

Java Threads

Introdução

- ◆ O único mecanismo de concorrência suportado explicitamente pela linguagem Java é multi-threading.
- ◆ Os mecanismos de gerenciamento e sincronização de threads foram incorporados diretamente à linguagem (java.lang).
- ◆ Um thread Java é representado por um objeto da classe Thread (java.lang.Thread).

Introdução

- ◆ Quando uma aplicação Java é executada:
 - ◆ A JVM cria um objeto do tipo Thread cuja tarefa a ser executada é descrita pelo método `main()`.
 - ◆ O Thread é iniciado automaticamente.
 - ◆ Os comandos descritos pelo método `main()` são executados sequencialmente até que o método termine e o thread acabe.

Criando Threads

- ◆ Existem duas formas de criar explicitamente um thread em Java:
 - ◆ Estendendo a classe `Thread` e instanciando um objeto desta nova classe.
 - ◆ Implementando a interface `Runnable`, e passando um objeto desta nova classe como argumento do construtor da classe `Thread`.
- ◆ Nos dois casos a tarefa a ser executado pelo thread deverá ser descrita pelo método `run()`.

Exemplo

Estendendo Thread

```
public class SimpleThread extends Thread {
    private String myName;
    public SimpleThread(String str) {
        myName = str;
    }
    public void run() {
        // Código executado pelo thread
        for (int i = 0; i < 10; i++)
            System.out.println(i + " " + myName);
        System.out.println("DONE! " + myName);
        // fim
    }
}
```

Exemplo

Estendendo Thread

- ◆ Para iniciar o thread é necessário instanciar um objeto da nova classe e invocar o método `start()` :

```
public class ThreadDemo {
    public static void main (String[] args) {
        (new SimpleThread("thread 1")).start();
        (new SimpleThread("thread 2")).start();
    }
}
```

A Interface Runnable

- ◆ Como Java não permite herança múltipla, a necessidade de estender a classe Thread restringe a criação de subclasses a partir de outras classes genéricas.
- ◆ Através da utilização da interface Runnable é possível criar classes que representem um thread sem precisar estender a classe Thread. Neste caso pode-se estender outras classes genéricas.
- ◆ A criação de um novo thread é feita através da instanciação de um objeto thread usando o objeto que implementa a interface Runnable.

Exemplo Implementando Runnable

```
class BasicThread2 implements Runnable{
    // método que descreve o código a ser
    // executado pelo thread
    public void run() {
        ...
    }
}

class Teste{
    public static void main(){
        // Cria um objeto que possui um método run()
        Runnable runnable = new BasicThread2();
        // Cria um novo thread usando um objeto runnable
        Thread thread = new Thread(runnable);
        // inicia o thread
        thread.start();
        ...
    }
}
```

Exemplo

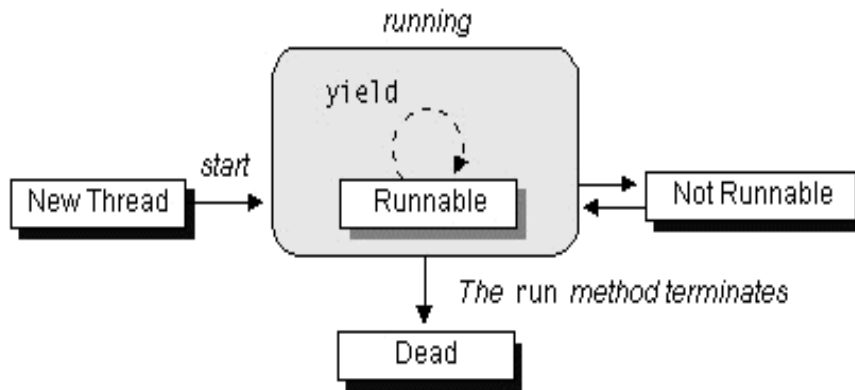
Implementando Runnable

```
public class Clock extends Applet implements Runnable {
    private Thread clockThread = null;
    public void start() {
        if (clockThread == null) {
            clockThread = new Thread(this, "Clock");
            clockThread.start();
        }
    }
    public void run() {
        Thread myThread = Thread.currentThread();
        while (clockThread == myThread) {
            repaint();
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e){}
        }
    }
    ...
}
```

Controlando a Execução

- ◆ **start()**
 - ◆ Inicia a execução do thread (só pode ser invocado uma vez).
- ◆ **yield()**
 - ◆ Faz com que a execução do thread corrente seja imediatamente suspensa, e outro thread seja escalonado.
- ◆ **sleep(t)**
 - ◆ Faz com que o thread fique suspenso por t segundos.
- ◆ **wait()**
 - ◆ Faz com que o thread fique suspenso até que seja explicitamente reativado por um outro thread.

Ciclo de Vida de um Thread



Estado Non-Runnable

- ◆ Um thread pode entrar no estado non-runnable como resultado de:
 - ◆ a execução pelo thread de uma chamada de I/O bloqueante.
 - ◆ a invocação explícita dos métodos `sleep()` ou `wait()` do objeto `Thread`.
- ◆ O método `yield()` não torna o thread non-runnable, apenas transfere a execução para outro thread.

Escalonamento

- ◆ Mecanismo que determina como os threads irão utilizar tempo de CPU.
- ◆ Somente os threads no estado runnable são escalonados para ser executados.
- ◆ Java permite a atribuição de prioridades para os threads.
- ◆ Threads com menor prioridade são escalonados com menor frequência.
- ◆ Os threads são escalonados de forma preemptiva seguindo uma disciplina “round robin”.

Prioridades

- ◆ Todo thread possui uma prioridade que:
 - ◆ pode ser alterada com `setPriority(int p)`
 - ◆ pode ser lida com `getPriority()`
- ◆ Algumas constantes incluem:
 - ◆ `Thread.MIN_PRIORITY`
 - ◆ `Thread.MAX_PRIORITY`
 - ◆ `Thread.NORM_PRIORITY`
 - ◆ o padrão é `Thread.NORM_PRIORITY`

Sincronização

- ◆ Cada thread possui uma pilha independente → dois threads que executam o mesmo método possuem versões diferentes das variáveis locais.
- ◆ A memória dinâmica (heap) de um programa é compartilhada por todos os threads → dois threads poderão portanto acessar os mesmos atributos de uma objeto de forma concorrente.
- ◆ Devido ao mecanismo de preempção, não há como controlar o acesso a recursos compartilhados sem um mecanismo específico de sincronização.

Exemplo

```
public class ContaCorrente {
    float saldo = 0;
    float tmp;

    public float getSaldo(){
        return saldo;
    }
    public void depositar(float valor){
        tmp = getSaldo();
        saldo = tmp+valor;
    }
    public void sacar(float valor){
        tmp = getSaldo();
        saldo = tmp-valor;
    }
}
```


Exemplo

- ◆ Considere dois threads acessando uma mesma conta simultaneamente, um invocando o método **depositar** e outro o método **sacar**.

Thread 1	Thread 2	saldo
---	---	0
<code>conta.depositar(100);</code>	---	0
<code>tmp = saldo;</code>	---	0
---	<code>conta.sacar(50);</code>	0
---	<code>tmp = conta;</code>	0
---	<code>saldo = tmp - 50;</code>	-50
<code>saldo = tmp + 100;</code>	---	100

Sincronização

- ◆ Java permite restringir o acesso a métodos de um objeto ou trechos de código através do modificador **synchronized**.
- ◆ Cada objeto Java possui um (e apenas um) lock.
- ◆ Para invocar um método **synchronized** é necessário adquirir (implicitamente) o lock associado ao objeto.
- ◆ Se dois ou mais métodos de um objeto forem declarados como **synchronized**, apenas um poderá ser acessado de cada vez.

Exemplo

```
public class ContaCorrente {  
    float saldo = 0;  
    float tmp;  
  
    public float getSaldo(){  
        return saldo;  
    }  
    public synchronized void  
    depositar(float valor){  
        tmp = Linguagens de Programação I getSaldo();  
    }  
}
```

19

wait() e notifyAll()

- ◆ O mecanismo de sincronização de Java é baseado em monitores, e utiliza uma variável condicional que é o próprio lock do objeto.
- ◆ As operações `wait(cond)` e `signal(cond)` usada em monitores são representadas em Java pelos métodos `wait()` e `notifyAll()`.
- ◆ `wait()` e `notifyAll()` só podem ser usados dentro de métodos ou trechos de código **synchronized**, e não recebem parâmetros, já que atuam sobre o lock do objeto.

wait() e notifyAll()

- ◆ Threads podem ser suspensas ao executarem o comando wait().
- ◆ Ao ser suspenso dentro de um método **synchronized**, o thread libera imediatamente o lock do objeto correspondente.
- ◆ O thread suspenso só será reativado quando um outro thread executar o comando notifyAll().
- ◆ Vários threads podem estar suspensos a espera do lock de um objeto.

Exemplo

```
class BlockingQueue extends Queue {
    public synchronized Object remove() {
        while (isEmpty()) {
            wait();    // this.wait()
        }
        return super.remove();
    }

    public synchronized void add(Object o) {
        super.add(o);
        notifyAll(); // this.notifyAll()
    }
}
```