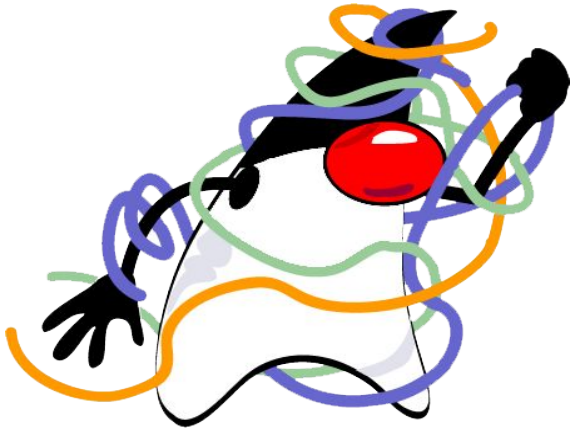
Universidade Federal do Amazonas
Instituto de Computação
Projeto de Programas
Técnicas Avançadas de Programação



Threads

Horácio Fernandes
horacio@icomp.ufam.edu.br

ColabWeb: bit.ly/pp-colabweb

Introdução

- Diversas coisas acontecem **simultaneamente**:
  - Corpo Humano:
    - *Respiração, circulação, digestão*
    - *Sentidos: visão, olfato, tato, paladar e audição*
  - Automóvel:
    - *Acelerar, manobrar, ar condicionado, som*
  - Computadores também:
    - *Compilação, leitura de e-mail, impressão, etc*

Introdução

- Ironicamente, dentro de um **mesmo processo** (programa), há um **fluxo sequencial**.

- Já imaginou se tivéssemos que:
  - Esperar um arquivo terminar de fazer download para continuar navegando na Internet?
  - Esperar um vídeo inteiro ser copiado para então começarmos a vê-lo?

Threads

- Porém, um mesmo programa também pode fazer **diversas coisas ao mesmo tempo**. Como?
  - Através das ***Threads*** – Linhas de Execução
- O suporte a *threads* permite que **múltiplos processamentos** ocorram literalmente **em paralelo**
  - Entretanto, em computadores com apenas um único processador/núcleo, as *threads* não são executadas em paralelo, mas sim concorrentemente dando a impressão de paralelismo. Mas um processador com um único núcleo é menos comum hoje em dia.

Threads em Java

- Apesar de até o momento termos feito apenas programas que executam um comando por vez, sequencialmente
  - Ou seja, usando apenas uma *thread*
- Java tem suporte à execução de **várias *threads*** ao mesmo tempo
  - ***Multithreading***
- Algumas linguagens, como Python, não permitem a execução de *threads* em paralelo, mas possuem mecanismos para execução de vários processos (programas)
  - *Multiprocessing*

Multithreading × Multiprocessing

- *Multithreading*:
  - O **mesmo programa** possui partes que executam em **paralelo** (threads)
  - As *threads* podem **compartilhar memória** (variáveis)

- *Multiprocessing*:
  - O programa é dividido em dois ou mais programas
  - Cada parte executa independente das outras (processos diferentes)
  - Não há compartilhamento de memória
  - Processos podem trocar informações através da troca de mensagens

- Como Java possui suporte completo a *threads*, iremos focar nisso nos slides seguintes.

Uso das Threads

- *Threads* podem ser usadas principalmente por dois motivos
  - Performance
    - *Permite usar mais de um núcleo (core) do processador ao mesmo tempo*
    - *Uma thread pode ficar executando enquanto outra está fazendo entrada/saída (e.g., escrevendo em um arquivo)*
  - Executar tarefas demoradas sem bloquear o programa
    - *Ex: fazer download de um arquivo enquanto você continua navegando*
    - *Ex: fazer uma requisição web sem travar a tela do Android*

Threads em Java

- Em Java, *threads* são representadas pela **classe `Thread`**
  - A única forma de criar uma *thread* em Java é criando um objeto dessa classe
  - Cada *thread* diferente será um objeto dessa classe
  - A *thread* irá começar a executar em paralelo quando o método `start` do seu objeto for executado

- A classe `Thread` implementa a interface `Runnable`
  - Que possui o método abstrato `public void run()`
  - O método `run` precisa ser implementado e este será executado em paralelo com as outras partes do programa quando a *thread* for iniciada (ou seja, quando o método `start` for chamado)

Threads em Java

- A forma mais simples de se **criar uma *thread* em Java** é:
  - Criando uma nova classe que herda a classe `Thread`
  - Implementar o método `run`
  - Criar um objeto da nova classe
  - Executar o método `start` para iniciar a *thread*

Exemplo I

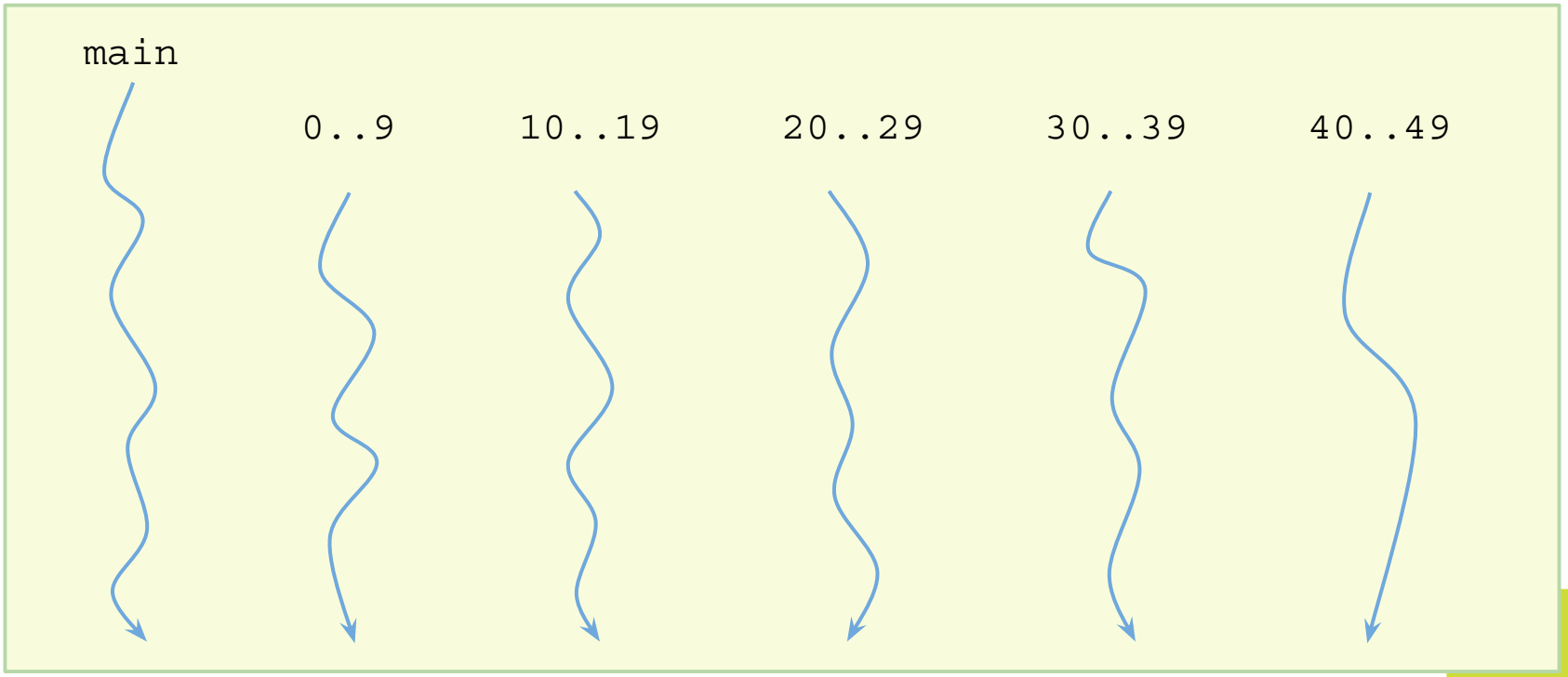
```
class PrintThread extends Thread {
  int from, to;

  public PrintThread(int from, int to) {
    this.from = from; this.to = to;
  }

  public void run() {
    System.out.println(Thread.currentThread());
    for (int i=from; i<to; i++) System.out.println("i=" + i);
  }

  public static void main(String[] args) {
    int i;
    for (i=0; i<5; i++) {
      PrintThread pi = new PrintThread(i*10, (i+1)*10);
      pi.start();
    }
  }

}
```

Para criar uma *thread*, estende-se a classe `Thread`

O método `run` será executado em paralelo quando a *thread* for iniciada

Para iniciar uma *thread*, instancie um objeto da classe e, em seguida, execute o método `start`

Exemplo I - Execução



Exemplo I - Execução

1a Execução

```
Thread[Thread-0,5,main]
Thread[Thread-1,5,main]
i == 0
i == 1
i == 2
i == 3
i == 10
Thread[Thread-2,5,main]
i == 20
i == 21
i == 22
i == 23
Thread[Thread-4,5,main]
i == 4
i == 5
i == 6
i == 7
i == 8
i == 9
i == 40
i == 24
i == 25 (...)
```

2a Execução

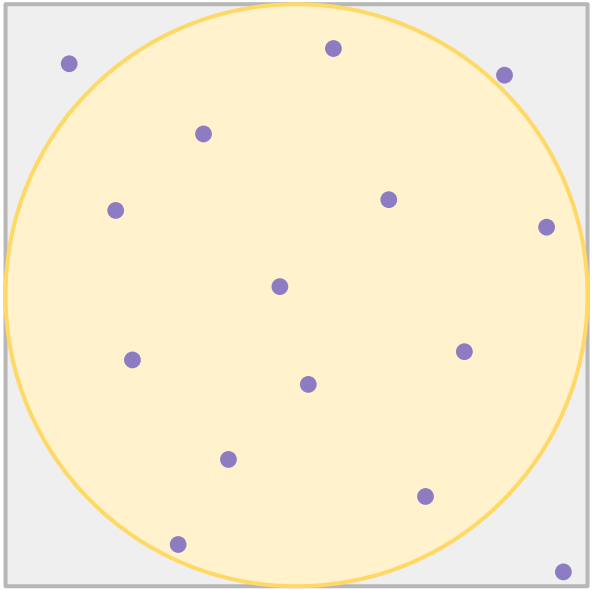
```
Thread[Thread-1,5,main]
Thread[Thread-4,5,main]
Thread[Thread-3,5,main]
i == 30
i == 31
i == 32
i == 33
i == 34
i == 35
i == 36
i == 37
i == 38
i == 39
Thread[Thread-2,5,main]
i == 20
i == 21
i == 22
i == 23
i == 24
i == 25
i == 26
i == 27 (...)
```

3a Execução

```
Thread[Thread-0,5,main]
Thread[Thread-1,5,main]
i == 0
i == 1
i == 2
i == 3
i == 4
i == 5
i == 6
i == 7
i == 8
i == 9
i == 10
i == 11
i == 12
Thread[Thread-3,5,main]
i == 30
i == 31
i == 32
i == 33
i == 34
i == 35 (...)
```

Exemplo II - Aproximação de Pi

- Neste exemplo, iremos usar *threads* para **aumentar a performance** do cálculo de uma **aproximação de Pi** usando o método de Monte Carlo
- Considere a figura ao lado que tem:
  - Círculo de raio 0.5
  - Quadrado de lado 1
- Se gerarmos `N` pontos aleatórios dentro do quadrado, e contarmos quantos (`C`) caíram dentro do círculo
  - `Pi = 4 * C / N`



https://computing.llnl.gov/tutorials/parallel_comp/#ExamplesPI

Exemplo II - Aproximação de Pi

- Quanto maior a quantidade de pontos, melhor será a aproximação de Pi
  - Entretanto, maior será o tempo de execução

- Este cálculo pode ser paralelizado:
  - Cada *thread* ficará responsável por gerar uma quantidade de pontos aleatórios e contar quantos caíram dentro do círculo
  - A *thread* inicial (`main`) irá aguardar as *threads* terminarem e agrupar os resultados individuais em um único resultado

Qtde Pontos (N)	Erro Médio
100	0.13104
1000	0.04616
10000	0.01250
100000	0.00483
1000000	0.00158
10000000	0.00033
100000000	0.00015
1000000000	0.00003
10000000000	0.00001

Exemplo II - Aproximação de Pi

```
import java.util.Random;

class PiThread extends Thread {
  int nPoints;
  int circleCount;
  Random rng = new Random();

  public PiThread(int nPoints) {
    this.nPoints = nPoints;
    this.start();
  }

  public void run() {
    for (int i = 0; i <= nPoints; i++) {
      double xCoord = rng.nextDouble();
      double yCoord = rng.nextDouble();

      if (Math.sqrt(Math.pow(0.5-xCoord, 2) +
          Math.pow(0.5-yCoord, 2)) <= 0.5)
            circleCount++;
    }

  }

  // MAIN no próximo slide

}
```

Quantidade de pontos aleatórios

Quantos caíram dentro do círculo

Geração de números aleatórios

Inicia a execução da *thread* quando uma instância (objeto) for criada

Executado `N` vezes

Gera um ponto (x, y) aleatório

O ponto aleatório gerado está dentro do círculo? Se sim, incrementa `C`

Exemplo II - Aproximação de Pi

```
public static void main(String[] args) {
  int numPoints = 1000000000;
  int numThreads = 8;

  PiThread threads[] = new PiThread[numThreads];
  for (int i=0; i<numThreads; i++)
    threads[i] = new PiThread(numPoints/numThreads);

  for (int i=0; i<numThreads; i++)
    try { threads[i].join(); }
    catch (InterruptedException e) {}

  int totalCount = 0;
  for (int i=0; i<numThreads; i++)
    totalCount += threads[i].circleCount;

  System.out.println(4.0 * totalCount/numPoints);
}
```

Quantidade de pontos aleatórios

Quantidade de *threads*

Cria e inicia a execução das *threads*

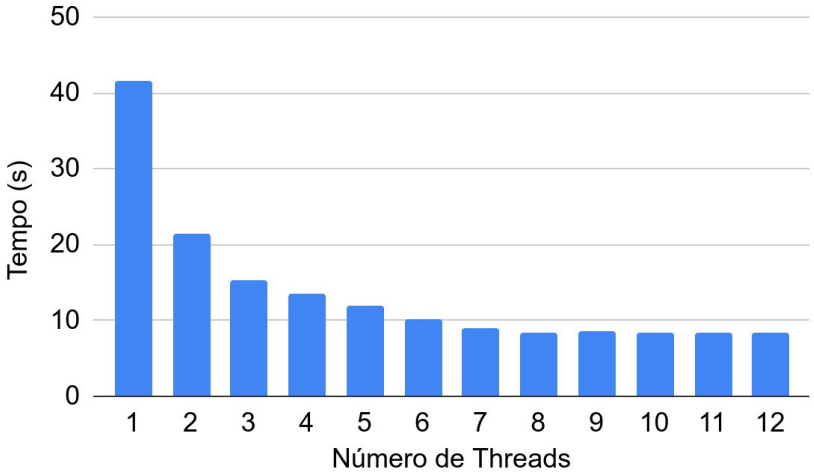
Espera até que todas as *threads* terminem de executar

Agrupa todos os contadores das *threads* em um contador total

Calcula Pi

Exemplo II - Aproximação de Pi

- Avaliação de performance ao variar a quantidade de *threads* de 1 a 12
  - Note como o tempo caiu quase pela metade de 1 para 2 *threads*
  - Com 8 *threads*, o tempo caiu de 41.5s para 8.4s, quase 5x mais rápido
  - Após 8 *threads*, o tempo parou de cair
    - *Motivo: o processador da máquina que executou os experimentos só tinha 4 núcleos com 2 threads cada (8 threads)*



Outra Forma de Criar Threads

- Em alguns casos pode não ser **conveniente herdar** a classe `Thread` (criar uma subclasse) só para criar uma *thread*
  - Principalmente se sua classe já herdar alguma outra classe
  - Pois Java não permite herança múltipla

- Solução:
  - Criar uma classe que **implementa** a interface **`Runnable`**
  - Criar um objeto da classe `Thread`, passando um objeto da sua classe como parâmetro

Interface `Runnable` e a Classe `Thread`

- A interface `Runnable` possui o método `run`:

```
package java.lang;

public interface Runnable {
    public abstract void run();
}
```

Esse é o código completo da interface `Runnable`

- A classe `Thread` também implementa a interface `Runnable`
  - E tem um construtor que aceita um objeto "`Runnable`" como parâmetro

```
public class Thread implements Runnable {
  // Código ...
  public Thread(Runnable target) {
    this(null, target, "Thread-" + nextThreadNum(), 0);
  }
  // Código ...
}
```

Exemplo III

```
class PrintThread implements Runnable {
  int from, to;

  public PrintThread(int from, int to) {
    this.from = from; this.to = to;
  }

  public void run() {
    System.out.println(Thread.currentThread());
    for (int i=from; i<to; i++) System.out.println("i == " + i);
  }

  public static void main(String[] args) {
    int i;
    for (i=0; i<5; i++) {
      PrintThread pir = new PrintThread(i*10, (i+1)*10);
      Thread pit = new Thread(pir);
      pit.start();
    }
  }
}
```

Para criar uma thread, implemente a interface `Runnable`

O método `run` será executado em paralelo quando a *thread* for iniciada

Para criar a thread, instancie um objeto da sua classe e, em seguida, instancie um objeto da classe `Thread` passando o primeiro objeto como parâmetro. Por fim, execute método `start`.

Exemplo IV

- Mais um exemplo de threads
  - Múltipla Impressão
  - Analise o próximo código, e tente deduzir a saída do programa

Exemplo IV

```
public class MultPrint implements Runnable {
  Thread thread;
  String string;
  int contador, tempoSleep;

  public MultPrint(String string, int contador, int tempoSleep) {
    this.string = string;
    this.contador = contador;
    this.tempoSleep = tempoSleep;
    this.thread = new Thread(this);
    this.thread.start();
  }

  public void run () {
    while (contador > 0) {
      System.out.println(string); contador--;
      try {
        Thread.sleep(tempoSleep);
      } catch (Exception e) {}
    }
  }

  public static void main (String args[]) {
    new MultPrint("ping", 5, 300);
    new MultPrint("pong", 5, 500);
  }
}
```

A *thread* será iniciada quando a classe for instanciada

O método `sleep` faz a *thread* atual "dormir" pelo tempo especificado (em *ms*)

Qual será a saída?

Exemplo IV

```
// Main (...)
public static void main (String args[]) {
  new MultPrint("ping", 5, 300);
  new MultPrint("pong", 5, 500);
}
```

ping (0)	pong (0)	ping (300)	pong (500)	ping (600)	ping (900)	pong (1000)	ping (1200)	pong (1500)	pong (2000)

```
$ java MultPrintThread
ping
pong
ping
pong
ping
ping
pong
ping
pong
pong
```

3a Forma de Criar Threads

- Para criar uma `Thread` simples, em precisar criar uma classe nova só para isso, pode-se usar **expressões lambda**:
  - *Problema: não pode acessar variáveis não-constantes (final) declaradas fora da thread*

```
class PrintThread3 {

  public static void main(String[] args) {
    for (int i=0; i<5; i++) {

      new Thread(() -> {
        System.out.println(Thread.currentThread());
        for (int j=0; j<10; j++) System.out.println("j == " + j);
      }).start();

    }
  }

}
```

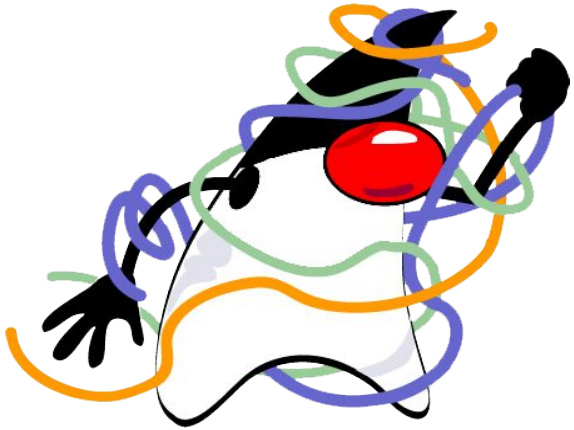
Cria uma nova thread usando expressões lambda

Prioridade

- Cada *thread* apresenta uma **prioridade de execução**
  - Pode ser alterada com `setPriority(int p)`

- Algumas constantes incluem:
  - `Thread.MIN_PRIORITY` (=1)
  - `Thread.MAX_PRIORITY` (=10)
  - `Thread.NORM_PRIORITY` (=5)
    - *O padrão é `Thread.NORM_PRIORITY`*

Universidade Federal do Amazonas
Instituto de Computação
Projeto de Programas
Técnicas Avançadas de Programação



Sincronização de Threads

ColabWeb: bit.ly/pp-colabweb

Horácio Fernandes
horacio@icomp.ufam.edu.br

Sincronização de *Threads*

- *Threads* permitem o **compartilhamento de recursos**
  - Recursos → variáveis, memória, arquivos, etc
- Quando duas ou mais *threads* acessam o mesmo recurso ao mesmo tempo, de forma descontrolada, alguns problemas podem acontecer. Por exemplo:
  - Se duas threads tentarem escrever no mesmo arquivo, isso pode resultar em perda de dados
  - Se duas threads escrevem ao mesmo tempo na mesma variável, é impossível prever qual dos dois valores será realmente armazenado
- Por isso, é necessário haver uma **sincronização** das *threads* para evitar o acesso simultâneo ao mesmo recurso

Sincronização - Exemplo do Problema

```
public class Contador {

  private int cont = 1;

  // Incrementa e retorna o valor anterior (i++)
  public int getAndInc() {
    int n;
    n = cont;      // Salva cont
    cont = n + 1; // Incrementa cont
    return n;      // Retorna o cont anterior
  }
}
```

```
public class MyThread extends Thread {
  Contador c;

  MyThread(Contador c) {
    this.c = c;
  }

  public void run() {
    System.out.println(c.getAndInc());
  }

  public static void main(String args[]) {
    Contador sharedC = new Contador();
    (new MyThread(sharedC)).start();
    (new MyThread(sharedC)).start();
  }
}
```

Um objeto dessa classe será o "recurso compartilhado"

Objeto (recurso) compartilhado

Duas *threads* foram criadas acessando o mesmo objeto (*sharedC*).

Sincronização - Exemplo do Problema

Thread 1	Thread 2	*Cont*
`c.getAndInc()`	`---`	1
`  n = cont;`	`---`	1
`  cont = n + 1;`	`---`	2
`  return n;`	`---`	2
`---`	`c.getAndInc()`	2
`---`	`  n = cont;`	2
`---`	`  cont = n + 1;`	3
`---`	`  return n;`	3

```
$ java MyThread
1
2
```

Experimento:
9319x de 10000 (93.19%)

Sincronização - Exemplo do Problema

Thread 1	Thread 2	*Cont*
`c.getAndInc()`	`---`	1
`  n = cont;`	`---`	1
`  cont = n + 1;`	`---`	2
`---`	`c.getAndInc()`	2
`---`	`  n = cont;`	2
`---`	`  cont = n + 1;`	3
`---`	`  return n;`	3
`  return n;`	`---`	3

```
$ java MyThread
2
1
```

Experimento:
580x de 10000 (5.80%)

Sincronização - Exemplo do Problema

Thread 1	Thread 2	*Cont*
`c.getAndInc()`	`---`	1
`  n = cont;`	`---`	1
`---`	`c.getAndInc()`	1
`---`	`  n = cont;`	1
`---`	`  cont = n + 1;`	2
`---`	`  return n;`	2
`  cont = n + 1;`	`---`	2
`  return n;`	`---`	2

```
$ java MyThread
1
1
```

Isso é um problema grave. Note que o contador não contou direito

Experimento:
101x de 10000 (1.01%)

Condição de Corrida

- Em geral, threads manipulam objetos compartilhados
  - Múltiplos acessos ao mesmo objeto podem gerar comportamentos não determinados caso estes acessos não sejam controlados
  - Na maioria das vezes o programa gerará resultados corretos
  - Entretanto, outras vezes ele produzirá resultados incorretos e nenhum erro será emitido, até que seja tarde

- O **resultado dependerá** da sequência ou **do tempo** de execução das *threads* ao acessar o recurso
  - Este problema é conhecido como **condição de corrida**
  - Do inglês, *race condition*

Exclusão Mútua

- Uma solução para a condição de corrida é a **exclusão mútua**:
  - Dar um **tempo de acesso exclusivo ao recurso** compartilhado, uma *thread* por vez
  - Durante esse tempo, outras *threads* tentando acessar o código terão que ficar esperando o recurso ficar disponível para acesso.
    - *Diz-se que a primeira thread "bloqueou" ou "reservou" o recurso*
    - *E a segunda thread está "bloqueada" ou "esperando"*
  - Neste caso, uma *thread* acessando o código exclui o acesso às outras threads.
    - *Por isso, esta técnica é conhecida como exclusão mútua*
  - O acesso ao recurso fica bloqueado durante toda a execução da **seção crítica**

Seção Crítica

- **Seção Crítica** é um **bloco** (sequência) de código que manipula um recurso compartilhado
- Pode ser composto de uma ou mais linhas que manipulam o recurso
  - Pode ter também outras linhas auxiliares, que não manipulam diretamente o recurso

Implementação de Exclusão Mútua

- Existem algumas formas de se implementar exclusão mútua:
  - Locks (mutexes), semáforos, monitores, dentre outros
  - Não iremos entrar nesses detalhes neste curso
    - *Isso é tratado na disciplina de Sistemas Operacionais*

- Em **Java**, exclusão mútua é implementada usando **Monitores**
  - Todo objeto (instância de classe) tem um monitor dentro dele

- Monitores:
  - Permite que *threads* tenham exclusão mútua a um recurso
  - Possui a habilidade de esperar até que o recurso fique disponível
  - O monitor “sinaliza” para uma *thread* que ela pode voltar a executar

Resumindo

- Em Java, **sincronização de threads** usa **exclusão mútua**, baseada em **monitores**, para controlar a execução de uma **seção crítica** e evitar a **condição de corrida**

Sincronização em Java

- Em Java, existem **duas formas** de se criar uma seção crítica que usa exclusão mútua para sincronização:
  - Métodos `synchronized`
  - Blocos `synchronized`

Métodos `synchronized`

- É possível marcar um método da classe como sendo `synchronized`
- Neste caso, o **método todo será a seção crítica**
- O método só será executado por uma *thread* se não tiver nenhuma outra *thread* executando-o
  - Caso contrário, ela ficará esperando a outra *thread* terminar de executar o método
- Java usa o monitor do objeto para controlar o acesso
  - Ou seja, nenhuma outra *thread* conseguirá bloquear o objeto enquanto o método estiver sendo executado

Métodos `synchronized` - Exemplo

```
public class Contador {

  private static int cont = 1;

  // Retorna cont e depois incrementa (i++)
  public synchronized int getAndInc() {
    int n;
    n = cont;        // Salva cont
    cont = n + 1;    // Incrementa cont
    return n;        // Retorna n
  }
}
```

Bloqueia a execução deste método por uma *thread* enquanto outra *thread* estiver executando ela

```
$ java MyThread
1
2
```

Experimento:
9367x de 10000 (93.67%)

```
$ java MyThread
2
1
```

Experimento:
633x de 10000 (6.33%)

```
$ java MyThread
1
1
```

Experimento:
0x de 10000 (0%)

Blocos `synchronized`

- Uma forma mais precisa de indicar a seção crítica é através dos **blocos `synchronized`**
  - Ao invés de indicarmos um método inteiro como `synchronized`, podemos indicar apenas uma parte dele

- A vantagem é que o **tamanho** da seção crítica é **menor**
  - Isso **aumenta a performance**, pois as threads ficarão menos tempo esperando para entrar na seção crítica

- É necessário indicar o recurso (objeto) a ser bloqueado
  - Veja a sintaxe no próximo slide

Blocos `synchronized` - Exemplo

```
public class Contador {

  private int cont = 1;

  // Retorna cont e depois incrementa (i++)
  public int getAndInc() {
    int n;
    synchronized(this) {
      n = cont; // Salva cont
      cont = n + 1; // Incrementa cont
    }
    return n; // Retorna n
  }
}
```

A entrada neste bloco é sincronizada. Apenas uma *thread* por vez pode entrar aqui. O monitor do objeto atual (`this`) é usado para controlar o acesso.

```
$ java MyThread
1
2
```

Experimento:
9627 de 10000 (96.27%)

```
$ java MyThread
2
1
```

Experimento:
373x de 10000 (3.73%)

```
$ java MyThread
1
1
```

Experimento:
0x de 10000 (0%)

Deadlock

- O uso de exclusão mútua pode gerar alguns problemas
- O mais conhecido deles é o *deadlock*
- No *deadlock*:
  - Uma *thread* `1` bloqueia um *recurso* `A` e, ainda em sua seção crítica, tenta bloquear um segundo *recurso* `B`
  - Entretanto, uma *thread* `2` já havia bloqueado o *recurso* `B` e, dentro de sua seção crítica tenta bloquear o *recurso* `A` (que está bloqueado pela *thread* `1`)
  - Como a *thread* `1` só irá liberar o *recurso* `A` quando puder acessar o *recurso* `B` e como a *thread* `2` só irá liberar o *recurso* `B` quando puder acessar o *recurso* `A`, uma *thread* ficará esperando pela outra e ninguém irá ceder
  - Resultado: **deadlock**
    - *O programa "trava"*

Deadlock - Exemplo

```
public class DoisRecursos extends Thread {
  String rec1, rec2;

  DoisRecursos(String rec1, String rec2) {
    this.rec1 = rec1;
    this.rec2 = rec2;
  }

  public void run() {

    synchronized (rec1) {
      System.out.println(this.getName() + " bloqueia " + rec1);

      try { Thread.sleep(50); } catch (InterruptedException e) {}

      synchronized (rec2) {
        System.out.println(this.getName() + " bloqueia " + rec2);
      }
    }
  }
}
```

A *thread* vai acessar e bloquear o objeto `rec1`

Em seguida, vai tentar acessar e bloquear o objeto `rec2`

Deadlock - Exemplo

```
public class DuasThreads {
  public static void main(String[] args) {
    String recurso1 = "Recurso A";
    String recurso2 = "Recurso B";

    (new DoisRecursos(recurso1, recurso2)).start();
    (new DoisRecursos(recurso2, recurso1)).start();
  }
}
```

Recursos que serão compartilhados

2 threads são criadas e acessarão os recursos

Mas em ordem invertida

```
$ java DuasThreads
Thread-0 bloqueia Recurso A
Thread-1 bloqueia Recurso B
    (o programa "trava")
```

Deadlock - Exemplo

- Porque o programa “trava”? Observe a ordem de execução:
  - *Thread* `1` acessa e bloqueia o *recurso* `A`
  - *Thread* `2` acessa e bloqueia o *recurso* `B`
  - *Thread* `1` tenta acessar o *recurso* `B`. Fica esperando ele ficar livre.
  - *Thread* `2` tenta acessar o *recurso* `A`. Fica esperando ele ficar livre.

Deadlock

- Definição:
  - Quando dois ou mais processos (*threads*) estão cada um esperando pelo outro para liberar um recurso
- São particularmente problemáticos porque não existe uma solução “geral” e são difíceis de prever
  - Entretanto, são fáceis de se detectar
- Só podem ocorrer quando
  - Pelo menos dois processos (*threads*) compartilham
  - Pelo menos dois recursos usando
  - Exclusão mútua e quando
  - Bloqueiam um recurso enquanto tentam acessar outro

Soluções

- *Deadlocks* podem ser evitados de algumas formas:
  - Não usar exclusão mútua
  - Ou não bloquear um recurso e tentar acessar outro
- Ou, pode-se deixar o *deadlock* acontecer e
  - Detectar que ele aconteceu, e
  - Parar/Reiniciar um dos dois processos
  - Exemplo (compra na DealExtreme):