# Java Básico

*Programação Concorrente*

*Prof. Fernando Vanini*

*IC - UNICAMP*

# Conceitos Básicos

- Processo
  - do ponto de vista do Sistema Operacional, um *processo* é a ativação de um *programa*
  - Um mesmo programa pode dar origem a vários processos. Um exemplo: várias ativações do navegador - esses processos são executados *em paralelo*.
  - Se a máquina dispõe de mais de um *processador*, os processos podem ser executados por processadores *diferentes*.

# Conceitos Básicos

- Processos paralelos e processos concorrentes:
  - Dois processos são ditos paralelos se executam de forma *independente* (não compartilham nenhum *recurso* como um dado em comum, arquivo, interface).
  - Dois processos são ditos concorrentes se estes *compartilham* algum tipo de recurso.

# Conceitos Básicos

- Concorrência e paralelismo numa aplicação:
  - Em muitas situações, linhas de execução paralelas ou concorrentes constituem a forma mais adequada de representar o comportamento da aplicação

Um exemplo: um sistema de controle industrial onde cada máquina tem um padrão de comportamento independente das demais.

# Concorrência em Java

- Motivação:
  - tratamento de eventos assíncronos como aqueles gerados por hardware específico (ex. interfaces de rede).
  - desempenho: em máquinas com mais de um processador, a máquina virtual Java pode alocar diferentes linhas de execução em diferentes processadores.
  - Facilidade de programação: em alguns casos, é mais fácil modelar uma aplicação como um conjunto de processos.

# Concorrência em Java

- A classe Thread:
  - define uma ‘linha de execução’.
  - o método **`start()`** dispara a execução de um objeto Thread.
  - o método **`run()`** define o comportamento de um objeto Thread durante o seu ‘tempo de vida’.
  - pode ser estendida, de forma a atender às necessidades da aplicação:
    - apenas o método **`run()`** precisa ser redefinido.

# Um exemplo (1)

```
class MyThread extends Thread{
    String name;  // um nome para este objeto
    int times;    // indica quantas vezes o texto será escrito em run().


    /** Construtor */
    public MyThread(String n, int k){
        name = n;
        times = k;
    }

    @Override
    public void run() {
        for(int i = 0;  i < times; i++){
            System.out.println(" i:"+i+"=========>   SimpleThread:"+name);
        }
    System.out.println("------------>   Encerrando "+name+".");
    }
}
```

# Um exemplo (2)

Neste exemplo são criadas 3 linhas de execução (*threads*):

- 2 *threads* correspondentes à ativação dos objetos **`t1`** e **`t2`**
- 1 *thread* correspondente à ativação do método **`main()`**, pela JVM.

```
public class ThreadEx1 {

    /**
     * @param args
     * @throws InterruptedException
     */
    public static void main(String[] args) {
        Thread t1 = new MyThread("thread1",100);        t1.start();
        Thread t2 = new MyThread("       thread2",100); t2.start();
        System.out.println("encerrando o método main().");
    }

}
```

# Concorrência em Java

- A interface Runnable:
  - define o método **`run()`** que estabelece o comportamento de um objeto Thread durante o seu ‘tempo de vida’.
  - a partir de um objeto Runnable é possível criar uma thread.
  - o uso da interface Runnable, possibilita a criação de uma thread a partir de uma classe que não é derivada de Thread, dando um grau adicional de liberdade ao projetista.

# Runnable: um exemplo (1)

```
class MyRunnable implements Runnable{
    String name;  // um nome para este objeto
    int times;    // indica quantas vezes o texto será escrito em run().


    /** Construtor */
    public MyThread(String n, int k){
        name = n;
        times = k;
    }

    @Override
    public void run() {
        for(int i = 0;  i < times; i++){
            System.out.println(" i:"+i+"=========>   SimpleThread:"+name);
        }
    System.out.println("------------>   Encerrando "+name+".");
    }
}
```

# Runnable: um exemplo (2)

São criadas 3 linhas de execução (*threads*):

- 2 *threads* correspondentes à ativação dos objetos **`r1`** e **`r2`**
- 1 *thread* correspondente à ativação do método **`main()`**, pela JVM.

```
public class RunnableEx1 {

    /**
     * @param args
     * @throws InterruptedException
     */
    public static void main(String[] args) {
        Thread r1 = new Thread(new MyRunnable("runnable1",100));
        r1.start();
        Thread r2 =
          new Thread(new MyRunnable("thread2",100));r2.start();
        System.out.println("encerrando o método main().");
    }
```

# Threads e a memória

- No modelo de threads da JVM:
  - cada *thread* tem a sua própria área de pilha: variáveis locais aos métodos criados pela thread só são accessíveis pela própria thread.
  - a área de *heap*, onde são alocados os objetos (criados através de *new*) é compartilhada por todas as *threads*.
  - Uma ou mais *threads* podem acessar um mesmo objeto (alocado no *heap*) a partir de uma referência local (alocada na pilha).

# Race conditions (1)

- Quanto mais de uma thread compartilha um recurso (como p. ex. um objeto), os acessos ao mesmo podem levar a situações de inconsistência.
- Um exemplo: suponha um objeto ContaCorrente compartilhado por de várias threads e operado através das seguintes operações:

```
public class ContaCorrente {
    private float saldo;

    public float getSaldo(){ return saldo; }

    public void credito(float valor) { saldo += valor; }

    public void debito(float valor)  { saldo -= valor; }
}
```

# Race conditions (2)

```
public class OperadorConta extends Thread{
   private ContaCorrente conta;
   ...
   public boolean lancaDebito(ContaCorrente cc, float valor){
       float saldo = conta.getSaldo();
       if(saldo < valor) return false;
       conta.debito(valor);
       return true;
   }
   ...
   public OperadorConta(ContaCorrente cc){ conta = cc; }
}
```

- Suponha 2 threads t1 e t2, criadas a partir da classe acima e compartilhando uma mesma conta corrente.
  - Mesmo que t1 e t2 usem apenas o método lancaDebito(), não se pode garantir que o saldo da conta (compartilhada) nunca será negativo.

# Synchronized

- O modificador synchronized, quando aplicado a um método, garante que a execução do mesmo será feita de forma ‘atômica’: nenhuma outra thread acessa esse método enquanto o mesmo não terminar sua execução.
- Quando aplicado a um objeto, o efeito é semelhante.

```
public class OperadorConta {
    ...
    public synchronized boolean lancaDebito(
                                              ContaCorrente cc,
                                              float valor
                                            ){
        float saldo = conta.getSaldo();
        if(saldo < valor) return false;
        conta.debito(valor);
        return true;
    }
    ...
}
```

# Join

- Uma thread pode ‘esperar’ pelo término de outra thread através do método **`join()`**.
- No exemplo abaixo:
  - a thread ‘main’ cria as threads t1 e t2
  - espera pela conclusão das mesmas para continuar sua execução.

```
public static void main(String[] args) {
    Thread t1 = new MyThread("thread1",100);        t1.start();
    Thread t2 = new MyThread("       thread2",100); t2.start();
    t1.join(); // espera pela conclusão de t1
    t2.join(); // espera pela conclusão de t2
    System.out.println("encerrando o método main().");
}
```