

Programação de Sistemas Distribuídos - Java para Web

1 - Threads

Citeforma

Jose Aser Lorenzo, Pedro Nunes, Paulo Jorge Martins

jose.l.aser@sapo.pt, pedro.g.nunes@gmail.com,
paulojsm@gmail.com

Março de 2010

V 1.3

Sumário

1- Threads.....	1-3
1.1- Objectivos.....	1-4
1.2- Preparar o ambiente de trabalho.....	1-5
1.2.1- Instalar o NetBeans.....	1-6
1.2.2- Criar um projecto.....	1-7
1.3- O que são threads?	1-9
1.3.1- A diferença entre processo e thread.....	1-10
1.3.2- Os estados de uma thread.....	1-12
1.4- A thread main	1-15
1.4.1- Propriedades da thread main.....	1-17
1.4.2- Exercício com a thread main	1-18
1.5- Criar threads.....	1-22
1.5.1- Como criar uma thread?.....	1-23
1.5.2- Como controlar a execução da thread?	1-24
1.5.3- Exercício sobre criação de thread	1-25
1.6- join() e isAlive().....	1-28
1.6.1- Método join()	1-29
1.6.2- Método isAlive().....	1-30
1.6.3- Exercícios sobre join() e isAlive()	1-31
1.7- Atribuir prioridade	1-33
1.7.1- Como atribuir prioridade a uma thread?	1-34
1.7.2- Exercício sobre prioridade de threads (#1/2).....	1-35
1.7.3- Exercício sobre prioridade de threads (#2/2).....	1-37
1.8- Semáforo.....	1-40
1.8.1- Acesso concorrente a recursos partilhados	1-41
1.8.2- Como criar um semáforo?.....	1-42
1.8.2.1- Num objecto ou numa variável	1-43
1.8.2.2- Num método.....	1-44
1.8.3- Exercício: recurso partilhado sem semáforo (#1/2)	1-45
1.8.4- Exercício: recurso partilhado sem semáforo (#2/2)	1-47
1.8.5- Exercício: recurso partilhado com semáforo	1-49
1.8.5.1- Semáforo no método	1-49
1.8.5.2- Semáforo no objecto	1-50



Programação de Sistemas Distribuídos - Java para a Web

Capítulo 1 – Threads

José Aser Lorenzo
Pedro Nunes
Paulo Jorge Martins



Java Web
© Citeforma

1- Threads

Objectivos

- Compreender a diferença entre processo e thread;
- Criar threads e alterar as suas propriedades;
- Escrever pequenos programas que usam threads para programação concorrente;
- Usar semáforos nas situações em que a concorrência coloca em perigo a integridade dos recursos partilhados;



1.1- Objectivos

Todas as classes Java foram concebidas tendo em conta um dos objectivos principais da linguagem: permitir a execução de várias threads em simultâneo. No fim deste capítulo o formando compreenderá o que é uma thread, quais as classes envolvidas na sua programação e será capaz de escrever programas que usam as capacidades “*multithreading*” da linguagem Java.

Esta característica da linguagem é muito requerida, visto que os novos processadores não escalam na frequência de relógio, mas sim no número de cores. Os programas que usam várias threads aproveitam de forma mais eficiente os vários “cores”.

Sumário

- Preparar ambiente de trabalho;
- O que são threads?
- A thread main;
- Criar threads;
- join() e isAlive();
- Atribuir prioridade;
- Semáforo;



1.2- Preparar o ambiente de trabalho

Vamos preparar o ambiente de trabalho para este capítulo.

NetBeans – Instalação

- Instalar o JDK - requer privilégios de administrador porque configura “plugin” para o browser;
 - Se não tiver privilégios de administrador pode copiar a estrutura de directorias a partir de uma instalação já feita;
- Instalar o NetBeans:
 - Fazer download da versão que inclui Tomcat;
 - Executar o instalador, escolher “Customize” e escolher o Tomcat, em vez de GlassFish;
 - O NetBeans pergunta qual o JDK que deve usar (v1.5 ou posterior);



1.2.1- Instalar o NetBeans

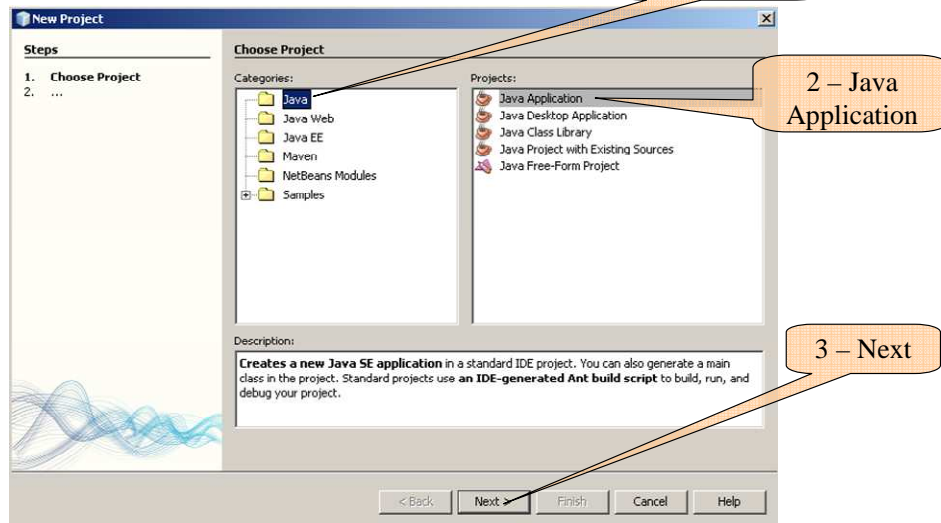
Neste curso vamos usar o NetBeans como ferramenta de apoio ao desenvolvimento. Trata-se de um IDE (*Integrated Development Environment*) desenvolvido pela comunidade em Open Source, que tem sido apoiado pela Sun. Pode ser obtido no seguinte endereço:

<http://www.netbeans.org>

NetBeans - Criar um projecto

#1/3

○ File → New Project



Java Web
© Citeforma

Capítulo 1 - Threads

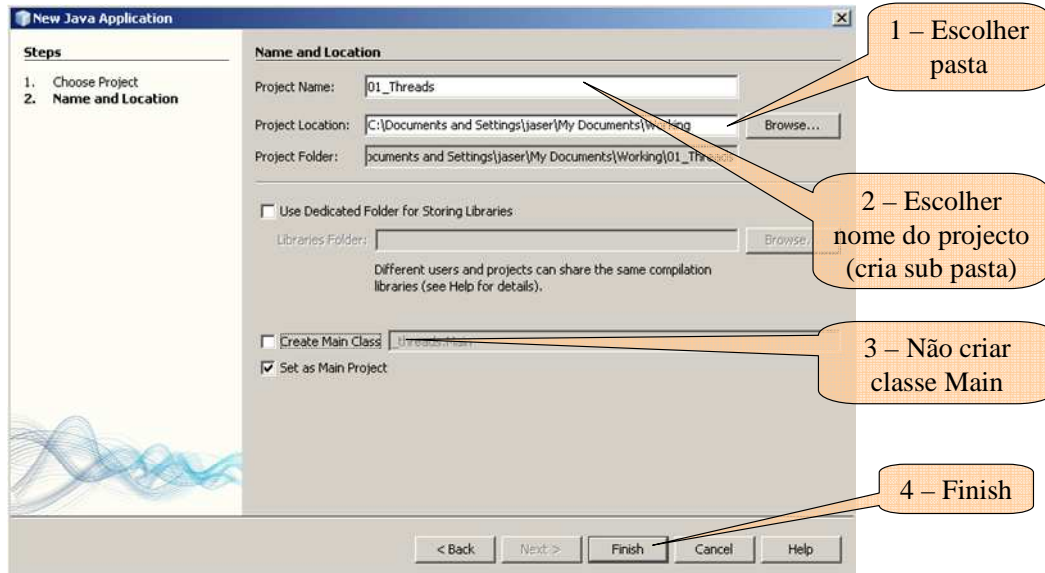
4

1.2.2- Criar um projecto

Vamos criar um projecto que irá conter os exercícios deste capítulo:

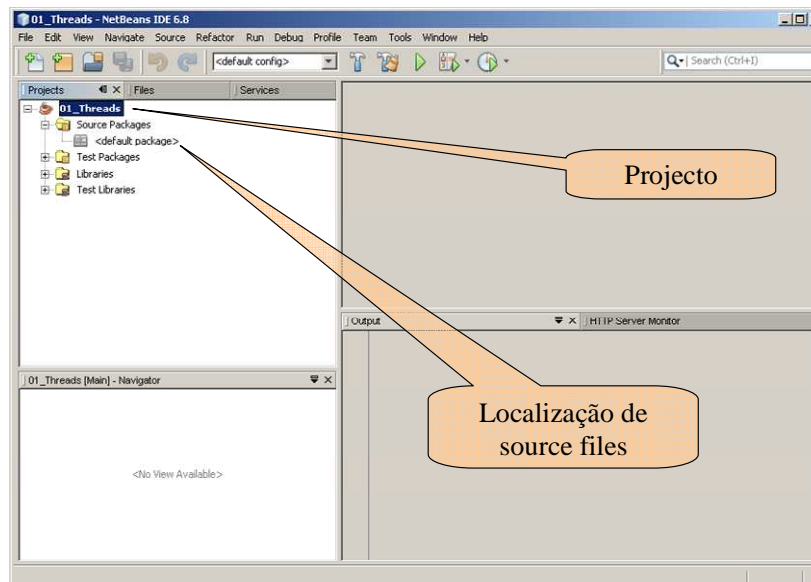
NetBeans - Criar um projecto

#2/3



NetBeans - Criar um projecto

#3/3



Sumário

- Preparar o ambiente de trabalho: JDev/NetBeans
- O que são threads?
- A thread main;
- Criar threads;
- join() e isAlive();
- Atribuir prioridade;
- Semáforo;



1.3- O que são threads?

A diferença entre processo e thread

- Um processo possui um **ambiente de execução próprio** e independente dos outros processos;
- Um sistema operativo multitarefa permite que vários processos sejam executados em simultâneo, de forma concorrente, **sem partilharem dados** entre si;
- Uma thread **corre dentro de um processo**, usando os recursos que o sistema operativo atribuiu ao processo;
- Enquanto os processos não podem **partilhar dados** entre si, as threads incluídas no mesmo processo podem fazê-lo, o que lhes aumenta a flexibilidade;
- A **troca de uma thread** por outra, dentro do mesmo processo, é uma operação muito mais leve para o sistema operativo que a troca de um processo por outro;



1.3.1- A diferença entre processo e thread

De uma forma simplista um processo é um programa executado pelo sistema operativo que possui um ambiente de execução próprio e independente dos outros processos. Esse ambiente de execução é formado pelas instruções e variáveis do programa e pelas estruturas de controlo criadas pelo sistema operativo. Um sistema operativo multitarefa permite que vários processos sejam executados em simultâneo, de forma concorrente, sem partilharem dados entre si.

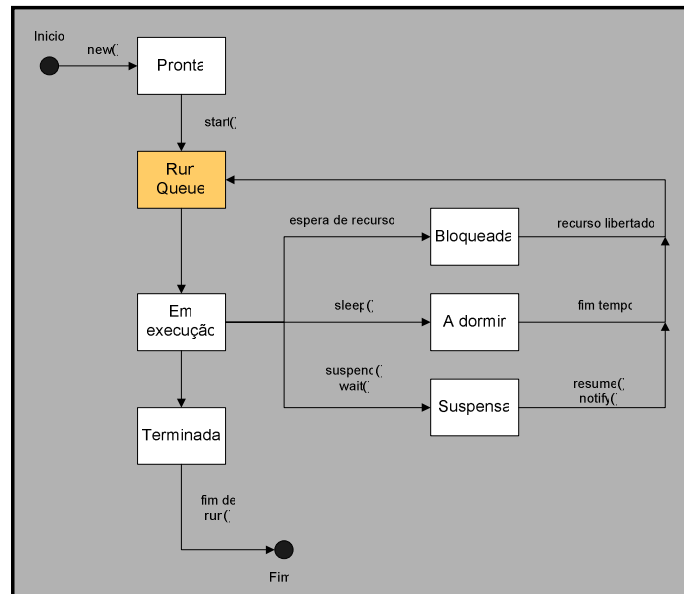
A multitarefa permite tirar mais rendimento dos recursos do computador, principalmente do processador, um recurso dispendioso e muito mais rápido que os outros. A título de exemplo compare a velocidade de funcionamento do processador (na ordem de 10^{-9} segundo) com a velocidade do disco (10^{-3} segundo). Para aproveitar este diferencial de tempo, o processo que está em execução é “adormecido” pelo sistema operativo enquanto espera por um recurso mais lento, dando o seu lugar a outro processo que entretanto será “acordado” e retoma a sua execução. Mesmo assim, enquanto um processo está em execução há períodos de desperdício que as threads tentam aproveitar.

Uma thread corre dentro de um processo, usando os recursos que o sistema operativo atribuiu ao processo. Este pode conter várias threads, cada uma com o seu ambiente de execução. Enquanto os processos não podem partilhar dados entre si, as threads incluídas no mesmo processo podem fazê-lo, o que lhes aumenta a flexibilidade.

Cada processador executa um processo de cada vez, assim como em cada processo, num determinado instante só há uma thread em execução. Mas a troca de uma thread por outra, dentro do mesmo processo, é uma operação muito mais leve para o sistema operativo que a troca de um processo por outro. Por estas razões a multitarefa com threads é mais leve para o sistema operativo que a multitarefa com processos.

Os estados de uma thread

#1/3



1.3.2- Os estados de uma thread

O ciclo de vida de uma thread é representado pelo diagrama de estados acima.

Os estados de uma thread

#2/3

- **Pronta** para ser executada - Fica neste estado quando é criada com `new()`. Após `start()` a thread vai para a Run Queue;
- **Run Queue** - Fila de threads que aguardam tempo de CPU. É gerida por FIFO se todas as threads têm a mesma prioridade. Se uma thread tem mais prioridade passa à frente das outras;
- **Em execução** – Se a thread é executada pela primeira vez inicia o método `run()`. Se a thread retoma a sua execução continua o método `run()` a partir do ponto onde parou;



- **Pronta** para ser executada. Fica neste estado quando é criada com **new()**, estando o objecto pronto a ser executado. Logo que o método **start()** seja chamado a thread vai para a Run Queue (fila de execução).
- **Run Queue (Fila de execução)** – Neste estado a thread está numa fila onde se encontram todas as threads que aguardam tempo de CPU. A fila é gerida pelo princípio FIFO (First In First Out) se todas as threads que aguardam têm a mesma prioridade. Se uma thread tiver maior prioridade passa à frente das outras.
- Em **execução** – O processador vai à fila de execução (Run Queue) retirar uma thread. Se essa thread é executada pela primeira vez então inicia o método **run()**. Se a thread retoma a sua execução então continua o método **run()** a partir do ponto onde parou da última vez;

Os estados de uma thread

#3/3

- **Bloqueada** – A execução da thread é interrompida porque está à espera de um recurso externo, por exemplo I/O. Logo que esta operação termine a thread passa para a fila de execução;
- **A dormir** – A execução da thread é interrompida pelo método `sleep(x)` durante x milissegundo. Passado este tempo regressa à fila de execução;
- **Suspensa** – A execução da thread foi interrompida porque os métodos `suspend()` ou `wait()` foram executados. A thread sai deste estado quando é notificada pelo método `notify()` ou `notifyAll()`, passando para a fila de execução;
- **Terminada** – A execução do método `run()` terminou;



- **Bloqueada** – A execução da thread é interrompida porque está à espera de um recurso externo, por exemplo I/O. Logo que esta operação termine a thread passa para a fila de execução;
- **A dormir** – A execução da thread é interrompida pelo método **`sleep(x)`** durante x milissegundos. Passado este tempo a thread regressa à fila de execução;
- **Suspensa** – A execução da thread foi interrompida porque os métodos **`suspend()`** ou **`wait()`** foram executados. A thread sai deste estado quando é notificada pelo método **`notify()`** ou **`notifyAll()`**, passando para a fila de execução;
- **Terminada** – A execução do método **`run()`** terminou pelo que terminou o ciclo de vida da thread;

Sumário

- Preparar o ambiente de trabalho: JDev/NetBeans
- O que são threads?
- A thread main;
- Criar threads;
- join() e isAlive();
- Atribuir prioridade;
- Semáforo;



1.4- A thread main

A thread Main

- Todos os programas Java possuem várias threads;
- As mais importantes são:
 - Main – Controla o programa principal;
 - Garbage collector (GC) – limpa objectos em memória que já não são utilizados para libertar recursos;
- O próximo programa mostra as propriedades da thread Main;



Propriedades da thread main

```
public class ThreadMain {  
    public static void main(String args[ ]) {  
        Thread t = Thread.currentThread();  
        System.out.println("Thread activa = "+t);  
        t.setName("Nome novo");  
        System.out.println("Thread activa = "+t);  
        try {  
            for (int n=5; n>0; n- -) {  
                System.out.println(n);  
                Thread.sleep(1000); //1 segundo  
            }  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
        System.out.println("Thread main terminada.");  
    }  
}
```

Ver a thread activa

Alterar o nome da thread

Alterar o tempo de execução da thread



1.4.1- Propriedades da thread main

Quando um programa Java entra em execução é iniciada a thread main. Todas as outras threads criadas no programa estarão dependentes desta. Quando o programa termina, esta thread termina, o que não obriga a que todas as threads dela dependentes tenham terminado.

Todas as threads podem ser controladas usando a classe Thread. O exemplo acima mostra algumas propriedades da thread main.

Exercício com a thread main

#1/4

- Escrever o programa do slide anterior e ver o resultado produzido;



Java Web
© Citeforma

Capítulo 1 - Threads

15

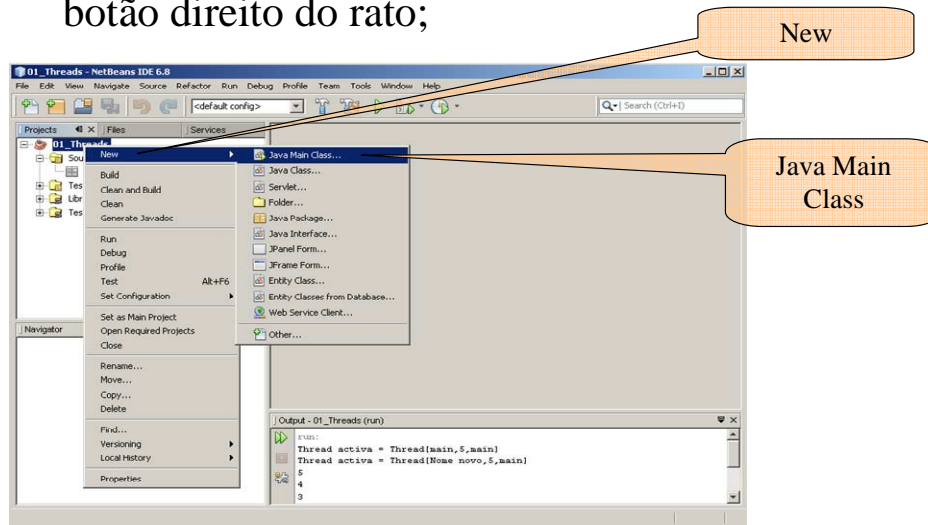
1.4.2- Exercício com a thread main

Vamos codificar o programa apresentado o slide anterior. Para isso precisamos criar uma classe.

Exercício com a thread main

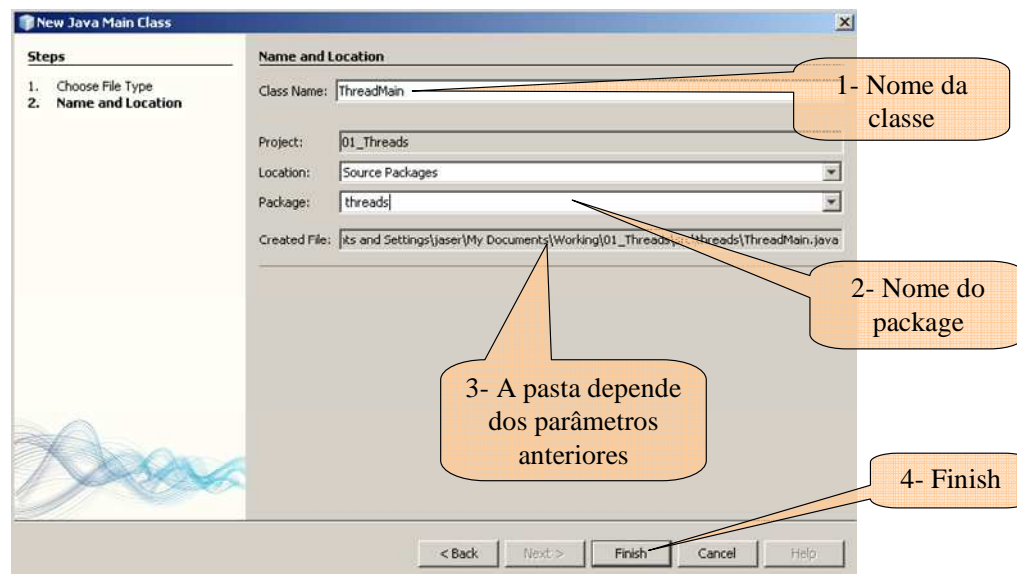
#2/4

- Seleccionar o projecto (01_Threads) e click com botão direito do rato;



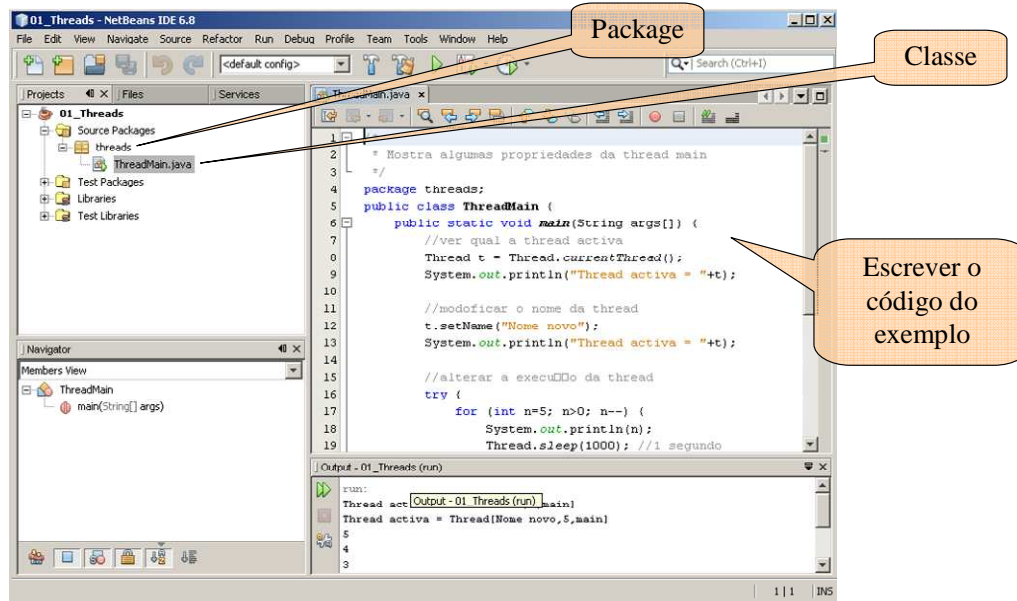
Exercício com a thread main

#3/4



Exercício com a thread main

#4/4



Java Web
© Citeforma

Capítulo 1 - Threads

18

A listagem abaixo contém o código completo do exemplo:



```

package threads;
public class ThreadMain {
    public static void main(String args[]) {
        //ver qual a thread activa
        Thread t = Thread.currentThread();
        System.out.println("Thread activa = "+t);

        //modificar o nome da thread
        t.setName("Nome novo");
        System.out.println("Thread activa = "+t);

        //alterar a execução da thread
        try {
            for (int n=5; n>0; n--) {
                System.out.println(n);
                Thread.sleep(1000); //1 segundo
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Thread main terminada.");
    }
}

```



Este programa produz o seguinte resultado:



```
Thread activa = Thread[main,5,main]
Thread activa = Thread[Nome novo,5,main]
5
4
3
2
1
Thread main terminada.
```



Para controlar a thread main é necessário criar uma referência sobre ela. Para obter essa referência usamos o método **currentThread()** da classe Thread.

O método **setName()** permite alterar o nome interno da thread. O nome da thread pode ser obtido usando o método **getName()**.



Se as características da thread não fossem alteradas o ciclo seria executado num instante. Dentro do ciclo a thread é forçada a adormecer por um segundo, o que retarda a sua execução. O método **sleep()** coloca a thread em estado “**dormindo**”, sendo o seu argumento o número de milissegundos de interrupção. O uso deste método obriga a tratar a excepção **InterruptedException**.



O método **toString()** da classe Thread imprime 3 argumentos:

- O nome interno da thread (primeiro “main” e depois “Novo nome”);
- A prioridade da thread, que por omissão é 5;
- O grupo à qual pertence a thread (main);

Sumário

- Preparar o ambiente de trabalho;
- O que são threads?
- A thread main;
- Criar threads;
- join() e isAlive();
- Atribuir prioridade;
- Semáforo;



1.5- Criar threads

Existem duas alternativas para criar uma thread dentro de um programa Java:

- Por herança da classe Thread redefinindo os seus métodos;
- Implementando a interface **Runnable**;

A primeira forma (herança) só é recomendada quando se pretendem alterar outras características da thread além do método run(). Isto porque o método **run()**, de todos os métodos definidos em Thread, é o único que normalmente é redefinido para programar threads. Este é o método implementado pela interface **Runnable**, que constitui a segunda alternativa. Para programar applets só a segunda forma é permitida, pois estas necessitam estender a classe **Applet**, tornando a implementação da interface a única forma de assumir o compromisso com o método run(). Todos os exemplos deste manual seguem a segunda alternativa.

Como criar uma thread?

- Qualquer classe pode criar uma thread desde que:
 1. Crie um objecto da classe Thread;
 2. Implemente a interface Runnable → tem que definir o método run();

```
Thread t = new Thread(Runnable obj, String nome);
```

Objecto da classe que possui o método run() que vai controlar a thread

Nome da thread

```
t.start()
```

Iniciar run()



Java Web
© Citeforma

Capítulo 1 - Threads

20

1.5.1- Como criar uma thread?

Qualquer classe pode controlar uma thread desde que:

- Crie um objecto da classe Thread;
- Implemente a interface Runnable usando o objecto anterior.

Para implementar a interface **Runnable** a classe tem que definir o método **run()**, que contém o código a executar na thread. Este método pode declarar variáveis, chamar outros métodos e criar objectos. A thread termina com o fim do método run().



```
Thread (Runnable objRun, String nome)
```

O primeiro parâmetro é um objecto da classe que implementa o interface Runnable e define qual o método **run()** a executar na thread. O segundo parâmetro define o nome interno da thread. Depois de criada, uma thread só entra em execução quando é invocado o método **start()** pertencente à classe Thread, este por sua vez é que invoca o método **run()**.

Se o método run() for invocado directamente, em vez de ser invocado normalmente pelo método start() da thread, o mais natural é que se trate de um erro de lógica, pois o código será executado no contexto da thread actual (possivelmente a thread main) e não em paralelo noutra thread.

Como controlar a execução da thread?

- A execução da thread é controlada pelo método `run()`:

```
public void run() {  
    try {  
        for (int i=iteracoes; i>0; i--) {  
            System.out.println(t.getName()+" "+i);  
            Thread.sleep(1000);  
        }  
    } catch (InterruptedException e) {  
        System.out.println(t.getName()+" interrompida");  
    }  
    System.out.println(t.getName()+" terminada");  
}
```

Ciclo for com um "sleep time"

O método `sleep()` obriga a tratar a exceção

Quando o `run()` terminar a thread termina



1.5.2- Como controlar a execução da thread?

No exemplo acima o método **run()** faz uma contagem decrescente desde o valor da variável "iteracoes" até 1, com intervalos de 1 segundo.

Exercício sobre criação de threads

- Definir a classe **MinhaThread** que cria e controla uma thread:
 - O seu construtor recebe 2 parâmetros:
 - O nome interno da thread;
 - O número de iterações que o respectivo método run() vai executar;
 - Define o método run() que controla a thread;
 - Não tem método main();
- Definir a classe **MinhaThreadRun** que cria três objectos da classe MinhaThread e os executa em concorrência, juntamente com a thread main;



1.5.3- Exercício sobre criação de thread

Código da classe MinhaThread:



```
package threads;
public class MinhaThread implements Runnable {
    public Thread t;
    private int iteracoes;

    //construtor
    public MinhaThread(String s, int n) {
        t= new Thread(this,s);
        System.out.println("Inicio da thread: "+t);
        iteracoes = n;
        t.start();
    }

    //definicao do comportamento da thread
    public void run() {
        try {
            for (int i=iteracoes; i>0; i--) {
                System.out.println(t.getName()+":"+i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println(t.getName()+" interrompida");
        }
        System.out.println(t.getName()+" terminada");
    }
}
```

```
}
}
```



O nome interno da thread é um parâmetro recebido no construtor.



O método **run()** faz uma contagem decrescente de um número recebido no construtor até 1, com intervalos de 1 segundo. O próprio construtor da classe inicia a thread.

No próximo exemplo criamos três threads da classe definida anteriormente, cada uma com 5 iterações. Quando os objectos são criados arrancam os respectivos métodos **run()**, o que põe as 3 threads em execução simultânea, em concorrência com a thread main.



```
package threads;
public class MinhaThreadRun {
    public static void main(String args[]) {
        System.out.println("Inicio da thread main");
        MinhaThread t1=new MinhaThread("MinhaThread-1",5);
        MinhaThread t2=new MinhaThread("MinhaThread-2",5);
        MinhaThread t3=new MinhaThread("MinhaThread-3",5);
        try {
            for (int n=10; n>0; n--) {
                System.out.println("Thread main:"+n);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println("Thread main interrompida.");
        }
        System.out.println("Thread main terminada");
    }
}
```



O resultado da execução deste programa pode variar, estando listado abaixo um possível output:



```
Inicio da thread main
Inicio da thread: Thread[MinhaThread-1,5,main]
Inicio da thread: Thread[MinhaThread-2,5,main]
Inicio da thread: Thread[MinhaThread-3,5,main]
MinhaThread-1:5
MinhaThread-2:5
MinhaThread-3:5
Thread main:10
MinhaThread-1:4
MinhaThread-2:4
MinhaThread-3:4
Thread main:9
```

```
MinhaThread-1:3
MinhaThread-2:3
MinhaThread-3:3
Thread main:8
MinhaThread-1:2
MinhaThread-2:2
MinhaThread-3:2
Thread main:7
MinhaThread-1:1
MinhaThread-2:1
MinhaThread-3:1
Thread main:6
MinhaThread-1 terminada
MinhaThread-2 terminada
MinhaThread-3 terminada
Thread main:5
Thread main:4
Thread main:3
Thread main:2
Thread main:1
Thread main terminada
```



Todas as threads têm 1 segundo de **sleep()**. A thread main é controlada por um ciclo com 10 iterações, enquanto cada uma das threads dependentes tem um ciclo com 5 iterações. Isto potencia que as threads dependentes terminem antes de main, mas não garante, como veremos a seguir.

Sumário

- Preparar o ambiente de trabalho;
- O que são threads?
- A thread main;
- Criar threads;
- **join() e isAlive();**
- Atribuir prioridade;
- Semáforo;



1.6- join() e isAlive()

Os métodos **isAlive()** e **join()** pertencem ambos à classe **Thread**. O método **join()** pode ser utilizado para garantir que a próxima instrução é executada apenas quando a thread actual tiver terminado. O método **isAlive()** permite verificar se a thread actual está em execução.

Método join()

- Pertence à classe Thread;
- A execução de join() só termina quando a thread tiver terminado, ou seja, é executado enquanto a thread estiver viva;
- É usado para garantir que a thread terminou;

```
MinhaThread t1=new MinhaThread("MinhaThread-1",2);  
  
t1.t.join();  
System.out.println("Fim join() t1");
```



1.6.1- Método join()

O método **join()** só termina quando a thread respectiva tiver terminado. Como se trata de um método bloqueante, podemos também indicar o tempo máximo de espera para a conclusão da thread.

Método `isAlive()`

- Pertence à classe `Thread`;
- Devolve `false` quando a thread terminou e `true` em todos os outros casos:

```
MinhaThread t1=new MinhaThread("MinhaThread-1",2);  
  
System.out.println("MinhaThread1 esta' em execucao? "+t1.t.isAlive());
```



1.6.2- Método `isAlive()`

O método **`isAlive()`** pode ser usado para efectuar um teste sobre o estado da thread, pois devolve **`true`** caso a thread esteja em execução e **`false`** se esta já tiver terminado. Quando devolve **`true`** significa que a execução da thread está algures depois da evocação do método `start()` e antes do método `run()` ter terminado.

Exercício sobre join() e isAlive()

- Definir a classe **MinhaThreadMultiJoin** que:
 - Cria 3 objectos da classe **MinhaThread** e os executa em concorrência, juntamente com a thread main;
 - Usa o método **join()** para garantir que a thread main só termina depois das outras threads terem terminado;
 - Usa o método **isAlive()** para verificar o resultado;



1.6.3- Exercícios sobre join() e isAlive()

Neste exercício são criados 3 objectos da classe **MinhaThread** (definida anteriormente) com número de iterações crescente. Vamos recorrer ao método **join()** para garantir que a thread main só termina depois das outras. Para isso invocamos **join()** sobre todas as threads criadas dentro do método **main()**.



```
package threads;
public class MinhaThreadMultiJoin {
    public static void main(String args[]) {
        System.out.println("Inicio da thread main");
        MinhaThread t1=new MinhaThread("MinhaThread-1",2);
        MinhaThread t2=new MinhaThread("MinhaThread-2",4);
        MinhaThread t3=new MinhaThread("MinhaThread-3",6);

        System.out.println("MinhaThread1 esta' em execucao? "+t1.t.isAlive());
        System.out.println("MinhaThread2 esta' em execucao? "+t2.t.isAlive());
        System.out.println("MinhaThread3 esta' em execucao? "+t3.t.isAlive());

        try {
            System.out.println("Esperar que as threads terminem");
            t1.t.join();
            System.out.println("Fim join() t1");
            t2.t.join();
            System.out.println("Fim join() t2");
            t3.t.join();
            System.out.println("Fim join() t3");
        }
    }
}
```

```

    } catch(InterruptedException e) {
        e.printStackTrace();
    }

    System.out.println("MinhaThread1 esta' em execucao? "+t1.t.isAlive());
    System.out.println("MinhaThread2 esta' em execucao? "+t2.t.isAlive());
    System.out.println("MinhaThread3 esta' em execucao? "+t3.t.isAlive());

    System.out.println("Thread main terminada");
}
}

```



O resultado da execução deste programa pode variar. Este é um possível output:



```

Início da thread main
Início da thread: Thread[MinhaThread-1,5,main]
Início da thread: Thread[MinhaThread-2,5,main]
Início da thread: Thread[MinhaThread-3,5,main]
MinhaThread1 esta' em execucao? true
MinhaThread2 esta' em execucao? true
MinhaThread3 esta' em execucao? true
Esperar que as threads terminem
MinhaThread-1:2
MinhaThread-2:4
MinhaThread-3:6
MinhaThread-1:1
MinhaThread-3:5
MinhaThread-2:3
MinhaThread-2:2
MinhaThread-3:4
MinhaThread-1 terminada
Fim join() t1
MinhaThread-2:1
MinhaThread-3:3
MinhaThread-2 terminada
Fim join() t2
MinhaThread-3:2
MinhaThread-3:1
MinhaThread-3 terminada
Fim join() t3
MinhaThread1 esta' em execucao? false
MinhaThread2 esta' em execucao? false
MinhaThread3 esta' em execucao? false
Thread main terminada

```



Pelo output do programa é possível notar que a execução do método **join()** na thread **t** de **t1** só termina com o fim dessa thread. Nessa altura executa o **println()** e vai executar o **join()** sobre **t** de **t2**. Com o fim deste vai executar o **join()** de **t** de **t3**.



Usámos um número de iterações crescente para aumentar a probabilidade das threads **t2** e **t3** ainda se encontrarem em execução antes de executarmos **t2.t.join()** e **t3.t.join()**.

Sumário

- Preparar o ambiente de trabalho;
- O que são threads?
- A thread main;
- Criar threads;
- join() e isAlive();
- Atribuir prioridade;
- Semáforo;



1.7- Atribuir prioridade

A linguagem Java permite a atribuição de prioridade a cada thread. Quando estão várias threads na fila de execução a que tiver maior prioridade será escolhida em detrimento de todas as outras. Isto significa que a prioridade só se nota quando há várias threads à espera de CPU, pois aumenta a probabilidade da thread com maior prioridade receber mais CPU. Se não houver concorrência no acesso ao CPU ("conflito de interesses") a diferença nas prioridades não será notada.

Como atribuir prioridade a uma thread?

- O método `setPriority()` define a prioridade da thread;
- O seu argumento é um número inteiro:
 - `Thread.MIN_PRIORITY`;
 - `Thread.NORM_PRIORITY`;
 - `Thread.MAX_PRIORITY`;
- A thread com maior prioridade sai primeiro da Run Queue;
- A prioridade só é importante em **situações de escassez de recursos**;



1.7.1- Como atribuir prioridade a uma thread?

Quando uma thread é criada recebe a mesma prioridade da thread que a lançou. A thread main recebe a prioridade associada ao número da constante **Thread.NORM_PRIORITY**.

O valor da prioridade da thread pode ser alterado usando o método **setPriority()**, que recebe como argumento um número inteiro entre **Thread.MIN_PRIORITY** e **Thread.MAX_PRIORITY**. A alteração deve ser feita antes de se executar o método `start()`.

Exercício sobre prioridade de threads #1/2

- Definir a classe **ThreadPrioridade** que cria e controla uma thread. É diferente de **MinhaThread** porque:
 - O construtor recebe a prioridade a atribuir à thread e o tempo de sleep expresso em mili + nano segundos;
 - Só entra em execução quando se invoca o seu método **iniciar()**;
 - Só pára quando se invoca o método **parar()**;
 - Usamos o método **join()** para ter a certeza que pára;
 - O tempo de CPU recebido pela thread é medido por um contador **n**;



1.7.2- Exercício sobre prioridade de threads (#1/2)

A classe abaixo cria uma thread seguindo um mecanismo semelhante ao que foi usado nos exemplos anteriores. A diferença está no construtor que recebe como parâmetro a prioridade que será atribuída à thread antes da execução do método **start()**:



```
package threads;
public class ThreadPrioridade implements Runnable {
    private Thread t;
    private long n;
    private int tempoMili, tempoNano;
    private boolean emExecucao;

    public ThreadPrioridade(String nome, int p, int tMili, int tNano) {
        n = 0;
        t = new Thread(this, nome);
        t.setPriority(p);
        this.tempoMili = tMili;
        this.tempoNano = tNano;
    }

    public void run() {
        while (emExecucao) {
            n++;
            try {
                //Paragem em milissegundo e nanosegundos
                Thread.sleep(tempoMili, tempoNano);
            } catch (InterruptedException e) {}
        }
    }
}
```

```
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public void iniciar() {
        emExecucao = true;
        t.start();
    }

    public void parar() {
        emExecucao = false;
        try {
            t.join();
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
    }

    public void vern() {
        System.out.println(t.getName() + ".n=" + n);
    }
}
```



Ao contrário dos exemplos anteriores esta thread não entra em execução quando o construtor é invocado, mas apenas quando se chama o método **iniciar()**. A execução da thread termina quando se chama o método **parar()** e neste usamos o método **join()** para ter a certeza que a thread pára.

Exercício sobre prioridade de threads #2/2

- Definir a classe **ThreadPrioridadeRun** que:
 - Cria três objectos da classe **ThreadPrioridade** atribuindo-lhes prioridades muito diferentes;
 - Cria condições de concorrência colocando tempos de sleep() muito baixos;
 - Executa as 3 threads em concorrência, arrancando a de menor prioridade em primeiro lugar e terminando-a em último;
 - Verifica o tempo de CPU que cada thread recebeu;



1.7.3- Exercício sobre prioridade de threads (#2/2)

Para testar a prioridade entre threads vamos criar a classe abaixo:



```
package threads;
public class ThreadPrioridadeRun {
    public static void main(String args[]) throws InterruptedException {
        //prioridade máxima para main
        Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
        //definir diferentes prioridades e tempos de sleep
        ThreadPrioridade baixa =
            new ThreadPrioridade("Baixa",Thread.MIN_PRIORITY,0,0);
        ThreadPrioridade media =
            new ThreadPrioridade("Media",Thread.NORM_PRIORITY,0,0);
        ThreadPrioridade alta =
            new ThreadPrioridade("Alta",Thread.MAX_PRIORITY,0,0);
        baixa.iniciar();
        media.iniciar();
        alta.iniciar();
        Thread.sleep(10000);
        alta.parar();
        media.parar();
        baixa.parar();
        baixa.vern();
        media.vern();
        alta.vern();
    }
}
```



Note que a thread de maior prioridade é parada em primeiro lugar, para não lhe conferir mais tempo de execução.



O tempo usado no método **sleep()** faz a diferença, pois quanto maior for o seu valor, menor será a concorrência pelo CPU. Para o demonstrar vamos executar a classe com 3 valores diferentes:

Definir sleep(100,0):



```
ThreadPrioridade baixa =  
    new ThreadPrioridade("Baixa",Thread.MIN_PRIORITY,100,0);  
ThreadPrioridade media =  
    new ThreadPrioridade("Media",Thread.NORM_PRIORITY,100,0);  
ThreadPrioridade alta =  
    new ThreadPrioridade("Alta",Thread.MAX_PRIORITY,100,0);
```

Definir sleep(0,10):



```
ThreadPrioridade baixa =  
    new ThreadPrioridade("Baixa",Thread.MIN_PRIORITY,0,10);  
ThreadPrioridade media =  
    new ThreadPrioridade("Media",Thread.NORM_PRIORITY,0,10);  
ThreadPrioridade alta =  
    new ThreadPrioridade("Alta",Thread.MAX_PRIORITY,0,10);
```

Definir sleep(0,0):



```
ThreadPrioridade baixa =  
    new ThreadPrioridade("Baixa",Thread.MIN_PRIORITY,0,0);  
ThreadPrioridade media =  
    new ThreadPrioridade("Media",Thread.NORM_PRIORITY,0,0);  
ThreadPrioridade alta =  
    new ThreadPrioridade("Alta",Thread.MAX_PRIORITY,0,0);
```

O quadro abaixo mostra como tempo de CPU recebido pela thread varia em função do tempo de **sleep()**:

Prioridade	sleep(100)	sleep(0,10)	sleep(0)
Baixa	92	5067	1919154
Média	92	5097	2436884
Elevada	92	5102	2455168



Se o teste for executado com 100 milissegundos não haverá concorrência e portanto todas as threads recebem o mesmo tempo de CPU. Isto porque enquanto uma “dorme” sobra CPU para atender as outras.

Com 10 nanossegundos já existe alguma concorrência, pois em algumas situações encontram-se várias na fila de execução e portanto é escolhida a que tem maior prioridade, recebendo esta mais CPU.

Quando não existe tempo de sleep() existe muita concorrência e nesta situação a thread que tem maior prioridade é escolhida muitas vezes em detrimento das outras.



Para evitar conflitos (concorrência) o programador deve assegurar-se que a thread é suspensa por um período de tempo que permita ao sistema dedicar tempo às outras threads

Sumário

- Preparar o ambiente de trabalho;
- O que são threads?
- A thread main;
- Criar threads;
- join() e isAlive();
- Atribuir prioridade;
- Semáforo;



1.8- Semáforo

Acesso concorrente a recursos partilhados

- As threads executam trabalho em paralelo e podem **partilhar recursos** entre si;
- O acesso em simultâneo a um recurso partilhado pode criar **problemas de consistência** ou coerência;
- Para evitar este problema usamos **semáforos**;
- O semáforo é **activado pela primeira** thread que acede ao recurso;
- Enquanto o semáforo estiver activado as outras threads **aguardam numa fila**;



1.8.1- Acesso concorrente a recursos partilhados

As threads executam trabalho em paralelo e podem partilhar recursos entre si, o que pode criar problemas no acesso aos recursos partilhados. Suponha que foi criada uma lista encadeada em memória e que duas threads inserem, alteram e eliminam nós concorrentemente. O que sucede se uma thread estiver a alterar um nó enquanto a outra o está a apagar?

Para evitar estes conflitos a linguagem Java usa um mecanismo de semáforos que permite bloquear o acesso a um objecto ou às variáveis manipuladas por um método desse objecto. Quando uma thread activa um semáforo sobre um recurso, todas as outras threads que o tentam aceder ficam no estado **bloqueada** até que o recurso seja libertado.

Como criar um semáforo?

- O qualificador **synchronized** cria um semáforo que:
 - Evita os acessos concorrentes aos recursos por ele abrangidos;
 - Cria uma fila de espera;
- O qualificador **synchronized** pode ser usado:
 - Num objecto;
 - Numa variável de objecto;
 - Numa variável de classe;
 - Num método de objecto;
 - Num método de classe;



1.8.2- Como criar um semáforo?

Em Java criamos um semáforo com o método `synchronized()` que pode ser utilizado em diferentes contextos:

Synchronized

#1/2

○ Num objecto:

```
public void metodo() {  
    ...  
    synchronized (this) {  
        /* this (este objecto) está reservado */  
    }  
    ...  
}
```


○ Numa variável de objecto:

```
public void metodo() {  
    ...  
    synchronized (recurso) {  
        /* a variável recurso está reservada */  
    }  
    ...  
}
```




1.8.2.1- Num objecto ou numa variável

Para serializar o acesso a um objecto devemos usar o método `synchronized()` desta forma:



```
public void metodo() {  
    ...  
    synchronized (this) {  
        /* this reservado */  
    }  
    ...  
}
```

O mesmo método pode ser usado para serializar o acesso a uma variável de objecto ou de classe:



```
public void metodo() {  
    ...  
    synchronized (recurso) {  
        /* recurso reservado */  
    }  
    ...  
}
```

Synchronized

#2/2

- Num método de objecto:

```
public synchronized void metodo() {  
    ...  
}
```

- Num método de classe:

```
public static synchronized void metodo() {  
    ...  
}
```



1.8.2.2- Num método

O qualificador **synchronized** também pode ser usado sobre um método de objecto ou de classe, como mostram os dois exemplos abaixo:



```
public synchronized void metodo() {  
    ...  
}
```



```
public static synchronized void metodo() {  
    ...  
}
```

Exercício: recurso partilhado **sem** semáforo #1/2

- Criar a classe **SemaforoRecursoPartilhado** que:
 - Possui uma variável de objecto do tipo String;
 - Define o método escreve() que recebe duas Strings: começa por guardar a primeira String na variável de objecto, adormece por 1 segundo e quando acorda adiciona a segunda String à mesma variável. Termina escrevendo o valor final da variável de objecto;
 - A String é o **recurso partilhado** que será acedido concorrentemente;
- Criar a classe **SemaforoThreadChamadora** que:
 - Cria uma thread que recebe um objecto da classe anterior;
 - O método run() vai invocar o método escreve() desse objecto;



1.8.3- Exercício: recurso partilhado sem semáforo (#1/2)

No exemplo seguinte é feito um acesso concorrente a um recurso partilhado que não tem a protecção de um semáforo.

A classe **SemaforoRecursoPartilhado** escreve no “*standard output*” o conteúdo de uma String, cujo valor é recebido em dois parâmetros. O primeiro valor define o conteúdo inicial da String. O segundo valor é adicionado à String depois do método adormecer por 1 segundo:




```
package threads;

public class SemaforoRecursoPartilhado {
    String s = "";

    public void escreve(String s1, String s2) {
        s = s1;
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        s = s + s2;
        System.out.println(s);
    }
}
```

A listagem abaixo contém a classe **SemaforoThreadChamadora**, que cria uma thread que recebe um objecto da classe **SemaforoRecursoPartilhado** e duas Strings. Estas serão passadas para o método **escreve()** quando o método **run()** o invocar, o que é feito durante a execução da thread.



```
package threads;

public class SemaforoThreadChamadora implements Runnable {
    Thread t;
    String s1, s2;
    SemaforoRecursoPartilhado srp;

    public SemaforoThreadChamadora(SemaforoRecursoPartilhado e, String s1, String
s2) {
        srp = e;
        this.s1 = s1;
        this.s2 = s2;
        t = new Thread(this);
        t.start();
    }

    @Override
    public void run() {
        srp.escreve(s1, s2);
    }
}
```

Exercício: recurso partilhado **sem** semáforo #2/2

- Criar a classe **SemaforoRun** que cria um objecto da classe **SemaforoRecursoPartilhado** e 3 objectos da classe **SemaforoThreadChamadora**;
- A variável String da classe **SemaforoRecursoPartilhado** será acedida de forma concorrente pelos 3 objectos;
- Verificar a **confusão** gerada nos valores armazenados na variável;



1.8.4- Exercício: recurso partilhado sem semáforo (#2/2)

No exemplo seguinte a classe **SemaforoRun** cria o recurso **SemaforoRecursoPartilhado** e 3 objectos da classe **SemaforoThreadChamadora** que partilham esse recurso.



```
package threads;
public class SemaforoRun {
    public static void main(String args[]) {
        SemaforoRecursoPartilhado srp = new SemaforoRecursoPartilhado();
        SemaforoThreadChamadora t1 =
            new SemaforoThreadChamadora(srp, "[1", "1]");
        SemaforoThreadChamadora t2 =
            new SemaforoThreadChamadora(srp, "[2", "2]");
        SemaforoThreadChamadora t3 =
            new SemaforoThreadChamadora(srp, "[3", "3]");
    }
}
```



A execução desta classe produz diferentes resultados, que dependem da forma como as threads são retiradas da "run queue". Um desses resultados é o seguinte:



```
[31]  
[31]2]  
[31]2]3]
```



Isto não é o que pretendíamos obter. A primeira thread atribui o valor [1 à variável partilhada e adormece. A segunda thread recebe CPU e “esmaga” o valor da variável com [2, adormecendo depois disso. A terceira thread entra em execução e esmaga o valor anterior, escrevendo [3.

Entretanto a primeira thread acorda e adiciona ao conteúdo da variável 1], escrevendo o resultado e terminando a sua execução. Entra então em execução a segunda thread, que adiciona à variável 2], escreve e termina. A terceira thread vai fazer o mesmo, adicionando 3].

Exercício: recurso partilhado **com** semáforo

- Corrigir o código do exercício anterior:
 1. Colocando um semáforo no **método** partilhado;
 2. Colocando um semáforo no **objecto** partilhado;



1.8.5- Exercício: recurso partilhado com semáforo

1.8.5.1- Semáforo no método

Para corrigir o problema anterior vamos colocar um semáforo no método **escreve()**, o que é feito com o qualificador **synchronized**. O semáforo é activado pela primeira thread que executa o método e fica activo enquanto o método estiver em execução. Durante esse tempo todas as threads que pretendam aceder ao recurso ficam à espera na fila. Logo que o semáforo é retirado é escolhida uma thread da fila de espera, que entra em execução e repete o processo.



```
public synchronized void escreve(String msg)
```



O output produzido após esta alteração aparece ordenado, pois o acesso ao recurso é feito em série e não em paralelo:



[11]
[22]
[33]

1.8.5.2- Semáforo no objecto

Como alternativa podemos colocar o semáforo no objecto, como mostra o exemplo abaixo:

```
public class SemaforoThreadChamadora implements Runnable {  
  
    . . .  
  
    public void run() {  
        synchronized (srp) {  
            srp.escreve(s1, s2);  
        }  
    }  
}
```

Esta alternativa provoca um aviso por parte do IDE, pois estamos a colocar o qualificador `synchronized` sobre uma variável que não é final.

Sumário

- Preparar o ambiente de trabalho;
- O que são threads?
- A thread main;
- Criar threads;
- join() e isAlive();
- Atribuir prioridade;
- Semáforo;

