

7.1. Criação e Execução de Threads

7.1.1. Definição de Threads

As *threads* permitem que um programa execute várias tarefas simultaneamente e de forma independente. Em Java, isso é possível graças ao suporte nativo ao **multithreading**.

Uma thread representa um fluxo independente de execução dentro de um programa. Quando um programa Java é iniciado, ele é executado por uma **thread principal** chamada **main**, criada automaticamente pela JVM. O programa se torna *multithread* quando novas threads são criadas.

No contexto de aplicações multithread, existe o conceito de **thread safety** (*segurança de thread*). Um trecho de código é considerado *thread-safe* se ele pode ser executado corretamente, mesmo quando acessado simultaneamente por várias threads.

Quando um código **não é thread-safe**, sua execução paralela pode gerar **resultados inesperados ou incorretos**, já que uma thread pode interferir nas ações de outra. Um exemplo clássico desse problema é a **atualização de contadores compartilhados** por várias threads.

Para lidar com esse tipo de situação, é necessário aplicar **mecanismos de sincronização**, que garantem que apenas uma thread por vez possa acessar determinados trechos de código ou recursos compartilhados.

7.1.2. Criação de Threads

Existem duas maneiras principais de criar threads em Java:

1. **Estendendo a classe **Thread****
2. **Implementando a interface **Runnable****

Embora ambos os métodos sejam válidos, **a abordagem recomendada é implementar a interface **Runnable****, pois Java **não permite herança múltipla**. Ao estender a classe **Thread**, você perde a possibilidade de estender outra classe.

A seguir, veremos exemplos utilizando as duas abordagens.

Código 34: Criando uma thread estendendo a classe **Thread**

```
public class ExemploThread extends Thread {  
    public void run() {  
        for (int i = 0; i < 10; i++) {  
            System.out.print("x");  
            try {
```

```

        Thread.sleep(200);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}
}

```

O método `run()` contém o código que será executado pela thread. Importante: **não devemos chamar `run()` diretamente** — a execução da thread começa com a chamada ao método `start()`.

Código 35: Classe que utiliza a thread

```

public class TesteThread {
    public static void main(String[] args) {
        ExemploThread thread = new ExemploThread();
        thread.start(); // inicia a thread

        for (int i = 0; i < 10; i++) {
            System.out.print("y");
            try {
                Thread.sleep(200);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

```

Ao executar esse código, os caracteres “x” (da nova thread) e “y” (da thread principal) são impressos de forma alternada ou desordenada, pois ambas as threads estão sendo executadas **concomitantemente e sem controle de ordem**.

Código 36: Criando uma thread com a interface `Runnable`

```

public class ExemploThreadRunnable implements Runnable {
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.print("x");
            try {
                Thread.sleep(200);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

```

```
}
```

Código 37: Executando uma thread com `Runnable`

```
public class TesteThread02 {  
    public static void main(String[] args) {  
        Runnable tarefa = new ExemploThreadRunnable();  
        Thread thread = new Thread(tarefa);  
        thread.start();  
  
        for (int i = 0; i < 10; i++) {  
            System.out.print("y");  
            try {  
                Thread.sleep(200);  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
    }  
}
```

Neste exemplo, criamos um objeto da interface `Runnable` e o passamos como argumento para o construtor da classe `Thread`. O comportamento é o mesmo do exemplo anterior.

7.1.3. Definindo um Nome para a Thread

Em Java, podemos definir um nome para uma thread de duas maneiras:

1. Utilizando o método `setName(String name)`
2. Através do construtor da classe `Thread`

Essas abordagens são demonstradas nos exemplos a seguir.

Código 38: Classe `ExemploThreadRunnable02` – obtendo o nome da thread

```
public class ExemploThreadRunnable02 implements Runnable {
    public void run() {
        String nome = Thread.currentThread().getName();
        for (int i = 0; i < 5; i++) {
            System.out.println(nome + " está executando...");
            try {
                Thread.sleep(300);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Código 39: Classe `TesteThread03` – atribuindo um nome à thread no construtor

```
public class TesteThread03 {
    public static void main(String[] args) {
        Runnable tr = new ExemploThreadRunnable02();
        Thread t = new Thread(tr, "thread_pessoal");
        t.start();

        String nomePrincipal = Thread.currentThread().getName();
        System.out.println("Thread principal: " + nomePrincipal);
    }
}
```

Saída esperada:

```
thread_pessoal está executando...
thread_pessoal está executando...
...
Thread principal: main
```

A chamada `Thread.currentThread().getName()` retorna o nome da thread atualmente em execução. Quando essa instrução está no `main()`, o nome retornado será `"main"`, que é o nome padrão da thread principal.

Já no caso da nova thread, o nome atribuído no construtor (`"thread_pessoal"`) é utilizado na execução do método `run()`.

Código 40: Refatorando para definir o nome depois da criação

```
public class TesteThread03 {
    public static void main(String[] args) {
        Runnable tr = new ExemploThreadRunnable02();
        Thread t = new Thread(tr);
        t.setName("thread_personalizada");
        t.start();

        System.out.println("Thread principal: " + Thread.currentThread().getName());
    }
}
```

Neste exemplo, usamos `setName()` para alterar o nome da thread **após sua criação**, mas **antes de iniciá-la** com `start()`.

7.1.4. Executando Múltiplas Threads

É comum haver várias threads sendo executadas simultaneamente, muitas vezes compartilhando o mesmo objeto. O exemplo a seguir mostra como isso funciona.

Código 41: Classe `ThreadMultiplo`

```
public class ThreadMultiplo implements Runnable {
    public void run() {
        String nome = Thread.currentThread().getName();
        for (int i = 0; i < 5; i++) {
            System.out.println(nome + " executando...");
            try {
                Thread.sleep(300);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Código 42: Classe **TesteThread04** – executando múltiplas threads sobre o mesmo objeto

```
public class TesteThread04 {  
    public static void main(String[] args) {  
        Runnable tarefa = new ThreadMultiplo();  
  
        Thread t1 = new Thread(tarefa, "Thread-A");  
        Thread t2 = new Thread(tarefa, "Thread-B");  
  
        t1.start();  
        t2.start();  
    }  
}
```

Saída esperada (exemplo):

```
Thread-A executando...  
Thread-B executando...  
Thread-A executando...  
Thread-B executando...  
...
```

Note que as threads são executadas **concorrentemente** e operam sobre **o mesmo objeto Runnable**, mas seus fluxos são **independentes**.

A **interleaving (intercalação)** das saídas ocorre porque a ordem de execução das threads é controlada pela **JVM e pelo sistema operacional**, o que significa que **não há garantia de ordem na impressão dos resultados**.

7.2. Estados de Threads

7.2.1. A Classe Thread

Durante o ciclo de vida de uma thread em Java, ela pode assumir diferentes **estados**, conforme mostra o **diagrama de estados da Figura 9**, baseado em *Sierra e Bates (2008)*.

Figura 9 – Diagrama de Estados da Thread

(*New* → *Runnable* → *Running* → *Waiting/Blocked* → *Runnable* → *Dead*)

Estado	Descrição
New	A thread é criada com o construtor <code>Thread()</code> e ainda não foi iniciada . Nenhum recurso de sistema foi alocado.
Runnable / Running	A thread está pronta para ser executada (<code>Runnable</code>). Quando o agendador a escolhe, ela passa para o estado <code>Running</code> , e seu método <code>run()</code> é executado pela CPU.
Waiting / Blocked	A thread está esperando ou bloqueada . Isso ocorre por <code>sleep()</code> , <code>wait()</code> , operações de I/O, ou sincronização.
Dead	A thread finalizou sua execução naturalmente (fim do <code>run()</code>), ou foi encerrada por um método como <code>stop()</code> (<i>obsoleto e inseguro</i>).

Quadro 8 – Descrição dos Estados da Thread

Exemplos de Código para os Estados

✓ New

```
Runnable tm = new MinhaTarefa();
Thread t1 = new Thread(tm, "João");
Thread t2 = new Thread(tm, "Maria");
// As threads foram criadas, mas ainda não iniciadas (estado New)
```

✓ Runnable / Running

```
t1.start();
t2.start();
// As threads estão no estado Runnable.
// O escalonador define qual entra em execução (Running).
```

💡 Quando uma thread está **Running**, ela está também **Runnable**, pois o estado **Runnable** inclui a possibilidade de estar executando ou aguardando a CPU.

✅ **Waiting / Blocked**

```
try {
    Thread.sleep(1000); // A thread entra em espera por 1 segundo
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

Outras causas para esse estado:

- **suspend()** (obsoleto)
- **wait()** → espera por um **notify()**
- I/O bloqueado
- **join()** → esperando outra thread terminar

Saídas do estado Waiting/Blocked:

- **sleep()** → termina o tempo de espera
- **wait()** → recebe **notify()** ou **notifyAll()**
- I/O → operação finaliza
- **suspend()** → requer **resume()** (ambos obsoletos)

✅ **Dead**

```
public void run() {
    try {
        String nome = Thread.currentThread().getName();
        for (int i = 0; i < 5; i++) {
            System.out.println("Thread: " + nome + ", posição: " + i);
            Thread.sleep(200);
        }
    } catch (InterruptedException e) {
        System.out.println("Thread interrompida.");
    }
}
// A thread termina naturalmente quando o loop termina (Dead)
```


⚠ O método `stop()` **não é seguro** e está obsoleto, pois pode causar inconsistências em objetos compartilhados entre threads.

7.3. Prioridades e Agendamento de Threads

Em Java, **cada thread possui uma prioridade**, e o **escalonador de threads** (scheduler) pode usar essa prioridade como critério para decidir qual thread será executada primeiro.

Prioridades Padrão

A classe `Thread` define três constantes para controle de prioridade:

Constante	Valor	Descrição
<code>Thread.MIN_PRIORITY</code>	1	Prioridade mínima
<code>Thread.NORM_PRIORITY</code>	5	Prioridade normal (padrão)
<code>Thread.MAX_PRIORITY</code>	10	Prioridade máxima

Por padrão, uma nova thread herda a prioridade da thread que a criou.

Exemplo: Definindo Prioridades

✓ Classe `ExemploPrioridades`

```
public class ExemploPrioridades implements Runnable {
    public void run() {
        String nome = Thread.currentThread().getName();
        for (int i = 0; i < 5; i++) {
            System.out.println("Thread: " + nome + ", posição: " + i);
        }
    }
}
```

✓ Classe `TesteThread05`

```
public class TesteThread05 {
    public static void main(String[] args) {
        Runnable tarefa = new ExemploPrioridades();
    }
}
```


```
Thread t1 = new Thread(tarefa, "Thread-1");
Thread t2 = new Thread(tarefa, "Thread-2");

t1.setPriority(Thread.MIN_PRIORITY); // prioridade 1
t2.setPriority(Thread.MAX_PRIORITY); // prioridade 10

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

Resultado Esperado

Embora a prioridade **não garanta a ordem de execução**, geralmente a **t2** (com prioridade maior) tende a ser executada antes da **t1**.

 A priorização depende da **implementação da JVM** e do **sistema operacional**. Nem sempre será respeitada rigidamente.

7.4. Relacionamentos entre Threads – Sincronismo

Por definição, as **threads são executadas independentemente** e compartilham o mesmo espaço de memória dentro de um processo. Isso significa que, **por padrão, não existe coordenação entre elas** — mas, com o uso de sincronismo, é possível controlar seu comportamento em relação a objetos compartilhados.

Exemplo: Concorrência em uma Conta Corrente

Imagine que temos uma **conta corrente** acessada simultaneamente por **duas threads**, como no exemplo descrito abaixo:

- **Ambas consultam o saldo**
- **Efetuem o saque, se houver saldo suficiente**

Se não houver controle, **ambas podem sacar ao mesmo tempo**, levando o saldo a valores incorretos — inclusive negativos.

Códigos envolvidos no exemplo

✓ Código 45: **ContaCorrente.java**

```
public class ContaCorrente {
    private double saldo;

    public ContaCorrente(double saldoInicial) {
        this.saldo = saldoInicial;
    }

    public double getSaldo() {
        return saldo;
    }

    public void sacar(double valor) {
        saldo -= valor;
    }
}
```

✓ Código 46: **ProcessaContaCorrente.java**

```
public class ProcessaContaCorrente implements Runnable {
    private ContaCorrente conta;
    private String nome;
```

```

public ProcessaContaCorrente(ContaCorrente conta, String nome) {
    this.conta = conta;
    this.nome = nome;
}

@Override
public void run() {
    efetuarOperacao();
}

public synchronized void efetuarOperacao() {
    if (conta.getSaldo() >= 20) {
        System.out.println("Valor sacado por " + nome + ": 20.0");
        conta.sacar(20);
        System.out.println("Saldo para " + nome + ": " + conta.getSaldo());
    } else {
        System.out.println("Saldo insuficiente para " + nome);
    }
}
}

```

✓ Código 47: **TesteContaCorrente.java**

```

public class TesteContaCorrente {
    public static void main(String[] args) {
        ContaCorrente conta = new ContaCorrente(100.0);

        Thread t1 = new Thread(new ProcessaContaCorrente(conta, "Maria"));
        Thread t2 = new Thread(new ProcessaContaCorrente(conta, "João"));

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

```

Resultado sem sincronização

Quando executamos esse programa **sem o uso do `synchronized`**, o resultado é imprevisível, como por exemplo:

```

Saldo para Maria: 100.0
Valor sacado por João: 20.0
Saldo para João: 80.0
Valor sacado por Maria: 20.0

```

Saldo para Maria: 60.0

⚠ O problema aqui é que **Maria leu o saldo antes de João sacar**, levando a resultados inconsistentes.

A Solução: Sincronismo

Para evitar esse tipo de condição de corrida (**race condition**), usamos o modificador **synchronized** no método **efetuarOperacao()**. Isso garante que **apenas uma thread por vez** possa acessar o método e realizar as operações críticas sobre o saldo.

Código 48: Método sincronizado

```
public synchronized void efetuarOperacao() {  
    if (conta.getSaldo() >= 20) {  
        System.out.println("Valor sacado por " + nome + ": 20.0");  
        conta.sacar(20);  
        System.out.println("Saldo para " + nome + ": " + conta.getSaldo());  
    } else {  
        System.out.println("Saldo insuficiente para " + nome);  
    }  
}
```

✅ Agora, as operações de consulta e saque são realizadas de forma **atômica**, sem interferência de outras threads.

Resultado com sincronismo

Valor sacado por Maria: 20.0
Saldo para Maria: 80.0
Valor sacado por João: 20.0
Saldo para João: 60.0

Agora a execução está ordenada e segura — **sem saldo negativo, sem leitura incorreta**, e com **consistência dos dados**.