Introdução à Programação Distribuída com Java

Nabor C. Mendonça
Universidade de Fortaleza
nabor@unifor.br

Referências

- Livro-texto:
  - E. R. Harold, *Java Network Programming*, 3rd Edition, O'Reilly & Associates, 2004
- Material suplementar:
  - Tutoriais e apresentações sobre a programação de aplicações distribuídas com Java disponíveis publicamente na Internet

Roteiro

- Por que Java?
- O que faz uma aplicação distribuída?
- Programação distribuída com Java
  - Fundamentos
    - Recursos de E/S
    - Concorrência
    - Endereços de rede
  - Comunicação usando *sockets*
    - Comunicação assíncrona
    - Comunicação síncrona
  - Aplicação piloto

Por que Java?

- Primeira linguagem de programação concebida desde o início visando um mundo distribuído
- Oferece um conjunto de soluções cruciais para o desenvolvimento aplicações distribuídas
  - Independência de plataforma
  - Segurança
  - Suporte para caracteres internacionais

Por que Java?

- Facilita a vida do programador
  - APIs para comunicação entre aplicações abstraem muitos dos detalhes de configuração e implementação necessários em outras linguagens
  - Diminui consideravelmente o volume de código da aplicação devotado para aspectos de comunicação e recursos de rede
  - Maior chance de reuso

O que faz uma aplicação distribuída?

- Obtém dados de fontes remotas
  - Páginas HTML, arquivos de imagens, dados relacionais e semi-estruturados (XML), etc
- Acessa continuamente informações de conteúdo dinâmico
  - Cotação de ações, notícias, monitoramento remoto de sistemas
- Envia dados para fontes remotas
  - Servidores de arquivos, computação massivamente paralela (ex.: projeto SETI@home)

O que faz uma aplicação distribuída?

- Interage com outras aplicações de modo “ponto-a-ponto” (P2P)
  - Jogos, bate-papo, compartilhamento de arquivos
- Busca informações na Web
- Realiza transações de Comércio Eletrônico
- E muito mais!
  - Computação móvel/ubíqua (J2ME)
  - TV interativa (ex.: SBTVD)
  - Trabalho colaborativo

Exercício 1.1

- Descreva outros recursos que Java oferece e que também podem ser úteis na implementação de aplicações distribuídas.
- Descreva o papel da Máquina Virtual Java (JVM) e explique como essa tecnologia contribui para que Java seja independente de plataforma.
- Pesquise na Internet pelo menos três exemplos de *middleware* disponíveis para a linguagem Java.

Fundamentos

- Recursos de E/S
- Concorrência
- Endereços de rede

Recursos de E/S

- Classes *Stream*
- Subclasses de *Stream*
- Classes base para comunicação via rede

Classes *Stream*

- O que é um *stream*?
  - Uma seqüência (“fluxo”) de bytes, geralmente de origem externa ao programa
  - Solução adotada por Java para padronizar a forma pela qual os programas realizam operações de I/O
  - Dois tipos básicos: entrada (*input stream*) e saída (*output stream*)
    - Um mesmo conjunto de métodos utilizado para enviar/receber dados através do *stream* independentemente do mecanismo de I/O subjacente

Streams

- Onde são usados?
  - Console: System.in, System.out, System.err
  - Estruturas para manipulação de dados e comunicação entre processos: array de bytes, *pipes*, etc.
  - Arquivos
  - Conexões de rede (!)

Subclasses de *Stream*

- Pacote java.io:
  - *BufferedInputStream*
  - *BufferedOutputStream*
  - *ByteArrayInputStream*
  - *ByteArrayOutputStream*
  - *DataInputStream*
  - *DataOutputStream*
  - *FileInputStream*
  - *FileOutputStream*
  - *FilterInputStream*
  - *FilterOutputStream*
  - *ObjectInputStream*
  - *ObjectOutputStream*
  - *PipedInputStream*
  - *PipedOutputStream*
  - *PrintStream*
  - *PushbackInputStream*
  - *SequenceInputStream*
  - *LineNumberInputStream*
  - *StringBufferInputStream*

Subclasses de *Stream*

- Pacote java.util.zip:
  - *CheckedInputStream*
  - *CheckedOutputStream*
  - *InflaterInputStream*
  - *DeflaterOutputStream*
  - *GZIPInputStream*
  - *GZIPOutputStream*
  - *ZipInputStream*
  - *ZipOutputStream*

Subclasses de *Stream*

- Pacote java.util.jar:
  - *JarInputStream*
  - *JarOutputStream*
- Pacote java.security:
  - *DigestInputStream*
  - *DigestOutputStream*
- Pacote javax.crypto:
  - *CipherInputStream*
  - *CipherOutputStream*

Subclasses de *Stream*

- Pacote sun.net:
  - *TelnetInputStream*
  - *TelnetOutputStream*
- Além das subclasses disponíveis em pacotes de terceiros ou que você mesmo venha a criar para as suas aplicações!

Streams para comunicação via rede

- *java.io.OutputStream*
- *java.io.InputStream*
- *java.io.Writer*
- *java.io.Reader*

A classe *OutputStream*

- *OutputStream* é uma classe abstrata que envia bytes “crus” de dados para um alvo específico como a console ou um servidor de rede
- Utilizada para especificar o retorno de muitos métodos no JDK
  - Manipulação geralmente feita apenas através dos métodos da classe abstrata
  - O polimorfismo esconde os detalhes do programador

A classe *OutputStream*

- Vários métodos de uso geral:

```
public abstract void write(int b) throws IOException
public void write(byte[] data) throws IOException
public void write(byte[] data, int offset, int length) throws
   IOException
public void flush() throws IOException
public void close() throws IOException
```

- O método write() envia bytes crus de dados para quem quer que esteja lendo dados desse *stream*

Exemplo: enviando dados para um *OutputStream*

```
import java.io.*;

public class HelloOutputStream {

  public static void main(String[] args) {

    String s = "Hello World\r\n";

    // Convert s to a byte array
    byte[] b = new byte[s.length()];
    s.getBytes(0, s.length()-1, b, 0);
    try {
      System.out.write(b);
      System.out.flush();
    }
    catch (IOException e) {
      System.err.println(e);
    }
  }
}
```

Descarregando um *OutputStream*

- Alguns *OutputStreams* são “buferizados” pelo S.O. para aumentar a performance
- O método flush() força os dados a serem enviados esteja o *buffer* cheio ou não
- Esta buferização é diferente daquela realizada pela subclasse *BufferedOutputStream*, que fica sob o controle da JVM
  - Nesse caso, uma chamada para flush() causa o esvaziamento de ambos os *buffers*

Fechando um *OutputStream*

- O método close() fecha o *stream* e libera qualquer recurso a ele associado
  - Qualquer dado ainda no *buffer* do *stream* será perdido!
- Uma vez fechado, qualquer tentativa de enviar dados pelo *stream* gerará uma exceção do tipo IOException

Criando subclasses para *OutputStream*

- Deve-se implementar:

```
public abstract void write(int b) throws IOException
```

- Semântica esperada:
  - *b* é um inteiro entre 0 e 255
  - Se *b* estiver fora dessa faixa, então apenas o byte menos significativo do inteiro deve ser enviado e os três bytes restantes devem ser descartados
- Outros métodos também podem ser sobrescritos

Exemplo: uma subclasse para *OutputStream*

```
package com.macfaq.io;

import java.io.*;

public class NullOutputStream extends OutputStream {
  public void write(int b) {
  }
  public void write(byte[] data) {
  }
  public void write(byte[] data, int offset, int length) {
  }
}
```

Exercício 2.1

- Crie uma subclasse para *OutputStream* chamada *InvertOutputStream* que inverte a ordem dos bytes de toda seqüência de bytes enviada através dos métodos write(byte[] data) e write(byte[] data, int offset, int length).
- O construtor da subclasse deve receber como parâmetro um *OutputStream* de destino, para onde as seqüências de bytes invertidas serão enviadas.
- Teste a sua implementação utilizando como destino a saída padrão da console (System.out).

A classe *InputStream*

- *InputStream* é uma classe abstrata que contém os métodos básicos para ler bytes “crus” de dados de um *stream*
- Tal como a *OutputStream*, também é utilizada para especificar o retorno de muitos métodos no JDK
  - Manipulação geralmente feita apenas através dos métodos disponíveis na classe abstrata

A classe *InputStream*

- Métodos:

```
public abstract int read() throws IOException
public int read(byte[] data) throws IOException
public int read(byte[] data, int offset, int length) throws
   IOException
public long skip(long n) throws IOException
public int available() throws IOException
public void close() throws IOException
public synchronized void mark(int readlimit)
public synchronized void reset() throws IOException
public boolean markSupported()
```


O método read()

```
public abstract int read() throws IOException
```

- Lê um único byte de dados (sem sinal) do *stream*
- Retorna um inteiro de valor entre 0 e 255
- Retorna -1 no final do *stream*
- Pode bloquear

Exemplo: método read()

```
import java.io.*;

public class Echo {
  public static void main(String[] args) {
    try {
      while (true) {
        // Note que embora seja lido um byte, na verdade é retornado um
        // valor inteiro entre 0 e 255. Este então é convertido para
        // um caractere ISO Latin-1 da mesma faixa antes de ser impresso.
        int i = System.in.read();
        // O valor -1 é retornado para indicar final de stream
        if (i == -1) break;
        // Sem a conversão uma string numérica como "65"
        // seria impressa ao invés do caractere "A"
        char c = (char) i;
        System.out.print(c);
      }
    }
    catch (IOException e) {
      System.err.println(e);
    }
    System.out.println();
  }
}
```


Lendo vários bytes de uma vez

- É mais eficiente ler múltiplos bytes de cada vez:

  `public int read(byte[] data) throws IOException`

  `public int read(byte[] data, int offset, int length) throws IOException`

  - O primeiro tenta preencher por completo o array especificado em *data*, enquanto o segundo tenta preencher apenas um subconjunto de até *length* bytes de *data* começando na posição especificada em *offset*
  - Ambos bloqueiam até que haja algum dado disponível
  - Ambos retornam o número de bytes realmente lidos ou -1 no final do *stream*

Contando os bytes disponíveis

- O método available() testa quantos bytes estão disponíveis para leitura no *stream* sem bloquear

```
public class EfficientEcho {
  public static void main(String[] args){
    try {
      while (true) {
        int n = System.in.available();
        if (n > 0) {
          byte[] b = new byte[n];
          int result = System.in.read(b);
          if (result == -1) break;
          String s = new String(b);
          System.out.print(s);
        }
      }
    } catch (IOException e) {
      System.err.println(e);
    }
  }
}
```


Pulando bytes

- O método skip() “pula” um número de bytes especificado sem lê-los

```
public int skip(long n) throws IOException
```

- Mais útil para leitura de arquivos com acesso randômico

Marcando e re-setando

```
public synchronized void mark(int readlimit)
public synchronized void reset() throws IOException
public boolean markSupported()
```

- O método markSupported() retorna verdadeiro se o *stream* suporta o mecanismo de marcação e falso caso contrário
- O método mark() coloca um marcador na posição atual do *stream* para onde pode-se retornar posteriormente usando o método reset()
- Só pode haver um único marcador no *stream* por vez. Um novo marcador apaga o marcador anterior
- Se o *stream* não suportar marcação, reset() poderá retornar IOException (dependendo do tipo do *stream*)

Fechando um *InputStream*

- O método close() fecha o *stream* e libera qualquer recurso a ele associado

  ```
  public void close() throws IOException
  ```
- Uma vez fechado, qualquer tentativa de ler novos dados do *stream* gerará uma exceção do tipo IOException

Criando subclasses para *InputStream*

- Deve-se implementar:
  `public abstract int read() throws IOException`
- Outros métodos também podem ser sobrescritos

Exemplo: uma subclasse para *InputStream*

```
import java.util.*;
import java.io.*;

public class RandomInputStream extends InputStream {
  private transient Random generator = new Random();
  public int read() {
    int result = generator.nextInt() % 256;
    if (result < 0) result = -result;
    return result;
  }
  public int read(byte[] data, int offset, int length) throws IOException {
    byte[] temp = new byte[length];
    generator.nextBytes(temp);
    System.arraycopy(temp, 0, data, offset, length);
    return length;
  }
  public int read(byte[] data) throws IOException {
    generator.nextBytes(data);
    return data.length;
  }
  public long skip(long bytesToSkip) throws IOException {
    return bytesToSkip; // Pulo sem efeito já que tudo é aleatório!
  }
}
```

Exercício 2.2

- Crie uma subclasse para *InputStream* chamada *InvertInputStream* que inverte a ordem dos bytes de toda seqüência de bytes lida através dos métodos read(byte[] data) e read(byte[] data, int offset, int length).
- O construtor da subclasse deve receber como parâmetro um *InputStream* de origem, de onde as seqüências de bytes a serem invertidas serão lidas.
- Teste a sua implementação utilizando como origem a entrada padrão da console (System.in).

Exercício 2.3

- Descubra quais são as subclasses de *InputStream* disponíveis no pacote java.io que suportam o mecanismo de marcação (markSupported() retorna verdadeiro).
- Baseado na resposta do item anterior, discuta até que ponto incluir métodos relativos ao mecanismo de marcação na classe abstrata adere às boas práticas do paradigma de orientação a objeto.

Codificação de caracteres em Java

- Java usa o padrão Unicode
  - Pode ser serializado em uma variedade de formatos (UTF-8, UCS-2, UCS-4, etc)
- A maioria dos arquivos de texto utiliza outros padrões de codificação:
  - ASCII
  - Latin-1
  - MacRoman
  - ...
- É importante atentar para os diferentes padrões de codificação de caracteres quando se lê e escreve texto em Java!

Readers e *Writers*

- As classes *Reader* e *Writer* são superclasses abstratas para classes que lêem e escrevem dados baseados em caracteres
- Sua principal função é realizar a conversão entre diferentes padrões de codificação de caracteres

```
public abstract class Reader extends Object
public abstract class Writer extends Object
```

A classe *Writer*

- A classe *Writer* possui métodos deliberadamente similares aos métodos da classe *OutputStream. A* diferença é que trabalham com caracteres ao invés de bytes

```
public void write(int c) throws IOException
public void write(char[] text) throws IOException
public abstract void write(char[] text, int offset, int
   length) throws IOException
public void write(String s) throws IOException
public void write(String s, int offset, int length) throws
   IOException
```


A classe *Writer*

- O método básico write() escreve um único caractere de dois bytes com um valor entre 0 e 65535. O valor é computado com base nos dois bytes menos significativos do inteiro passado como parâmetro
- Os outros métodos permitem escrever uma seqüência (*array*) de caracteres, uma sub-seqüência de caracteres, uma string, ou uma sub-string.
- Tal como um *OutputStream*, um *Writer* pode ser buferizado. Para forçar a escrita, basta chamar flush():

```
public abstract void flush() throws IOException
```

- Finalmente, o método close() fecha o *Writer* e libera qualquer recurso a ele associado:

```
public abstract void close() throws IOException
```

A classe *OutputStreamWriter*

- A classe *OutputStreamWriter* serve como uma ponte entre um *stream* de caracteres e um *stream* de bytes
- Ela recebe caracteres no padrão Unicode e os envia para um *OutputStream* subjacente convertidos para bytes de acordo com um padrão de codificação especificado
- O padrão de codificação pode ser especificado no construtor, ou ser aceito o padrão *default* da plataforma:
  - ISO Latin-1 para Windows e Solaris
  - MacRomam para Macintosh
- A lista completa dos padrões de codificação suportados em Java 1.5 está disponível no endereço:

  http://java.sun.com/j2se/1.5.0/docs/guide/intl/encoding.doc.html

A classe *OutputStreamWriter*

- Construtores:

```
public OutputStreamWriter(OutputStream out, String enc)
   throws UnsupportedEncodingException
public OutputStreamWriter(OutputStream out)
```

- Os outros métodos apenas sobre-escrevem os métodos da super classe, mantendo o comportamento

```
public void write(int c) throws IOException
public void write(char c[], int offset, int length) throws
   IOException
public void write(String s, int offset, int length) throws
   IOException
public void flush() throws IOException
public void close() throws IOException
```

A classe *OutputStreamWriter*

- Por exemplo, para se escrever um arquivo codificado no padrão Macintosh Symbol deve-se fazer:

```
FileOutputStream fout = new fileOutputStream("symbol.txt");
OutputStreamWriter osw = new OutputStreamWriter(fout,
   "MacSymbol");
```


A classe *Reader*

- Os métodos da classe *Reader* são deliberadamente similares aos métodos da classe *InputStream*
- O método básico read() lê um único caractere (que pode conter entre um e quatro bytes dependendo do padrão de codificação) e retorna o caractere como um inteiro entre 0 e 65535, ou -1 no final do *stream*

```
public int read() throws IOException
```

- Também é possível ler uma seqüência de caracteres de uma vez

```
public int read(char[] text) throws IOException
public abstract int read(char[] text, int offset, int length)
   throws IOException
```

A classe *Reader*

- Todos os métodos de leitura bloqueiam até que haja dados disponíveis, ocorra um erro de *I/O*, ou tenha chegado o final do *stream*
- O método ready() retorna verdadeiro se o Reader estiver pronto para ser lido, do contrário retorna falso

```
public boolean ready() throws IOException
```

- *Readers* também podem suportar marcação, como acontece com outros *InputStreams*
- Finalmente, o método close() fecha o *Reader* e libera qualquer recurso a ele associado

```
public abstract void close() throws IOException
```


A classe *InputStreamReader*

- A classe *InputStreamReader* serve como uma ponte entre um *stream* de bytes e um *stream* de caracteres
- Ela lê bytes de um *InputStream* subjacente e os converte para caracteres de acordo com um padrão de codificação especificado
- O padrão de codificação pode ser especificado no construtor, ou ser aceito o padrão *default* da plataforma

```
public InputStreamReader(InputStream in)
public InputStreamReader(InputStream in, String encoding)
   throws UnsupportedEncodingException
```

A classe *InputStreamReader*

- Os outros métodos apenas sobre-escrevem os métodos da super classe, mantendo o comportamento

```
public int read() throws IOException
public int read(char c[], int off, int length) throws
   IOException
public boolean ready() throws IOException
public void close() throws IOException
```

- Por exemplo, para se ler um arquivo codificado no padrão Macintosh Symbol deve-se fazer:

```
FileInputStream fin = new FileInputStream("symbol.txt");
InputStreamReader reader = new InputStreamReader(fin,
   "MacSymbol");
```


Outros *Readers* e *Writers*

- *BufferedReader*
- *BufferedWriter*
- *CharArrayReader*
- *CharArrayWriter*
- *FileReader*
- *FileWriter*
- *FilterReader*
- *FilterWriter*
- *LineNumberReader*
- *PipedReader*
- *PipedWriter*
- *PrintWriter*
- *PushbackReader*
- *StringReader*
- *StringWriter*

A classe *PrintWriter*

- A classe *PrintWriter* é uma subclasse de *Writer* que permite usar os conhecidos métodos print() e println()
- Pode ser encadeada a um *OutputStreamWriter*
- O método println() causa o descarregamento (*flushing*) do *stream*, mas isso não acontece quando um caractere de final ou início de linha é escrito individualmente
- Cuidado!
  - Cada println() adiciona um caractere de final de linha ao *stream* que é dependente de plataforma
    - "\n" (*linefeed*) no Unix
    - "\r" (*carriage return*) no Macintosh
    - "\r\n" no Windows

A classe *BufferedReader*

- A classe *BufferedReader* é uma subclasse de *Reader* que pode ser encadeada a outra subclasse para buferizar a leitura de caracteres
- O tamanho do *buffer* pode ser especificado no construtor, ou ser aceito o tamanho *default* de 8.192 caracteres

```
public BufferedReader(Reader in, int bufferSize)
public BufferedReader(Reader in)
```

- Oferece o conhecido método readLine() que permite ler um texto uma linha de cada vez

```
public String readLine() throws IOException
```

  - Pode bloquear se linhas não terminam com "\n"!
- Não suporta marcação, exceto até o tamanho do *buffer*

Exemplo: método readLine()

```
// Implementa o utilitário 'cat' do Unix em Java

import java.io.*;

class cat  {
  public static void main (String args[]) {
    String thisLine;
   //Cicula pelos argumentos
   for (int i=0; i < args.length; i++) {
     //Abre o arquivo para leitura
     try {
       BufferedReader br = new BufferedReader(new FileReader(args[i]));
       while ((thisLine = br.readLine()) != null) {
         System.out.println(thisLine);
       } // final do while
     } // final do try
     catch (IOException e) {
       System.err.println("Error: " + e);
     }
  } // final do for
} // final do main
}
```


Exercício 2.4

- Crie uma subclasse para *Writer* chamada *CipherWriter* que "cifra" os caracteres escritos no *stream* da seguinte forma:
  - Vogais e consoantes devem ser substituídas pelas suas respectivas letras sucessoras no alfabeto, mantendo o mesmo estilo de "capitalização" (considere o alfabeto como uma lista circular).
  - Todos os outros tipos de caracteres (acentos, sinais de pontuação, símbolos, etc) devem ser mantidos inalterados.
- O construtor da subclasse deve receber como parâmetro um *OutputStreamWriter* de destino, para onde os caracteres cifrados serão escritos.
- Teste a sua implementação utilizando como destino a saída padrão da console (System.out).

Exercício 2.5

- Crie uma subclasse para *Reader* chamada *CipherReader* que "decifra" os caracteres cifrados pela classe implementada no exercício 2.4.
- O construtor da subclasse deve receber como parâmetro um *InputStreamReader* de origem, de onde os caracteres a serem decifrados serão lidos.
- Teste a sua implementação utilizando como origem um *InputStreamReader* que leia caracteres de um texto previamente cifrado pela classe *CipherWriter.*

Concorrência

- Por que usar concorrência?
- A classe *Thread*
- Escalonamento de *threads* em Java
- Recursos de sincronização

Por que usar concorrência?

- A concorrência possibilita aumentar o desempenho de um programa através da criação de múltiplas linhas de execução dentro de um mesmo processo
- O preço pago pelo concorrência é um aumento significativo na complexidade do programa
  - Necessidade de mecanismos de sincronização para evitar interferências indevidas nos dados
  - Mau uso desses mecanismos pode levar a situações de *deadlock*
  - Programa fica muito mais difícil de projetar, implementar e testar
- Java foi uma das primeiras linguagens de programação a oferecer suporte nativo para concorrência
  - Implementada através de *Threads*

Por que usar concorrência? (cont.)

- Dividir o trabalho de um único programa através de múltiplas *threads* deve, em geral, aumentar o tempo total de execução do programa
- *Threads* ajudam, no entanto, se o programa vai interagir com coisas imprevisivelmente lentas como:
  - Pessoas
  - Dispositivos de E/S
  - Conexões de rede
- Enquanto uma *thread* fica “bloqueada” aguardando dados de uma pessoa ou dispositivo, outras *threads* podem continuar executando normalmente

O que é uma *thread*?

- Um caminho de execução independente na JVM
- *Threads* possibilitam que um programa Java execute múltiplas tarefas "simultaneamente"
- Todas as *threads* de um programa compartilham o mesmo espaço de endereçamento de memória
  - Uma mesma região de memória pode ser acessada e alterada por diferentes *threads*
  - Risco de uma *thread* interferir indevidamente com as variáveis e estruturas de dados de outra *thread*

Sincronização de *threads*

- Para evitar interferências indevidas nos dados, Java oferece *mecanismos de sincronização*
  - Permitem que uma *thread* obtenha o direito de uso exclusivo sobre um ou mais recursos compartilhados
- A sincronização entre *threads* deve ser usada com cautela, uma vez que o benefício da concorrência é perdido caso o programa inteiro pare aguardando que um recurso compartilhado de uso exclusivo volte a ser liberado
- A escolha de quais objetos e métodos de um programa devem ser sincronizados é a parte mais importante (e difícil de dominar) da programação com *threads*

A classe *Thread*

- Java oferece dois modos de implementar *threads*:
  - Criando uma subclasse para a classe *java.lang.Thread*, ou
  - Criando uma classe que implemente a interface java.lang.Runnable e passando uma instância dessa classe como parâmetro para o construtor de *Thread*
- Em ambos os casos, o programador deve prover a *thread* com um método run(), que corresponderá ao seu método “main()”

Exemplo: estendendo *Thread*

```
import java.io.*;

public class ContadorThread extends Thread {
  public void run() {
    for(int I=0;I<=10;I++) {
      System.out.println("Contando: "  +I);
}}}

public class Exemplo1{
  public static void main(String[] args) {
    ContadorThread contador = new ContadorThread();
    contador.start();
}}
```

Exemplo: implementando *Runnable*

```
import java.io.*;

public class ContadorClass implements Runnable {
  public void run() {
    for(int I=0;I<=10;I++) {
      System.out.println("Contando: "  +I);
}}}

public class Exemplo2{
  public static void main(String[] args) {
    ContadorClass contador = new ContadorClass();
    Thread thread = new Thread(contador);
    thread.start();
}}
```


A classe *Thread* (cont.)

- A classe *Thread* tem três métodos principais:

  ```
  public void start()
  public void run()
  public final void stop()
  ```

  - O médod start() prepara a thread para ser executada
  - O método run() implementa a tarefa da *thread* propriamente dita
  - O método stop() pára a execução da *thread*
- A *thread* é encerrada quando o método run() chega ao fim ou quando o método stop() é invocado
- O método run() nunca deve ser chamado explicitamente
  - Ele é chamado automaticamente pela JVM ao se invocar start()

A classe *Thread* (conc.)

- A classe *Thread* oferece ainda vários outros métodos para:
  - Suspender e continuar a execução de uma *thread*
  - Por uma *thread* para dormir e depois acordá-la
  - Fazer a *thread* que está executando ceder a vez para outras
  - Esperar até que uma *thread* encerre sua execução
  - E muito mais!

Exercício 3.1

- Considere o programa abaixo:

```
public class Exemplo3{
  public static void main(String[] args) {
    System.out.println("Criando os contadores...");
    ContadorThread contador1 = new ContadorThread();
    ContadorThread contador2 = new ContadorThread();
    ContadorThread contador3 = new ContadorThread();
    System.out.println("Iniciando os contadores...");
    contador1.start(); contador2.start(); contador3.start();
    System.out.println("Contadores foram iniciados.");
}}
```

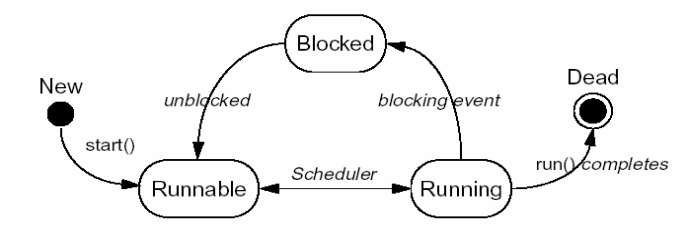
  - O que vai acontecer com o texto impresso na tela por cada um dos contadores? E com o texto impresso na tela pelo método main(), antes e após os contadores serem iniciados?
  - Execute o programa e verifique se o resultado foi o que você esperava. Caso tenha sido diferente, explique o que pode ter acontecido.

Exercício 3.2

- Escreva um programa para compactar uma lista de arquivos cujos nomes são passados como parâmetro para o método main(). Implemente o programa em duas versões:
  - Na primeira, os arquivos devem ser compactados seqüencialmente;
  - Na segunda, os arquivos devem ser compactados “em paralelo”, cada arquivo por uma *thread* em separado.
- Ao final, o programa deve informar o tempo total gasto na compactação dos arquivos.
- Teste as duas versões do programa com diferentes números e tamanhos de arquivos. O que pode ser observado com relação ao desempenho das duas versões do programa?

Estados de uma *Thread*

- Os estados possíveis para uma Thread são:
  - *Running* – o estado a que todas as *threads* aspiram
  - *Blocked* – vários estados de espera (aguardando notificação, dormindo, suspensa, aguardando recurso de I/O)
  - *Runnable* ou *Ready* – aguardando apenas a CPU
  - *Dead* – método run() concluído



- O acesso à CPU é feito exclusivamente pelo escalonador de *threads* (*scheduler*) da JVM

Escalonamento de *threads* em Java

- A especificação de Java não dita como as *threads* devem ser escalonadas
- A maioria das JVMs usa um escalonador pre-emptivo
  - O escalonador pode "forçar" a *thread* com a CPU a ceder a vez para outras *threads*
- *Threads* podem ter diferentes prioridades
  - Prioridade varia entre 1 (mínima) e 10 (máxima) – default é 5
  - Uma *thread* de baixa prioridade nunca é executada enquanto houver uma outra de prioridade mais alta pronta para execução
    - Escalonamento de *threads* de mesma prioridade depende do escalonador
  - Manipulada através dos métodos getPriority e setPriority

Recursos de sincronização

- Utilizados para permitir a sincronização entre as diferentes *threads* de um programa
- Principais recursos:
  - Métodos sincronizados
  - Blocos de código sincronizados
  - Empacotadores sincronizados
  - Espera e notificação

Métodos sincronizados

- Permitir acesso concorrente a objetos compartilhados pode ser muito perigoso
  - Por exemplo, no caso de uma lista, uma *thread* pode estar removendo o "último" elemento da lista ao mesmo tempo em que outra *thread* está inserindo um novo elemento
- Para prevenir acessos concorrentes a objetos compartilhados, em Java métodos podem ser declarados como "sincronizados" (*synchronized*)

Travamento ("*locking*") de objetos

- Todo objeto em Java tem uma "trava" associada a ele
  - A trava tem uma única *thread* como proprietária (proprietário é nulo se o objeto está destravado)
  - A trava tem um contador para contar a profundidade de travamento (utilizado para controlar chamadas recursivas)
- Qualquer invocação de um método sincronizado deve primeiro obter a trava para o objeto alvo da invocação
  - Travamento é por objeto – execução de um método sincronizado implica no bloqueio da execução de qualquer método sincronizado do mesmo objeto pelas outras *threads*
  - Invocação de métodos estáticos deve obter a trava para o objeto que representa a classe

Processo de travamento

- Entrando num método quando o objeto está destravado:
  - A *thread* invocadora torna-se proprietária da trava associada ao objeto, travando o objeto
  - A profundidade da trava é definida para 1, e a *thread* é então autorizada a continuar a execução do método
- Entrando num método quando o objeto está travado, mas a trava é da própria *thread* invocadora:
  - A profundidade da trava é incrementada de 1

Processo de travamento (cont.)

- Entrando num método quando o objeto está travado e a trava não pertence à *thread* invocadora:
  - A *thead* é suspensa e colocada numa fila de sincronização associada ao objeto alvo
- Saindo de um método quando a profundidade da trava é maior que 1
  - A profundidade da trava é decrescida de 1

Processo de travamento (conc.)

- Saindo de um método quando a profundidade da trava é 1 e não há nenhuma *thread* na fila de sincronização:
  - A profundidade da trava é zerada, o proprietário é definido como nulo, e o objeto é destravado
- Saindo de um método quando a profundidade da trava é 1 e há pelo menos uma *thread* na fila de sincronização:
  - Uma *thread* é retirada da fila e sua execução é retomada
  - A *thread* escolhida é definida como a nova proprietária do objeto (profundidade da trava continua 1)

DeadlocK

- *Deadlock* é um estado onde duas ou mais *threads* estão aguardando mutuamente uma às outras, de modo que nenhuma delas jamais conseguirá continuar sua execução
- *Deadlock* é relativamente raro quando se usa apenas métodos sincronizados, mas pode ocorrer se os métodos chamam outros métodos
  - A chave para evitar *deadlock* é garantir que os objetos são sempre travados numa ordem pré-defina, evitando a formação de ciclos de espera

Transações

- Considere uma aplicação onde a semântica de transação é importante, por exemplo na troca de objetos entre duas listas
  - Não é permitido que nenhum objeto "desapareça" momentaneamente das listas enquanto a troca é efetuada (ou seja, todos os objetos devem sempre pertencer a uma das listas)
- Para trocar um objeto de lista de forma "atômica", deve-se:
  1. Travar ambas as listas
  2. Retirar o objeto de sua lista original e inseri-lo na nova lista
  3. Destravar as listas

Blocos de código sincronizados

- O problema da transação atômica é impossível de ser resolvido em Java usando apenas simples coleções de objetos (ex.: vetores) e métodos sincronizados
- Java permite definir blocos de código sincronizados exatamente para esta situação:

```
// código para mover um objeto de v1 para v2
synchronized (v1) { // trava v1
  synchronized (v2) { // trava v2
    v2.add(v1.remove(0));
  } // libera v2
} // libera v1
```

Empacotadores sincronizados

- A classe *Collections* oferece empacotadores ("*wrappers*") sincronizados para as coleções padrões de Java (*List*, *Set,* etc)

```
Vector unSafe = new Vector();
List safe = Collections.synchronizedList(unSafe);
```

- Cada empacotador oferece métodos sincronizados para manipular (incluir/remover) objetos cujas implementações simplesmente invocam os métodos correspondentes na coleção subjacente

Empacotadores sincronizados (cont.)

- Empacotadores não garantem que a iteração sobre os objetos de uma coleção será feita de forma atômica
  - Nesses casos, deve-se usar blocos sincronizados:

```
safe.add(x); // OK.  add é sincronizado
synchronized (safe) {
  ListIterator i = safe.listIterator();
  while (i.hasNext()) {
        x = i.next();
}}
```

- Sem a sincronização, a iteração pode gerar exceções do tipo *ConcurrentModificationException*, caso algum objeto da coleção for modificado concorrentemente por outra *thread* após o início da iteração

Espera e notificação

- Travas não são a melhor solução para muitos problemas de sincronização
  - Considere, por exemplo, o caso em que uma *thread* precisa aguardar até que uma determinada lista não esteja mais vazia
  - Métodos sincronizados permitem checar se a lista está ou não vazia de forma segura, mas a *thread* ainda teria que ficar testando essa condição repetidamente
- Java oferece o recurso *wait/notify* para auxiliar neste tipo de necessidade de sincronização

Espera e notificação (cont.)

- Além das filas de sincronização usadas na implementação de métodos e blocos de código sincronizados, cada objeto em Java também tem associada a ele uma fila de "espera"
- Uma *thread* só pode entrar na fila de espera de um objeto de forma voluntária, invocando o método wait() nesse objeto
  - Antes de invocar wait(), a *thread* deve primeiro obter a trava referente ao objeto
  - Ao invocar wait(), a trava do objeto é imediatamente liberada (mesmo que a execução do método ou bloco sincronizado ainda não tenha sido encerrada)

Espera e notificação (cont.)

- A *thread* que invoca wait() é suspensa e colocada na fila de espera do objeto
- A *thread* suspensa permanece na fila de espera até ela seja "notificada" por outra *thread*
- Após retomar sua execução, a *thread* notificada deve re-adquirir a trava do objeto para então poder prosseguir de forma segura
  - Uma chamada para *notify* transfere uma *thread* da fila de espera do objeto para a sua fila de sincronização
  - A maioria das implementações usa filas do tipo FIFO
- O programador deve se certificar de que as *threads* apenas invocam wait() nos locais apropriados!

Cuidados ao usar *wait/notify*

- Notificação é "silenciosa" se não há nenhuma *thread* suspensa aguardando pelo objeto
  - *Threads* muito lentas podem "perder" suas notificações
- O mecanismo de notificação deve (pelo menos por convenção) ser usado para sinalizar um mudança relevante no estado do objeto alvo
  - O novo estado ainda assim pode não ser apropriado para que uma *thread* que estava aguardando notificação possa continuar sua execução

Convenção para uso de *wait/notify*

- Use **while** para checar o estado do objeto e aguardar por notificações

```
synchronized doit() {
  while (stateIsNotPerfectForMe) {
    try { wait(); }
    catch(InterruptedException annoying) {}
  }
  /* ok to doit now */
}
```

- Use **notifyAll()** após cada mudança de estado

Exemplo: buffer finito

- Problema: implemente uma coleção em Java que possa ser usada por dois conjuntos de *threads* para compartilhar dados
  - Produtoras inserem objetos na coleção
  - Consumidoras removem objetos da coleção
- Sincronização é necessária de três maneiras:
  1. Sincronização no acesso aos métodos da coleção
  2. Consumidoras devem aguardar quando o buffer estiver vazio
  3. Produtoras devem aguardar quando o buffer estiver cheio

Implementação do buffer finito

```
public synchronized void add(Object x) {
    while (isFull()) {
        try { buff.wait(); }
        catch(InterruptedException annoying) {}
    }
    buff.add(x);
    capacity -= 1;
    notifyAll();
}

public synchronized Object remove(int index) {
    while (isEmpty()) {
        try { buff.wait(); }
        catch(InterruptedException annoying) {}
    }
    capacity += 1;
    notifyAll();
    return buff.remove(index);
}
```


Threads e coleta de lixo

- Embora haja coletores de lixo “concorrentes”, em geral é melhor pressupor que estes nem sempre estarão disponíveis
  - Nesse caso, todas as *threads* de um programa são interrompidas pelo coletor enquanto a coleta de lixo é efetuada

Exercício 3.3

- Escreva uma nova versão para o programa concorrente de compactação de arquivos, agora usando um número fixo (pré-definido) de *threads* de compactação.
  - As *threads* deverão ser criadas e iniciadas no início do programa.
  - A comunicação da *thread* principal (método main) com as *threads* de compactação se dará através de um *buffer* finito de *Strings* contendo o nome dos arquivos a serem compactados.
  - O tamanho máximo do *buffer* corresponderá ao número total de *threads* criadas no início do programa.
- Compare o desempenho da nova versão com as duas versões anteriores, utilizando diferentes quantidades e tamanhos de arquivos.

Endereços de rede

- A classe *InetAddress*

A classe *InetAddress*

- Classe genérica utilizada para representar endereços da Internet (nomes de domínio e endereços IP) em Java
- Oferece métodos simples para converter entre nomes de domínio e endereços em formato numérico
- Encapsulada pela maioria dos mecanismos de comunicação em rede disponíveis no pacote *java.net*
- Duas subclasses (a partir de Java1.4):
  - *Inet4Address*
    - Representa endereços do protocolo IPv4 (32 bits)
  - *Inet6Address*
    - Representa endereços do protocolo IPv6 (128 bits)
    - Disponibilidade limitada (dependente de plataforma)

A classe *InetAddress* (cont.)

- Define dois atributos privados:
  - *hostName* (string com um nome do domínio)
  - *address* (inteiro de 32 bits com um endereço IP em formato binário)
- Não possui construtores!
  - Instâncias devem ser criadas através de métodos estáticos apropriados

Métodos estáticos

`InetAddress getLocalHost() throws UnknownHostException`

- Retorna um objeto *InetAddress* com o endereço da máquina local

`InetAddress getByName(String host) throws UnkownHostException`

- Retorna um objeto *InetAddress* como o endereço da máquina cujo nome de domínio ou endereço IP foi especificado em *host*

`InetAddress[] getAllByName(String host) throws UnknownHostException`

- Retorna um vetor de objetos *InetAddress* com todos os endereços associados ao nome de domínio ou endereço IP especificado em *host*

Métodos de instância

`String getHostName()`

- Retorna uma string com o nome de domínio da máquina cujo endereço está representado pelo objeto alvo

`byte[] getAddress()`

- Retorna um vetor de bytes com o endereço IP do objeto alvo em formato binário

`String getHostAddress()`

- Retorna uma string com o endereço IP do objeto alvo em formato numérico
  - "X.X.X.X" (IPv4)
  - "X:X:X:X:X:X:X:X" (IPv6)

Métodos sobre-escritos de *Object*

`public boolean equals(Object o)`

- Retorna verdadeiro se o objeto alvo e o objeto passado como parâmetro são ambos instâncias da classe *InetAddress*, e têm o mesmo endereço IP (independente do nome de domínio de cada um)

`public int hashCode()`

- Retorna um inteiro gerado a partir dos valores da referência e dos atributos do objeto alvo
- Útil quando o objeto alvo é manipulado via tabelas *hash*

`public String toString()`

- Retorna uma string contendo o nome de domínio e o endereço IP (formato numérico) do objeto alvo

Exemplo: descobrindo o nome e o IP da máquina local

```
import java.net.*;

// Obtém o objeto InetAddress referente ao endereço da máquina local
InetAddress localaddr = InetAddress.getLocalHost();

System.out.println ("Nome de domínio: " + localaddr.getHostName());
System.out.println ("Endereço IP: " + localaddr.getHostAddress );
```

Exercício 4.1

- Escreva um programa Java que mostre quantos e quais endereços IP estão associados a um dado nome de domínio
- Teste seu programa utilizando nomes de servidores conhecidos da Web, preferencialmente aqueles que tenham um alto número de usuários/visitantes
- Baseado nos resultados obtidos, responda:
  - O número de endereços associados a um dado servidor é sempre o mesmo, ou pode variar?
  - Há alguma ordem pré-estabelecida com relação à lista de endereços retornada para cada servidor?

Comunicação usando *sockets*

- Comunicação assíncrona
- Comunicação síncrona

Comunicação assíncrona

- A classe *DatagramPacket*
- A classe *DatagramSocket*
- A classe *MulticastSocket*

A classe *DatagramPacket*

- Implementação em Java de um pacote do protocolo UDP
- Usada para enviar e receber dados de forma rápida e não confiável via objetos da classe *DatagramSocket*
- Pacotes de tamanho limitado:
  - Máximo teórico de 65507 bytes
  - Na prática, pacotes maiores do que 8k são truncados na maioria das plataformas
  - Uma implementação do protocolo UDP não é obrigada a aceitar pacotes maiores do que 576 bytes
  - Tamanho recomendado: 512 bytes

Construtores (pacotes de entrada)

`DatagramPacket (byte[] buffer, int length)`

- Dados do pacote são recebidos e armazenados em *buffer* (iniciando na posição *buffer[0]* e continuando até os dados serem totalmente armazenados ou *length* bytes terem sido escritos)

`DatagramPacket (byte[] buffer, int offset, int length)`

- Dados do pacote são recebidos e armazenados em *buffer* (iniciando na posição *buffer[offset]* e continuando até os dados serem totalmente armazenados ou *length* bytes terem sido escritos)
- *length* tem que ser menor ou igual a (*buffer.length - offset*)

Construtores (pacotes de saída)

```
DatagramPacket (byte[] data, int length, InetAddress
  destination, int port)
```

- Pacote é preenchido com *length* bytes do vetor *data* (iniciando na posição *data[0]*); *destination* e *port* indicam o endereço de rede e a porta do destino para onde o pacote será enviado
- *length* deve ser menor ou igual a *data.length*

```
DatagramPacket (byte[] data, int offset, int length,
  InetAddress destination, int port)
```

- Pacote é preenchido com *length* bytes do vetor *data* (iniciando na posição *data[offset]*); *destination* e *port* indicam o endereço de rede e a porta do destino para onde o pacote será enviado
- *length* deve ser menor ou igual a (*data.length – offset*)

Métodos de instância

`InetAddress getAddress()`
- Devolve o endereço de rede (IP) definido para o pacote

`int getPort()`
- Devolve a porta associada ao pacote

`byte[] getData()`
- Devolve um vetor de bytes contendo os dados do pacote

`int getLength()`
- Devolve o número de bytes dos dados do pacote (pode ser menor que *getData().length*!)

Métodos de instância (cont.)

`int getOffset()`
- Devolve a posição no vetor devolvido por *getData()* onde os dados do pacote começam

`void setAddress(InetAddres ip)`
- Define o endereço de rede do pacote

`void setPort(int port)`
- Define a porta associada ao pacote

`void setLength()`
- Define o número de bytes do buffer interno que devem ser considerados como os dados do pacote

Métodos de instância (cont.)

`void setData(byte[] data)`

- Define os bytes do buffer interno do pacote

`void setData(byte[] data, int offset, int length)`

- Define os bytes do buffer interno do pacote (iniciando na posição *data[offset]* e continuando até a posição *data[offset + length]*

Exemplo: criando um pacote de saída

```
import java.net.*
...
// Define os dados do pacote a partir de uma string
String s = "Hello World!";
byte[] dados = s.getBytes("ASCII");
try {
   // Define o IP e a porta a serem usados como destino do pacote
   InetAddress ip = InetAddress.getByName("unifor.br");
   int porta = 7;
   // Cria o pacote passando os dados da string,o IP, e a porta
   DatagramPacket pacote = new DatagramPacket (dados, dados.length, ip,
   porta);
   // Mostra os dados do pacote na tela
   System.out.println(new String(pacote.getData(), "ASCII"));
}
catch (IOException e) {
}
```

A classe *DatagramSocket*

- Implementação em Java de um *socket* do tipo *datagram* (protocolo UDP)
- Uma programa deve criar uma instância de *DatagramSocket* para enviar/receber pacotes para/de outros programas através do protocolo UDP
  - *socket* deve estar acoplado a uma porta local
  - Programa que envia o primeiro pacote (cliente) não precisa especificar a porta (S.O. aloca a primeira disponível)
  - Programa que recebe o primeiro pacote (servidor) deve escolher uma porta específica, conhecida pelos programas clientes

Construtores

`DatagramSocket() throws SocketException`

- Cria um *socket* acoplado a um porta local qualquer (usado no lado do cliente)

`DatagramSocket(int port) throws SocketException`

- Cria um *socket* acoplado à porta especificada em *port* (usado no lado do servidor)

```
DatagramSocket(int port, InetAddress address) throws
  SocketException
```

- Cria um *socket* acoplado à porta especificada em *port* e que apenas pode receber pacotes enviados para o endereço especificado em *address* (usado em computadores com múltiplos IPs)

Métodos de instância

`void send(DatagramPacket dp) throws IOException`

- Envia o pacote UDP especificado em *dp* para o seu endereço de destino (processo é liberado imediatamente após o pacote ser enviado)

`void receive(DatagramPacket dp) throws IOException`

- Recebe um único pacote UDP da rede e o armazena no objeto especificado em *dp* (processo fica bloqueado até a chegada do pacote)

`void close()`

- Fecha o *socket* e libera a porta à qual ele estava acoplado

`int getLocalPort()`

- Devolve o valor da porta local acoplada ao *socket*

Métodos de instância (cont.)

`void connect(InetAddress address, int port)`

- Restringe o uso do *socket* apenas para o endereço IP e a porta especificados em *address* e *port*, respectivamente

`int getPort()`

- Devolve o valor da porta definido pelo método `connect()`

`InetAddress getInetAddress()`

- Devolve o endereço IP definido pelo método `connect()`

`void disconnect()`

- Retira todas as restrições definidas pelo método `connect()`

Exemplo: enviando pacotes

```
import java.net.*
...
try {
   // Define um endereço de destino (IP + porta)
   InetAddress servidor = InetAddress.getByName("unifor.br");
   int porta = 1024;
   // Cria o socket
   DatagramSocket socket = new DatagramSocket()
   // Laço para ler linhas do teclado e enviá-las ao endereço de destino
   BufferedReader teclado = new BufferedReader(new InputStreamReader(System.in));
   String linha = teclado.readLine();
   while (!linha.equals(".")) {
         // Cria um pacote com os dados da linha
         byte[] dados = linha.getBytes();
         DatagramPacket pacote = new DatagramPacket(dados, dados.length, servidor, porta)
         // Envia o pacote ao endereço de destino
         socket.send(pacote);
         // Lê a próxima linha
         linha = teclado.readLine();
   }
} catch (UnkownHostException e) {}
catch (SocketException e){} catch(IOException){}
```


Exemplo: recebendo pacotes

```
import java.net.*
...
int porta = 1024; // Define porta
byte[] buffer = new byte[1000]; // Cria um buffer local
try {
   // Cria o socket
   DatagramSocket socket = new DatagramSocket(porta)
   // Cria um pacote para receber dados da rede no buffer local
   DatagramPacket pacote = new DatagramPacket(buffer, buffer.length);
   // Laço para receber pacotes e mostrar seus conteúdos na saída padrão
   while (true) {
         try {
            socket.receive(pacote);
            String conteudo = new String(pacote.getData(), 0, pacote.getLength());
            System.out.println("Pacote recebido de " + pacote.getAddress());
            System.out.println("Conteúdo do pacote: " + conteudo);
            // Redefine o tamanho do pacote
            pacote.setLength(buffer.length)
         } catch (IOException e) {}
   }
} catch (SocketException e){}
```

Exercício 5.1

- Escreva um programa em Java para criar vários *sockets* do tipo *datagram* no lado do cliente. Para cada *socket* criado, o programa deve mostrar a porta que lhe foi atribuída pelo S.O.
- Execute o programa repetidas vezes, se possível em diferentes computadores. Baseado nas suas observações, responda:
  - Há alguma relação entre os valores de porta obtidos?
  - Os valores mudam de execução para execução? Por que?
- Altere o programa de modo que cada *socket* criado seja explicitamente fechado antes da criação do próximo *socket.* Re-execute o programa modificado.
  - Houve alguma mudança em relação aos valores de porta obtidos com a versão anterior do programa? Por que?

Exercício 5.2

- Implemente um “servidor de eco” em Java para pacotes UDP. O servidor deve aguardar pacotes numa porta específica. A cada pacote recebido, o servidor deve mostrar seu conteúdo na tela e em seguida re-enviar o pacote de volta ao seu endereço de origem.
- Implemente uma aplicação cliente para enviar uma dada quantidade de pacotes ao servidor de eco. A aplicação deve verificar e informar se todo os pacotes enviados ao servidor estão sendo recebidos de volta, e se a ordem de entrega é a mesma em que eles foram enviados.

Exercício 5.2 (cont.)

- Execute a aplicação repetidas vezes, com diferentes quantidades de pacotes. O que pode ser observado com relação ao recebimento e à ordem de entrega dos pacotes, tanto no servidor quanto na aplicação cliente?

A classe *MulticastSocket*

- Subclasse de *DatagramSocket* para difusão seletiva baseada no protocolo UDP
- Necessária para receber pacotes enviados a “grupos” de processos
  - Grupos são identificados por endereços IP da classe D (224.0.0.0 a 239.255.255.255), também conhecidos como endereços de difusão seletiva (*multicast addresses*)
  - Grupos podem ser permanentes ou transientes
    - Permanentes – existem mesmo sem membros
    - Transientes – só existem enquanto houver membros
  - Processos podem livremente criar, se juntar a, ou sair de um grupo qualquer

A classe *MulticastSocket* (cont.)

- Uso e comportamento similares aos de *DatagramSocket*
- Acrescenta métodos para um processo se juntar a um grupo de difusão seletiva (*joingGroup*), enviar pacotes ao grupo (*send*), e posteriormente sair do grupo (*leaveGroup*)
  - Ao se juntar a um grupo, o processo passa a receber todos os pacotes enviados para o endereço do grupo
  - Ao sair de um grupo, o processo deixa de receber os pacotes enviados ao endereço do grupo até que o *socket* seja fechado ou o processo volte a se juntar ao grupo
- Não é necessário se juntar a um grupo para enviar pacotes para os seus membros!
  - Pacotes podem ser enviados através de um *DatagramSocket*

Construtores

`MulticastSocket() throws SocketException`

- Cria um *socket* acoplado a um porta local qualquer (usado no lado do cliente)

`MulticastSocket(int port) throws SocketException`

- Cria um *socket* acoplado à porta especificada em *port* (usado no lado do servidor)
- Porta não pode estar sendo usada por outro *DatagramSocket*

Métodos de instância

`void joinGroup(InetAddress group) throws IOException`

- Associa o *socket* ao grupo cujo endereço IP é passado em *group* (gera exceção se endereço não for da classe D)
- Um mesmo *socket* pode se juntar a múltiplos grupos (pacotes de um mesmo grupo podem ser identificados pelo endereço de destino de cada pacote recebido)

`void leaveGroup(InetAddress group) throws IOException`

- Desassocia o *socket* do grupo cujo endereço IP é passado em *group* (gera exceção se endereço não for da classe D; sem efeito se o *socket* não fazia parte do grupo)

`void send(DatagramPacket dp, byte ttl) throws IOException`

- Envia o pacote UDP especificado em *dp* para o seu grupo de destino, utilizando como "tempo de sobrevida" o valor especificado em *ttl*
- *ttl* = 1 quando pacote é enviado pelo método *send* da superclasse

Métodos de instância (cont.)

`void setTimeToLive(int ttl) throws IOException`

- Redefine o valor do tempo de sobrevida para pacotes enviados pelo método *send* da superclasse

`int getTimeToLive() throws IOException`

- Devolve o valor do tempo de sobrevida utilizado no envio de pacotes pelo método *send* da superclasse

Exemplo: um *sniffer* para endereços de difusão seletiva

```
import java.net.*;
import java.io.*;

public class MulticastSniffer {
  public static void main(String[] args) {
    InetAddress group = null;
    int port = 0;
    // Lê o endereço do grupo (IP + porta) da linha de comando
    try {
      group = InetAddress.getByName(args[0]);
      port = Integer.parseInt(args[1]);
    }
    catch (Exception e) {
      // Erro na leitura dos argumentos ou endereço inválido
      System.err.println("Uso: java MulticastSniffer endereço porta");
      System.exit(1);
    }
    ...
```


Exemplo: um *sniffer* para grupos de difusão seletiva (cont.)

```
    MulticastSocket ms = null;
    try {
      // Cria um socket associado ao endereço do grupo
      ms = new MulticastSocket(port);
      ms.joinGroup(group);
      // Cria uma área de dados para receber conteúdo dos pacotes
      byte[] buffer = new byte[80];
      // Laço para recebimento de pacotes e impressão do conteúdo
      while (true) {
        DatagramPacket dp = new DatagramPacket(buffer, buffer.length);
        ms.receive(dp);
        String s = new String(dp.getData());
        System.out.println(s);
      }
    }
    catch (IOException e) {
      System.err.println(e);
    }
    ...
```

Exemplo: um *sniffer* para grupos de difusão seletiva (conc.)

```
    ...
    // Em caso de erro ou interrupção do programa,
    // sinaliza saída do grupo e fecha socket
    finally {
      if (ms != null) {
        try {
          ms.leaveGroup(group);
          ms.close();
        }
        catch (IOException e) {}
      } // if
    } // finally
  } // main
} // class
```


Exercício 5.3

- Utilize a aplicação cliente implementada como parte do Exercício 5.2 para testar o *sniffer* cujo código foi mostrado nos três slides anteriores.
  - Execute o *sniffer* em múltiplas maquinas de uma mesma rede local, passando o endereço “all-systems.mcast.net” e uma porta de sua escolha como parâmetros na linha de comando.
  - Execute a aplicação cliente em uma máquina qualquer da mesma rede da máquina do *sniffer*, de modo que ela envie uma quantidade variada de pacotes UPD para o endereço e a porta especificados acima.
  - O que pode ser observado em relação ao recebimento dos pacotes pelos *sniffers*? Há mudanças na ordem de recebimento de um sniffer em relação à dos outros?

Comunicação síncrona

- A classe *Socket*
- A classe *ServerSocket*

A classe *Socket*

- Implementação em Java de um *socket* do tipo *stream* (protocolo TCP)
- Um cliente deve criar uma instância de *Socket* para se conectar a um servidor remoto
  - A criação do *Socket* conecta automaticamente o cliente ao servidor
  - Cliente precisa informar o nome ou IP da máquina e a porta de comunicação do servidor

Construtores

`Socket (String host, int port) throws IOException`

- Cria um *socket* conectado ao servidor cujo endereço (nome ou IP) e porta são especificados em *host* e *port*, respectivamente

`Socket (InetAddress address, int port) throws IOException`

- Cria um *socket* conectado ao servidor cujo endereço IP (objeto da classe *InetAddress*) e porta são especificados em *address* e *port*, respectivamente

Métodos de instância

`void close() throws IOException`

- Fecha o *socket*, desconectando os canais de recebimento e de envio de dados

`InetAddres getInetAddress()`

- Retorna o endereço IP da máquina do servidor ao qual o *socket* está conectado

`int getPort()`

- Retorna a porta do servidor ao qual o *socket* está conectado

Métodos de instância (cont.)

`InputStream getInputStream() throws IOException`

- Retorna um objeto do tipo *InputStream* correspondente ao canal de recebimento de dados do *socket*

`OutPutStream getOutputStream() throws IOException`

- Retorna um objeto do tipo *OutputStream* correspondente ao canal de envio de dados do *socket*

Exemplo: estabelecendo uma conexão

```
import java.net.*
import java.io.*
 ...
// Cria um socket conectado ao servidor web da Amazon.com
Socket socket = new Socket ("www.amazon.com", 80);

// Obtém canal de envio de dados
OutputStream out = socket.getOutputStream();

// Obtém canal de recebimento de dados
InputStream in = socket.getInputStream();
```

Exercício 5.4

- Escreva um programa em Java que mostre em quais portas de um dado endereço de rede há servidores aguardando conexões TCP. Um servidor está aguardando conexão em uma determinada porta de um endereço de rede se for possível criar um socket conectado a esse endereço nessa porta.
  - Reduza o tempo de execução do programa limitando a busca a uma determinada faixa de valores de porta
  - Caso o tempo de verificação de cada porta seja demasiadamente longo, pesquise a documentação da classe Socket e descubra como diminuir o tempo máximo de espera (timeout) do socket para estabelecimento de conexões

Exercício 5.5

- Implemente um programa em Java que funcione como um cliente de servidores HTTP. Na Internet, um cliente interage com um servidor HTTP através de conexões baseadas no protocolo TCP, normalmente utilizando a porta 80.
- O cliente deve requisitar um arquivo ".html" qualquer do servidor, e então mostrar os dados do arquivo, tais como recebidos do servidor, na sua saída padrão.
- O comando HTTP (versão 1.0) para a requisição de arquivos deve ser uma string da forma:

  `"GET nome_do_arquivo.html HTTP/1.0\r\n\n"`

Exercício 5.5 (cont.)

- Tente requisitar o mesmo arquivo do servidor, mas agora utilizando outros comandos do protocolo HTTP (por exemplo, TRACE, HEAD, etc).
- Compare o resultado desses comandos com aqueles obtidos com comando GET, e discuta qual seria a utilidade desses outros comandos na prática.

A classe *ServerSocket*

- Implementação em Java do mecanismo para aceitar conexões do tipo *stream* (protocolo TCP) no lado do servidor
- Um objeto *ServerSocket* deve ser criado acoplado a alguma porta local
  - Valores de porta variam entre 1 e 65535
  - Portas de 1 a 1023 são reservadas para serviços do sistema: eco (7), hora do dia (13), FTP (21), Telnet (23), SMTP (25), Finger (79), HTTP (80), etc.
- A cada nova conexão aceita, o *ServerSocket* retorna um novo *socket* (da classe *Socket*) já conectado ao *socket* do cliente

Construtores

`ServerSocket (int port) throws IOException`

- Cria um *ServerSocket* que irá aguardar conexões na porta especificada em *port* (com 50 conexões pendentes, no máximo)
- Se *port* = 0, a escolha da porta fica a critério do S.O.

`ServerSocket (int port, int backlog) throws IOException`

- Igual ao anterior, mas com o número máximo de conexões pendentes especificado em *backlog*

Métodos de instância

`Socket accept() throws IOException`

- Bloqueia o servidor até que seja estabelecida uma conexão com algum cliente
- Retorna um *socket* (da classe *Socket*) já conectado ao *socket* do cliente que solicitou a conexão

`void close() throws IOException`

- Fecha o *ServerSocket* para futuras conexões (conexões pendentes continuarão sendo tratadas)

`int getLocalPort()`

- Retorna o valor da porta à qual o *ServerSocket* está acoplado

Exemplo: aceitando novas conexões

```
import java.net.*
...
// Cria um ServerSocket acoplado à porta 1024
ServerSocket server = new ServerSocket(1024);
// Laço para tratamento de conexões
while (true) {
   // Aguarda uma nova solicitação de conexão de um cliente
   Socket cliente = server.accept();
   // Trata a nova conexão aceita
   ...
   // Fecha a conexãote.close();
}
```


Exercício 5.6

- Implemente um programa *seqüencial* em Java, que funcione como um servidor de eco para mensagens enviadas via TCP. O servidor deve aguardar conexões em uma porta (>1023) a ser definida pelo usuário.
  - A cada nova conexão aceita, o servidor deve enviar ao cliente a mensagem “Seja bem-vindo!”
  - A partir daí, toda nova mensagem recebida de um cliente deve ter seu conteúdo exibido na saída padrão do servidor, e então ser imediatamente re-enviada de volta para o cliente
- Teste o seu programa utilizando várias aplicações Telnet como clientes do servidor.

Exercício 5.7

- Re-implemente o servidor de eco do exercício anterior, agora utilizando múltiplas *threads* para atender às conexões de forma concorrente.
- Compare e discuta as diferenças na execução das duas versões do programa.

Exercício 5.8

- Projete e implemente uma API genérica de comunicação em Java. A API deve fornecer um conjunto comum de interfaces de comunicação que poderão ser implementadas utilizando diferentes mecanismos de comunicação (por exemplo, UDP ou TCP).
- Utilize o padrão de projeto *Factory Method* ou *Abstract Factory* para permitir que uma aplicação instancie as abstrações de comunicação fornecidas pela API (como endereços, portas, conexões e mensagens) sem especificar as suas classes concretas diretamente.

Exercício 5.8 (cont.)

- Escreva uma aplicação de teste para demonstrar o uso da API proposta na prática. A aplicação dever ser compilada e executada utilizando pelo menos duas instâncias diferentes da API, cada qual baseada em um diferente mecanismo de comunicação.
- Dica 1: baseie o projeto da API na classes que implementam comunicação baseada em *socket* TCP de Java, e adapte esse modelo para funcionar sobre *socket* UDP.
- Dica 2: considere que as aplicações que utilizam a API apenas transmitem mensagens textuais (tipo *String*) de tamanho reduzido (até no máximo 512 bytes).

Aplicação piloto: Bate-papo

Descrição da aplicação

- Implemente uma versão simplificada de uma aplicação de bate-papo (*chat*) para a Internet.
- A aplicação deve ser implementada seguindo o modelo cliente/servidor, com a comunicação entre os clientes e o servidor feita exclusivamente através da API proposta no Exercício 5.8.

Especificação do servidor

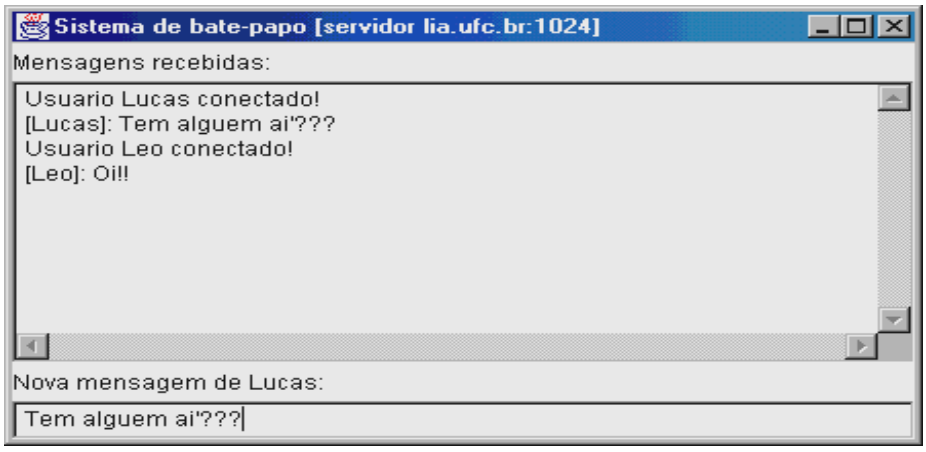
- O servidor deve implementar um **serviço de difusão de mensagens**, através do qual os clientes poderão "bater-papo" enviando mensagens ao servidor.
- O servidor deve manter uma lista com os clientes atualmente conectados. Ao receber uma mensagem de um cliente, o servidor deve mostrá-la na saída padrão (terminal) e imediatamente reenviá-la para todos os clientes da lista (incluindo o próprio cliente de origem).

Especificação do cliente

- O cliente deve oferecer uma interface gráfica contendo basicamente:
  - uma área de saída de texto, para mostrar as mensagens recebidas do servidor;
  - uma linha de entrada de texto, para digitação e envio de novas mensagens ao servidor.
- Para permitir a identificação das mensagens, cada cliente deve possuir um nome (ou apelido) próprio, que pode ser fornecido como parâmetro de execução da aplicação.

Especificação do cliente (cont.)

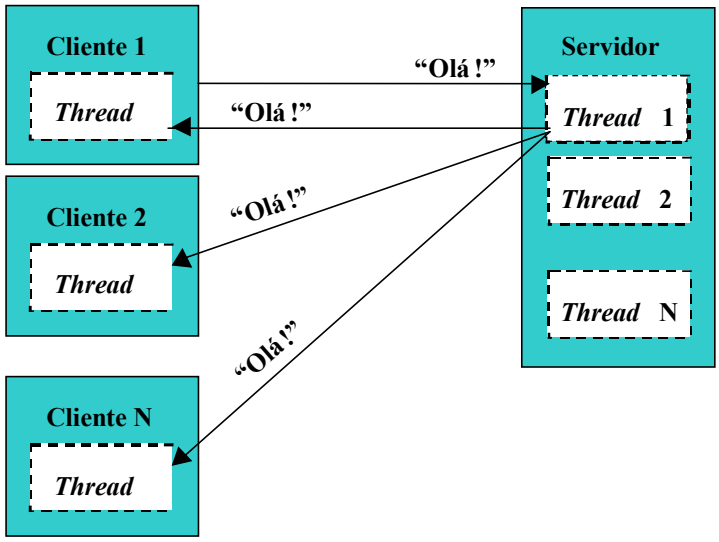
- Exemplo de interface gráfica:



Dicas de implementação

- No servidor, crie múltiplas *threads* para tratar de maneira concorrente o recebimento e a difusão das mensagens enviadas por cada cliente.
- No cliente, crie uma nova *thread* exclusivamente para receber as mensagens do servidor e atualizar o conteúdo da área de saída de texto.
- Não esqueça de sincronizar os métodos necessários no servidor (qualificador *synchronized*), para evitar inconsistências na lista de conexões.

Esquema de execução



Sugestões para outras aplicações

- Trabalho cooperativo
  - Quadro de desenho/pintura compartilhado
  - Edição de texto compartilhada
- Jogos em rede
  - Forca
  - Memória
  - Dominó
  - Palitinhos (“porrinha”)
  - Seqüência (“genius”)
  - Serpente
  - Labirinto
  - ...e o que mais a sua imaginação criar!