Multithreading em JAVA

Prof. MSc. Daniel Menin Tortelli

Multithreading em JAVA

- O JAVA disponibiliza a concorrência para o programador de aplicativos por meio de suas APIs.

- O programador especifica os aplicativos que contém **threads de execução**, em que cada thread designa uma parte de um programa que pode executar concorrentemente com outras threads.

- Essa capacidade é denominada de **Multithreading**.

Multithreading em JAVA

- Um problema com aplicativos de uma única thread é que atividades longas devem ser concluídas antes que outras atividades iniciem.

- Em um aplicativo com múltiplas threads, as threads podem ser distribuídas por múltiplos processadores, de modo que múltiplas tarefas são realizadas concorrentemente e o aplicativo pode operar de modo mais eficiente.

- O multithreading também pode aumentar o desempenho em sistemas de único processador que simula concorrência – quando uma thread não puder prosseguir, outra pode utilizar o processador.

Multithreading em JAVA

- Threads - Linhas de execução:
  - Cada linha é uma parte de um programa que pode executar concorrentemente com outras linhas (multithreading).
  - Isso dá a Java poderosas capacidades não existentes em linguagens single-threaded.
- Exemplo: download de um clipe de video no Youtube:
  - Em vez de ter que baixar o clipe inteiro e depois tocá-lo: Download de uma parte, toca aquela parte, download da próxima parte, toca aquela parte... Essas atividades prosseguem concorrentemente.
  - Para evitar a reprodução instável, sincroniza as threads de modo que a thread que está reproduzindo o video apenas inicie quando houver uma quantidade suficiente do clipe carregado pela thread de download.

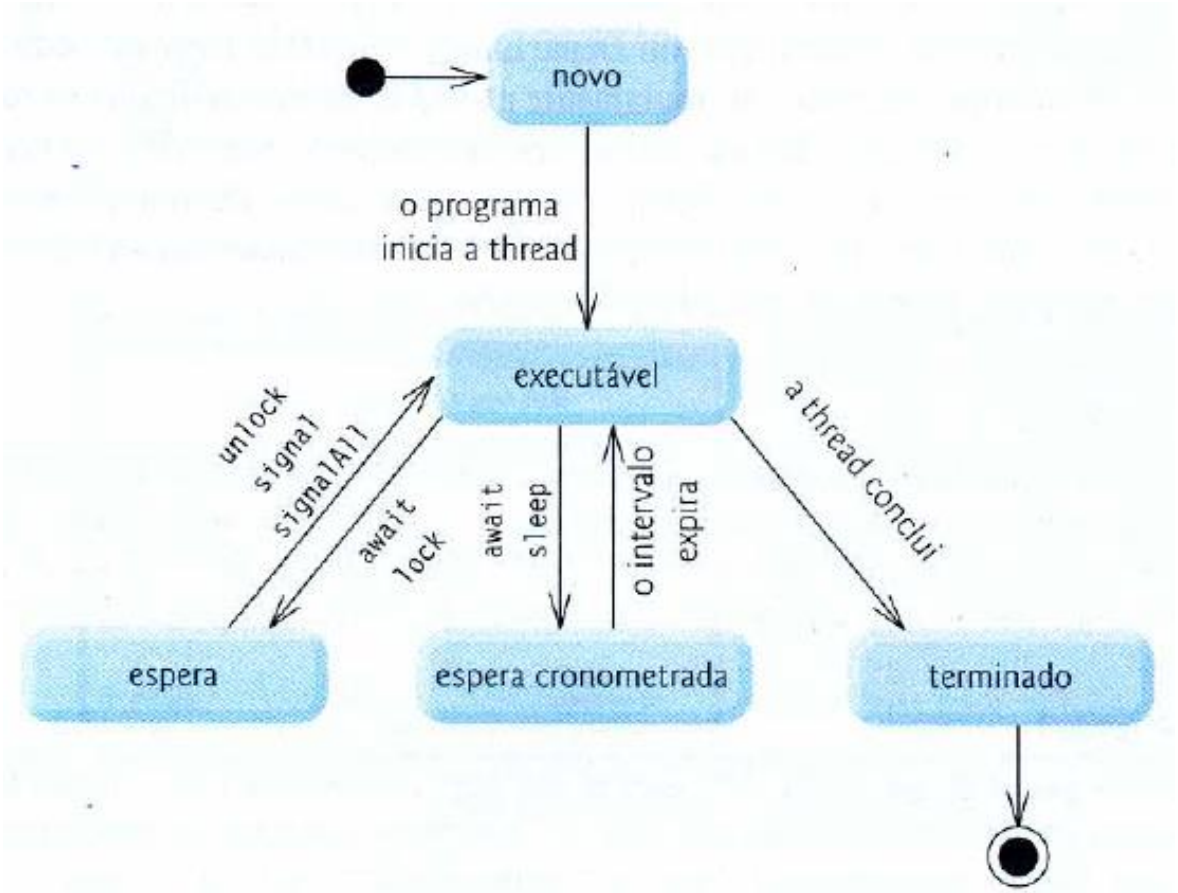
Multithreading em JAVA

- Exemplo: coleta de lixo do JAVA (Garbage Collection)
  - Linguagens como o C/C++ exigem que o programador reivindique memória dinamicamente alocada de modo explícito.
  - O JAVA fornece uma **thread coletora de lixo** que reivindica a memória que não e mais necessária.

Estados de Thread

Estados de Thread

- A qualquer dado momento, diz-se que uma thread está em um dos vários **estados de thread**:



Estados de Thread

- Uma nova thread inicia seu ciclo de vida no estado **Novo**.
- Ela permanece nesse estado até o programa iniciar a thread, o que a coloca no estado **Executável**.
- Considera-se que uma thread nesse estado está executando sua tarefa.

Estados de Thread

- Às vezes uma thread entra no estado de **Espera** enquanto espera outra thread realizar uma tarefa.

- Uma vez nesse estado, a thread só volta ao estado *executável* quando outra thread sinalizar a thread de espera para retomar a execução.

Estados de Thread

- Uma thread *executável* pode entrar no estado de **Espera Sincronizada** por um intervalo específico de tempo.

- Uma thread nesse estado volta para o estado *executável* quando esse intervalo de tempo expira ou quando ocorre o evento que ele está esperando.

- As threads de *espera sincronizada* não podem utilizar o processador, mesmo que haja um disponível.

Estados de Thread

- Uma thread pode transitar para o estado de *espera sincronizada* se fornecer um intervalo de espera opcional quando ela estiver esperando outra thread realizar uma tarefa.

- Essa thread retornará ao estado *executável* quando ela for sinalizada por outra thread ou quando o intervalo sincronizado expirar – o que ocorrer primeiro.

Estados de Thread

- Outra maneira de colocar uma thread no estado de *espera sincronizada* é colocá-la para “dormir”.

- Uma “thread adormecida” permanece no estado de espera sincronizada por um período designado de tempo (denominado **intervalo de adormecimento**) no ponto em que ele retorna para o estado *executável*.

- As threads dormem quando, por um breve período, não tem de realizar nenhuma tarefa. Ex.: auto-save Word.

Estados de Thread

- Uma thread *executável* entra no estado **Terminado** quando completa sua tarefa ou, caso contrário, termina (talvez devido a uma condição de erro).

Estados de Thread

- Quando uma thread entra pela primeira vez no estado *executável* a partir do estado *novo*, a thread está no estado **pronto**.

- Uma thread pronta entra no estado de execução (isto é, começa a executar) quando o S.O. atribui a thread a um processador (despachar a thread).

Estados de Thread

- Na maioria dos S.O., cada thread recebe uma pequena quantidade de tempo de processador – denominada **quantum** ou **fração de tempo** – com o qual realiza sua tarefa.

- Quando o *quantum* da thread expirar, a thread retornará ao estado *pronto* e o S.O. atribuirá outra thread ao processador.

- ***As transições entre esses estados são tratadas unicamente pelo S.O.***

- O processo que utiliza um S.O. para decidir qual thread despachar é conhecido como **agendamento de thread** e depende das prioridades da thread.

Prioridades e Escalonamento de Threads

Prioridades e escalonamento de Threads

- Todos os applets e os aplicativos Java são multithreaded.

- Threads têm prioridades de 1 a 10 (constantes declaradas na classe Thread):
  - **Thread.MIN_PRIORITY - 1**
  - **Thread.NORM_PRIORITY - 5** (default)
  - **Thread.MAX_PRIORITY - 10**

- Threads novas herdam a prioridade da thread que as criou.

Prioridades e escalonamento de Threads

- Problemas:
  - Uma prioridade de thread Java pode ser mapeada de forma diferente para as prioridades de threads do S.O. subjacente:
    - Solaris tem 2^{32-1} níveis de prioridades;
    - Windows NT tem apenas 7 níveis de prioridades;
    - Windows 7 tem apenas 6 níveis de prioridades;
      - Gerenciador de Tarefas...

Prioridades e escalonamento de Threads

- Todas as threads com prioridade mais alta são mais importantes para um programa e devem ser alocadas em tempo de processador antes das threads de prioridade mais baixa.

- As propriedades de thread não podem garantir a ordem em que elas são executadas. Ex.: várias threads com a mesma prioridade.

Prioridades e escalonamento de Threads

- O trabalho do **escalonador de threads** do S.O. é determinar a próxima thread que entra em execução.

- Ele mantém a thread de prioridade mais alta executando o tempo todo e, se houver mais de uma thread de prioridade idêntica, isso assegura que cada uma delas executa por um quantum no estilo rodízio **(round-robin)**.

Prioridades e escalonamento de Threads

- Quando uma thread de prioridade mais alta entra no estado de *pronto*, o S.O. geralmente faz **preempção** da thread em *execução*.

- Dependendo do S.O., as threads de prioridade mais alta poderiam adiar – por um tempo indeterminado – a execução de threads de prioridade mais baixa.

- Esse adiamento indefinido é chamado de **inanição** (**Starvation**).

Criando e Executando Threads Usando a Interface *Runnable*

Criando e Executando Threads Usando a Interface *Runnable*

- Em JAVA, uma maneira de implementar um aplicativo com suporte a múltiplas threads é a interface ***Runnable*** (pacote java.lang).

- Utiliza-se classes e métodos predefinidos para criar as threads que executam os objetos *Runnables*.

- A interface declara um único método abstrato chamado ***run***.

Criando e Executando Threads Usando a Interface *Runnable*

- *Runnables* são executados por um objeto de uma classe que implementa a interface ***Executor*** (pacote java.util.concurrent).

- Essa interface declara um único método chamado ***execute***.

- Um objeto *Executor* cria e gerencia um grupo de threads denominado **pool de threads**.

- Essas threads executam os objetos *Runnables* passados para o método **execute**.

Criando e Executando Threads Usando a Interface *Runnable*

- O Executor atribui cada *Runnable* a uma das threads disponíveis no pool de threads.

- Se não houver nenhuma thread no pool, o *Executor* cria uma nova thread ou espera que uma se torne disponível para atribuir a ela um *Runnable* que foi passado para o método ***execute***.

- Dependendo o tipo de *Executor*, há um limite para o número de threads que podem ser criadas.

Exemplo 1: Classe PrintTask

```
package runnabletester;

import java.util.Random;


// Classe PrintTask dorme por um tempo aleatório de 0 a 5 segundos
public class PrintTask implements Runnable
{
    private int sleepTime; // tempo de adormecimento aleatório para a thread
    private String threadName; // nome da thread
    private static Random generator = new Random(); // gerador de números aleatórios

    // Construtor: atribui nome a thread
    public  PrintTask(String name)
    {
        this.threadName = name; // atribui nome da thread

        // seleciona o tempo aleatório entre 0 e 5 segundos
        this.sleepTime = generator.nextInt(5000);
    }
```

Exemplo 1: Classe PrintTask (continuação...)

```
    // Método run: método a ser executado pela thread
    public void run()
    {
        // coloca a thread para dormir pela quantidade de tempo em sleepTime
        try
        {
            System.out.printf("A thread %s irá dormir por %d milissegundos. \n",
                    this.threadName, this.sleepTime);

            Thread.sleep(this.sleepTime);
        } // fim try
        // se a thread for interrompida enquanto dormia, imprime o rastreamento
        // da pilha
        catch (InterruptedException exception)
        {
            exception.printStackTrace();
        } // fim catch

        // imprime o nome da thread que acordou
        System.out.printf("\nA thread %s acordou", this.threadName);

    } // fim metodo run

} // fim classe PrintTask
```

Exemplo 1: Classe RunnableTester

```
package runnabletester;

import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;

public class RunnableTester
{
    public static void main(String[] args)
    {
        // cria algumas threads...
        PrintTask task1 = new PrintTask("thread1");
        PrintTask task2 = new PrintTask("thread2");
        PrintTask task3 = new PrintTask("thread3");

        System.out.println("Iniciando as threads...");

        // Cria o ExecutorService para gerenciar as threads
        ExecutorService threadExecutor = Executors.newFixedThreadPool(3);

        // inicia as threads e as coloca em estado executável
        threadExecutor.execute(task1);  // inicia task1
        threadExecutor.execute(task2);  // inicia task2
        threadExecutor.execute(task3);  // inicia task3

        threadExecutor.shutdown();      // encerra as threads


        System.out.println("Fim da função main");

    } // fim main
} // fim classe RunnableTester
```

Criando e Executando Threads Usando a herança da classe *Thread*

Criando e Executando Threads Usando a herança da classe *Thread*

- Declara-se uma classe como sendo uma subclasse da classe *Thread* e override o método **run.**

- **A classe *Thread* implementa a interface *Runnable*.**

Criando e Executando Threads usando Herança da Classe *Thread*

```
package threadtester;

import java.util.Random;


public class PrintTask extends Thread
{
    private int sleepTime; // tempo de adormecimento aleatório para a thread
    private String threadName; // nome da thread
    private static Random generator = new Random(); // gerador de números aleatórios

    // Construtor: atribui nome a thread
    public  PrintTask(String name)
    {
        this.threadName = name; // atribui nome da thread

        // seleciona o tempo aleatório entre 0 e 5 segundos
        this.sleepTime = generator.nextInt(5000);
    }
```

Criando e Executando Threads usando Herança da Classe *Thread*

```
// Método run: método a ser executado pela thread
@Override
public void run()
{
    // coloca a thread para dormir pela quantidade de tempo em sleepTime
    try
    {
        System.out.printf("A thread %s irá dormir por %d milissegundos. \n",
                this.threadName, this.sleepTime);

        Thread.sleep(this.sleepTime);
    } // fim try
    // se a thread for interrompida enquanto dormia, imprime o rastreamento
    // da pilha
    catch (InterruptedException exception)
    {
        exception.printStackTrace();
    } // fim catch

    // imprime o nome da thread que acordou
    System.out.printf("\nA thread %s acordou", this.threadName);

} // fim metodo run

} // fim classe PrintTask
```

Criando e Executando Threads usando Herança da Classe *Thread*

```
package threadtester;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;


public class ThreadTester {


    public static void main(String[] args)
    {
        // cria algumas threads...
        PrintTask task1 = new PrintTask("thread1");
        PrintTask task2 = new PrintTask("thread2");
        PrintTask task3 = new PrintTask("thread3");

        System.out.println("Iniciando as threads...");

        // Cria o ExecutorService para gerenciar as threads
        ExecutorService threadExecutor = Executors.newFixedThreadPool(3);

        // inicia as threads e as coloca em estado executável
        threadExecutor.execute(task1);  // inicia task1
        threadExecutor.execute(task2);  // inicia task2
        threadExecutor.execute(task3);  // inicia task3

        threadExecutor.shutdown();      // encerra as threads


        System.out.println("Fim da função main");

    } // fim main
} // fim classe ThreadTester
```

Start – Sleep – Interrupt - Join

- O método Thread.start() é chamado para iniciar a execução de uma thread.

- O método Thread.sleep() é usado para suspender a execução de uma thread por um determinado período de tempo (em milissegundos).
  - Esse é uma forma eficiente de tornar o tempo do processador disponível para outras threads de uma aplicação, ou outras aplicações rodando no sistema.
  - Não há garantia de que o tempo de espera configurado no parâmetro da função sleep seja preciso. Isso depende do S.O.
  - Um período de suspensão pode ser quebrado com interrupções (Thread.interrupt), terminando forçadamente a execução da thread.

Start – Sleep – Interrupt - Join

```
class Count extends Thread
{
    Count()
    {
        //super("my extending thread");
        //System.out.println("my thread created" + this);
        this.start();
    }

    public void run()
    {
        try
        {
            for (int i = 0; i < 10; i++)
            {
                System.out.println("Imprimindo o contador " + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e)
        {
            System.out.println("Child Thread interrompida");
        }
        System.out.println("Child Thread terminou a execução");
    }
}
```

Start – Sleep – Interrupt - Join

```
public static void main(String[] args)
{
    Count cnt = new Count();

    try
    {
        while (cnt.isAlive())
        {
            System.out.println("A Main Thread permanecerá ativa "
                    + "enquanto as threads filhas estiverem executando!");
            Thread.sleep(1500);
        }
    }
    catch (InterruptedException e)
    {
        System.out.println("Main Thread interrompida");
    }

    System.out.println("Main Thread terminou a execução");
}
```

Start – Sleep – Interrupt - Join

```
A Main Thread permanecerá ativa enquanto as threads filhas estiverem executando!
Imprimindo o contador 0
Imprimindo o contador 1
A Main Thread permanecerá ativa enquanto as threads filhas estiverem executando!
Imprimindo o contador 2
A Main Thread permanecerá ativa enquanto as threads filhas estiverem executando!
Imprimindo o contador 3
Imprimindo o contador 4
A Main Thread permanecerá ativa enquanto as threads filhas estiverem executando!
Imprimindo o contador 5
A Main Thread permanecerá ativa enquanto as threads filhas estiverem executando!
Imprimindo o contador 6
Imprimindo o contador 7
A Main Thread permanecerá ativa enquanto as threads filhas estiverem executando!
Imprimindo o contador 8
A Main Thread permanecerá ativa enquanto as threads filhas estiverem executando!
Imprimindo o contador 9
Child Thread terminou a execução
Main Thread terminou a execução
BUILD SUCCESSFUL (total time: 11 seconds)
```

Start – Sleep – Interrupt - Join

- O método Thread.interrupt() é chamado para indicar a uma thread que ela deve parar o que estiver fazendo, fazer outra coisa ou, simplesmente, terminar abruptamente sua execução.
  - O programador pode decidir como exatamente uma thread irá responder a uma interrupção.
  - Mas o comportamento mais comum é terminar a execução da thread.
  - Para que o mecanismo de interrupção funcione corretamente, a thread interrompida deve suportar sua própria interrupção.

Start – Sleep – Interrupt - Join

- O método Thread.join() permite que uma thread aguarde a finalização de outra thread antes de prosseguir com sua execução.

```
class JoinTest implements Runnable{

    @Override
    public void run()
    {
        Thread t = Thread.currentThread();
        System.out.println("Thread iniciou: " + t.getName());

        try
        {
            Thread.sleep(4000);
        }
        catch (InterruptedException ie)
        {
            ie.printStackTrace();
        }
        System.out.println("Thread terminou: " + t.getName());
    }
}
```

Start – Sleep – Interrupt - Join

```
    public static void main(String[] args)
    {
        // Inicio Parte 1
//        Thread th1 = new Thread(new JoinTest(), "th1");
//        Thread th2 = new Thread(new JoinTest(), "th2");
//        Thread th3 = new Thread(new JoinTest(), "th3");
//
//        th1.start();
//        th2.start();
//        th3.start();
        // Fim Parte 1


        // Inicio Parte 2
        Thread th1 = new Thread(new JoinTest(), "th1");
        Thread th2 = new Thread(new JoinTest(), "th2");
        Thread th3 = new Thread(new JoinTest(), "th3");

        // Start first thread immediately
        th1.start();

        /* Start second thread(th2) once first thread(th1) is dead */
        try
        {
            th1.join();
        }
        catch (InterruptedException ie)
        {
            ie.printStackTrace();
        }
```

Start – Sleep – Interrupt - Join

```
    th2.start();

    /* Start third thread(th3) once second thread(th2) is dead */
    try
    {
        th2.join();
    }
    catch (InterruptedException ie)
    {
        ie.printStackTrace();
    }

    th3.start();

    // Displaying a message once third thread is dead
    try
    {
        th3.join();
    }
    catch (InterruptedException ie)
    {
        ie.printStackTrace();
    }
    System.out.println("Todas as threads terminaram sua execução!");

    // Fim Parte 2
}
```

```
run-single:
Thread iniciou: th1
Thread terminou: th1
Thread iniciou: th2
Thread terminou: th2
Thread iniciou: th3
Thread terminou: th3
All three threads have finished execution
BUILD SUCCESSFUL (total time: 12 seconds)
```

Exemplo 1

```
class Processor implements Runnable
{
    private int ID;

    public Processor(int _id)
    {
        this.ID = _id;
    }

    @Override
     public void run()
    {
        System.out.println("Iniciado: " + this.ID);

        try
        {
            Thread.sleep(5000);
        }
        catch (InterruptedException ex)
        {
        }

        System.out.println("Completado: " + this.ID);
    }
}
```

- Cria um programa com *ExecutorService* que cria um pool de threads para controlar a quantidade de threads em execução:

```java
public static void main(String[] args)
{
    // Cria um pool de threads que limita a execução de um número x
    // de threads.
    ExecutorService executor = Executors.newFixedThreadPool(2);

    for (int i = 0; i < 5; i++)
    {
        executor.submit(new Processor(i));
    }

    // Espera TODAS as threads terminarem sua tarefa e termina a execução
    executor.shutdown();

    System.out.println("Todas as threads envidadas...");

    try
    {
        executor.awaitTermination(1, TimeUnit.DAYS);
    } catch (InterruptedException ex)
    {
        ex.printStackTrace();
    }

    System.out.println("Todas as threads terminadas!");
}
```

```
run-single:
Todas as threads envidadas...
Iniciado: 0
Iniciado: 1
Completado: 1
Completado: 0
Iniciado: 2
Iniciado: 3
Completado: 3
Iniciado: 4
Completado: 2
Completado: 4
Todas as threads terminadas!
BUILD SUCCESSFUL (total time: 15 seconds)
```

Exemplo 2

- Esse programa consiste em duas threads:
- A primeira é a thread principal que todo programa possui (*main*)
- A thread principal cria uma nova thread a partir de um objeto ***Runnable***, chamada *MessageLoop*, e espera até que ela termine sua execução.
- Se a thread *MessageLoop* demorar tempo demais para terminar sua execução, a thread principal irá interrompê-la.

```
static void threadMessage(String message)
{
    String threadName = Thread.currentThread().getName();

    System.out.printf("%s: %s%n", threadName, message);
}

private static class MessageLoop implements Runnable
{
    Random random = new Random();

    @Override
    public void run()
    {
        String poema [] =
        {
            "Quero todo o teu espaço e todo o teu tempo",
            "Quero todas as tuas horas e todos os teus beijos",
            "Quero toda a tua noite",
            "e todo o teu silêncio."
        };

        try
        {
            for (int i = 0; i < poema.length; i++)
            {
                int sleepTime = random.nextInt(5) * 1000 + 1000;

                Thread.sleep(sleepTime);

                threadMessage(poema[i] + " -> " + sleepTime);
            }
        }
        catch (InterruptedException ex)
        {
            threadMessage("Ainda não terminei!");
        }
    }
}
```

```
    public static void main(String[] args)
    {
        // Tempo (em milissegundo) antes de interromper a thread 'MessageLoop' (1 hora)
        long patience = 10000;

        // Se um inteiro for passado na linha de comando da aplicação
        // substitui o tempo definido na variável 'patience' (em segundos)
//        if (args.length > 0)
//        {
//            try
//            {
//                patience = Long.parseLong(args[0]) * 1000;
//            }
//            catch (NumberFormatException ex)
//            {
//                System.err.println("Argumento deve ser um número inteiro!");
//                System.exit(1);
//            }
//        }

        threadMessage("Iniciando Thread 'MessageLoop'");

        long startTime = System.currentTimeMillis();
        Thread t = new Thread(new MessageLoop());
        t.start();

        threadMessage("Aguardando a thread 'MessageLoop' terminar...");
```

```
        // Execute enquanto a thread estiver ativa
        while (t.isAlive())
        {
            //threadMessage("Aguardando...");

            // Aguarda 1 segundo (no máximo) para a thread terminar
//              try
//              {
//                  t.join();
//              }
//              catch (InterruptedException ex)
//              {
//                  ex.printStackTrace();
//              }

            if (((System.currentTimeMillis() - startTime) > patience) && t.isAlive())
            {
                threadMessage("Cansado de esperar");

                t.interrupt();
            }
        }

        threadMessage("Fim MAIN!");
    }
```

```
run-single:
main: Iniciando Thread 'MessageLoop'
main: Aguardando a thread 'MessageLoop' terminar...
Thread-0: Quero todo o teu espaço e todo o teu tempo -> 3000
Thread-0: Quero todas as tuas horas e todos os teus beijos -> 5000
main: Cansado de esperar
main: Cansado de esperar
main: Cansado de esperar
main: Cansado de esperar
main: Cansado de esperar
main: Cansado de esperar
main: Cansado de esperar
main: Cansado de esperar
main: Cansado de esperar
main: Cansado de esperar
main: Cansado de esperar
main: Cansado de esperar
main: Cansado de esperar
main: Cansado de esperar
main: Cansado de esperar
main: Cansado de esperar
Thread-0: Ainda não terminei!
main: Cansado de esperar
main: Fim MAIN!
BUILD SUCCESSFUL (total time: 10 seconds)
```

Exercício 1

Crie um programa que cria 100 threads e armazene-as em um array.

Executar as threads que devem, cada uma, escrever uma mensagem na tela contendo seu número.

```
run:
Iniciando as threads

A thread 0 executou
A thread 9 executou
A thread 8 executou
A thread 7 executou
A thread 1 executou
A thread 2 executou
A thread 3 executou
A thread 6 executou
A thread 4 executou
A thread 5 executou
```

Exercício 2

Crie duas threads onde:

Uma thread fica enviando notícias aleatórias a cada 5 segundos (as notícias ficam em um array, no total de 5 notícias cadastradas).

Enquanto a outra thread fica enviando a hora atual do sistema a cada 10 segundos.

O programa deve terminar quando 10 notícias forem impressas.

```
Iniciando as threads...07:18:32
Noticia 0
Noticia 3
 07:18:42
Noticia 1
Noticia 2
 07:18:52
Noticia 2
Noticia 3
 07:19:02
Noticia 0
Noticia 0
 07:19:12
Noticia 3
Noticia 0
```

Exercício 3

Escreva um programa que realize o cálculo das somas dos valores das linhas de uma matriz quadrada qualquer de números inteiros e imprima o resultado na tela.

- Faça com que o cálculo do somatório de cada linha seja realizado em paralelo por uma thread.
- A matriz deve ser gerada no início do programa e seus elementos, bem como sua dimensão, devem ser criados aleatoriamente.
- No final, exiba a matriz e o valor da soma dos elementos de suas linhas logo após a impressão do último elemento da linha da matriz.

Exercício 4

Crie uma classe T1 do tipo Thread com um método construtor que receba um número de identificação da Thread e um método **run** que fique em loop infinito imprimindo na tela a frase "Thread **xx** executando", onde **‘xx’** é o número de identificação da Thread.

Faça um programa que crie uma matriz de 5 Threads T1 e, com um laço FOR, inicialize todas as Threads.

Exercícios 5

Crie uma classe **ThreadImpar** que implementa da interface *Runnable*. O trabalho a ser feito é gerar um número inteiro aleatório de 1 a 15 toda vez que executar. Quando um número **ímpar** for encontrado, a thread entra em espera sincronizada por 3 segundos.

Crie uma classe **ThreadPar** que herda da classe *Thread*. O trabalho a ser feito é gerar um número inteiro aleatório de 1 a 15 toda vez que executar. Quando um número **par** for encontrado, a thread entra em espera sincronizada por 2 segundos.

Crie uma classe **ThreadDivFive** que herda da classe *Thread*. O trabalho a ser feito é gerar um número inteiro aleatório de 0 a 50 toda vez que executar. Quando um número **divisível por 5** for encontrado, a thread entra em espera sincronizada por 5 segundos.

Ambas as threads devem ser executadas concorrentemente 5 vezes.

No final, elas devem exibir quais os números ímpares e pares e divisíveis por 5 foram encontrados durante a sua execução, respectivamente, e quantas vezes as threads entraram no modo de espera sincronizada.

Exiba também, no final da execução da thread, a hora da conclusão do seu trabalho no formato (hh:mm:ss)

Exercícios 6

Usando Threads, crie um programa que simule uma corrida de cavalos onde 3 cavalos se encontram na pista e correm de maneira independente e concorrentemente.

A cada 3 segundos, um número de 0 – 10 é gerado aleatoriamente para cada cavalo. Esse número representa quantos metros o cavalo percorreu na pista.

Vence o cavalo que completar o percurso de 1000 metros da pista.

No final, imprima o número do cavalo que venceu a corrida, bem como o número do segundo e terceiro colocados, respectivamente.