Concorrência em Java

Threads em Java

Máquina Virtual Java, Processos e Threads

– Cada instância da JVM corresponde a um processo do sistema operacional hospedeiro.
– A JVM não possui o conceito de processos – apenas implementa threads internamente.
– Ao ser iniciada, a JVM executa o método **main()** do programa na thread principal, e novas threads podem ser criadas a partir desta.
– Threads de uma mesma JVM compartilham memória, portas de comunicação, arquivos e outros recursos.

Threads

- Implementação de Threads no Java
  - Green Threads
    - Usadas em sistemas sem suporte a threads.
    - Threads e escalonamento implementados pela máquina virtual Java.
  - Threads Nativas
    - Usa threads nativas e escalonador do S.O.
    - Usada nos sistemas suportados oficialmente.
      - Linux, Solaris e Windows
    - Ideal em máquinas com >1 processador.

Ciclo de Vida de uma Thread

Estados das Threads em Java

- **Pronta**: poderia ser executada, mas o processador está ocupado.
- **Rodando**: em execução.
- **Suspensa**:
  - Bloqueada: aguarda operação de I/O.
  - Em Espera: aguarda alguma condição.
  - Dormindo: suspensa por um tempo.

Classe **Thread** e Interface **Runnable**

```
public class Thread extends Object
    {
        implements Runnable {
            // Métodos da classe thread
        }
        public interface Runnable {
            public void run();   // Código executado
                                 por uma thread
        }
    }
```

Principais métodos de **java.lang.Thread**

- **Construtores:** Thread(), Thread (Runnable), Thread(String), Thread(Runnable, String), ...
- **Método run()**: contém o código executado pela Thread; herdado da interface **Runnable**; implementado pela Thread ou por um objeto passado como parâmetro para seu construtor.
- **Método start()**: inicia a Thread em paralelo com a Thread que a criou, executando o código contido no método run().

Threads na Linguagem Java

- O **conceito de thread** está presente em Java através da classe **java.lang.Thread** .
- Java também oferece :
  - Mecanismos para **sincronização e controle de concorrência entre threads**.
  - Classes para **gerenciamento de grupos (pools) de threads**.
  - Classes da API que podem ser acessadas concorrentemente (thread-safe).

Criando Threads no Java usando herança

- Definir uma classe que estenda **Thread** e implemente o método **run()**:

```
public  class  MyThread extends Thread {
     public void run() {
              // código da thread
     }
 }
```

- Criar thread e chamar **start()**, que executa **run()**:

```
... ...

MyThread  t   = new MyThread();    // cria a thread
   t.start();   // inicia a thread
... ...
```

Criando Threads no Java usando herança

- Definir uma classe que implemente **Runnable**:

```
public class MyRun implements Runnable {
  public void run() {
        // código da thread
    ...
}
```

- Criar uma **Thread** passando uma instância do **Runnable** como parâmetro e chamar **start()**:

```
...
Thread  t  =  new  Thread (new MyRun());  // cria a thread

t.start();   // inicia thread
...
```

Criando Threads no Java sem usar herança

- Criar um **Runnable**, definindo o método **run()**, instanciar a thread passando o **Runnable** e chamar **start()** :

```
...
Runnable  myRun  =  new Runnable() {
   // cria o runnable
   public void run() {
      ... // código da thread
   }
};
...
new Thread(myRun).start();  // cria e inicia a thread
…
```

Criando Threads no Java sem usar herança

- Criar uma **Thread**, definindo o método **run(),** e chamar **start()** :

```
...

Thread  myThread  =  new Thread() {
    // cria a thread
    public void run() {
        ... // código da thread
    }
};
...
myThread.start();   // executa a thread
```

Nome da Thread

- Identificador não-único da Thread.
- Pode ser definido ao criar a Thread com **Thread(String)** ou **Thread(Runnable, String)**
- Se não for definido na criação, o nome da Thread será “Thread-n” (com n incremental)
- Pode ser redefinido com **setName(String)**
- Pode ser obtido através do método **getName()**

Prioridade de Threads

- Valor de 1 a 10; 5 é o default.
- Threads herdam prioridade da *thread* que a criou.
- Modificada com **setPriority(int)** .
- Obtida com **getPriority()** .

Prioridades de Threads

- 10 A -> B -> C
- 9 D -> E
- 8 F
- 7 G -> H
- 6 I
- 5 J -> K -> L
- 4 M
- 3 N -> O
- 2 P
- 1 Q

Escalonamento de Threads

- **Threads com prioridades iguais** são escalonadas em ***round-robin*** (rodízio), com cada uma ativa durante um ***quantum***.

Outros métodos de **java.lang.Thread**

- Obter Referência da Thread em Execução:

  currentThread()
- Suspender Execução: **sleep(long)**, **interrupt()**
- Aguardar fim da Thread: **join()** , **join(long)**
- Verificar Estado: **isAlive()** , **isInterrupted()**
- Liberar o Processador: **yield()**
- Destruir a Thread: **destroy()**

Exemplo Thread Sleep

```
public class ThreadSleep extends Thread {
    private long tempo = 0;
    public ThreadSleep(long tempo) { this.tempo = tempo; } // Construtor
    public void run() { // Código da Thread
        System.out.println(getName() +" vai dormir por "+ tempo +" ms.");
        try {
            sleep(tempo);
            System.out.println(getName() + " acordou.");
        } catch (InterruptedException e) { e.printStackTrace(); }
    }
    public static void main(String args[]) {
          for (int i=0; i<10; i++)
          // Cria e executa as Threads
          new ThreadSleep((long)(Math.random()*10000)).start();
    }
}
```

Grupos de Threads

- Às vezes é útil identificar várias threads como pertencentes a um grupo de threads.
- A classe ThreadGroup contém métodos para criar e manipular grupos de threads.
- Durante a construção, o grupo recebe um nome único, através de um argumento String.

Grupos de Threads

- A classe **ThreadGroup** define um **grupo de *threads*** que são controladas conjuntamente.

- O **grupo de *threads*** é definido usando o construtor Thread(ThreadGroup, ...).

Criando grupo de threads

```
// Cria grupo de Threads
ThreadGroup  myGroup  =  new ThreadGroup("MyThreads");
...
// Cria Threads e as insere no grupo
Thread  myThread1  = new MyThread(myGroup,"MyThread"+i);
Thread  myThread2  = new MyThread(myGroup,"MyThread"+i);
...
// Interrompe todas as Threads do grupo
group.interrupt();
...
```

Variáveis Locais em Threads

- A classe **ThreadLocal** permite criar variáveis locais para Threads (ou seja, que têm valores distintos para cada Thread)

Variáveis Locais em Threads

```
// Cria variável local
static ThreadLocal  valorLocal  = new ThreadLocal();
...
// Define o valor da variável para esta Thread
valorLocal.set( new  Integer(10) );
// Obtém o valor de var correspondente a esta Thread
int  valor  = ( (Integer) valorLocal.get() ).intValue(); // valor = 10
// Outra thread pode acessar a mesma variável e obter outro
    valor
int  valor  = ( (Integer) valorLocal.get() ).intValue(); // valor = 0
```

Gerenciando
Grupos de Threads

- java.lang.Object
  java.util.concurrent.Executors

- A classe **Executors** estende a classe raiz **Object**.

- A classe **Executors** gerencia um **grupo de threads**.

  – Interface **Executor**
  – Interface **ExecutorService**

Gerenciando a Execução de Threads

- Gerenciamento através da classe **Executors**:
- **SingleThreadExecutor** usa uma **thread** para executar atividades seqüencialmente.
- **FixedThreadPool** usa um **grupo de threads** de tamanho fixo para executar atividades.
- **CachedThreadPool** cria **threads sob demanda**.

Threads e Atividades

- Java usa ***threads*** – fluxos de execução independentes de outros fluxos – para representar **atividades independentes simultâneas**.

Escalonamento de Atividades

- Um **ScheduledExecutorService** permite escalonar a **execução de atividades**, definindo um atraso para início da atividade e/ou um período de repetição entre execuções.
  - **SingleThreadScheduledExecutor** usa uma thread para todas as atividades escalonadas.
  - **ScheduledThreadPool** cria um grupo de threads para executar as atividades.

Exemplo de uso de Executors

```
import java.util.concurrent.*;
  public class ExecutorTest {
    public static void main(String args[]) {
      ExecutorService exec = Executors newSingleThreadExecutor();
// ExecutorService exec = Executors.newFixedThreadPool(5);
// ExecutorService exec = Executors.newCachedThreadPool();
      for (int i=0; i<10; i++) {
      // Cria e executa as threads
          exec.execute (new ThreadSleep( ( long )( Math.Random()*10000)));
      }
      // Encerra o executor assim que as threads terminarem
      exec.shutdown();
    }
  }
```

Escalonamento com **ScheduledExecutorService**

```
import java.util.concurrent.*;
public class ScheduleThread extends Thread {
   public void run () { System.out.println(getName() + " executada."); }
   public static void main(String args[]) {
      ScheduledExecutorService exec =
         Executors.newSingleThreadScheduledExecutor() ;
      for (int i=0; i<10; i++) {
         long delay = (long) (Math.random()*10000);
         System.out.println("Thread-" + i + " será executada em "+ tempo + "ms.");
        // Escalona a execução da thread
        exec.schedule(newScheduleThread(),
                              delay, TimeUnit.MILLISECONDS);
      }
      exec.shutdown();
  }
}
```

Execução periódica de Threads com **ScheduledExecutorService**

```
import java.util.concurrent.*;
public class CountdownThread extends Thread {
    private static long tempo = 0, cont = 10, espera = 5, intervalo = 1;
    public CountdownThread(long tempo) { this.tempo = tempo; }
    public void run () {
       System.out.println(tempo +"segundos para o encerramento.");
       tempo--; }
    public static void main(String args[]) {
       ScheduledExecutorService  exec  =  Executors.new
     SingleThreadScheduledExecutor();
            exec.scheduleAtFixedRate(new  CountdownThread(cont),
                                              espera, intervalo, TimeUnit.SECONDS);
       try { exec.awaitTermination(cont+espera, TimeUnit.SECONDS);
       } catch (InterruptedException ie) { ie.printStackTrace(); }
       exec.shutdown();
    }
}
```