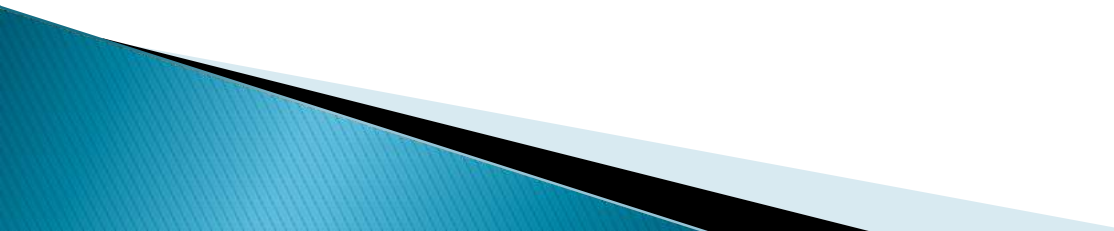


Linguagem de Programação Orientada a Objetos 2

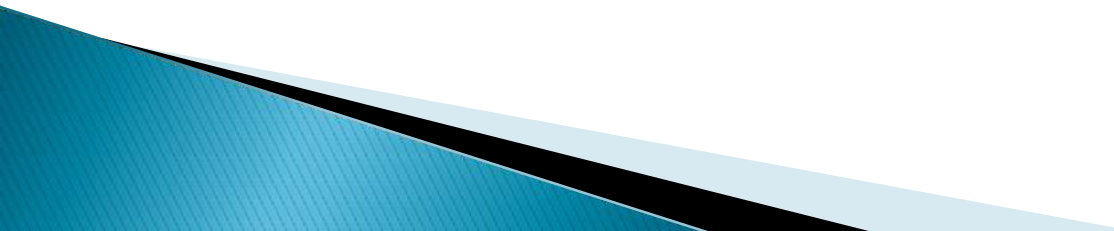
Threads – Introdução
Prof. Tales Viegas

<https://facebook.com/ProfessorTalesViegas>

Introdução

- ▶ Em várias situações necessitamos “rodar duas coisas ao mesmo tempo”
 - ▶ Imaginem vocês programando Java sem um navegador aberto
 - ▶ Ou prestar atenção no professor sem estar com o computador ligado fazendo outra coisa!
 - ▶ Em programação, também temos a necessidade de fazer coisas em paralelo. Para isto, em Java, utilizamos o conceito de Threads
- 

Introdução

- ▶ Um programa multithreaded contém duas ou mais partes que podem rodar simultaneamente.
 - ▶ Cada uma dessas partes é chamada de *thread*, e cada thread define um caminho de execução.
 - ▶ Assim, multithreading é uma forma especializada de multitarefa.
- 

Introdução

- ▶ A multithreading permite escrever programas muito eficientes, que utilizam ao máximo a capacidade da CPU, porque o tempo ocioso é minimizado.

Introdução

▶ Threads x Processos

- Processos: tarefas em espaços de endereços diferentes. Se comunicam usando pipes oferecidos pelo SO
- Threads: tarefas dentro do espaço de endereços da aplicação.

Importante

- ▶ O comportamento de um código que utiliza threads não é garantido:
 - JVMs diferentes podem executar threads de maneiras diferentes
 - Não há como garantir que os segmentos serão executados na ordem em que foram iniciados
 - Só o que podemos garantir é que *“Cada segmento será iniciado e executado até a sua conclusão”*

Exemplos de Threads

- ▶ Garbage Collector
- ▶ Tratamento de eventos (botões, caixas, de texto, etc)
- ▶ Todo programa em Java é executado em uma Thread

```
public class Exemplo1 {  
    public static void main(String[ ] args) {  
        System.out.println("Esta é a thread" +  
                            Thread.currentThread().getName()  
    }  
}
```

Criando Threads

- ▶ Opção 1: estender a classe `java.lang.Thread`
 - Construtores :
 - `Thread()`
 - `Thread(String name)`
 - Métodos básicos :
 - `run()`: Ponto de entrada para início da execução de uma thread
 - `start()`: Dá partida numa thread, chamando seu método `run()`
 - `getName()`: Obtém o nome de uma thread

Exemplo 2

```
public class Exemplo2 extends Thread {  
  
    Exemplo2(String nomeThread) {  
        super(nomeThread);  
    }  
  
    public void run() {  
        for (int i = 0; i < 6; i++) {  
            System.out.println(this.getName() + ": " + i);  
        }  
    }  
}
```

```
public class Exemplo2Main{  
    public static void main(String[] args) {  
        Exemplo2 ex2 = new Exemplo2("Thread Ex2");  
        ex2.start();  
        System.out.println("Thread em execucao = "  
+ Thread.currentThread().getName());  
    }  
}
```

Criando Threads

- ▶ Opção 2 : implementar a interface `java.lang.Runnable`
 - Abordagem mais indicada
 - Permite que outras classes sejam estendidas
 - Construtores :
 - `Thread(Runnable target)`
 - `Thread(Runnable target, String name)`

Exemplo 3

```
public class Exemplo3 implements Runnable{
```

```
    public void run() {
```

```
        for (int i = 0; i < 6; i++) {
```

```
            System.out.println(Thread.currentThread().getName() + " : "
```

```
+ i);
```

```
        }
```

```
    }
```

```
}
```

```
public class Exemplo3Main {
```

```
    public static void main(String[] args) {
```

```
        Exemplo3 ex3 = new Exemplo3();
```

```
        Thread t = new Thread(ex3, "Thread Ex3");
```

```
        t.start();
```

```
        System.out.println("Thread em execucao = "
```

```
+
```

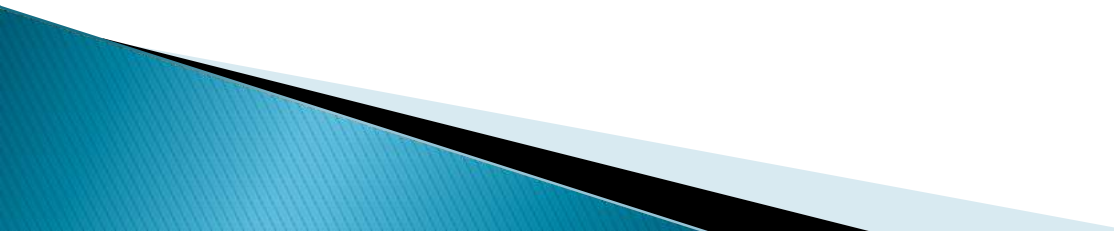
```
Thread.currentThread().getName());
```

```
    }
```

```
}
```

Múltiplas Linhas de Execução

```
public class Exemplo4 implements Runnable {  
    public void run() {  
        for (int i=1; i<4; i++){  
            System.out.println("Thread " +  
                Thread.currentThread().getName());  
        }  
    }  
}
```



Múltiplas Linhas de Execução

```
public class Exemplo4Main{  
    public static void main(String args[]){  
        Exemplo4 ex4 = new Exemplo4();  
        Thread primeira = new Thread(ex4);  
        primeira.setName("Primeira");  
  
        Thread segunda = new Thread(ex4);  
        segunda.setName("Segunda");  
  
        Thread terceira = new Thread(ex4);  
        terceira.setName("Terceira");  
  
        primeira.start();  
        segunda.start();  
        terceira.start();  
    }  
}
```

Realmente é Importante

- ▶ Não há como garantir que os segmentos serão executados na ordem em que foram iniciados
- ▶ Só o que podemos garantir é que *“Cada segmento será iniciado e executado até a sua conclusão”*

Pergunta 1

1) Dado o código a seguir :

```
1. class Pergunta1 extends Thread {  
2.  
3.     public static void main(String [] args) {  
4.         Pergunta1 t = new Pergunta1();  
5.         t.run();  
6.     }  
7.  
8.     public void run() {  
9.         for(int i=1;i<3;++i) {  
10.            System.out.print(i + "..");  
11.        }  
12.    }  
13. }
```

Qual será o resultado ?

- A. Não irá compilar devido a linha 4.
- B. Não irá compilar devido a linha 5.
- C. 1..2..
- D. 1..2..3..

Pergunta 2

2) Dado o código a seguir,

```
1. public class Pergunta2 implements Runnable {  
2.     public void run() {  
3.         // some code here  
4.     }  
5. }
```

Qual dessas opções criará e iniciará essa Thread ?

- A. `new Runnable(Pergunta2).start();`
- B. `new Thread(Pergunta2).run();`
- C. `new Thread(new Pergunta2()).start();`
- D. `new Pergunta2().start();`

Pergunta 3

3) Dado o código a seguir ,

```
1. class Pergunta3 extends Thread {  
2.  
3.     public static void main(String [] args) {  
4.         Pergunta3 t = new Pergunta3();  
5.         t.start();  
6.         System.out.print("one. ");  
7.         t.start();  
8.         System.out.print("two. ");  
9.     }  
10.  
11.     public void run() {  
12.         System.out.print("Thread ");  
13.     }  
14. }
```

Qual será o resultado de sua execução?

A. A compilação falhará

B. Será gerada uma exceção em tempo de execução.

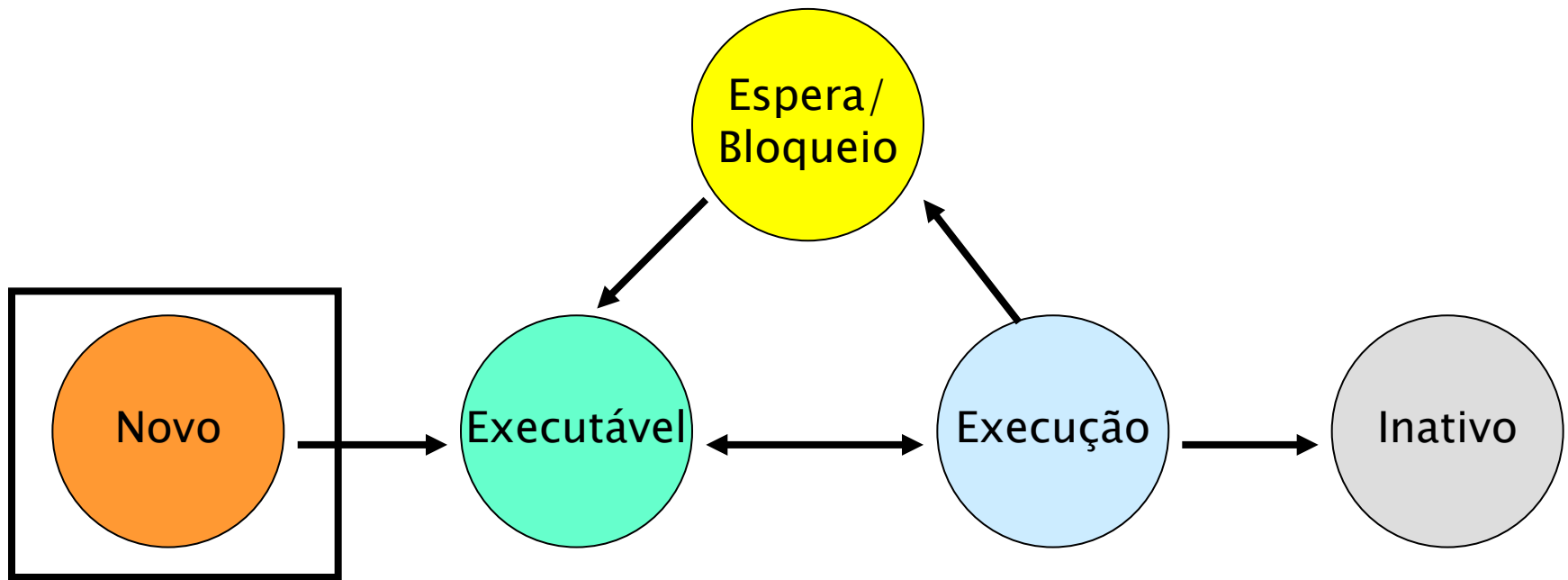
C. Thread one. Thread two.

D. A saída não pode ser determinada

Thread Scheduler

- ▶ Decide qual segmento deve ser executado
- ▶ A ordem na qual os segmentos são selecionados não é garantido.
- ▶ Decide através do estado em que a Thread se encontra e no tempo de espera

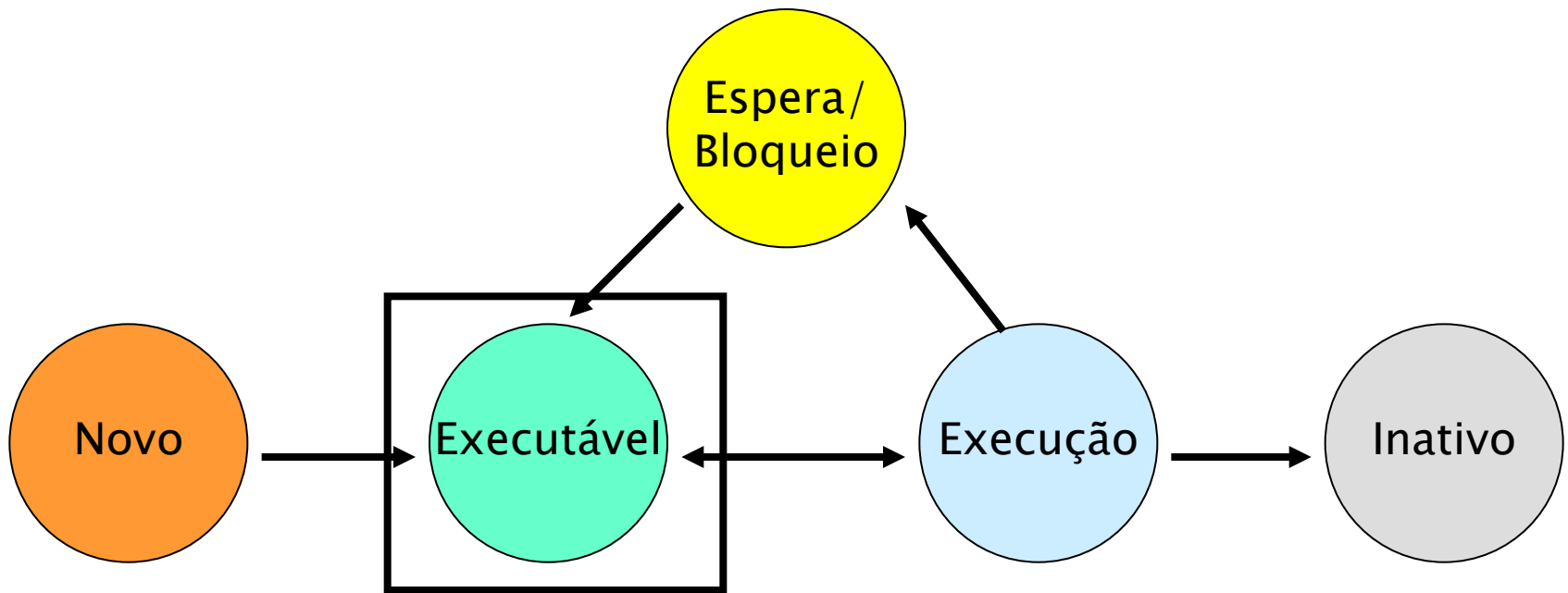
Estados de uma Thread



Estados de uma thread

- ▶ Novo
 - É o estado em que o segmento se encontra logo depois de ter sido criada uma instância da classe Thread.
 - É um segmento inativo.

Estados de uma Thread

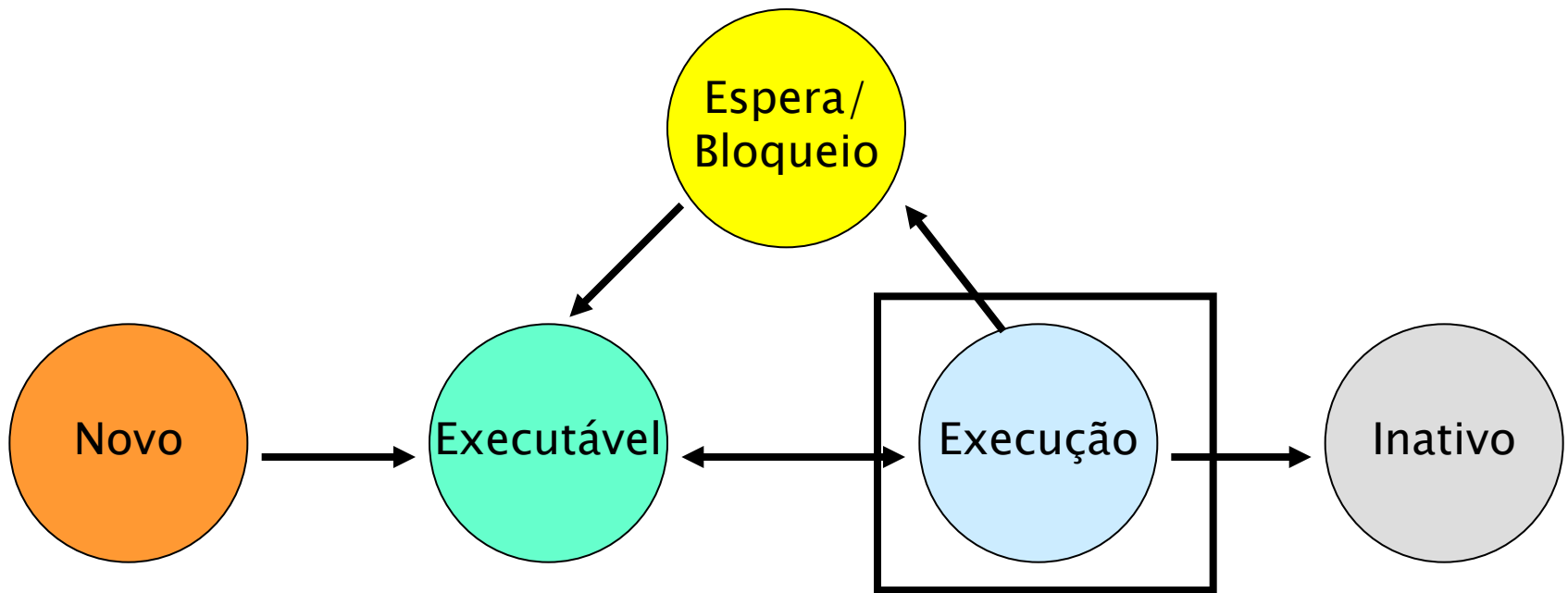


Estados de uma thread

▶ Executável:

- O segmento torna-se executável após a execução do método `start()`
- É um segmento ativo, mas o scheduler ainda não o selecionou para ser processado.
- Também torna-se executável após voltar de um estado de espera

Estados de uma Thread

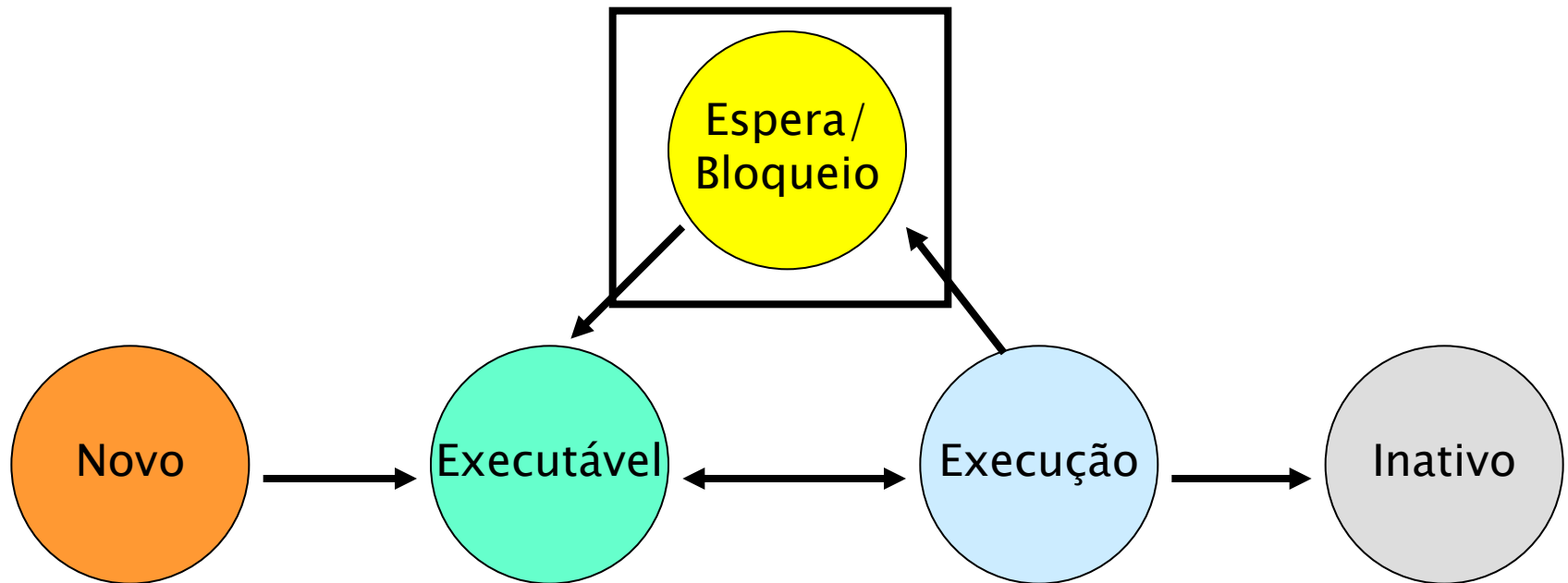


Estados de uma thread

▶ Execução:

- Esse é o estado em que o segmento se encontra quando o scheduler o seleciona para ser executado imediatamente
- É um segmento ativo

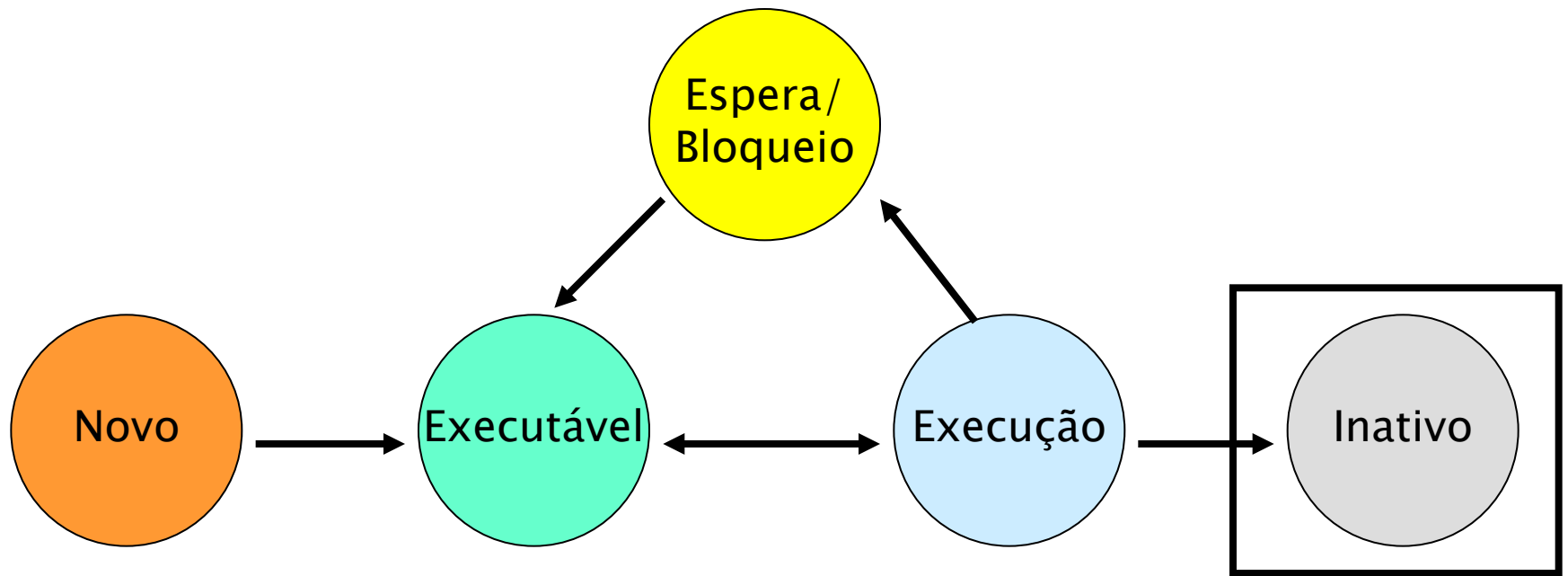
Estados de uma Thread



Estados de uma thread

- ▶ Espera/bloqueio/suspensão:
 - O segmento está ativo, mas não é executável.
 - Um segmento pode ficar bloqueado esperando por algum recurso (ex. E/S)
 - Um segmento pode ficar suspenso porque o código o informou para ficar inativo por algum tempo
 - Um segmento pode ficar na espera porque o código de execução a provocou.
 - O método `isAlive()` pode ser usado para determinar se o segmento ainda está ativo

Estados de uma Thread



Estados de uma thread

▶ Inativo:

- O segmento passa para este estado quando o seu método `run()` for concluído.
- É um segmento inativo e nunca poderá ser ativado novamente.
- O objeto Thread pode continuar sendo utilizado, mas se o método `start()` for executado, será gerada uma exceção em tempo de execução.