Linguagem Java:

Threads

Rômulo Silva de Oliveira
Instituto de Informática - UFRGS

romulo@inf.ufrgs.br
http://www.inf.ufrgs.br/~romulo

Threads

- Java suporta programação multithread como parte da linguagem
- Maioria das linguagens
  - Não suporta
  - Prove uma biblioteca a parte
- **Multitasking**
  - Capacidade de executar vários programas simultaneamente
- **Multithreading**
  - Capacidade de realizar vários fluxos de execução simultaneamente
  - Dentro do mesmo programa

Threads

- Exemplo de programa Java com uma única thread

```
class MainIsRunInAThread {
    public static void main(String[] args) {
          System.out.println(Thread.currentThread());
          for (int i=0; i<1000; i++) {
            System.out.println("i == " + i);
        }
    }
}
```

- Quando uma aplicação Java inicia
  - A máquina virtual (VM) executa o método main()
  - Dentro de uma thread

Threads

- Threads permitem escrever programas que
  - fazem várias coisas ao mesmo tempo
- Cada thread representa uma sequencia de controle independente
- Por exemplo,
  - Uma thread pode escrever um arquivo em disco
  - Uma thread diferente responde ao acionamento de teclas do usuário
- Outro exemplo, ao mesmo tempo
  - Obter uma imagem através da rede e
  - Solicitar uma informação atualizada de preço de ação
  - Rodar diversas animações

Threads - Exemplo

```
class CountThreadTest extends Thread {
    int from, to;
    public CountThreadTest(int from, int to) {
        this.from = from;
        this.to = to;
    }

    // the run() method is like main() for a thread
    public void run() {
        for (int i=from; i<to; i++) {
            System.out.println("i == " + i);
        }
    }
```


Threads - Exemplo

```
public static void main(String[] args) {
// spawn 5 threads, each counts 200 numbers

        for (int i=0; i<5; i++) {
            CountThreadTest t = new
                    CountThreadTest(i*200, (i+1)*200);

// starting a thread will launch a separate sequence of
// control and execute the run() method of the thread
            t.start();
        }
    }

}
```

Criando Novas Threads

- Criar uma nova thread significa
  - Escrever o código que será executado pela thread
  - Escrever o código que dispara a thread
- A execução da thread inicia pelo método run()

```
public void run( );
```

- A implementação do método run() pode ser feita de duas formas:
- Criando uma subclasse da classe java.lang.Thread
- Criando um método que implementa a interface Runnable

Subclasse da Classe Thread

- Aplicação copia arquivo de um diretório para outro
- Arquivo grande tranca a aplicação
  Não responde a eventos
- Solução: copiar o arquivo com uma thread separada
- Cria uma subclasse de Thread
  com a lógica necessária no método run()
- Exemplo:
  - Classe FileCopyThread

Disparando a Execução da Thread

- Para usar a Thread é necessário iniciar sua execução
- Feito através do seu método start( )
- Exemplo:

```
File from = getCopyFrom();
File to = getCopyTo();

// create an instance of the thread class
Thread t = new FileCopyThread(from, to);

// call start() to activate the thread asynchronously
t.start();
```


Implementando a Interface Runnable

- Existem situações quando
  NÃO é conveniente criar uma subclasse de Thread
- Por exemplo, tornar executável uma classe que não herda de Thread
- Solução possível com a interface java.lang.Runnable
- Interface possui um único método

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

- Reimplementar FileCopyThread com a interface Runnable é fácil

```
class FileCopyRunnable implements Runnable {
    // the rest of the class remains mostly the same
    ...
}
```

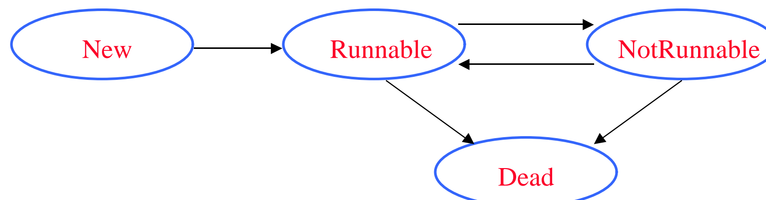
Disparando a Execução da Thread

- Para usar a interface Runnable
  - Necessária a cooperação de uma Thread
- Exemplo de como iniciar uma thread com FileCopyRunnable:

```
File from = new File("file.1");
File to = new File("file.2");
// create an instance of the Runnable
Runnable r = new FileCopyRunnable(from, to);
// create an instance of Thread, passing the Runnable
Thread t = new Thread(r);
// start the thread
t.start();
```


Estados das Threads

- New para Runnable: Chamada do método start
- Dead: Método run() termina, método stop() or destroy() é chamado
- Runnable para Notrunnable
  - Thread esperando por operação de e/s
  - Thread colocada para dormir pelo método sleep()
  - Método wait() foi chamado
  - Thread foi suspensa pelo método suspend()



Construtores da Classe Thread

- public Thread();
- public Thread(Runnable target);
- public Thread(Runnable target, String name);
- public Thread(String name);
- public Thread(ThreadGroup group, Runnable target);
- public Thread(ThreadGroup group, Runnable target, String name);
- public Thread(ThreadGroup group, String name);

- name é o nome String da Thread, podendo ser Thread-N
- target é a instância Runnable cujo método run() será executado
- group é o ThreadGroup ao qual a Thread será adicionada

Nomes das Threads

- Toda thread tem um nome definido pelo construtor
- Pode acessar com os métodos

- public final String getName();
- public final void setName(String name);

- Nomes são importantes
- Permitem ao programador identificar threads durante depuração

Start( ) e Stop( )

- Para iniciar e terminar Threads depois de criadas:

- public void start();
  - Inicia a execução da thread
- public final void stop();
  - Faz a thread terminar através do lançamento de uma exceção ThreadDeath
- public final void stop(Throwable obj);
  - Idem, outro tipo de exceção
- public void destroy();
  - Similar, mas sem o ThreadDeath, sem limpeza

- Exemplo: DyingThread

Escalonamento e Prioridades

- Thread no estado RUNNABLE, prioridade mais alta é executada
- Prioridade mais alta tira o processador da prioridade mais baixa
- Outros detalhes dependem da plataforma
  - Threads com prioridades iguais ?
- Thread.yield() passa processador para outra Thread, mesma prioridade
- Na criação, thread herda prioridade da thread criadora

- public final static int MAX_PRIORITY = 10;
- public final static int MIN_PRIORITY = 1;
- public final static int NORM_PRIORITY = 5;
- public final int getPriority();
- public final void setPriority(int newPriority);

Interrupção de uma Thread

- Uma thread pode interromper outra lançando uma exceção InterruptedException, também liga uma flag
- `public void interrupt();`
- `public static boolean interrupted();`
  - Testa e Desliga a flag
- `public boolean isInterrupted();`
  - Testa e Não desliga a flag
- É possível suspender temporariamente uma thread
- `public final void suspend();`
- `public final void resume();`
- É possível suspender a thread por determinado tempo
- `public static void sleep(long millisecond);`

Esperando uma Thread Acabar

- Método join é usado para esperar a Thread acabar
- public final void join();
- public final void join(long millisecond);
- public final void join(long millisecond, int nanosecond);

```
Thread t = new OperationINeedDoneThread();
t.start();
.... // do some other stuff
t.join(); // wait for the thread to complete
```

- Quando pode sair por timeout,
  - deve usar o método isAlive() para determinar motivo

Threads Daemon

- Algumas threads são do tipo "background" ou "daemon"
- Elas apenas fornecem algum serviço para outras threads
- Quando apenas threads daemon permanecem vivas,
  o programa termina
- Ao menos uma thread daemon existe: o coletor de lixo
- Coletor de lixo possui a prioridade mais baixa de todas as threads

```
public final boolean isDaemon();
public final void setDaemon(boolean on);
```


Outros Métodos

```
public int countStackFrames();
```

- Retorna o número de ativações de métodos na pilha desta thread
- Thread deve estar suspensa

```
public final boolean isAlive();
```

- Retorna true se start() desta thread já foi chamado e a Thread não morreu ainda

```
public static Thread currentThread();
```

- Retorna o objeto Thread associado com este fluxo de execução

Outros Métodos

```
public static void dumpStack();
```

- Usado para depuração
- Envia para System.err uma lista método por método da pilha desta Thread

```
public String toString();
```

- Retorna um String para depuração que descreve a Thread

```
public static int activeCount();
```

- Retorna o número de Threads no ThreadGroup da Thread em questão

```
public static int enumerate(Thread tarray[]);
```

- Retorna uma lista de todas as Threads no ThreadGroup desta Thread

Grupos de Threads

- Cada Thread pertence exatamente a uma instância de ThreadGroup
- Classe ThreadGroup ajuda na gerência de grupos de Threads similares
- Por exemplo
  - Usado por Web browsers para agrupar threads de um applet

- Objetos ThreadGroup formam uma estrutura tipo árvore
- Grupos podem conter tanto Threads como outros grupos
- O grupo de threads raiz é chamado "system"
  - Contém Threads do sistema
  - Contém o ThreadGroup "main"
- O grupo de Threads "main" contém a Thread que executa main( )

Grupos de Threads

- Classe ThreadGroup possui dois construtores
- Se o grupo pai não é indicado, usa o grupo da Thread executando
- Inicialmente um ThreadGroup não possui nenhum elemento

```
public ThreadGroup(String name);
public ThreadGroup(ThreadGroup parent, String name);
```

```
public final ThreadGroup getThreadGroup();
```

- Retorna o ThreadGroup ao qual esta Thread pertence
- Cada Thread pertence a exatamente um ThreadGroup

Grupos de Threads

- Métodos atuam sobre todos os elementos do grupo, recursivo

```
public final void suspend();
public final void resume();
public final void stop();
public final void destroy();
```

- Controlam a prioridade máxima dentro do grupo

```
public final int getMaxPriority();
public final void setMaxPriority(int pri);
```

Grupos de Threads - Navegação

```
public int activeCount();
public int activeGroupCount();
```

- Número de Threads e ThreadsGroups na árvore, recursivo

```
public int enumerate(Thread list[]);
public int enumerate(Thread list[],
                        boolean recurse);
public int enumerate(ThreadGroup list[]);
public int enumerate(ThreadGroup list[],
                        boolean recurse);
```

- Lista os elementos do grupo

Grupos de Threads

```
public final boolean parentOf(ThreadGroup g);
```

- Testa se o ThreadGroup em questão é o pai de g

```
public final ThreadGroup getParent();
```

- Retorna o ThreadGroup pai do ThreadGroup em questão
- Retorna null se o ThreadGroup for a raiz da árvore

```
public void list();
```

- Coloca informações para depuração em System.out

Grupos de Threads - Outros Métodos

```
public final String getName();

public final boolean isDaemon();

public final void setDaemon(boolean daemon);

public String toString();

public void uncaughtException(Thread t,
                              Throwable e);
```


Concorrência entre Threads

- Java admite múltiplas Threads simultâneas
- Por exemplo, simultaneamente
  - Fazer o download de um arquivo da Internet
  - Recalcular de uma planilha
  - Imprimir um documento
- Concorrência exige cuidados especiais

```
public class Counter {
   private int count = 0;
   public int incr ( ) {
      int n = count;
      count = n + 1;
      return n;
   }
}
```

Concorrência entre Threads

Thread 1	Thread 2	Count
cnt = counter.incr();	---	0
n = count; // 0	---	0
count = n + 1; // 1	---	1
return n; // 0	---	1
---	cnt = counter.incr();	1
---	n = count; // 1	1
---	count = n + 1; // 2	2
---	return n; // 1	2

Concorrência entre Threads

Thread 1	Thread 2	Count
cnt = counter.incr();	---	0
n = count; // 0	---	0
---	cnt = counter.incr();	0
---	n = count; // 0	0
---	count = n + 1; // 1	1
---	return n; // 0	1
count = n + 1; // 1	---	1
return n; // 0	---	1

Métodos Sincronizados

- Conceito de monitor usado para impor
  acesso mutuamente exclusivo aos métodos
- Métodos são identificados pelo modificador "synchronized"
- Quando um método sincronizado é chamado
  - Monitor é consultado
  - Se nenhum outro método sincronizado estiver em execução, continua
  - Se método sincronizado em execução, chamada tem que aguardar
- Cada monitor é associado com um OBJETO
  e não com um trecho de código
- Métodos sincronizados de diferentes objetos da mesma classe
  PODEM ser executados ao mesmo tempo

Métodos Sincronizados

```
public class Counter2 {
    private int count = 0;
    public synchronized int incr() {
        int n = count;
        count = n + 1;
        return n;
    }
}
```

Métodos Sincronizados

Thread 1	Thread 2	Count
cnt = counter.incr();	---	0
(acquires the monitor)	---	0
n = count; // 0	---	0
---	cnt = counter.incr();	0
---	(can't acquire monitor)	0
count = n + 1; // 1	(blocked)	1
return n; // 0	(blocked)	1
(releases the monitor)	(acquires the monitor)	1
---	n = count; // 1	1
---	count = n + 1; // 2	2
---	return n; // 1	2
---	(releases the monitor)	2

Métodos Sincronizados

- Métodos estáticos podem ser sincronizados
  - Somente um da classe executa a cada momento
- Expressões podem ser sincronizados
  - Acesso exclusivo ao objeto que resulta da expressão

```
void safe_lshift(byte[] array, int count) {
    synchronized(array) {
        System.arraycopy(array, count, array, 0,
  array.size - count);
    }
}
void call_method(SomeClass obj) {
    synchronized(obj) {
        obj.variable = 5;
    }
}
```

Deadlock

- Situação onde duas ou mais Threads estão bloqueadas,
  uma esperando a outra
- Máquina virtual de Java não realiza nenhum algoritmo para
  Detectar ou
  Notificar deadlocks

- Exemplo: DEADLOCK

Condições

- Uma condição é uma expressão lógica que:
  - Deve ser verdadeira para a Thread prosseguir
  - Quando a condição é falsa a Thread deve esperar

```
while ( ! the_condition_I_am_waiting_for ) {
    wait();
}
```

- O método wait() pausa a Thread corrente e a coloca na fila de wait
- O método notify() acorda uma única Thread da fila
- O método notifyAll() acorda todas as Threads da fila

```
wait(long milliseconds);
wait(long milliseconds, int nanoseconds);
```

- Exemplo: BUFFER, BUFFER2

Condições

- Todos os OBJETOS Java podem participar da sincronização com wait() e notify()
- No caso de arrays, é necessário uma expressão sincronizada

```
// wait for an event on this array
Object[] array = getArray();
synchronized (array) {
array.wait();
}
...
// notify waiting threads
Object[] array = getArray();
synchronized (array) {
array.notify();
}
```


Linguagem Java:

Programando com Sockets

Rômulo Silva de Oliveira
Instituto de Informática - UFRGS

romulo@inf.ufrgs.br
http://www.inf.ufrgs.br/~romulo

Sockets - Introdução

- SOCKET
  - É uma abstração
  - Representa um ponto de conexão em uma rede TCP/IP
- Para dois computadores trocarem informações
  - Cada um utiliza um socket
- Um computador é o SERVIDOR
  - Abre o socket e escuta a espera de conexões
- O outro computador é o CLIENTE
  - Chama o socket do servidor para iniciar a conexão

Sockets - Introdução

- Cada computador em uma rede TCP/IP possui endereço único
  - Endereço IP
- Portas representam conexões individuais com este endereço
- BINDING: Quando o socket é criado,
  deve ser associado com uma porta específica
- Para estabelecer uma conexão o cliente precisa conhecer
  - Endereço IP da máquina onde o servidor executa
  - Número da porta que o servidor escuta
- Alguns números de portas são bem conhecidos:

7-echo	13-daytime	21-ftp	23-telnet
25-smtp	79-finger	80-http	110-pop3

- Número de portas são definidos pela IANA
  - Internet Assigned Numbers Authority

Sockets - Introdução

- Existem dois modos de operação
  - Orientado a conexão, TCP - Transport Control Protocol
  - Sem conexão, UDP - User Datagram Protocol
- **Sockets orientados a conexão**
  - Conexão deve ser estabelecida antes do envio dos dados
  - Conexão deve ser terminada ao final da comunicação
  - Os dados chegam na mesma ordem que foram enviados
- **Sockets sem conexão**
  - Entrega não é garantida
  - Dados podem chegar em ordem diferente da que foram enviados
- Modo usado depende das necessidades da aplicação
- Conexão implica em
  - Maior Confiabilidade
  - Maior Overhead

Sockets - Cliente Orientado a Conexão

- Procedimento diferente para Clientes e Servidores
- Socket cliente criado e conectado pelo construtor da classe

```
Socket clientSocket = new Socket("merlin", 80);
```

- Acesso através de streams, classes do pacote java.io

```
DataOutputStream outbound = new DataOutputStream(
                    clientSocket.getOutputStream() );
BufferedReader inbound = new BufferedReader( new
   InputStreamReader(clientSocket.getInputStream()) );
```

- Conexão deve ser fechada ao final
  - Streams são fechados antes

Sockets - Cliente Orientado a Conexão

- Clientes seguem a mesma receita
  - Cria um socket com conexão cliente
  - Associa streams de leitura e escrita com o socket
  - Utiliza os streams conforme o protocolo do servidor
  - Fecha os streams
  - Fecha o socket
- Existem opções para sockets clientes

- SO_LINGER
  - Time-out usado para acks associados com a desconexão
  - Permite forçar uma desconexão abortiva, dados podem ser perdidos

```
public int getSoLinger()
public void setSoLinger( boolean on, int val)
```


Sockets - Cliente Orientado a Conexão

- SO_TIMEOUT
  - Leitura de um socket conectado bloqueia até que
    - Dados sejam recebidos ou Socket seja fechado
  - Opção estabelece um time-out para este bloqueio, -1 é default
    - Neste caso retorna InterruptedIOException

```
public synchronized int getSoTimeout()
public synchronized void setSoTimeout(int timeout)
```

- TCP_NODELAY
  - Controla a aplicação do algoritmo de Nagle
  - Pacotes são enviados somente quando
    todos os anteriores foram confirmados ou buffer atinge certo tamanho
  - Opção ligada permite o envio imediato dos pacotes

```
public boolean getTcpNoDelay()
public void setTcpNoDelay(boolean on)
```

Sockets - Servidor Orientado a Conexão

- Servidores não criam conexões
- Servidores esperam por um pedido de conexão de um cliente
- Construtor de ServerSocket é usado

```
ServerSocket serverSocket = new ServerSocket( 80, 5);
```

- Primeiro parâmetro é o número da porta a ser escutada
- Segundo parâmetro é o tamanho da pilha de escuta
  - Servidor pode receber várias requisições de conexão ao mesmo tempo
  - As requisições são processadas uma a uma
  - A fila de requisições ainda não atendidas é chamada de Pilha de Escuta
  - Parâmetro informa quantas requisições manter
  - Por exemplo, apenas as últimas 5 requisições devem ser mantidas
  - Valor default é 50

Sockets - Servidor Orientado a Conexão

- Método accept() é chamado para retirar requisições da fila
  - Bloqueia até chegar uma requisição

```
Socket client = serverSocket.accept();
```

- Método retorna um socket conectado com o cliente
- Socket do servidor não é usado, um novo é criado para os dados
- Socket do servidor continua enfileirando pedidos de conexão
- Finalmente, streams são criados:

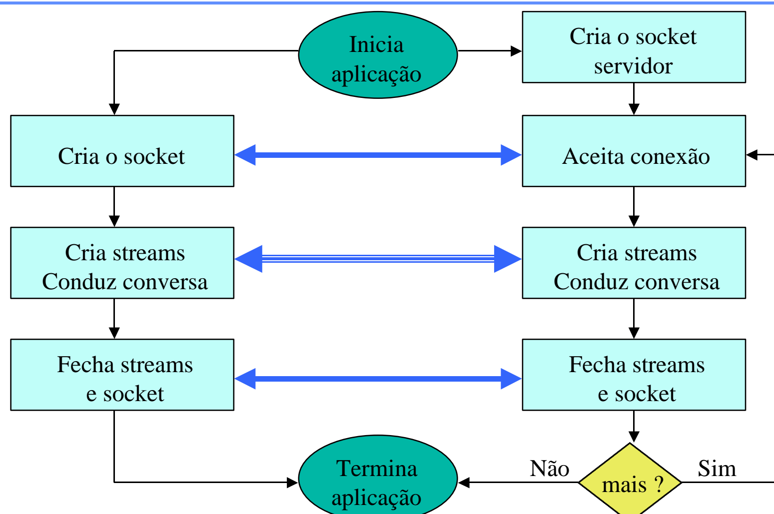
```
BufferedReader inbound = new BufferedReader( new
   InputStreamReader(client.getInputStream()) );
OutputStream outbound = client.getOutputStream() );
```

- Após o final da sessão, `accept()` é usado para nova conexão
- Ao fechar “serverSocket”, requisições enfileiradas são canceladas

Sockets - Servidor Orientado a Conexão

- Esquema básico para o servidor:
  - Cria um socket servidor e inicia a escutar
  - Chama método accept() para pegar novas conexões
  - Cria streams de entrada e saída para o socket da conexão
  - Conduz a troca de dados conforme o protocolo em questão
  - Fecha os streams e o socket da conexão
  - Repete várias vezes
  - Fecha o socket servidor
- Servidor apresentado processa uma conexão de cada vez
- Servidor concorrente
  - Cria uma nova thread para cada conexão aberta
  - Consegue processar várias conexões simultaneamente
  - Exemplo, servidores Web comerciais são todos concorrentes

Sockets - Orientado a Conexão



Sockets - Datagramas

- Mesma classe é usada por cliente e servidor
  - **Classe DatagramSocket**

```
DatagramSocket serverSocket = new DatagramSocket(4545);
DatagramSocket clientSocket = new DatagramSocket( );
```

- Servidor deve especificar sua porta
- Omissão do parâmetro significa “use a próxima porta livre”

- **Classe DatagramPacket** é usada para enviar e receber dados
  - Contém informações para conexão e os próprios dados
- Para receber dados de um socket datagrama
  - Método receive bloqueia até a recepção dos dados

```
DatagramPacket packet =
             new DatagramPacket(new byte[512],512);
clientSocket.receive(packet);
```


Sockets - Datagramas

- Para enviar dados para um socket datagrama
  - Necessário um endereço, **classe InetAddress**
- Não possui construtor, instanciada por métodos estáticos

```
InetAddress getByName( String host);
InetAddress[] getAllByName( String host);
InetAddress[] getLocalHost();
```

- Exemplo:

```
InetAddress addr=InetAddress.getByName(“inf.ufrgs.br”);
```

- Podem lançar uma exceção UnknownHostException
  - Computador não está conectado a um Domain Name Server
  - Host não foi encontrado

Sockets - Datagramas

- Envio de um string para um socket destinatário

```
String toSend = "This is the data to send!";
byte sendbuf[] = toSend.getBytes();
DatagramPacket sendPacket = new DatagramPacket(
  sendbuf, sendbuf.length, addr, port);
clientSocket.send( sendPacket);
```

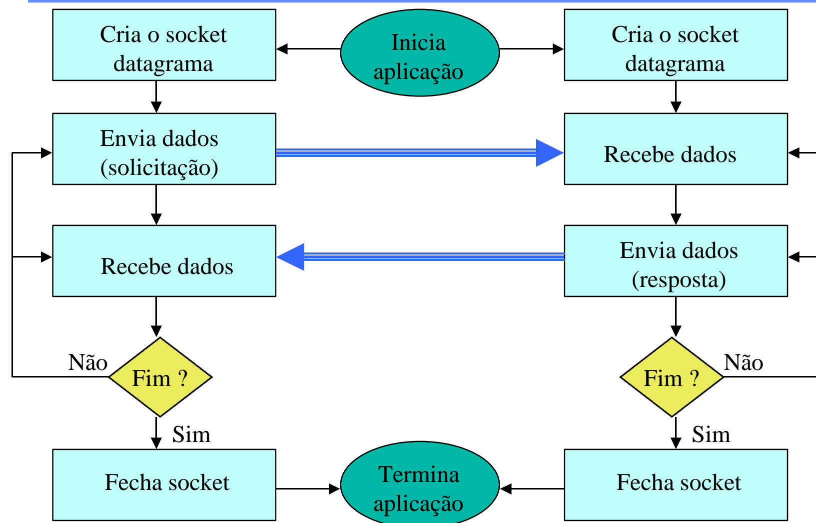
- String deve ser convertido para array de bytes
- Endereço destino incluído no DatagramPacket
- Como servidor descobre quem enviou o pacote ?

```
pacoteRecebido.getAddress();
pacoteRecebido.getPort();
```


Sockets - Datagramas

- Algoritmo do Servidor
  - Cria um socket datagrama em uma porta específica
  - Chama receive() para esperar por pacotes
  - Responde ao pacote recebido conforme protocolo em questão
  - Repete várias vezes
  - Fecha o socket
- Algoritmo do Cliente
  - Cria um socket datagrama em qualquer porta livre
  - Cria um endereço destinatário
  - Envia os dados conforme o protocolo em questão
  - Espera por dados com a resposta
  - Repete várias vezes
  - Fecha o socket datagrama

Sockets - Datagramas



Sockets - Multicast

- Endereço multicast:
  - Endereço classe D no intervalo entre 224.0.0.1 e 239.255.255.255
- Classe Multicast descende de Datagram
  - Capaz de transmitir para todos os hosts escutando, simultaneamente
  - O próprio socket também recebe tudo que envia
- Normalmente funciona somente em uma rede local
  - Maioria dos roteadores não propaga pacotes multicast para Internet
- Baseado no IGMP - Internet Group Management Protocol
- Para receber transmissões multicast, host deve ingressar em um Grupo Multicast

```
public void joinGroup(InetAddress mcastaddr);
public void leaveGroup(InetAddress mcastaddr);
public void setTTL(byte ttl);
```

Sockets - Servidor HTTP Exemplo

- Exemplo: Um servidor HTTP
- Suporta um subconjunto do HTTP versão 1.0
  - Suporta apenas requisições de arquivos
- HTTP usa conexão, TCP, tipicamente porta 80
- Todo o protocolo funciona com formato texto simples
- Requisição do cliente: GET FILE HTTP/1.0
- Primeira palavra é o "método" da requisição
  - GET: obtém um arquivo
  - HEAD: obtém apenas informações sobre o arquivo
  - POST: Envia dados para o servidor
  - PUT: Envia dados para o servidor
  - DELETE: Apaga um recurso
  - LINK: Vincula dois recursos
  - UNLINK: Desvincula dois recursos

Sockets - Servidor HTTP Exemplo

- Segundo parâmetro é o nome do arquivo
- http://www.qnet.com/
  `GET / HTTP/1.0`
- http://www.qnet.com/index.html
  `GET /index.html HTTP/1.0`
- http://www.qnet.com/classes/applet.html
  `GET /classes/applet.html HTTP/1.0`
- Requisição termina com uma linha branca `"\r\n"`
- Linhas adicionais podem seguir, exemplo:

```
GET / HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/2.0 (Win95; I)
Host: merlin
Accept: image/gif, image/x-xbitmap, image/jpeg, */*
```

Sockets - Servidor HTTP Exemplo

- Resposta usa um cabeçalho semelhante, termina com linha em branco

```
HTTP/1.0 200 OK
Content-type: text/html
Content-Length: 128
```

- Texto após o código de status é opcional
- Após o cabeçalho, o arquivo pedido é enviado
- Quando o arquivo foi completamente transmitido, conexão é fechada
- Cada par de solicitação-resposta necessita uma nova conexão
- Mais informações sobre HTTP em http://www.w3.org

Sockets - Servidor HTTP Exemplo

- Códigos de status para HTTP 1.0
  - Texto opcional pode ser omitido ou modificado

- 200 OK 201 Created
- 202 Accepted 204 No Content
- 300 Multiple Choices 301 Moved Permanently
- 302 Moved Temporarily 304 Not Modified
- 400 Bad Request 401 Unauthorized
- 403 Forbidden 404 Not Found
- 500 Internal Server Error 501 Not Implemented
- 502 Bad Gateway 503 Service Unavailable

Sockets - Servidor HTTP Exemplo

- HttpServer
  - Objeto principal, algoritmo geral do servidor
- HttpServerSocket
  - Retorna HttpSocket no lugar de Socket
- HttpSocket
  - Análise da solicitação, geração da resposta, conhece os formatos
- NameValue
  - Armazena pares de String, usado para os cabeçalhos