quando a **Thread** é criada, mas ainda não foi iniciada.
3. Em execução (Runnable): A **Thread** está pronta para ser executada e aguarda a disponibilidade da CPU.
4. Bloqueada (Blocked): A **Thread** está aguardando por um recurso ou uma ação específica para prosseguir.
5. Esperando (Waiting): A **Thread** está aguardando indefinidamente por uma notificação de outra **Thread**.
6. Terminada (Terminated): A **Thread** finalizou sua execução.

Criação e Inicialização de Threads

Criação de Threads

Em Java, as threads são criadas por meio da implementação da interface Runnable ou pela extensão da classe Thread. Essa abordagem permite a definição do comportamento específico da thread.

Inicialização de Threads

Após a criação, as threads precisam ser inicializadas. Isso é feito chamando o método start(), que irá executar o código definido na thread.

Ciclo de Vida

As threads passam por um ciclo de vida composto por estados como NEW, RUNNABLE, BLOCKED, WAITING, TIMED_WAITING e TERMINATED. O gerenciamento desses estados é crucial para a correta execução das threads.

Exemplo Prático

Veja um exemplo de criação e inicialização de uma thread em Java: Thread myThread = new Thread(() -> { // Código a ser executado pela thread }); myThread.start();

Paralelismo x Concorrência

Paralelismo: Execução de vários trechos de Código no mesmo instante.

Concorrência: várias execuções de Código concorrendo pelo mesmo recurso.

Sincronização de Threads

1

Exclusão Mútua

Garantir que apenas uma thread acesse um recurso compartilhado por vez, evitando conflitos e inconsistências.

2

Espera Ocupada

Threads aguardando a liberação de um recurso bloqueado entram em um estado de espera ativa, consumindo ciclos de CPU.

3

Bloqueio e Desbloqueio

Mecanismos para adquirir e liberar o controle sobre recursos compartilhados, garantindo a coordenação entre threads.

Comunicação entre Threads

A comunicação entre threads é fundamental para a coordenação e colaboração entre diferentes fluxos de execução em um programa Java. Existem diversos mecanismos disponíveis para permitir que as threads troquem informações e sincronizem suas atividades.

3

Métodos

As principais formas de comunicação entre threads são: o uso de variáveis compartilhadas, a passagem de mensagens e a utilização de mecanismos de sincronização como semáforos e monitores.

2

Paradigmas

Existem dois paradigmas principais de comunicação entre threads: a comunicação síncrona, em que as threads bloqueiam até que a comunicação seja concluída, e a comunicação assíncrona, em que as threads podem continuar executando outras tarefas enquanto aguardam a resposta.

Um exemplo comum de comunicação entre threads é o uso de variáveis compartilhadas, onde uma thread produz dados que são consumidos por outra thread. Isso requer cuidados especiais para evitar condições de corrida e garantir a consistência dos dados.

Desafios e Boas Práticas

Concorrência e Sincronização

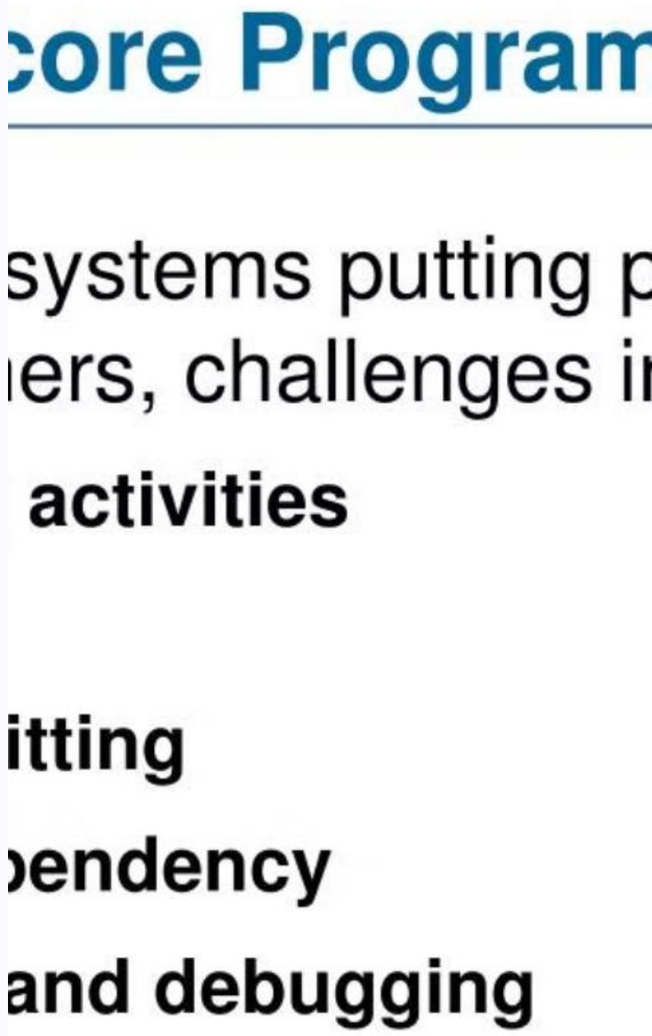
Gerenciar a concorrência e sincronização entre threads pode ser um desafio complexo, requerendo cuidado para evitar problemas como deadlocks e race conditions.

Escalabilidade e Desempenho

Com o aumento do número de cores e processadores, é crucial projetar aplicações multithreaded de forma escalável, maximizando o uso dos recursos disponíveis.

Depuração e Monitoramento

Bugs relacionados a threads podem ser difíceis de reproduzir e depurar. Ferramentas de monitoramento e depuração especializadas são essenciais.



Minha Primeira Thread

```
public class Threads_1 {

    public static void main(String[] args) {
        Thread t = Thread.currentThread();
        System.out.println(t.getName());
    }
}
```

Classe MeuRunnable

```
public class MeuRunnable implements Runnable {

    public void run(){
        System.out.println("Olá Mundo!");
    }
}
```

Classe Thread_1

```
public class Threads_1 {

    public static void main(String[] args) {
        Thread t = Thread.currentThread();
        System.out.println(t.getName());

        //Nova thread
        Thread t0 = new Thread(new MeuRunnable());
        t0.run();
    }
}
```

Classe MeuRunnable

```
public class MeuRunnable implements Runnable{

    public void run(){
        String name = Thread.currentThread().getName();
        System.out.println(name);
    }
}
```

Classe Thread_1

```
public class Threads_1 {
    public static void main(String[] args) {
        Thread t = Thread.currentThread();
        System.out.println(t.getName());

        //Nova thread
        Thread t0 = new Thread(new MeuRunnable());
        t0.run();
    }
}
```

Classe Thread_1

```
public class Threads_1 {
    public static void main(String[] args) {
        Thread t = Thread.currentThread();
        System.out.println(t.getName());

        //Nova thread
        Thread t0 = new Thread(new MeuRunnable());
        // t0.run(); // apenas execitando na mesma thread
        t0.start(); //executando em uma nova thread
    }
}
```

Classe Thread_1 com lambda

```
public class Threads_1 {
        public static void main(String[] args) {
                Thread t = Thread.currentThread();
                System.out.println(t.getName());

                //Nova thread
                Thread t0 = new Thread(new MeuRunnable());
                // t1.run(); // apenas execitando na mesma thread
                t0.start(); //executando em uma nova thread

                // Rinnable com lambda
                Thread t1 = new Thread(
                        () -> System.out.println("LP-III"));
                t1.start();
        }
}
```

```java
public class Threads_1 {
	public static void main(String[] args) {
		Thread t = Thread.currentThread();
		System.out.println(t.getName());
		Runnable meuRunnable = new MeuRunnable();

		//Nova thread
		Thread t1 = new Thread(meuRunnable);
		// t1.run(); // apenas execitando na mesma thread
		t1.start(); //executando em uma nova thread

		// Rinnable com lambda
		Thread t2 = new Thread(
			() -> System.out.println("LP-III"));
		t2.start();

		Thread t3 = new Thread(meuRunnable);
		t3.start();
	}
}
```

```java
public class Threads_1 {
	public static void main(String[] args) {
		Thread t = Thread.currentThread();
		System.out.println(t.getName());
		Runnable meuRunnable = new MeuRunnable();

		//Nova thread
		Thread t1 = new Thread(meuRunnable);
		// t1.run(); // apenas execitando na mesma thread

		// Rinnable com lambda
		Thread t2 = new Thread(
			() -> System.out.println("LP-III"));

		Thread t3 = new Thread(meuRunnable);

		t1.start(); //executando em uma nova thread
		t2.start();
		t3.start();
	}
}
```

Classe Synchronized_1 – Sincronização de Threads

```
public class Synchronized_1 {

        static int i = - 1;

        public static void main(String[] args) {
                MeuRunnable runnable = new MeuRunnable();

                Thread t0 = new Thread(runnable);
                Thread t1 = new Thread(runnable);
                Thread t2 = new Thread(runnable);
                Thread t3 = new Thread(runnable);
                Thread t4 = new Thread(runnable);

                t0.start();
                t1.start();
                t2.start();
                t3.start();
                t4.start();
        } ...
```

Classe Synchronized_1 – Sincronização de Threads

```
public class Synchronized_1 {

	…

	public static class MeuRunnable implements Runnable {
		@Override
		public void run() {
			i++;
		}
	}
}
```

Classe Synchronized_1 – Sincronização de Threads

```
public class Synchronized_1 {

	...

	public static class MeuRunnable implements Runnable {
		@Override
		public void run() {
			i++;
			String name = Thread.currentThread().getName();
			System.out.println(name + ":" + i);
		}
	}
}
```

Classe Synchronized_1 – Sincronização de Threads

```
public class Synchronized_1 {

	...

	public static class MeuRunnable implements Runnable {
		@Override
		public synchronized void run() {
			i++;
			String name = Thread.currentThread().getName();
			System.out.println(name + ":" + i);
		}
	}
}
```

Classe Synchronized_1 – Sincronização de Threads

```
public class Synchronized_1 {

        …

        public static class MeuRunnable implements Runnable {
                @Override
                public void run() {
                        synchronized(this){
                                i++;
                                String name = Thread.currentThread().getName();
                                System.out.println(name + ":" + i);
                        }
                }
        }
}
```

Classe Synchronized_1 – Sincronização de Threads

```
public class Synchronized_1 {
        ...
        public static class MeuRunnable implements Runnable {
                @Override
                public void run() {
                        synchronized(this){
                                i++;
                                String name = Thread.currentThread().getName();
                                System.out.println(name + ":" + i);
                        }
                        synchronized(this){
                                i++;
                                String name = Thread.currentThread().getName();
                                System.out.println(name + ":" + i);
                        }
                }
        }
}
```

Classe Synchronized_1 – Sincronização de Threads

```
public class Synchronized_1 {
        ...
         public static class MeuRunnable implements Runnable {
                static Object lock1 = new Object();
                static Object lock2 = new Object();

                @Override
                public void run() {
                        synchronized(lock1){
                                i++;
                                String name = Thread.currentThread().getName();
                                System.out.println(name + ":" + i);
                        }
                        ...
```

Classe Synchronized_1 – Sincronização de Threads

```
public class Synchronized_1 {
        …
         public static class MeuRunnable implements Runnable {
                static Object lock1 = new Object();
                static Object lock2 = new Object();

                @Override
                public void run() {
                        ...
                        synchronized(lock2){
                                i++;
                                String name = Thread.currentThread().getName();
                                System.out.println(name + ":" + i);
                        }
                }
        }
}
```

Classe Synchronized_1 – Sincronização de Threads

```
public class Synchronized_1 {
        ...

        public static void imprime() {
                i++;
                String name = Thread.currentThread().getName();
                System.out.println(name + ":" + i);
        }

        public static class MeuRunnable implements Runnable {
                @Override
                public void run() {
                        imprime();
                }
        }
}
```

Classe Synchronized_1 – Sincronização de Threads

```
public class Synchronized_1 {
        ...

        public static void imprime() {
                synchronized (Synchronized_1.class) {
                        i++;
                        String name = Thread.currentThread().getName();
                        System.out.println(name + ":" + i);
                }
        }

        public static class MeuRunnable implements Runnable {
                @Override
                public void run() {
                        imprime();
                }
        }
}
```

Na vida real como usar a sincronização?

Classe Synchronized_2 – Sincronização de Threads

```
public class Synchronized_2 {

        private static int i = 0;

        public static void main(String[] args) {
                MeuRunnable runnable = new MeuRunnable();

                Thread t0 = new Thread(runnable);
                Thread t1 = new Thread(runnable);
                Thread t2 = new Thread(runnable);
                Thread t3 = new Thread(runnable);
                Thread t4 = new Thread(runnable);

                t0.start();
                t1.start();
                t2.start();
                t3.start();
                t4.start();
        } ...
```

Classe Synchronized_2 – Sincronização de Threads

```
public class Synchronized_2 {
        ...
        public static class MeuRunnable implements Runnable {

                @Override
                public void run() {
                        int j;

                        synchronized (this) {
                                i++;
                                j = i * 2;
                        }

                        double jElevadoA100 = Math.pow(j, 100);
                        double sqrt = Math.sqrt(jElevadoA100);
                        System.out.println(sqrt);
                }
        }
}
```

Sincronizar Coleções

Classe SincronizarColecoes – Sincronização de Coleções

```
public class SincronizarColecoes {

    private static List<String> lista = new ArrayList<>();

    public static void main(String[] args) {
        MeuRunnable runnable = new MeuRunnable();
        Thread t0 = new Thread(runnable);
        Thread t1 = new Thread(runnable);
        Thread t2 = new Thread(runnable);
        t0.start();
        t1.start();
        t2.start();
        System.out.println(lista);
    }

    public static class MeuRunnable implements Runnable {

        public void run() {
            lista.add("LP-III");
        }
    }
}
```

Classe SincronizarColecoes – Sincronização de Coleções

```
public class SincronizarColecoes {

    private static List<String> lista = new ArrayList<>();

    public static void main(String[] args) {
        ...
    }

    public static class MeuRunnable implements Runnable {

        public void run() {
            lista.add("LP-III");
            String name = Thread.currentThread().getName();
            System.out.println(name + " inseriu na lista!");
        }
    }
}
```

Classe SincronizarColecoes – Sincronização de Coleções

```
public class SincronizarColecoes {

    private static List<String> lista = new ArrayList<>();

    public static void main(String[] args) {
        lista = Collections.synchronizedList(lista);
        ...
    }

    public static class MeuRunnable implements Runnable {

        public void run() {
            ...
        }
    }
}
```

Coleções para Concorrência

Classe ColecoesParaConcorrencia – Thread-Safe

```
public class ColecoesParaConcorrencia {

    private static List<String> lista = new CopyOnWriteArrayList<>();

    public static void main(String[] args) {
        MeuRunnable runnable = new MeuRunnable();
        Thread t0 = new Thread(runnable);
        Thread t1 = new Thread(runnable);
        Thread t2 = new Thread(runnable);
        t0.start();
        t1.start();
        t2.start();
        System.out.println(lista);
    }
    public static class MeuRunnable implements Runnable {
        public void run() {
            lista.add("LP-III");
            String name = Thread.currentThread().getName();
            System.out.println(name + " inseriu na lista!");
        }
    }
}
```

Classe ColecoesParaConcorrencia – Thread-Safe

```
public class ColecoesParaConcorrencia {

   // private static List<String> lista = new CopyOnWriteArrayList<>();
   private static Map<Integer, String> mapa = new ConcurrentHashMap<>();

   public static void main(String[] args) {
      ...
      // System.out.println(lista);
      System.out.println(mapa);

   }
   public static class MeuRunnable implements Runnable {
      public void run() {
         // lista.add("LP-III");
         mapa.put(new Random().nextInt(), "LP-III");
         String name = Thread.currentThread().getName();
         System.out.println(name + " inseriu na lista!");
      }
   }
}
```

Classe ColecoesParaConcorrencia – Thread-Safe uso de filas

```
public class ColecoesParaConcorrencia {

    // private static List<String> lista = new CopyOnWriteArrayList<>();
    // private static Map<Integer, String> mapa = new ConcurrentHashMap<>();
    private static BlockingQueue<String> fila = new LinkedBlockingQueue<>();

    public static void main(String[] args) {
        ...
        // System.out.println(lista);
        // System.out.println(mapa);
        System.out.println(fila);

    }
    public static class MeuRunnable implements Runnable {
        public void run() {
            // lista.add("LP-III");
            // mapa.put(new Random().nextInt(), "LP-III");
            fila.add("LP-III");
            String name = Thread.currentThread().getName();
            System.out.println(name + " inseriu na lista!");
        }
    }
}
```

Evitando Synchronized
Classes Atômicas

Classe ClassesAtômicas

```
public class ClassesAtomicas {

   static int i = - 1;

   public static void main(String[] args) {
      MeuRunnable runnable = new MeuRunnable();
      Thread t0 = new Thread(runnable);
      Thread t1 = new Thread(runnable);
      Thread t2 = new Thread(runnable);
      t0.start();
      t1.start();
      t2.start();
   }

   public static class MeuRunnable implements Runnable {
      @Override
      public void run() {
         i++;
         String name = Thread.currentThread().getName();
         System.out.println(name + ":" + i);
      }
   }
}
```

Classe ClassesAtomicas - Inteiro

```
public class ClassesAtomicas {
    static AtomicInteger i = new AtomicInteger(-1);

    public static void main(String[] args) {
        MeuRunnable runnable = new MeuRunnable();
        Thread t0 = new Thread(runnable);
        Thread t1 = new Thread(runnable);
        Thread t2 = new Thread(runnable);
        t0.start();
        t1.start();
        t2.start();
    }

    public static class MeuRunnable implements Runnable {
        @Override
        public void run() {
            String name = Thread.currentThread().getName();
            System.out.println(name + ":" + i.incrementAndGet());
        }
    }
}
```

Classe ClassesAtomicas - Lógico

```
public class ClassesAtomicas {
    // static AtomicInteger i = new AtomicInteger(-1);
    static AtomicBoolean b = new AtomicBoolean(false);

    public static void main(String[] args) {
        MeuRunnable runnable = new MeuRunnable();
        Thread t0 = new Thread(runnable);
        Thread t1 = new Thread(runnable);
        Thread t2 = new Thread(runnable);
        t0.start();
        t1.start();
        t2.start();
    }

    public static class MeuRunnable implements Runnable {
        @Override
        public void run() {
            String name = Thread.currentThread().getName();
            // System.out.println(name + ":" + i.incrementAndGet());
            System.out.println(name + ":" + b.compareAndExchange(false, true));
        }
    }
}
```

Classe ClassesAtomicas - Referencia

```
public class ClassesAtomicas {
    // static AtomicInteger i = new AtomicInteger(-1);
    // static AtomicBoolean b = new AtomicBoolean(false);
    static AtomicReference<Object> r = new AtomicReference<>(new Object());

    public static void main(String[] args) {
        MeuRunnable runnable = new MeuRunnable();
        Thread t0 = new Thread(runnable);
        Thread t1 = new Thread(runnable);
        Thread t2 = new Thread(runnable);
        t0.start();
        t1.start();
        t2.start();
    }
    public static class MeuRunnable implements Runnable {
        @Override
        public void run() {
            String name = Thread.currentThread().getName();
            // System.out.println(name + ":" + i.incrementAndGet());
            // System.out.println(name + ":" + b.compareAndExchange(false, true));
            System.out.println(name + ":" + r.getAndSet(new Object()));
        }
    }
}
```

Multithread – Em Espera

Classe Volatile

```
public class Volatile {

    private static int numero = 0;
    private static boolean preparado = false;

    private static class MeuRunnable implements Runnable {
        @Override
        public void run() {
            while (!preparado) {
                Thread.yield();
            }
            System.out.println("Número: " + numero);
        }
    }

    public static void main(String[] args) {
        Thread t0 = new Thread(new MeuRunnable());
        t0.start();
        numero = 42;
        preparado = true;
    }
}
```

Classe Volatile2 – O que está acontencendo?

```
public class Volatile2 {

    private static int numero = 0;
    private static boolean preparado = false;

    private static class MeuRunnable implements Runnable {
        @Override
        public void run() {
            while (!preparado) {
                Thread.yield();
            }
            if (numero != 42) {
                throw new IllegalStateException("LP-III");
            }
        }
    }

    public static void main(String[] args) {

        ...

    }
}
```

Classe Volatile2 – O que está acontencendo?

```
public class Volatile2 {
   ...
   public static void main(String[] args) {
      while (true) {
         Thread t0 = new Thread(new MeuRunnable());
         t0.start();
         Thread t1 = new Thread(new MeuRunnable());
         t1.start();
         Thread t2 = new Thread(new MeuRunnable());
         t2.start();
         numero = 42;
         preparado = true;

         while(t0.getState() != State.TERMINATED
             || t1.getState() != State.TERMINATED
             || t2.getState() != State.TERMINATED) {
            //espera
         }
         numero = 0;
         preparado = false;
      }
   }
}
```

Classe Volatile2 – Solução

```
public class Volatile2 {

    private static volatile int numero = 0;
    private static volatile boolean preparado = false;

    private static class MeuRunnable implements Runnable {

        ...

    }

    public static void main(String[] args) {

        ...

    }
}
```

Executores – Execução Simples

Classe Executors_SingleThread_Callable – Thread simples

```
public class Executors_SingleThread_Callable {

    public static void main(String[] args) {
        ExecutorService executor = Executors.newSingleThreadExecutor();
        executor.execute(new MeuRunnable());
    }

    public static class MeuRunnable implements Runnable {
        public void run() {
            String nome = Thread.currentThread().getName();
            System.out.println(nome + ": LP-III");
        }
    }
}
```

Classe Executors_SingleThread_Callable – Thread simples

```
public class Executors_SingleThread_Callable {

    public static void main(String[] args) {
        ExecutorService executor = Executors.newSingleThreadExecutor();
        executor.execute(new MeuRunnable());
        executor.shutdown();
    }

    public static class MeuRunnable implements Runnable {
        public void run() {
            String nome = Thread.currentThread().getName();
            System.out.println(nome + ": LP-III");
        }
    }
}
```

Classe Executors_SingleThread_Callable – Thread simples

```
public class Executors_SingleThread_Callable {

    public static void main(String[] args) throws InterruptedException {
        ExecutorService executor = null;
        try {
            executor = Executors.newSingleThreadExecutor();
            executor.execute(new MeuRunnable());
            executor.awaitTermination(5, TimeUnit.SECONDS);
        } catch (Exception e) {
            throw e;
        } finally {
            if (executor != null) {
                executor.shutdown();
            }
        }
    }
    ...
}
```

Classe Executors_SingleThread_Callable – Thread simples

```
public class Executors_SingleThread_Callable {

    public static void main(String[] args) throws InterruptedException {
        ExecutorService executor = null;
        try {
            executor = Executors.newSingleThreadExecutor();
            executor.execute(new MeuRunnable());
            executor.execute(new MeuRunnable());
            executor.execute(new MeuRunnable());
            Future<?> future = executor.submit(new MeuRunnable());
            System.out.println(future.isDone());
            executor.shutdown();
            executor.awaitTermination(10, TimeUnit.SECONDS);
            System.out.println(future.isDone());
        } catch (Exception e) {
            throw e;
        } finally {
            if (executor != null) {
                executor.shutdownNow();
            }
        }
    ...
}
```

Classe Executors_SingleThread_Callable – Thread simples

```
public class Executors_SingleThread_Callable {

    public static void main(String[] args) throws Exception {
        ExecutorService executor = null;
        try {
            executor = Executors.newSingleThreadExecutor();
            Future<String> future = executor.submit(new MeuCallable());
            System.out.println(future.isDone());
            System.out.println(future.get());
            // System.out.println(future.get(1, TimeUnit.SECONDS));
            System.out.println(future.isDone());
        } catch (Exception e) {
            throw e;
        } finally {
            if (executor != null) {
                executor.shutdownNow();
            }
        }
    }
    ...
}
```

Classe Executors_SingleThread_Callable – Thread simples

```
public class Executors_SingleThread_Callable {

    ...

    public static class MeuCallable implements Callable<String> {
        public String call() throws Exception {
            // Thread.sleep(1000);
            String nome = Thread.currentThread().getName();
            int nextInt = new Random().nextInt();
            return nome + ": LP-III " + nextInt;
        }
    }
}
```

Executores – Execução Multiplas

Classe Executors_MultiThread – Multiplas Excuções

```
public class Executors_MultiThread {

    public static void main(String[] args) throws Exception {
        ExecutorService executor = null;
        try {
            executor = Executors.newFixedThreadPool(4);
            Future<String> f1 = executor.submit(new Tarefa());
            Future<String> f2 = executor.submit(new Tarefa());
            Future<String> f3 = executor.submit(new Tarefa());
            System.out.println(f1.get());
            System.out.println(f2.get());
            System.out.println(f3.get());
            executor.shutdown();
        } catch (Exception e) {
            throw e;
        } finally {
            if (executor != null) {
                executor.shutdownNow();
            }
        }
    }
    ...
}
```

Classe Executors_MultiThread – Multiplas Excuções

```
public class Executors_MultiThread {

    ...

    public static class Tarefa implements Callable<String> {
        @Override
        public String call() throws Exception {
            String nome = Thread.currentThread().getName();
            int nextInt = new Random().nextInt();
            return nome + ": LP-III " + nextInt;
        }
    }
}
```

Classe Executors_MultiThread – Multiplas Excuções

```
public class Executors_MultiThread {

    public static void main(String[] args) throws Exception {
        ExecutorService executor = null;
        try {
            // executor = Executors.newFixedThreadPool(4);
            executor = Executors.newCachedThreadPool();
            Future<String> f1 = executor.submit(new Tarefa());
            Future<String> f2 = executor.submit(new Tarefa());
            Future<String> f3 = executor.submit(new Tarefa());
            System.out.println(f1.get());
            System.out.println(f2.get());
            System.out.println(f3.get());
            executor.shutdown();
        } catch (Exception e) {
            throw e;
        } finally {
            if (executor != null) {
                executor.shutdownNow();
            }
        }
    }
    ...
}
```

Classe Executors_MultiThread – Multiplas Excuções

```
public class Executors_MultiThread {

    ...

    public static class Tarefa implements Callable<String> {
        @Override
        public String call() throws Exception {
            String nome = Thread.currentThread().getName();
            int nextInt = new Random().nextInt();
            return nome + ": LP-III " + nextInt;
        }
    }
}
```

Classe Executors_MultiThread – Multiplas Excuções

```
public class Executors_MultiThread {

    public static void main(String[] args) throws Exception {
        ExecutorService executor = null;
        try {
            executor = Executors.newCachedThreadPool();
            Tarefa t1 = new Tarefa();
            Tarefa t2 = new Tarefa();
            Tarefa t3 = new Tarefa();
            Tarefa t4 = new Tarefa();
            List<Future<String>> list = executor.invokeAll(List.of(t1, t2, t3, t4));
            for (Future<String> future : list) {
                System.out.println(future.get());
            }
            executor.shutdown();
        } catch (Exception e) {
            throw e;
        } finally {
            if (executor != null) {
                executor.shutdownNow();
            }
        }
    }
    ...
}
```

Classe Executors_MultiThread – Multiplas Excuções

```
public class Executors_MultiThread {

    ...

    public static class Tarefa implements Callable<String> {
        @Override
        public String call() throws Exception {
            String nome = Thread.currentThread().getName();
            int nextInt = new Random().nextInt();
            return nome + ": LP-III " + nextInt;
        }
    }
}
```

Classe Executors_MultiThread – Multiplas Excuções

```
public class Executors_MultiThread {

    public static void main(String[] args) throws Exception {
        ExecutorService executor = null;
        try {
            executor = Executors.newCachedThreadPool();
            List<Tarefa> lista = new ArrayList<>();
            for (int i = 0; i < 10; i++) {
                lista.add(new Tarefa());
            }
            List<Future<String>> list = executor.invokeAll(lista);
            for (Future<String> future : list) {
                System.out.println(future.get());
            }
            executor.shutdown();
        } catch (Exception e) {
            throw e;
        } finally {
            if (executor != null) {
                executor.shutdownNow();
            }
        }
    }
    ...
}
```

Classe Executors_MultiThread – Multiplas Excuções

```
public class Executors_MultiThread {

    ...

    public static class Tarefa implements Callable<String> {
        @Override
        public String call() throws Exception {
            String nome = Thread.currentThread().getName();
            int nextInt = new Random().nextInt();
            return nome + ": LP-III " + nextInt;
        }
    }
}
```

Classe Executors_MultiThread – Multiplas Excuções

```
public class Executors_MultiThread {

    public static void main(String[] args) throws Exception {
        ExecutorService executor = null;
        try {
            executor = Executors.newCachedThreadPool();
            List<Tarefa> lista = new ArrayList<>();
            for (int i = 0; i < 10; i++) {
                lista.add(new Tarefa());
            }

            String string = executor.invokeAny(lista);
            System.out.println(string);

            executor.shutdown();
        } catch (Exception e) {
            throw e;
        } finally {
            if (executor != null) {
                executor.shutdownNow();
            }
        }
    }
    ...
}
```

Classe Executors_MultiThread – Multiplas Excuções

```
public class Executors_MultiThread {

    ...

    public static class Tarefa implements Callable<String> {
        @Override
        public String call() throws Exception {
            String nome = Thread.currentThread().getName();
            int nextInt = new Random().nextInt();
            return nome + ": LP-III " + nextInt;
        }
    }
}
```

Executores – Agendamento

Classe Executors_Scheduled – Agendamento de Execução

```
public class Executors_Scheduled {

    public static void main(String[] args) throws InterruptedException, ExecutionException {
        ScheduledExecutorService executor = Executors.newScheduledThreadPool(4);
        ScheduledFuture<String> future = executor.schedule(new Tarefa(), 2, TimeUnit.SECONDS);
        System.out.println(future.get());
        executor.shutdown();
    }

    public static class Tarefa implements Callable<String> {
        @Override
        public String call() throws Exception {
            String nome = Thread.currentThread().getName();
            int nextInt = new Random().nextInt();
            return nome + ": LP-III " + nextInt;
        }
    }
}
```

Classe Executors_Scheduled – Agendamento de Execução

```
public class Executors_Scheduled {

    public static void main(String[] args) throws InterruptedException, ExecutionException {
        ScheduledExecutorService executor = Executors.newScheduledThreadPool(4);
        executor.schedule(new Tarefa(), 2, TimeUnit.SECONDS);
        executor.shutdown();
    }

    public static class Tarefa implements Callable<String> {
        @Override
        public void run() {
            String nome = Thread.currentThread().getName();
            int nextInt = new Random().nextInt();
            System.out.println(nome + ": LP-III " + nextInt);
        }
    }
}
```

Classe Executors_Scheduled – Agendamento recorrente

```
public class Executors_Scheduled {

    public static void main(String[] args) throws InterruptedException, ExecutionException {
        ScheduledExecutorService executor = Executors.newScheduledThreadPool(4);
        executor.scheduleAtFixedRate(new Tarefa(), 0, 1, TimeUnit.SECONDS);
    }

    public static class Tarefa implements Runnable {
        @Override
        public void run() {
            String nome = Thread.currentThread().getName();
            int nextInt = new Random().nextInt();
            System.out.println(nome + ": LP-III " + nextInt);
        }
    }
}
```

Classe Executors_Scheduled – Agendamento recorrente

```
public class Executors_Scheduled {

    public static void main(String[] args) throws InterruptedException, ExecutionException {
        ScheduledExecutorService executor = Executors.newScheduledThreadPool(4);
        executor.scheduleAtFixedRate(new Tarefa(), 0, 1, TimeUnit.SECONDS);
    }

    public static class Tarefa implements Runnable {
        @Override
        public void run() {
            try {
                Thread.sleep(1000);
            } catch (Exception e) {
                e.printStackTrace();
            }
            String nome = Thread.currentThread().getName();
            int nextInt = new Random().nextInt();
            System.out.println(nome + ": LP-III " + nextInt);
        }
    }
}
```

Classe Executors_Scheduled – Agendamento recorrente

```
public class Executors_Scheduled {

    public static void main(String[] args) throws InterruptedException, ExecutionException {
        ScheduledExecutorService executor = Executors.newScheduledThreadPool(4);
        executor.scheduleWithFixedDelay(new Tarefa(), 0, 1, TimeUnit.SECONDS);
    }

    public static class Tarefa implements Runnable {
        @Override
        public void run() {
            try {
                Thread.sleep(1000);
            } catch (Exception e) {
                e.printStackTrace();
            }
            String nome = Thread.currentThread().getName();
            int nextInt = new Random().nextInt();
            System.out.println(nome + ": LP-III " + nextInt);
        }
    }
}
```

Aguardando por outra Thread

Classe CyclicBarrier – Aguardando por outra execução

```
public class CyclicBarrier_1 {

   //432*3 + 3^14 + 45*127/12 = ?
   public static void main(String[] args) {
      ExecutorService executor = Executors.newFixedThreadPool(3);

      Runnable r1 = () ->{
         System.out.println(432d*3d);
         System.out.println("Terminei a execução");
      };
      Runnable r2 = () ->{
         System.out.println(Math.pow(3, 14));
         System.out.println("Terminei a execução");
      };
      Runnable r3 = () ->{
         System.out.println(45d*127d/12d);
         System.out.println("Terminei a execução");
      };
      executor.submit(r1);
      executor.submit(r2);
      executor.submit(r3);
      executor.shutdown();
   }
}
```

Classe CyclicBarrier – Aguardando por outra execução

```
public class CyclicBarrier_1 {

    //432*3 + 3^14 + 45*127/12 = ?
    public static void main(String[] args) {
        CyclicBarrier cycleBarrier = new CyclicBarrier(3);

        ExecutorService executor = Executors.newFixedThreadPool(3);

        Runnable r1 = () ->{
            System.out.println(432d*3d);
            await(cycleBarrier);
            System.out.println("Terminei a execução");
        };
        Runnable r2 = () ->{
            System.out.println(Math.pow(3, 14));
            await(cycleBarrier);
            System.out.println("Terminei a execução");
        };
        Runnable r3 = () ->{
            System.out.println(45d*127d/12d);
            await(cycleBarrier);
            System.out.println("Terminei a execução");
        };
        ...
```

Classe CyclicBarrier – Aguardando por outra execução

```
public class CyclicBarrier_1 {

    //432*3 + 3^14 + 45*127/12 = ?
    public static void main(String[] args) {
        ...
        executor.submit(r1);
        executor.submit(r2);
        executor.submit(r3);
        executor.shutdown();
    }

    private static void await(CyclicBarrier cycleBarrier) {
        try {
            cycleBarrier.await();
        } catch (InterruptedException | BrokenBarrierException e) {
            Thread.currentThread().interrupt();
            e.printStackTrace();
        }
    }
}
```

Classe CyclicBarrier – Coletando os resultados

```
public class CyclicBarrier_2 {

    private static BlockingQueue<Double> resultados = new LinkedBlockingQueue<>();

    //432*3 + 3^14 + 45*127/12 = ?
    public static void main(String[] args) {

        Runnable finalizacao = () -> {
            System.out.println("Somando tudo.");
            double resultadoFinal = 0;
            resultadoFinal += resultados.poll();
            resultadoFinal += resultados.poll();
            resultadoFinal += resultados.poll();
            System.out.println("Processaimento finalizado.
                        O resultado final é: " + resultadoFinal);
        };

        CyclicBarrier cycleBarrier = new CyclicBarrier(3, finalizacao);

        ExecutorService executor = Executors.newFixedThreadPool(3);

        ...
```

Classe CyclicBarrier – Coletando os resultados

```
public class CyclicBarrier_2 {

    private static BlockingQueue<Double> resultados = new LinkedBlockingQueue<>();

    //432*3 + 3^14 + 45*127/12 = ?
    public static void main(String[] args) {
        ...

        Runnable r1 = () ->{
            resultados.add(432d*3d);
            await(cycleBarrier);
        };

        Runnable r2 = () ->{
            resultados.add(Math.pow(3, 14));
            await(cycleBarrier);
        };

        Runnable r3 = () ->{
            resultados.add(45d*127d/12d);
            await(cycleBarrier);
        };

        ...
```

Classe CyclicBarrier – Coletando os resultados

```
public class CyclicBarrier_2 {

    private static BlockingQueue<Double> resultados = new LinkedBlockingQueue<>();

    //432*3 + 3^14 + 45*127/12 = ?
    public static void main(String[] args) {
        ...

        executor.submit(r1);
        executor.submit(r2);
        executor.submit(r3);
        executor.shutdown();

    }

    private static void await(CyclicBarrier cycleBarrier) {
        try {
            cycleBarrier.await();
        } catch (InterruptedException | BrokenBarrierException e) {
            Thread.currentThread().interrupt();
            e.printStackTrace();
        }
    }
}
```

Classe CyclicBarrier – executando em ciclos

```
public class CyclicBarrier_3 {

    private static BlockingQueue<Double> resultados = new LinkedBlockingQueue<>();
    private static ExecutorService executor = null;
    private static Runnable r1 = null;
    private static Runnable r2 = null;
    private static Runnable r3 = null;
    private static double resultadoFinal = 0;

    //432*3 + 3^14 + 45*127/12 = ?
    public static void main(String[] args) {

        Runnable sumarizacao = () -> {
            System.out.println("Somando tudo.");
            resultadoFinal += resultados.poll();
            resultadoFinal += resultados.poll();
            resultadoFinal += resultados.poll();
            System.out.println("Processaimento finalizado.
                        O resultado final é: " + resultadoFinal);
            System.out.println("------------------------------");
            restart();
        };
        ...
```

Classe CyclicBarrier – executando em ciclos

```
public class CyclicBarrier_3 {
   ...
   public static void main(String[] args) {
      ...
      CyclicBarrier cycleBarrier = new CyclicBarrier(3, sumarizacao);
      executor = Executors.newFixedThreadPool(3);

      r1 = () ->{
         resultados.add(432d*3d);
         await(cycleBarrier);
      };

      r2 = () ->{
         resultados.add(Math.pow(3, 14));
         await(cycleBarrier);
      };

      r3 = () ->{
         resultados.add(45d*127d/12d);
         await(cycleBarrier);
      };

      restart();
   }
   ...
```

Classe CyclicBarrier – executando em ciclos

```
public class CyclicBarrier_3 {
    ...

    private static void await(CyclicBarrier cycleBarrier) {
        try {
            cycleBarrier.await();
        } catch (InterruptedException | BrokenBarrierException e) {
            Thread.currentThread().interrupt();
            e.printStackTrace();
        }
    }

     private static void restart(){
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        executor.submit(r1);
        executor.submit(r2);
        executor.submit(r3);
    }
}
```

Classe CyclicBarrier – executando em ciclos

```
public class CyclicBarrier_3 {

    private static BlockingQueue<Double> resultados = new LinkedBlockingQueue<>();
    private static ExecutorService executor = null;
    private static Runnable r1 = null;
    private static Runnable r2 = null;
    private static Runnable r3 = null;
    private static double resultadoFinal = 0;

    //432*3 + 3^14 + 45*127/12 = ?
    public static void main(String[] args) {

        Runnable sumarizacao = () -> {
            System.out.println("Somando tudo.");
            resultadoFinal += resultados.poll();
            resultadoFinal += resultados.poll();
            resultadoFinal += resultados.poll();
            System.out.println("Processaimento finalizado.
                               O resultado final é: " + resultadoFinal);
            System.out.println("------------------------------");
            // restart();
        };

        ...
```

Classe CyclicBarrier – executando em ciclos

```
public class CyclicBarrier_3 {
   ...
   public static void main(String[] args) {
      ...
      CyclicBarrier cycleBarrier = new CyclicBarrier(3, sumarizacao);

      executor = Executors.newFixedThreadPool(3);

      r1 = () ->{
         while (true) {
            resultados.add(432d*3d);
            await(cycleBarrier);
         }
      };

      r2 = () ->{
         while (true) {
            resultados.add(Math.pow(3, 14));
            await(cycleBarrier);
         }
      };

      ...
```

Classe CyclicBarrier – executando em ciclos

```
public class CyclicBarrier_3 {
   ...
   public static void main(String[] args) {
      ...
      r3 = () ->{
         while (true) {
            resultados.add(45d*127d/12d);
            await(cycleBarrier);
         }
      };

      restart();
   }

    private static void await(CyclicBarrier cycleBarrier) {
       try {
          cycleBarrier.await();
       } catch (InterruptedException | BrokenBarrierException e) {
          Thread.currentThread().interrupt();
          e.printStackTrace();
       }
    }

    ...
```

Classe CyclicBarrier – executando em ciclos

```
public class CyclicBarrier_3 {
    ...
    private static void restart(){
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        executor.submit(r1);
        executor.submit(r2);
        executor.submit(r3);
    }
}
```

Classe CyclicBarrier – executando em ciclos

```
public class CyclicBarrier_3 {

    private static BlockingQueue<Double> resultados = new LinkedBlockingQueue<>();
    private static ExecutorService executor = null;
    private static Runnable r1 = null;
    private static Runnable r2 = null;
    private static Runnable r3 = null;
    private static double resultadoFinal = 0;

    //432*3 + 3^14 + 45*127/12 = ?
    public static void main(String[] args) {

        Runnable sumarizacao = () -> {
            System.out.println("Somando tudo.");
            resultadoFinal += resultados.poll();
            resultadoFinal += resultados.poll();
            resultadoFinal += resultados.poll();
            System.out.println("Processaimento finalizado.
                               O resultado final é: " + resultadoFinal);
            System.out.println("------------------------------");
            // restart();
        };

        ...
```

Classe CyclicBarrier – executando em ciclos

```
public class CyclicBarrier_3 {
   ...
   public static void main(String[] args) {
      ...
      CyclicBarrier cycleBarrier = new CyclicBarrier(3, sumarizacao);

      executor = Executors.newFixedThreadPool(3);

      r1 = () ->{
         while (true) {
            resultados.add(432d*3d);
            await(cycleBarrier);
            sleep();
         }
      };

      r2 = () ->{
         while (true) {
            resultados.add(Math.pow(3, 14));
            await(cycleBarrier);
            sleep();
         }
      };
      ...
```

Classe CyclicBarrier – executando em ciclos

```
public class CyclicBarrier_3 {
   ...
   public static void main(String[] args) {
      ...
      r3 = () ->{
         while (true) {
            resultados.add(45d*127d/12d);
            await(cycleBarrier);
            sleep();
         }
      };

      restart();
   }

   private static void await(CyclicBarrier cycleBarrier) {
      try {
         cycleBarrier.await();
      } catch (InterruptedException | BrokenBarrierException e) {
         Thread.currentThread().interrupt();
         e.printStackTrace();
      }
   }
   ...
```

Classe CyclicBarrier – executando em ciclos

```
public class CyclicBarrier_3 {
    ...

    private static void restart(){
        sleep();
        executor.submit(r1);
        executor.submit(r2);
        executor.submit(r3);
    }

    private static void sleep(){
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Executando Threads após uma quantidade específica de execuções

Classe CountDownLatch

```
public class CountDownLatch_1 {

    private static volatile int i = 0;

    public static void main(String[] args) {
        ScheduledExecutorService executor = Executors.newScheduledThreadPool(3);

        Runnable r1 = () -> {
            int j = new Random().nextInt();
            int x = i * j;
            System.out.println(i + " x " + j + " = " + x);
        };
        executor.scheduleAtFixedRate(r1, 0, 1, TimeUnit.SECONDS);

        while (true) {
            sleep();
            i = new Random().nextInt();
        }
    }
    ...
```

Classe CountDownLatch

```
public class CountDownLatch_1 {

    ...
    public static void sleep(){
        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Classe CountDownLatch

```
public class CountDownLatch_1 {

    private static volatile int i = 0;
    private static CountDownLatch latch = new CountDownLatch(3);

    public static void main(String[] args) {
        ScheduledExecutorService executor = Executors.newScheduledThreadPool(3);

        Runnable r1 = () -> {
            int j = new Random().nextInt();
            int x = i * j;
            System.out.println(i + " x " + j + " = " + x);
            latch.countDown();
        };
        executor.scheduleAtFixedRate(r1, 0, 1, TimeUnit.SECONDS);

        while (true) {
            await();
            i = new Random().nextInt();
        }
    }
    ...
```

Classe CountDownLatch

```
public class CountDownLatch_1 {

    ...
    public static void await(){
        try {
            latch.await();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Classe CountDownLatch

```
public class CountDownLatch_1 {

    private static volatile int i = 0;
    private static CountDownLatch latch = new CountDownLatch(3);

    public static void main(String[] args) {
        ScheduledExecutorService executor = Executors.newScheduledThreadPool(3);

        Runnable r1 = () -> {
            int j = new Random().nextInt();
            int x = i * j;
            System.out.println(i + " x " + j + " = " + x);
            latch.countDown();
        };
        executor.scheduleAtFixedRate(r1, 0, 1, TimeUnit.SECONDS);

        while (true) {
            await();
            i = new Random().nextInt();
            latch = new CountDownLatch(3);
        }
    }
    ...
```

Classe CountDownLatch

```
public class CountDownLatch_1 {

    ...
    public static void await(){
        try {
            latch.await();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Classe CountDownLatch – Vários Await

```
public class CountDownLatch_2 {

    private static volatile int i = 0;
    private static CountDownLatch latch = new CountDownLatch(3);

    public static void main(String[] args) {

        ScheduledExecutorService executor = Executors.newScheduledThreadPool(4);

        Runnable r1 = () -> {
            int j = new Random().nextInt();
            int x = i * j;
            System.out.println(i + " x " + j + " = " + x);
            latch.countDown();
        };
        Runnable r2 = () -> {
            await();
            i = new Random().nextInt(100);
        };
        Runnable r3 = () -> {
            await();
            latch = new CountDownLatch(3);
        };
        ...
```

Classe CountDownLatch – Vários Await

```
public class CountDownLatch_2 {
   ...
   public static void main(String[] args) {
      ...
      Runnable r4 = () -> {
         await();
         System.out.println("Terminei a execução! Vamos Começar Novamente");
      };

      executor.scheduleAtFixedRate(r1, 0, 1, TimeUnit.SECONDS);
      executor.scheduleWithFixedDelay(r2, 0, 1, TimeUnit.SECONDS);
      executor.scheduleWithFixedDelay(r3, 0, 1, TimeUnit.SECONDS);
      executor.scheduleWithFixedDelay(r4, 0, 1, TimeUnit.SECONDS);

   }

   public static void await(){
     try {
       latch.await();
     } catch (InterruptedException e) {
       e.printStackTrace();
     }
   }
}
```

Limitar a execução de várias Threads em um trecho de código
Semáfaros

Classe Semaphore

```
public class Semaphore_1 {

   public static void main(String[] args) {
      ExecutorService executor = Executors.newCachedThreadPool();

      Runnable r1 = () -> {
         String name = Thread.currentThread().getName();
         int usuario = new Random().nextInt(10000);
         System.out.println("Usuário " + usuario " matriculou-se em LP-III! " +
                           + "Usando a thread " + name);
      };

      for (int i = 0; i < 500; i++) {
         executor.execute(r1);
      }

      executor.shutdown();
   }
}
```

Classe Semaphore

```
public class Semaphore_1 {

    private static final Semaphore semaphore = new Semaphore(3);

    public static void main(String[] args) {
        ExecutorService executor = Executors.newCachedThreadPool();

        Runnable r1 = () -> {
            String name = Thread.currentThread().getName();
            int usuario = new Random().nextInt(10000);
            try {
                semaphore.acquire();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                e.printStackTrace();
            }
            System.out.println("Usuário " + usuario " matriculou-se em LP-III! " +
                               + "Usando a thread " + name);
        };

        ...
```

Classe Semaphore

```
public class Semaphore_1 {

   ...

   public static void main(String[] args) {

      ...

      for (int i = 0; i < 500; i++) {
         executor.execute(r1);
      }

      executor.shutdown();
   }
}
```

Classe Semaphore

```
public class Semaphore_1 {

    private static final Semaphore semaphore = new Semaphore(3);

    public static void main(String[] args) {
        ExecutorService executor = Executors.newCachedThreadPool();

        Runnable r1 = () -> {
            String name = Thread.currentThread().getName();
            int usuario = new Random().nextInt(10000);
            acquire();
            System.out.println("Usuário " + usuario + " matriculou-se em LP-III!"
                               + "Usando a thread " + name);
            sleep();
            semaphore.release();
        };

        for (int i = 0; i < 500; i++) {
            executor.execute(r1);
        }

        executor.shutdown();

    }
    ...
```

Classe Semaphore

```
public class Semaphore_1 {

    ...

    public static void acquire(){
        try {
            semaphore.acquire();
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            e.printStackTrace();
        }
    }

    public static void sleep(){
        try {
            int tempoEspera = new Random().nextInt(6);
            tempoEspera++;
            Thread.sleep(1000 * tempoEspera);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            e.printStackTrace();
        }
    }
}
```

Classe Semaphore - Ver quantas threads faltam executar

```
public class Semaphore_2 {

    private static final Semaphore semaphore = new Semaphore(3);
    private static AtomicInteger qtd = new AtomicInteger(0);

    public static void main(String[] args) {
        ScheduledExecutorService executor = Executors.newScheduledThreadPool(501);

        Runnable r1 = () -> {
            String name = Thread.currentThread().getName();
            int usuario = new Random().nextInt(10000);
            boolean conseguiu = false;
            qtd.incrementAndGet();
            while(!conseguiu){
                conseguiu = tryAcquire();
            }
            qtd.decrementAndGet();
            System.out.println("Usuário " + usuario + " matriculou-se em LP-III! " +
                               + " Usando a thread " + name);
            sleep();
            semaphore.release();
        };
        ...
```

Classe Semaphore - Ver quantas threads faltam executar

```
public class Semaphore_2 {
    ...
    public static void main(String[] args) {
        ...
        Runnable r2 = () -> {
            System.out.println(qtd.get());
        };

        for (int i = 0; i < 500; i++) {
            executor.execute(r1);
        }
        executor.scheduleAtFixedRate(r2, 0, 100, TimeUnit.MILLISECONDS);
    }

    public static boolean tryAcquire(){
        try {
            return semaphore.tryAcquire(1, TimeUnit.SECONDS);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            e.printStackTrace();
            return false;
        }
    }
    ...
```

Classe Semaphore - Ver quantas threads faltam executar

```
public class Semaphore_2 {
    ...

    public static void sleep(){
        try {
            int tempoEspera = new Random().nextInt(6);
            tempoEspera++;
            Thread.sleep(1000 * tempoEspera);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            e.printStackTrace();
        }
    }
}
```

Travando a execução de Threads
Classes de Locks

Classe ReentrantLock - Sincronizando a execução

```
public class ReentrantLock_1 {

    private static int i = -1;

    public static void main(String[] args) {

        ExecutorService executor = Executors.newCachedThreadPool();

        Runnable r1 = () -> {
            String name = Thread.currentThread().getName();
            i++;
            System.out.println(name + " lendo o incremento " + i);
        };

        for (int i = 0; i < 6; i++) {
            executor.execute(r1);
        }

        executor.shutdown();
    }
}
```

Classe ReentrantLock - Sincronizando a execução

```
public class ReentrantLock_1 {

    private static int i = -1;
    private static Lock lock = new ReentrantLock();

    public static void main(String[] args) {
        ExecutorService executor = Executors.newCachedThreadPool();

        Runnable r1 = () -> {
            lock.lock();
            //boolean conseguiu = lock.tryLock();
            //boolean conseguiu = lock.tryLock(1, TimeUnit.SECONDS);
            String name = Thread.currentThread().getName();
            i++;
            System.out.println(name + " lendo o incremento " + i);
            lock.unlock();
        };
        for (int i = 0; i < 6; i++) {
            executor.execute(r1);
        }
        executor.shutdown();
    }
}
```

Classe ReentrantReadWriteLock - Sincronizando a execução

```
public class ReentrantReadWriteLock_1 {

    private static int i = -1;
    private static ReadWriteLock lock = new ReentrantReadWriteLock();

    public static void main(String[] args) {

        ExecutorService executor = Executors.newCachedThreadPool();

        Runnable r1 = () -> {
            Lock writeLock = lock.writeLock();
            writeLock.lock();
            String name = Thread.currentThread().getName();
            System.out.println(name + " - Escrevendo: " + i);
            i++;
            System.out.println(name + " - Escrito: " + i);
            writeLock.unlock();
        };

        ...
```

Classe ReentrantReadWriteLock - Sincronizando a execução

```
public class ReentrantReadWriteLock_1 {

    private static int i = -1;
    private static ReadWriteLock lock = new ReentrantReadWriteLock();

    public static void main(String[] args) {
        ...

        Runnable r2 = () -> {
            Lock readLock = lock.readLock();
            readLock.lock();
            System.out.println("Lendo: " + i);
            System.out.println("Lido: " + i);
            readLock.unlock();
        };

        for (int i = 0; i < 6; i++) {
            executor.execute(r1);
            executor.execute(r2);
        }

        executor.shutdown();
    }
}
```

Trocando dados entre duas Threads

Classe SynchronousQueue - Trocando informações

```
public class SynchronousQueue_1 {

    private static final SynchronousQueue<String> fila = new SynchronousQueue<>();

    public static void main(String[] args) {
        ExecutorService executor = Executors.newCachedThreadPool();
        Runnable r1 = () -> {
            put();
            System.out.println("Escreveu na fila!");
        };
        executor.execute(r1);
        executor.shutdown();
    }

    private static void put() {
        try {
            fila.put("LP-III");
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            e.printStackTrace();
        }
    }
}
```

Classe SynchronousQueue - Trocando informações

```
public class SynchronousQueue_1 {

    private static final SynchronousQueue<String> fila = new SynchronousQueue<>();

    public static void main(String[] args) {
        ExecutorService executor = Executors.newCachedThreadPool();

        Runnable r1 = () -> {
            put();
            System.out.println("Escreveu na fila!");
        };

        Runnable r2 = () -> {
            String msg = take();
            System.out.println("Pegou da fila! " + msg);
        };

        executor.execute(r1);
        executor.execute(r2);

        executor.shutdown();
    }
    ...
```

Classe SynchronousQueue - Trocando informações

```
public class SynchronousQueue_1 {
    ...
    private static String take() {
        try {
            return fila.take();
            // return fila.poll(timeout, unit);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            e.printStackTrace();
            return "Exceção!";
        }
    }

    private static void put() {
        try {
            fila.put("LP-III");
            // fila.offer(e, timeout, unit);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            e.printStackTrace();
        }
    }
}
```

Classe Exchanger - Trocando informações

```
public class Exchanger_1 {

    private static final Exchanger<String> EXCHANGER = new Exchanger<>();

    public static void main(String[] args) {
        ExecutorService executor = Executors.newCachedThreadPool();

        Runnable r1 = () -> {
            String msg = "Toma isso!";
            String retorno = exchange(msg);
            System.out.println(retorno);
        };

        Runnable r2 = () -> {
            String msg = "Obrigado!";
            String retorno = exchange(msg);
            System.out.println(retorno);
        };

        executor.execute(r1);
        executor.execute(r2);

        executor.shutdown();
    }
...
```

Classe Exchanger - Trocando informações

```
public class Exchanger_1 {

    ...

    private static String exchange(String msg) {
        try {
            return EXCHANGER.exchange(msg);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            e.printStackTrace();
            return "Exceção";
        }
    }
}
```

Classe Exchanger - Trocando informações

```
public class Exchanger_1 {

    private static final Exchanger<String> EXCHANGER = new Exchanger<>();

    public static void main(String[] args) {

        ExecutorService executor = Executors.newCachedThreadPool();

        Runnable r1 = () -> {
            String name = Thread.currentThread().getName();
            System.out.println(name + " toma isso");
            String msg = "Toma isso!";
            String retorno = exchange(msg);
            System.out.println(name + " - " + retorno);
        };

        Runnable r2 = () -> {
            String name = Thread.currentThread().getName();
            System.out.println(name + " obrigado");
            String msg = "Obrigado!";
            String retorno = exchange(msg);
            System.out.println(name + " - " + retorno);
        };
        ...
```

Classe Exchanger - Trocando informações

```
public class Exchanger_1 {
    ...
    public static void main(String[] args) {
        ...
        executor.execute(r1);
        executor.execute(r2);

        executor.shutdown();
    }

    private static String exchange(String msg) {
        try {
            return EXCHANGER.exchange(msg);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            e.printStackTrace();
            return "Exceção";
        }
    }
}
```

MultiThreads
Produtor vs Consumidor

Classe ProdutorConsumidor - Condição de corrida e Deadlock

```
public class ProdutorConsumidor_1 {

    private static final List<Integer> lista = new ArrayList<>(5);
    private static boolean produzindo = true;
    private static boolean consumindo = true;

    public static void main(String[] args) {
        Thread produtor = new Thread(() -> {
            while (true) {
                try {
                    simulaProcessamento();
                    if (produzindo) {
                        System.out.println("Produzindo");
                        int numero = new Random().nextInt(10000);
                        lista.add(numero);
                        if (lista.size() == 5) {
                            System.out.println("Pausando produtor.");
                            produzindo = false;
                        }
                        if (lista.size() == 1) {
                            System.out.println("Iniciando consumidor.");
                            consumindo = true;
                        }
                    ...
```

Classe ProdutorConsumidor - Condição de corrida e Deadlock

```
public class ProdutorConsumidor_1 {
    ...
    public static void main(String[] args) {

        Thread produtor = new Thread(() -> {
            while (true) {
                try {
                    ...
                    } else {
                        System.out.println("!!! Produtor dormindo!");
                    }
                } catch (Exception e) {
                    System.out.println(e.getMessage());
                }
            }
        });

        Thread consumidor = new Thread(() -> {
            while (true) {
                try {
                    simulaProcessamento();
                    if (consumindo) {
                        System.out.println("Consumindo");
                        ...
```

Classe ProdutorConsumidor - Condição de corrida e Deadlock

```
public class ProdutorConsumidor_1 {
    ...
    public static void main(String[] args) {
        ...
        Thread consumidor = new Thread(() -> {
            while (true) {
                try {
                    ...
                    if (consumindo) {
                        ...
                        Optional<Integer> numero = lista.stream().findFirst();
                        numero.ifPresent(n -> {
                            lista.remove(n);
                        });
                        if (lista.size() == 0) {
                            System.out.println("Pausando consumidor.");
                            consumindo = false;
                        }
                        if (lista.size() == 4) {
                            System.out.println("Iniciando produtor.");
                            produzindo = true;
                        }
                    ...
```

Classe ProdutorConsumidor - Condição de corrida e Deadlock

```
public class ProdutorConsumidor_1 {
    ...
    public static void main(String[] args) {
        ...
        Thread consumidor = new Thread(() -> {
            while (true) {
                try {
                    ...
                    } else {
                        System.out.println("??? Consumidor dormindo!");
                    }
                } catch (Exception e) {
                    System.out.println(e.getMessage());
                }
            }
        });

        Janelas.monitore(() -> String.valueOf(lista.size()));
        produtor.start();
        consumidor.start();
    }
    ...
```

Classe ProdutorConsumidor - Condição de corrida e Deadlock

```
public class ProdutorConsumidor_1 {

    ...

    private static final void simulaProcessamento() {
        int tempo = new Random().nextInt(40);
        try {
            Thread.sleep(tempo);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            e.printStackTrace();
        }
    }
}
```

Classe ProdutorConsumidor - Região crítica e Exclusão mútua

```
public class ProdutorConsumidor_2 {

    private static final BlockingQueue<Integer> fila = new LinkedBlockingDeque<>(5);
    private static volatile boolean produzindo = true;
    private static volatile boolean consumindo = true;
    private static final Lock lock = new ReentrantLock();

    public static void main(String[] args) {

        Thread produtor = new Thread(() -> {
            while (true) {
                try {
                    simulaProcessamento();
                    if (produzindo) {
                        lock.lock();
                        System.out.println("Produzindo");
                        int numero = new Random().nextInt(10000);
                        fila.add(numero);
                        if (fila.size() == 5) {
                            System.out.println("Pausando produtor.");
                            produzindo = false;
                        }
                        ...
```

Classe ProdutorConsumidor - Região crítica e Exclusão mútua

```
public class ProdutorConsumidor_2 {
    ...
    public static void main(String[] args) {

        Thread produtor = new Thread(() -> {
            while (true) {
                try {
                    ...
                    if (produzindo) {
                        ...
                        if (fila.size() == 1) {
                            System.out.println("Iniciando consumidor.");
                            consumindo = true;
                        }
                        lock.unlock();
                    } else {
                        System.out.println("!!! Produtor dormindo!");
                    }
                } catch (Exception e) {
                    System.out.println(e.getMessage());
                }
            }
        });
        ...
```

Classe ProdutorConsumidor - Região crítica e Exclusão mútua

```
public class ProdutorConsumidor_2 {
    ...
    public static void main(String[] args) {
        ...
        Thread consumidor = new Thread(() -> {
            while (true) {
                try {
                    simulaProcessamento();
                    if (consumindo) {
                        lock.lock();
                        System.out.println("Consumindo");
                        Optional<Integer> numero = fila.stream().findFirst();
                        numero.ifPresent(n -> {
                            fila.remove(n);
                        });
                        if (fila.size() == 0) {
                            System.out.println("Pausando consumidor.");
                            consumindo = false;
                        }
                        if (fila.size() == 4) {
                            System.out.println("Iniciando produtor.");
                            produzindo = true;
                        }
                        ...
```

Classe ProdutorConsumidor - Região crítica e Exclusão mútua

```
public class ProdutorConsumidor_2 {
    ...
    public static void main(String[] args) {
        ...
        Thread consumidor = new Thread(() -> {
            while (true) {
                try {
                    ...
                    if (consumindo) {
                        ...
                        lock.unlock();
                    } else {
                        System.out.println("??? Consumidor dormindo!");
                    }
                } catch (Exception e) {
                    System.out.println(e.getMessage());
                }
            }
        });
        Janelas.monitore(() -> String.valueOf(fila.size()));
        produtor.start();
        consumidor.start();
    }
    ...
```

Classe ProdutorConsumidor - Região crítica e Exclusão mútua

```
public class ProdutorConsumidor_2 {
    ...
    private static final void simulaProcessamento() {
        int tempo = new Random().nextInt(40);
        try {
            Thread.sleep(tempo);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            e.printStackTrace();
        }
    }
}
```

Classe ProdutorConsumidor - Usando as ferramentas certas

```
public class ProdutorConsumidor_3 {

    private static final BlockingQueue<Integer> fila = new LinkedBlockingDeque<>(5);

    public static void main(String[] args) {

        Runnable produtor = () -> {
            simulaProcessamento();
            System.out.println("Produzindo");
            int numero = new Random().nextInt(10000);
            try {
                fila.put(numero);
                System.out.println(numero);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                e.printStackTrace();
            }
        };
        ...
```

Classe ProdutorConsumidor - Usando as ferramentas certas

```
public class ProdutorConsumidor_3 {
    ...
    public static void main(String[] args) {
        ...
        Runnable consumidor = () -> {
            simulaProcessamento();
            System.out.println("Consumindo");
            try {
                Integer take = fila.take();
                System.out.println(take);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                e.printStackTrace();
            }
        };

        Janelas.monitore(() -> String.valueOf(fila.size()));
        ScheduledExecutorService executor = Executors.newScheduledThreadPool(2);
        executor.scheduleWithFixedDelay(produtor, 0, 10, TimeUnit.MILLISECONDS);
        executor.scheduleWithFixedDelay(consumidor, 0, 10, TimeUnit.MILLISECONDS);
    }

    ...
```

Classe ProdutorConsumidor - Usando as ferramentas certas

```
public class ProdutorConsumidor_3 {

   ...

   private static final void simulaProcessamento() {
      int tempo = new Random().nextInt(40);
      try {
         Thread.sleep(tempo);
      } catch (InterruptedException e) {
         Thread.currentThread().interrupt();
         e.printStackTrace();
      }
   }
```

Classe ProdutorConsumidor - Usando as ferramentas certas

```
public class ProdutorConsumidor_3 {

    private static final BlockingQueue<Integer> fila = new LinkedBlockingDeque<>(5);

    public static void main(String[] args) {

        Runnable produtor = () -> {
            // simulaProcessamento();
            simulaProcessamentoLento();
            System.out.println("Produzindo");
            int numero = new Random().nextInt(10000);
            try {
                fila.put(numero);
                System.out.println(numero);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                e.printStackTrace();
            }
        };
        ...
```

Classe ProdutorConsumidor - Usando as ferramentas certas

```
public class ProdutorConsumidor_3 {
    ...
    public static void main(String[] args) {
        ...
        Runnable consumidor = () -> {
            simulaProcessamento();
            //simulaProcessamentoLento();
            System.out.println("Consumindo");
            try {
                Integer take = fila.take();
                System.out.println(take);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                e.printStackTrace();
            }
        };

        Janelas.monitore(() -> String.valueOf(fila.size()));
        ScheduledExecutorService executor = Executors.newScheduledThreadPool(2);
        executor.scheduleWithFixedDelay(produtor, 0, 10, TimeUnit.MILLISECONDS);
        executor.scheduleWithFixedDelay(consumidor, 0, 10, TimeUnit.MILLISECONDS);
    }
    ...
```

Classe ProdutorConsumidor - Usando as ferramentas certas

```
public class ProdutorConsumidor_3 {

    ...

    private static final void simulaProcessamento() {
        int tempo = new Random().nextInt(40);
        try {
            Thread.sleep(tempo);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            e.printStackTrace();
        }
    }

    private static final void simulaProcessamentoLento() {
        int tempo = new Random().nextInt(400);
        try {
            Thread.sleep(tempo);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            e.printStackTrace();
        }
    }
}
```