CAPÍTULO 19

Apêndice - Sockets

"Olho por olho, e o mundo acabará cego."
— Mohandas Gandhi

Conectando-se a máquinas remotas.

19.1 - MOTIVAÇÃO: UMA API QUE USA OS CONCEITOS

APRENDIDOS

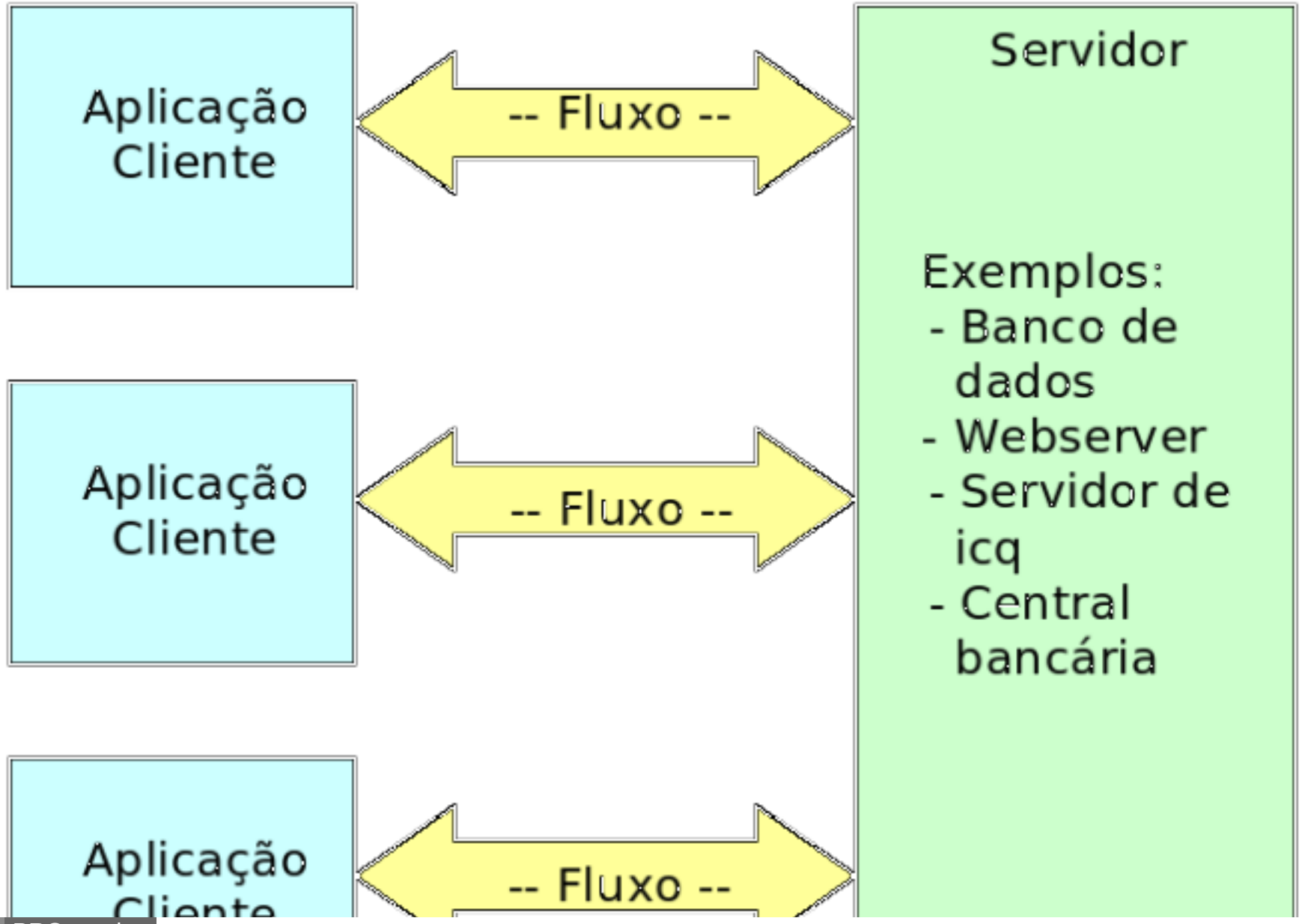
Neste capítulo, você vai conhecer a API de **Sockets** do java pelo pacote `java.net`.

Mais útil que conhecer a API, é você perceber que estamos usando, aqui, todos os conceitos e bibliotecas aprendidas durante os outros capítulos. Repare, também, que é relativamente simples aprender a utilizar uma API, agora que temos todos os conceitos necessários para tal.

19.2 - PROTOCOLO

Da necessidade de dois computadores se comunicarem, surgiram diversos protocolos que permitissem tal troca de informação: o protocolo que vamos usar aqui é o **TCP** (Transmission Control Protocol).

Através do **TCP**, é possível criar um fluxo entre dois computadores - como é mostrado no diagrama abaixo:

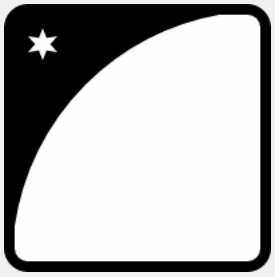


É possível conectar mais de um cliente ao mesmo servidor, como é o caso de diversos banco de dados, servidores Web, etc.

Ao escrever um programa em Java que se comunique com outra aplicação, não é necessário se preocupar com um nível tão baixo quanto o protocolo. As classes que trabalham com eles já foram disponibilizadas para serem usadas por nós no pacote `java.net`.

A vantagem de se usar TCP, em vez de criar nosso próprio protocolo de bytes, é que o TCP vai garantir a entrega dos pacotes que transferirmos e criar um protocolo base para isto é algo bem complicado.

Você pode também fazer o curso FJ-11 dessa apostila na Caelum



Querendo aprender ainda mais sobre Java e boas práticas de orientação a objetos? Esclarecer dúvidas dos exercícios?

Ouvir explicações detalhadas com um instrutor?

A Caelum oferece o **curso FJ-11** presencial nas cidades de São Paulo, Rio de Janeiro e Brasília, além de turmas incompany.

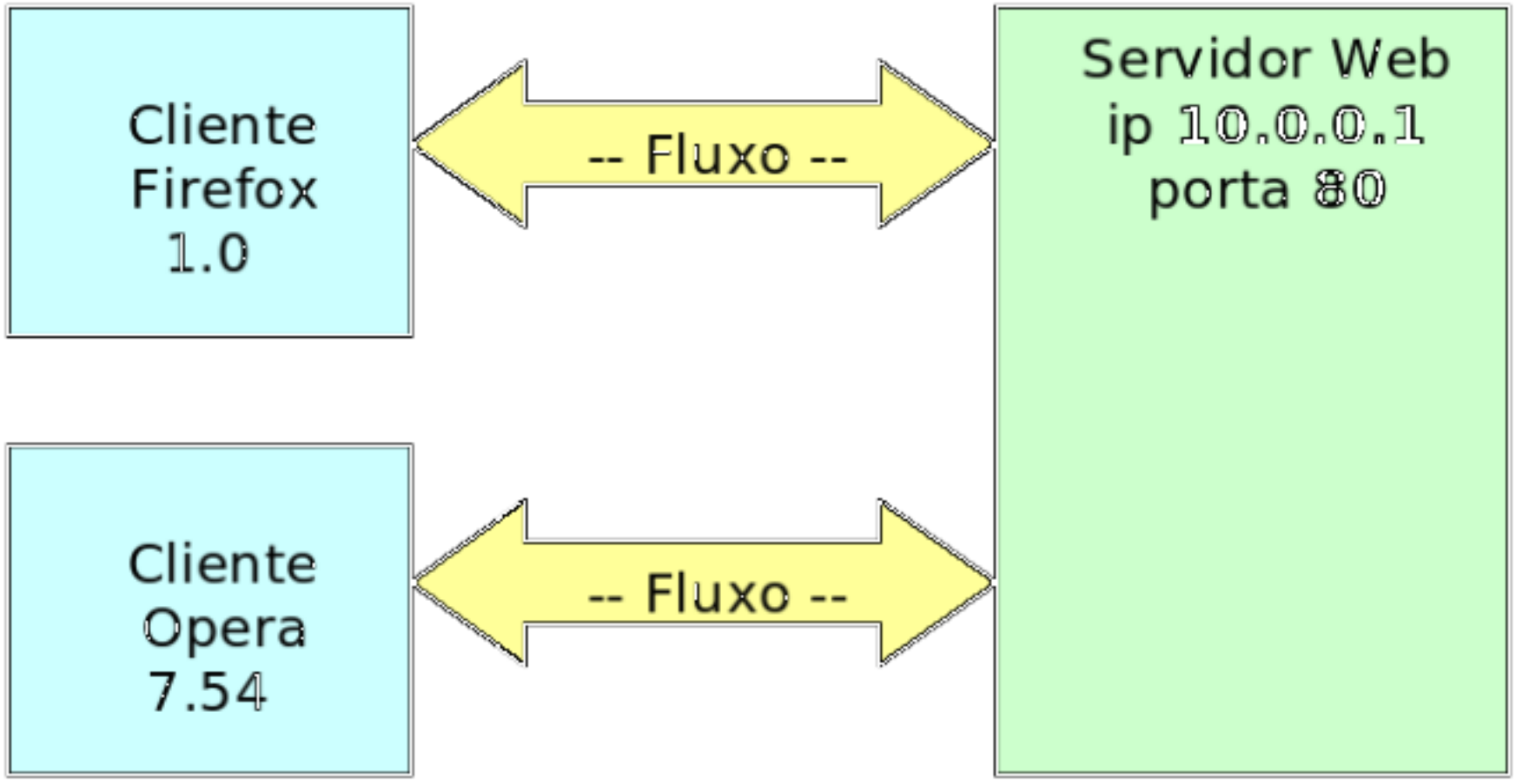
Consulte as vantagens do curso Java e Orientação a Objetos.

19.3 - PORTA

Acabamos de mencionar que diversos computadores podem se conectar a um só, mas, na realidade, é muito comum encontrar máquinas clientes com uma só conexão física. Então, como é possível se conectar a dois pontos? Como é possível ser conectado por diversos pontos?

Todas as aplicações que estão enviando e recebendo dados fazem isso

através da mesma conexão física, mas o computador consegue discernir, durante a chegada de novos dados, quais informações pertencem a qual aplicação. Mas como?



Assim como existe o **IP** para identificar uma máquina, a **porta** é a solução para identificar diversas aplicações em uma máquina. Esta porta é um número

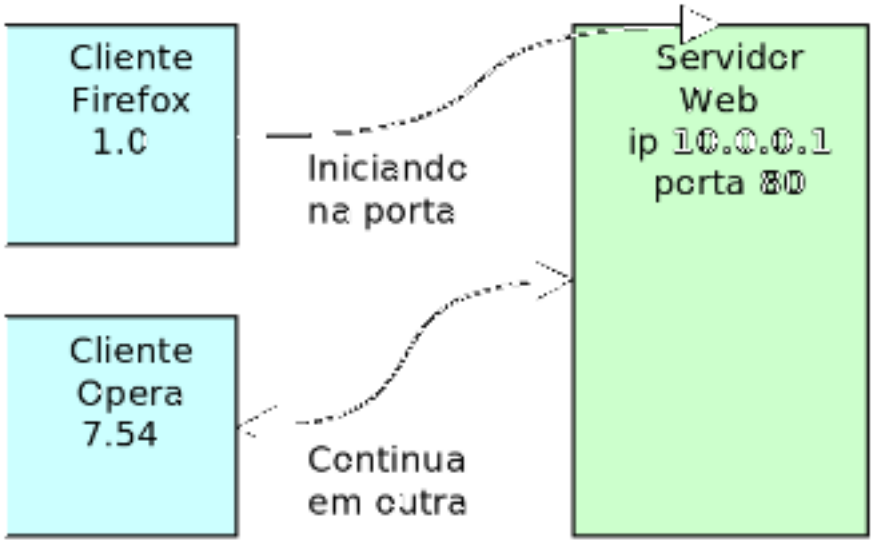
de 2 bytes, **varia de 0 a 65535**. Se todas as portas de uma máquina estiverem ocupadas, não é possível se conectar a ela enquanto nenhuma for liberada.

Ao configurar um servidor para rodar na porta 80 (padrão http), é possível se conectar a esse servidor através dessa porta que, junto com o ip, vai formar o endereço da aplicação. Por exemplo, o servidor web da caelum.com.br pode ser representado por: caelum.com.br:80

19.4 - SOCKET

Mas se um cliente se conecta a um programa rodando na porta 80 de um servidor, enquanto ele não se desconectar dessa porta, será impossível que outra pessoa se conecte?

Acontece que, ao efetuar e aceitar a conexão, o servidor redireciona o cliente de uma porta para outra, liberando novamente sua porta inicial e permitindo que outros clientes se conectem novamente.



Em Java, isso deve ser feito através de threads e o processo de aceitar a conexão deve ser rodado o mais rápido possível.

19.5 - SERVIDOR

Iniciando um modelo de servidor de chat, o serviço do computador que funciona como base deve, primeiro, abrir uma porta e ficar ouvindo até alguém tentar se conectar.

```java
import java.net.*;

public class Servidor {
  public static void main(String args[]) throws IOException {

    ServerSocket servidor = new ServerSocket(12345);
    System.out.println("Porta 12345 aberta!");
      // a continuação do servidor deve ser escrita aqui

  }
}
```

Se o objeto for realmente criado, significa que a porta 12345 estava fechada e foi aberta. Se outro programa possui o controle desta porta neste instante, é normal que o nosso exemplo não funcione, pois ele não consegue utilizar uma porta que já está em uso.

Após abrir a porta, precisamos esperar por um cliente através do método `accept` da `ServerSocket`. Assim que um cliente se conectar, o programa continuará, por isso dizemos que esse método é blocante, segura a thread até que algo o notifique.

```
Socket cliente = servidor.accept();
System.out.println("Nova conexão com o cliente " +
  cliente.getInetAddress().getHostAddress()
); // imprime o ip do cliente
```

Por fim, basta ler todas as informações que o cliente nos enviar:

```
Scanner scanner = new Scanner(cliente.getInputStream());

while (scanner.hasNextLine()) {
  System.out.println(scanner.nextLine());
}
```

Fechamos as conexões, começando pelo fluxo:

```
in.close();
cliente.close();
servidor.close();
```

O resultado é a classe a seguir:

```
1  public class Servidor {
```

```
2    public static void main(String[] args) throws IOException {
3      ServerSocket servidor = new ServerSocket(12345);
4      System.out.println("Porta 12345 aberta!");
5
6      Socket cliente = servidor.accept();
7      System.out.println("Nova conexão com o cliente " +
8        cliente.getInetAddress().getHostAddress()
9      );
10
11     Scanner s = new Scanner(cliente.getInputStream());
12     while (s.hasNextLine()) {
13       System.out.println(s.nextLine());
14     }
15
16     s.close();
17     servidor.close();
18     cliente.close();
19   }
20 }
```

Tire suas dúvidas no GUJ Respostas

O GUJ é um dos principais fóruns brasileiros de computação e o maior em português sobre Java. A nova versão do GUJ é baseada em uma ferramenta de perguntas e respostas (QA) e tem uma comunidade muito forte. São mais de 150 mil usuários pra ajudar você a esclarecer suas dúvidas.

Faça sua pergunta.

19.6 - CLIENTE

A nossa tarefa é criar um programa cliente que envie mensagens para o servidor... o cliente é ainda mais simples do que o servidor.

O código a seguir é a parte principal e tenta se conectar a um servidor no IP 127.0.0.1 (máquina local) e porta 12345:

```
Socket cliente = new Socket("127.0.0.1",12345);
System.out.println("O cliente se conectou ao servidor!");
```

Queremos ler os dados do cliente, da entrada padrão (teclado):

```
Scanner teclado = new Scanner(System.in);
while (teclado.hasNextLine()) {
  // lê a linha e faz algo com ela
}
```

Basta ler as linhas que o usuário digitar através do buffer de entrada (`in`), e jogá-las no buffer de saída:

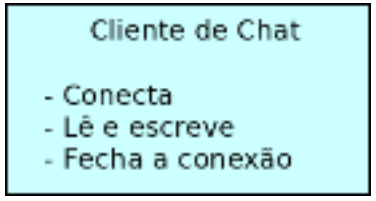
```
PrintStream saida = new PrintStream(cliente.getOutputStream());
Scanner teclado = new Scanner(System.in);
while (teclado.hasNextLine()) {
  saida.println(teclado.nextLine());
}
saida.close();
teclado.close();
```

Repare que usamos os conceito de `java.io` aqui novamente, para leitura do teclado e envio de mensagens para o servidor. Para as classes `Scanner` e `PrintStream`, tanto faz de onde que se lê ou escreve os dados: o importante é

que esse stream seja um `InputStream` / `OutputStream`. É o poder das interfaces, do polimorfismo, aparecendo novamente.

Nosso programa final:

```
1  public class Cliente {
2    public static void main(String[] args)
3            throws UnknownHostException, IOException {
4      Socket cliente = new Socket("127.0.0.1", 12345);
5      System.out.println("O cliente se conectou ao servidor!");
6
7      Scanner teclado = new Scanner(System.in);
8      PrintStream saida = new PrintStream(cliente.getOutputStream());
9
10     while (teclado.hasNextLine()) {
11       saida.println(teclado.nextLine());
12     }
13
14     saida.close();
15     teclado.close();
16     cliente.close();
17   }
18 }
```



Para testar o sistema, precisamos rodar primeiro o servidor e, logo depois, o cliente. Tudo o que for digitado no cliente será enviado para o servidor.

Multithreading

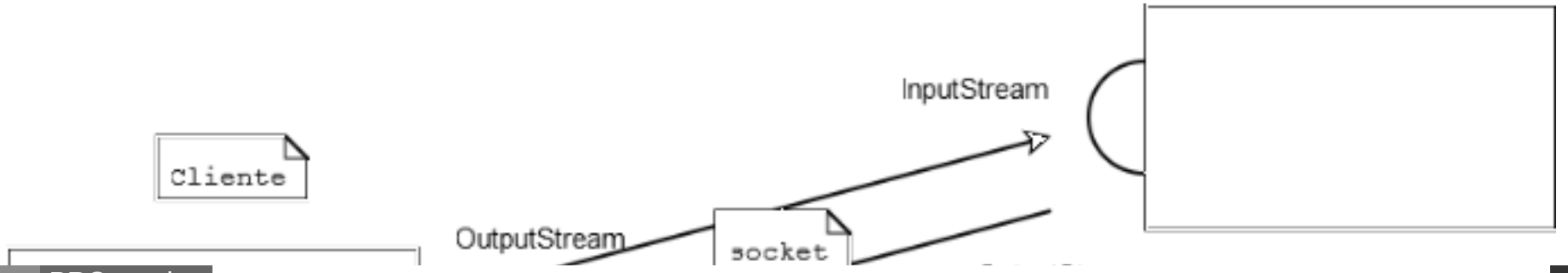
Para que o servidor seja capaz de trabalhar com dois clientes ao mesmo tempo é necessário criar uma thread logo após executar o método `accept`.

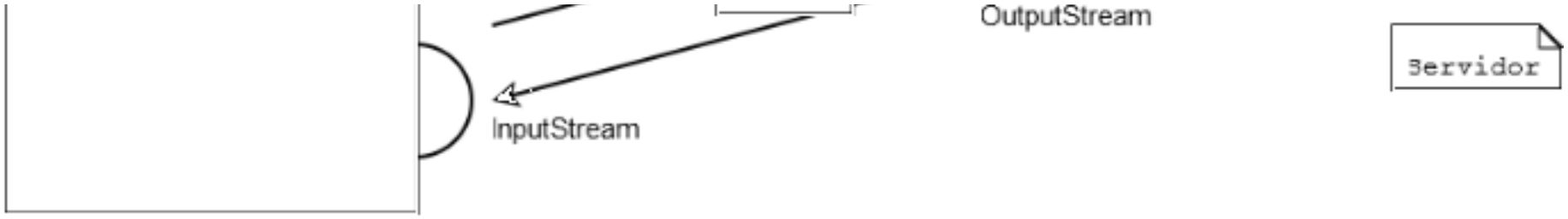
A thread criada será responsável pelo tratamento dessa conexão, enquanto o laço do servidor disponibilizará a porta para uma nova conexão:

```
while (true) {
```

```
    Socket cliente = servidor.accept();

    // cria um objeto que vai tratar a conexão
    TratamentoClass tratamento = new TratamentoClass(cliente);

    // cria a thread em cima deste objeto
    Thread t = new Thread(tratamento);

    // inicia a thread
    t.start();

  }
```

19.7 - IMAGEM GERAL





A socket do cliente tem um `InputStream`, que recebe do `OutputStream` do servidor, e tem um `OutputStream`, que transfere tudo para o `InputStream` do servidor. Muito parecido com um telefone!

Repare que cliente e servidor são rótulos que indicam um estado. Um micro (ou melhor, uma JVM) pode ser servidor num caso, mas pode ser cliente em outro caso.

19.8 - EXERCÍCIOS: SOCKETS

1. Crie um projeto `sockets`.

Vamos fazer um pequeno sistema em que tudo que é digitado no micro cliente

acaba aparecendo no micro servidor. Isto é, apenas uma comunicação unidirecional.

Crie a classe `Servidor` como vimos nesse capítulo. Abuse dos recursos do Eclipse para não ter de escrever muito!

```
1  package br.com.caelum.chat;
2
3  import java.io.IOException;
4  import java.net.ServerSocket;
5  import java.net.Socket;
6  import java.util.Scanner;
7
8  public class Servidor {
9    public static void main(String[] args) throws IOException {
10     ServerSocket servidor = new ServerSocket(12345);
11     System.out.println("Porta 12345 aberta!");
12
13     Socket cliente = servidor.accept();
14     System.out.println("Nova conexão com o cliente " +
15       cliente.getInetAddress().getHostAddress());
16
17     Scanner entrada = new Scanner(cliente.getInputStream());
```

```
18      while (entrada.hasNextLine()) {
19        System.out.println(entrada.nextLine());
20      }
21
22      entrada.close();
23      servidor.close();
24    }
25  }
```

2. Crie a classe `Cliente` como vista anteriormente:

```
1  package br.com.caelum.chat;
2
3  import java.io.IOException;
4  import java.io.PrintStream;
5  import java.net.Socket;
6  import java.net.UnknownHostException;
7  import java.util.Scanner;
8
9  public class Cliente {
10   public static void main(String[] args)
11           throws UnknownHostException, IOException {
12     Socket cliente = new Socket("127.0.0.1", 12345);
13     System.out.println("O cliente se conectou ao servidor!");
```

```
14
15      Scanner teclado = new Scanner(System.in);
16      PrintStream saida = new PrintStream(cliente.getOutputStream());
17
18      while (teclado.hasNextLine()) {
19        saida.println(teclado.nextLine());
20      }
21
22      saida.close();
23      teclado.close();
24    }
25  }
```

Utilize dos quick fixes e control espaço para os `imports` e o `throws`.

3. Rode a classe `Servidor`: repare no console do Eclipse que o programa fica esperando. Rode a classe `Cliente`: a conexão deve ser feita e o Eclipse deve mostrar os dois consoles para você (existe um pequeno ícone na view de Console para você alternar entre eles).

Digite mensagens no cliente e veja se elas aparecem corretamente no

servidor.

4. Teste seu programa com um colega do curso, usando comunicação remota entre as duas máquinas. Combinem entre si quem vai rodar o cliente e quem vai rodar o servidor. Quem for rodar o cliente deve editar o IP na classe para indicar o endereço da outra máquina (verifique também se estão acessando a mesma porta).

> **Descobrindo o ip da máquina**
>
> No Windows, abra o console e digite ipconfig para saber qual é o seu IP.
> No Linux (ou no BSD, Mac, Solaris), vá no console e digite ifconfig.

5. (opcional) E se você quisesse, em vez de enviar tudo o que o cliente digitou, transferir um arquivo texto do micro do cliente para servidor? Seria difícil?

Abuse do polimorfismo! Faça o cliente ler de um arquivo chamado `arquivo.txt` (crie-o!) e faça com que o servidor grave tudo que recebe num arquivo que chama `recebido.txt`.

19.9 - DESAFIO: MÚLTIPLOS CLIENTES

Quando o servidor aceita um cliente com a chamada ao `accept`, ele poderia chamar novamente este método para aceitar um novo cliente. E, se queremos aceitar vários clientes, simultâneos, basta chamar o `accept` várias vezes e tratar cada cliente em sua própria `Thread` (senão o método `accept` não será invocado novamente!).

Um esboço de solução para a classe `Servidor`:

```
ServerSocket servidor = new ServerSocket(12345);

// servidor fica eternamente aceitando clientes...
while (true) {
  Socket cliente = servidor.accept();
  // dispara uma Thread que trata esse cliente e já espera o próximo
}
```

19.10 - DESAFIO: BROADCAST DAS MENSAGENS

Agora que vários clientes podem mandar mensagens, gostaríamos que os clientes recebessem as mensagens enviadas pelas outras pessoas. Ao invés do servidor simplesmente escrever as mensagens no console, ele deve mandar cada mensagem para todos os clientes conectados.

Precisamos manter uma lista de clientes conectados e, quando chegar uma mensagem (de qualquer cliente), percorremos essa lista e mandamos para todos.

Use um `List` para guardar os `PrintStreams` dos clientes. Logo depois que o servidor aceitar um cliente novo, crie um `PrintStream` usando o `OutputStream` dele e adicione na lista. E, quando receber uma mensagem nova, envia para todos na lista.

Um esboço:

Adicionando na lista:

```
while (true) {
  Socket cliente = servidor.accept();
  this.lista.add(new PrintStream(cliente.getOutputStream()));

  // dispara uma Thread que trata esse cliente e já espera o próximo
}
```

Método que distribui as mensagens:

```
void distribuiMensagem(String msg) {
  for (PrintStream cliente : lista) {
    cliente.println(msg);
  }
}
```

Mas nosso cliente também recebe mensagens. Então precisamos fazer com que o Cliente, além de ler mensagens do teclado e enviar para o servidor, simultaneamente também possa receber mensagens de outros clientes enviadas pelo servidor.

Ou seja, precisamos de uma segunda Thread na classe `Cliente` que fica

recebendo mensagens do `InputStream` do servidor e imprimindo no console.

Um esboço:

```
Scanner servidor = new Scanner(cliente.getInputStream());
while (servidor.hasNextLine()) {
  System.out.println(servidor.nextLine());
}
```

Lembre que você precisará de no mínimo 2 threads para o cliente e 2 para o servidor. Então provavelmente você vai ter que escrever 4 classes.

Melhorias possíveis:

- Faça com o a primeira linha enviada pelo cliente seja sempre o nick dele. E quando o servidor enviar a mensagem, faça ele enviar o nick de cada cliente antes da mensagem.

- E quando um cliente desconectar? Como retirá-lo da lista?

- É difícil fazer o envio de arquivos pelo nosso sistema de chats? Sabendo que a leitura de um arquivo é feita pelo `FileInputStream`, seria difícil mandar esse `InputStream` pelo `OutputStream` da conexão de rede?

19.11 - SOLUÇÃO DO SISTEMA DE CHAT

Uma solução para o sistema de chat cliente-servidor com múltiplos clientes proposto nos desafios acima. Repare que a solução não está nem um pouco elegante: o `main` já faz tudo, além de não tratarmos as exceptions. O código visa apenas mostrar o uso de uma API. É uma péssima prática colocar toda a funcionalidade do seu programa no `main` e também de jogar exceções para trás.

Nesta listagem, faltam os devidos **imports**.

Primeiro, as duas classes para o cliente. Repare que a única mudança grande é a classe nova, Recebedor:

```java
public class Cliente {
  public static void main(String[] args)
        throws UnknownHostException, IOException {
    // dispara cliente
    new Cliente("127.0.0.1", 12345).executa();
  }

  private String host;
  private int porta;

  public Cliente (String host, int porta) {
    this.host = host;
    this.porta = porta;
  }

  public void executa() throws UnknownHostException, IOException {
    Socket cliente = new Socket(this.host, this.porta);
    System.out.println("O cliente se conectou ao servidor!");

    // thread para receber mensagens do servidor
    Recebedor r = new Recebedor(cliente.getInputStream());
    new Thread(r).start();

    // lê msgs do teclado e manda pro servidor
```

```
25      Scanner teclado = new Scanner(System.in);
26      PrintStream saida = new PrintStream(cliente.getOutputStream());
27      while (teclado.hasNextLine()) {
28        saida.println(teclado.nextLine());
29      }
30
31      saida.close();
32      teclado.close();
33      cliente.close();
34    }
35  }
```

```
1   public class Recebedor implements Runnable {
2
3     private InputStream servidor;
4
5     public Recebedor(InputStream servidor) {
6       this.servidor = servidor;
7     }
8
9     public void run() {
10      // recebe msgs do servidor e imprime na tela
11      Scanner s = new Scanner(this.servidor);
12      while (s.hasNextLine()) {
```

```
13          System.out.println(s.nextLine());
14        }
15      }
16  }
```

Já o Servidor sofreu bastante modificações. A classe `TrataCliente` é a responsável por cuidar de cada cliente conectado no sistema:

```
1  public class Servidor {
2
3    public static void main(String[] args) throws IOException {
4      // inicia o servidor
5      new Servidor(12345).executa();
6    }
7
8    private int porta;
9    private List<PrintStream> clientes;
10
11   public Servidor (int porta) {
12     this.porta = porta;
13     this.clientes = new ArrayList<PrintStream>();
14   }
15
```

```
16    public void executa () throws IOException {
17      ServerSocket servidor = new ServerSocket(this.porta);
18      System.out.println("Porta 12345 aberta!");
19
20      while (true) {
21        // aceita um cliente
22        Socket cliente = servidor.accept();
23        System.out.println("Nova conexão com o cliente " +
24          cliente.getInetAddress().getHostAddress()
25        );
26
27        // adiciona saida do cliente à lista
28        PrintStream ps = new PrintStream(cliente.getOutputStream());
29        this.clientes.add(ps);
30
31        // cria tratador de cliente numa nova thread
32        TrataCliente tc =
33            new TrataCliente(cliente.getInputStream(), this);
34        new Thread(tc).start();
35      }
36
37    }
38
39    public void distribuiMensagem(String msg) {
```

```
40      // envia msg para todo mundo
41      for (PrintStream cliente : this.clientes) {
42        cliente.println(msg);
43      }
44    }
45  }
```

```
1   public class TrataCliente implements Runnable {
2
3     private InputStream cliente;
4     private Servidor servidor;
5
6     public TrataCliente(InputStream cliente, Servidor servidor) {
7       this.cliente = cliente;
8       this.servidor = servidor;
9     }
10
11    public void run() {
12      // quando chegar uma msg, distribui pra todos
13      Scanner s = new Scanner(this.cliente);
14      while (s.hasNextLine()) {
15        servidor.distribuiMensagem(s.nextLine());
16      }
17      s.close();
```

```
18     }
19  }
```

Você encontra a Caelum também em: