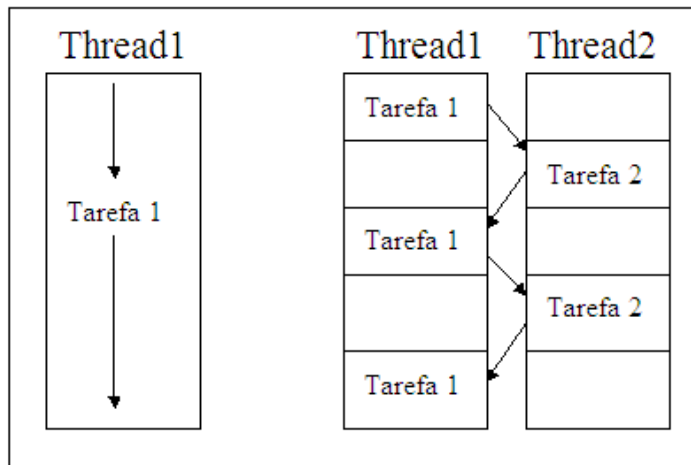


16. Threads:

16.1. Conceituando Threads:

Uma das principais características da linguagem Java é a sua capacidade de executar múltiplas *threads*, mas, o que são *threads*? Em computação *threads* podem ser definidas como uma seqüência de linhas de execução que executam tarefas específicas.

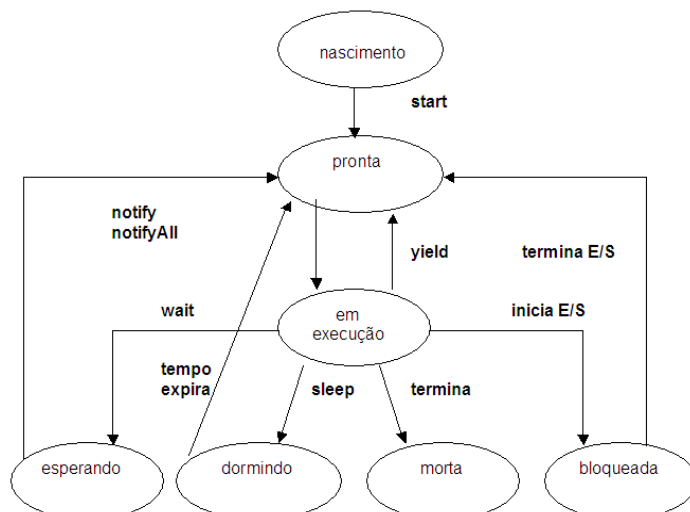


O principal propósito de se utilizar múltiplas *threads* é explorar o paralelismo nas aplicações. Desta forma é atingida mais eficazmente a capacidade de processamento das CPUs, uma vez que enquanto uma *thread* fica aguardando algum evento outra *thread* poderá ser executada. Por exemplo: uma aplicação que utiliza recursos gráficos poderia disparar uma *thread* para apresentar uma imagem e outra *thread* para ler alguma informação

de algum arquivo em disco. A figura a seguir representa duas aplicações: uma que utiliza *threads* e a outra que não utiliza.

As aplicações com múltiplas *threads* têm a capacidade de aproveitar o tempo ocioso existente entre as pausas naturais ocorridas na execução de uma tarefa.

16.2. Ciclo de Vida de uma Thread:



A figura ao lado apresenta os ciclos de vida de uma *thread*. Estes ciclos são representados pelos seguintes estados:

- **Nascimento** – compreende o processo de criação da *thread*.
- **Pronta** – a *thread* já foi criada e está esperando ser despachada para o processamento.
- **Em Execução** – a *thread* está sendo executada pela

CPU.

- **Bloqueada** – a *thread* requisitou alguma operação ao sistema operacional e fica aguardando ser atendida.
- **Dormindo** – a *thread* colocou-se voluntariamente para dormir por uma determinada quantidade de tempo.
- **Esperando** – a *thread* está esperando que algum recurso por ela compartilhado com outra *thread* seja liberado.
- **Morta** – a *thread* encerrou a sua tarefa e liberará todos os recursos alocados por ela.

Quem trata dos despachos das *threads* é o **Thread Scheduler** integrante da Java Virtual Machine.

É importante observar que as *threads* residem em processos e que os processos são despachados e tratados pelo sistema operacional.

Quando a JVM chama o método *main()* ele cria uma *thread* principal e a coloca em uma **pilha** específica, todos os métodos acionados no método *main()* são colocados nesta **pilha**. As *threads* criadas voluntariamente por nós desenvolvedores são colocadas em uma outra pilha, denominada de **pilha do usuário**. Cabe a JVM selecionar quais *threads* despachar das diferentes **pilhas**.

16.3. Criando Threads:

Para criarmos uma *thread* deve ser utilizada a classe **Thread**. Esta classe implementa a interface **Runnable** que define um único método abstrato denominado **run()**, sendo assim, todo aplicativo que deseja utilizar *threads* precisa implementar este método.

O código abaixo apresenta um caso simples de utilização de *threads*:

```
public class MinhaThread implements Runnable{

    public void run(){
        vai();
    }

    public void vai(){
        imprimeAlgo();
    }

    public void imprimeAlgo(){
        System.out.println("Thread Disparada !!");
    }
}
```

```
public static void main(String[] args){  
  
    Runnable disparaThread = new MinhaThread();  
    Thread  testaThread = new Thread(disparaThread);  
    testaThread.start();  
    System.out.println("Disparou !!");  
}  
}
```

A instrução `Runnable disparaThread = new MinhaThread();` identifica qual o objeto que deverá ser executado como uma nova *thread* em nosso exemplo será o objeto **disparaThread**.

A instrução `Thread testaThread = new Thread(disparaThread);` passa para o objeto **Thread** o que será executado. Podemos dizer que o objeto **Thread** é o trabalhador e o objeto **Runnable** identifica o serviço a ser executado por este trabalhador.

A instrução `testaThread.start();` faz com que a *thread* seja colocada no estado de pronta onde aguardará a JVM para ser despachada para execução.

Os métodos `run()`, `vai()` e `imprimeAlgo()` são colocados na pilha de *threads* dos usuários.

Uma outra forma de atingirmos o mesmo objetivo seria com o código abaixo:

```
public class MinhaThread extends Thread{  
  
    public void run(){  
        vai();  
    }  
  
    public void vai(){  
        imprimeAlgo();  
    }  
  
    public void imprimeAlgo(){  
        System.out.println("Thread Disparada !!");  
    }  
  
    public static void main(String[] args){  
  
        MinhaThread  testaThread = new MinhaThread();  
        testaThread.start();  
        System.out.println("Disparou !!");  
    }  
}
```

Neste código a classe **MinhaThread** herda da classe **Thread** logo, quando

você instancia esta classe o objeto **Thread** já está pronto para ser ativado, bastando, para isto, acionar o método **start()**. O grande inconveniente em se utilizar esta forma de codificação é que a linguagem Java não suporta herança múltipla, assim, se estamos herdando da classe **Thread** não poderemos herdar de mais nenhuma outra classe.

16.4. Colocando Threads para Dormir:

Para garantir que outras *threads* tenham a oportunidade de serem processadas ou para garantir o sincronismo de eventos você pode colocar uma *thread* para dormir por uma determinada quantidade de tempo, para tal é disponibilizado o método **sleep()**.

Este método recebe como argumento um número inteiro especificando o tempo, em milisegundos, que a *thread* ficará dormindo. Este método deverá estar dentro de uma estrutura de exceção *try/catch*.

No código a seguir a classe **MinhaThread** possui as variáveis de instância **palavra** e **delay** que representam, respectivamente, a palavra que será apresentada na console e o tempo que a *thread* ficará dormindo.

```
public class MinhaThread implements Runnable {

    private String palavra;
    private int delay;

    public MinhaThread(String oQueFalar, int tempoDelay){
        palavra=oQueFalar;
        delay=tempoDelay;
    }

    public void run() {
        try {
            for (;;) {
                System.out.print(palavra + " ");
                Thread.sleep(delay);
            }
        } catch (Exception e) {
            System.err.println("Erro");
            return;
        }
    }

    public static void main(String[] args) {
        Runnable ping=new MinhaThread("PING", 500);
        Runnable pong=new MinhaThread("PONG", 1000);
        new Thread(ping).start();
        new Thread(pong).start();
    }
}
```

16.5 Atribuindo e obtendo nomes das Threads:

Os métodos *setName()* e *getName()* permitem, respectivamente, que sejam atribuídos nomes às *threads* e que estes nomes possam ser posteriormente resgatados.

O código a seguir apresenta a utilização destes métodos.

```
public class MinhaThread extends Thread {

    private String palavra;
    private int delay;

    public MinhaThread(String oQueFalar, int tempoDelay){
        palavra=oQueFalar;
        delay=tempoDelay;
    }

    public void run() {
        try {

            System.out.print("Nome: " + getName());
            for (;;) {
                System.out.print(palavra + " ");
                Thread.sleep(delay);
            }

        }catch (Exception e) {
            System.err.println("Erro");
            return;
        }

    }

    public static void main(String[] args) {
        Runnable ping=new MinhaThread("PING", 500);
        Runnable pong=new MinhaThread("PONG", 1000);
        Thread PING = new Thread(ping);
        PING.setName("Super PING");
        Thread PONG = new Thread(pong);
        PONG.setName("Super PONG");
        PING.start();
        PONG.start();
    }
}
```

16.6. Atribuindo prioridades às Threads

Uma *thread* possui, inicialmente, a mesma prioridade da *thread* que a criou. Esta prioridade pode ser alterada através do método ***setPriority()***. Seu valor pode variar entre **MIN_PRIORITY** (valor padrão=1) e **MAX_PRIORITY** (valor padrão=10). Se nenhuma prioridade é especificada o valor **NORM_PRIORITY** (valor padrão=5) é assumido. O método ***getPriority()*** obtém o valor corrente da prioridade.

O código a seguir apresenta a utilização das prioridades em *threads*:

```
public class PingPongNome extends Thread {

    private String palavra;
    private int delay;

    public PingPongNome(String oQueFalar, int tempoDelay){

        palavra=oQueFalar;
        delay=tempoDelay;
    }
    public void run() {
        try {

            System.out.println("Nome :"+ getName() + " Prioridade " +
                getPriority());
            for (;;) {
                System.out.print(palavra + " ");
                Thread.sleep(delay);
            }
        } catch (Exception e) {
            System.err.println("Erro");
            return;
        }
    }

    public static void main(String[] args) {
        PingPongNome ping=new PingPongNome("ping", 500);
        ping.setName("Super PING");
        ping.setPriority(Thread.NORM_PRIORITY+5);
        ping.start();
        PingPongNome pong=new PingPongNome("PONG", 500);
        pong.setName("Super PONG");
        pong.start();
    }
}
```

16.7. Renunciando ao uso de CPU

O método ***yield()*** permite que uma *thread* renuncie ao uso da CPU, permitindo que outras *threads* possam ser disparadas. O método ***yield()*** informa ao *thread scheduler* que a *thread* não necessita, momentaneamente, ser executada. A *thread* emissora do ***yield()*** fica em estado de pronta e, caso não exista nenhuma outra *thread* para ser despachada, ela entra novamente em execução. Este método não precisa de uma estrutura ***try/catch***.

```
public class TestaYield extends Thread {
    private String palavra;
    private int delay;
    private boolean renuncia;

    public TestaYield(String oQueFalar, int tempoDelay, boolean renunciaCpu){
        palavra=oQueFalar;
        delay=tempoDelay;
        renuncia=renunciaCpu;
    }

    public void run() {
        try{
            for (;;) {

                System.out.print(palavra + " ");
                if (renuncia)
                    Thread.yield();
                Thread.sleep(delay);
            }

        }catch(Exception e) {
            System.err.println(" Erro");
            return;
        }
    }

    public static void main(String[] args) {

        TestaYield ping=new TestaYield("primeira", 500, true);
        ping.start();
        TestaYield pang =new TestaYield("segunda", 500, false);
        pang.start();
        TestaYield peng =new TestaYield("terceira", 500, false);
        peng.start();
        TestaYield pung =new TestaYield("quarta", 500, false);
        pung.start();
        TestaYield pong =new TestaYield("quinta", 500, false);
        pong.start();
    }
}
```

16.8 Cancelando uma thread

O processo de cancelamento sem riscos de uma *thread* é feito através dos métodos ***interrupt()*** e ***interrupted()***. O processo de cancelamento envolve, necessariamente, a ordem da *thread* que está comandando o cancelamento e o consentimento da *thread* que está sendo cancelada. Uma interrupção não força a parada de uma *thread*, embora ela interrompa o “descanso” de uma *thread* em *wait* ou *sleeping*, ela apenas ativa o estado de interrupção da *thread*.

Quando uma *thread* emite um ***interrupt()*** para uma outra *thread* ela apenas irá ativar este estado de interrupção. A outra *thread* se desejar ser interrompida deverá utilizar o método ***interrupted()*** para verificar se o estado de interrupção foi ativado. Caberá a esta *thread* decidir se interromperá ou não o seu processamento.

```
public class InterrompeThread {
    public static void main(String[] args) {
        new Thread02().start();
    }
}

class Thread01 extends Thread{

    public void run() {
        int i=0;
        while(!interrupted()) {
            i++;
            System.out.print( i + " ");
        }
    }
}

class Thread02 extends Thread{

    public void run() {
        try {
            Thread01 ping =new Thread01();
            ping.start();
            Thread.sleep(1000);
            ping.interrupt();
        }catch (Exception e) {
            System.err.println("Erro");
        }
    }
}
```

16.9. Esperando uma thread terminar:

Às vezes precisamos esperar o término da execução de várias *threads* que foram disparadas para produzirmos um resultado final. Para uma *thread* esperar que outra *thread* termine deve ser utilizado o método ***join()***. Este método precisa ser utilizado em conjunto com uma estrutura ***try/catch***.

O código abaixo apresenta um exemplo do uso do método *join()*.

```
public class TestaJoin extends Thread {
    private String palavra;
    private int delay;

    public TestaJoin(String oQueFalar, int tempoDelay){
        palavra=oQueFalar;
        delay=tempoDelay;
    }

    public void run() {
        try {
            System.out.print(palavra + " ");
            Thread.sleep(delay);

        }catch (Exception e) {
            System.err.println("Erro");
            return;
        }
    }

    public static void main(String[] args) {
        TestaJoin ping=new TestaJoin("Teste do Join !", 10000);
        ping.start();
        try {
            ping.join();
            System.out.println("Esperei o término !");
        }catch (Exception e) {
            System.err.println("Erro");
            return;
        }
    }
}
```

16.10 Sincronização

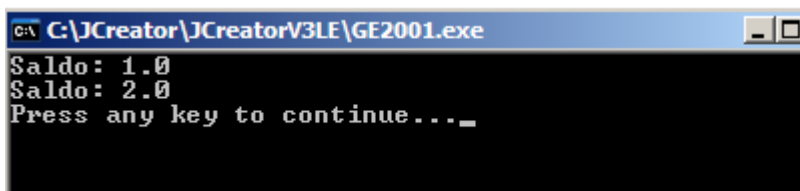
Para sincronizar o acesso a recursos compartilhados a linguagem Java utiliza monitores (semáforos). Todo o objeto contendo métodos com o atributo ***synchronized*** possui um monitor. O monitor permite que apenas uma *thread* por vez execute um método ***synchronized*** sobre o objeto. O objeto fica atômicamente bloqueado quando o seu método ***synchronized*** é invocado. As demais *threads* que tentarem invocar este método ficarão em estado de espera.

As variáveis de instância pertencem unicamente a uma instância de determinada classe e se esta classe implementa *threads*, cada *thread* disparada poderá ter a capacidade de compartilhar estas variáveis, o que poderá provocar sérios danos ao aplicativo.

No código abaixo temos um exemplo desta possível situação. São apresentadas as classes **MinhaClasse** e **MinhaThread**. **MinhaClasse** define uma variável de instância denominada **saldo** que tem o seu valor incrementado a cada invocação de seu método **calcula**. A classe **MinhaThread** disparará duas *threads* sobre uma única instância da classe **MinhaClasse**.

```
public class MinhaClasse {  
  
    private double saldo;  
    public MinhaClasse(){  
    }  
    public void calcula(){  
        saldo++;  
        System.out.println("Saldo: " + saldo);  
    }  
}  
  
public class MinhaThread implements Runnable {  
  
    private MinhaClasse soma = new MinhaClasse();  
  
    public void run() {  
        soma.calcula();  
    }  
    public static void main(String[] args) {  
        Runnable dispara = new MinhaThread();  
        Thread uma = new Thread(dispara);  
        Thread outra = new Thread(dispara);  
        uma.start();  
        outra.start();  
    }  
}
```

Quando executada a classe **MinhaThread** é produzido o resultado da figura abaixo.



Observe que o valor do saldo foi incrementado com a participação das duas *threads*. Portanto, podemos concluir que as duas

threads tiveram acesso à mesma variável de instância.

Imaginando a situação de gravação de alguma informação em um disco magnético esta operação de concorrência poderia provocar algum resultado inesperado. Da mesma forma seria se dois correntistas que compartilham a mesma conta-corrente pudessem ter acesso simultâneo para realizar operações de retirada.

No nosso código exemplo vamos utilizar o método *synchronized* para termos a garantia que apenas uma *thread* terá acesso ao recurso que queremos controlar.

Sendo assim, vamos colocar o *synchronized* na classe **MinhaClasse**, conforme o novo código abaixo.

```
public class MinhaClasse {  
    private double saldo;  
    public MinhaClasse(){  
    }  
  
    public synchronized void calcula(){  
        saldo++;  
        System.out.println("Saldo: " + saldo);  
    }  
}
```

Ao fazermos isto estamos tendo a certeza que o método *calcula* da classe **MinhaClasse** será executado atomicamente, qual seja, apenas uma *thread* por vez terá acesso a ele.

17.0 Tratamento de Exceções:

Para que você desenvolva uma aplicação que seja considerada robusta e confiável necessariamente você deverá se preocupar com o tratamento dos possíveis erros que poderão ocorrer durante a sua utilização pelos usuários em geral.

Um exemplo clássico seria a captura de informações de teclado fornecidas pelos usuários de seu aplicativo. O seu código está preparado para que seja digitado um valor numérico e, no entanto, foi digitado um valor não numérico. Obviamente quando for realizada a conversão dos dados ocorrerá um erro. O procedimento adequado que você deverá adotar é que aqueles erros possíveis de acontecerem sejam todos adequadamente tratados de forma que qualquer anomalia ocorrida seja sempre identificada.

Alguns possíveis problemas podem ser tratados já em tempo de compilação como, por exemplo, uma tentativa de se conectar a uma máquina servidora. Como existe a possibilidade de esta máquina estar indisponível o método que trata desta conexão obriga que você preveja esta possibilidade em seu código. Outro caso semelhante é a tentativa de ler algum dado gravado em um arquivo residente em um disco magnético. Este arquivo pode não mais existir! O seu código precisa prever esta situação.

Mas você não precisa guardar de cabeça todas as situações possíveis, o compilador sempre avisará quando o seu código precisar se precaver.

Mas nem sempre o compilador possui este poder. Alguns tipos de erros podem ser pouco prováveis como, por exemplo, não conseguir gravar um arquivo em um disco

magnético por falta de espaço. Neste caso o seu código compilará e somente em tempo de execução você perceberá o problema.

No código a seguir é solicitado ao usuário digitar dois números inteiros, via teclado. Os números serão divididos um pelo outro. No primeiro caso vamos fazer uma divisão por zero e vamos observar que será lançada uma exceção. Uma exceção é lançada quando o método em que ocorreu um determinado erro não consegue tratá-lo.

```
import java.util.Scanner;

public class TestaCalculadora{

    public static void main(String[] args){

        int numero1,numero2, resultado;
        Scanner entrada = new Scanner(System.in);

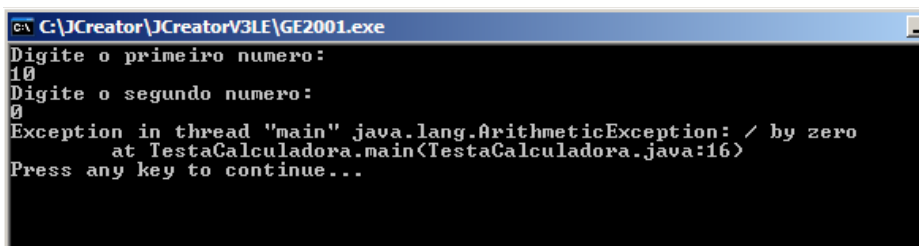
        System.out.println("Digite o primeiro numero:");
        numero1 = entrada.nextInt();

        System.out.println("Digite o segundo numero:");
        numero2 = entrada.nextInt();

        resultado = numero1 / numero2;

        System.out.printf("%s%d\n", "Resultado = ", resultado);

    }
}
```

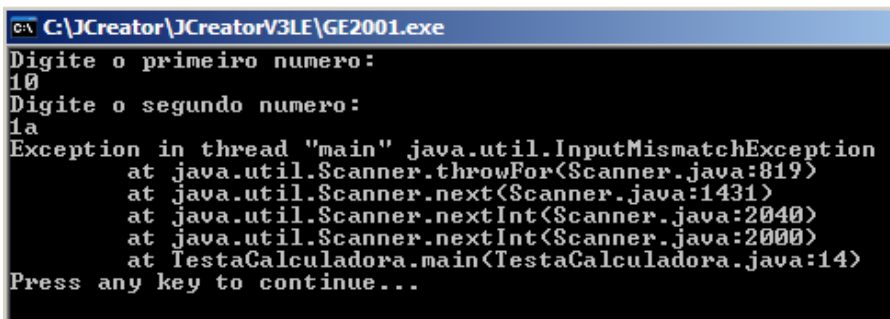


```
C:\JCreator\JCreatorV3LE\GE2001.exe
Digite o primeiro numero:
10
Digite o segundo numero:
0
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at TestaCalculadora.main<TestaCalculadora.java:16>
Press any key to continue...
```

A figura ao lado apresenta o resultado da execução da classe TestaCalculadora onde pode ser

observado o lançamento da exceção *java.lang.ArithmeticException* quando da execução da instrução **resultado = numero1 / numero2**.

Na execução do código anterior o usuário agora misturou números e letras na digitação do denominador. O resultado é apresentado na figura seguinte.



```
C:\JCreator\JCreatorV3LE\GE2001.exe
Digite o primeiro numero:
10
Digite o segundo numero:
1a
Exception in thread "main" java.util.InputMismatchException
    at java.util.Scanner.throwFor<Scanner.java:819>
    at java.util.Scanner.next<Scanner.java:1431>
    at java.util.Scanner.nextInt<Scanner.java:2040>
    at java.util.Scanner.nextInt<Scanner.java:2000>
    at TestaCalculadora.main<TestaCalculadora.java:14>
Press any key to continue...
```

Como pode ser observado desta vez a exceção

java.util.InputMismatchException é lançada. Desta vez a exceção não foi provocada pela divisão por zero e sim pela leitura de um dado não número inteiro através do método *nextInt()*.

A apresentação destas exceções para o usuário final do aplicativo demonstrará uma relativa fragilidade no aplicativo que nunca será bem aceita.

No código seguinte você verá como podemos tratar as exceções que foram lançadas.

```
import java.util.Scanner;
import java.util.InputMismatchException;

public class TestaCalculadora{

    public static void main(String[] args) {

        int numero1,numero2, resultado;
        boolean continua=true;
        Scanner entrada = new Scanner(System.in);
        do {
            try{
                System.out.println("Digite o primeiro numero:");
                numero1 = entrada.nextInt();
                System.out.println("Digite o segundo numero:");
                numero2 = entrada.nextInt();
                resultado = numero1 / numero2;
                System.out.printf("%s%d\n", "Resultado = ", resultado);
                continua = false;
            }
            catch(InputMismatchException erro1){
                System.err.printf("%s", "Você digitou errado. Entre com números inteiros\n");
                entrada.nextLine();
            }
            catch(ArithmeticException erro2){
                System.err.printf("%s", "Divisão por ZERO.\n");
                entrada.nextLine();
            }
        } while(continua);
    }
}
```

Para tal foi incluído um bloco *try* que é o bloco onde estão inseridas as instruções que poderão lançar as exceções. O bloco *catch* contém as instruções que tratarão as exceções. Para cada tipo de exceção lançada deve haver um bloco *catch*.

Todas as classes de exceção do Java herdam da classe *Exception* e temos os seguintes tipos de exceções mais usuais: *RuntimeException*, *IOException*, *AWTError*, *ThreadDeath*, *OutOfMemoryError*, *ArrayIndexOutOfBoundsException*, *InputMismatchException*, *ClassCastException*, *NullPointerException* e *ArithmeticException*.

Em algumas situações você poderá desejar executar algum código de seu aplicativo mesmo que uma exceção seja lançada, para tal existe o bloco ***finally***. Toda e qualquer instrução que integra este bloco será executada mesmo que uma exceção seja ou não lançada.

Este bloco deve ser inserido após os blocos ***try/catch***. O código a seguir reapresenta o segundo exemplo onde ocorreu uma exceção que foi tratada.

```
import java.util.Scanner;
import java.util.InputMismatchException;

public class TestaCalculadora{

    public static void main(String[] args) {

        int numero1,numero2, resultado;
        boolean continua=true;
        Scanner entrada = new Scanner(System.in);

        do {
            try{
                System.out.println("Digite o primeiro numero:");
                numero1 = entrada.nextInt();
                System.out.println("Digite o segundo numero:");
                numero2 = entrada.nextInt();
                resultado = numero1 / numero2;
                System.out.printf("%s%d\n", "Resultado = ", resultado);
                continua = false;
            }
            catch(InputMismatchException erro1){
                System.err.printf("%s", "Você digitou errado. Entre com números inteiros\n");
                entrada.nextLine();
            }
            catch(ArithmeticException erro2){
                System.err.printf("%s", "Divisão por ZERO.\n");
                entrada.nextLine();
            }
            finally{
                System.out.println("Passou por mim");
            }
        } while(continua);
    }
}
```

O resultado da execução é apresentado a seguir.

```

C:\JCreator\JCreatorV3LE\GE2001.exe
Digite o primeiro numero:
10
Digite o segundo numero:
0
Divisão por ZERO.
Passou por mim
Digite o primeiro numero:
10
Digite o segundo numero:
2
Resultado = 5
Passou por mim
Press any key to continue..._

```

Observe que a instrução inserida no bloco *finally* é executada ocorrendo erro ou não na entrada dos dados.

Você também poderá criar métodos que lançam exceções. Estas exceções podem ser aquelas que integram a família da classe *Exception* ou você pode criar o seu próprio conjunto de exceções.

Para lançar uma exceção, o método lançador deverá conter a seguinte instrução em sua assinatura:

throws tipo-da-exceção

E em seu interior:

throw new tipo-da exceção

O código a seguir apresenta um exemplo de utilização de uma classe lançando exceções para serem tratadas.

```

import java.util.Scanner;
public class LancaExcecao{
    public static void main(String args[]) {
        Excecao trata = new Excecao();
        try{
            trata.leAlgo();
        }
        catch(Exception e){
            System.err.println("Usuario não digitou nada");
        }
    }
}
class Excecao {
    private String nome;
    public void leAlgo() throws Exception{
        Scanner entrada = new Scanner(System.in);
        System.out.println("Digite algo:");
        nome = entrada.nextLine();

        if ((nome.trim()).equals("")){
            throw new Exception();
        }
    }
}

```

Neste exemplo é lançada a exceção do tipo *Exception* caso o usuário nada tenha digitado.

18.0 Processando Arquivos:

Nas aplicações em geral os dados normalmente são armazenados em bases de dados residentes em discos magnéticos de onde, posteriormente, poderão ser consultados. Conseqüentemente, é de fundamental importância dominar as técnicas que permitem a leitura e a gravação de dados nestes arquivos.

Os arquivos podem ser classificados em dois tipos:

Arquivos texto – um arquivo que contém caracteres do tipo texto. Os registros ou campos são por delimitados por caracteres especiais definidos pelo próprio desenvolvedor da aplicação.

Arquivos binários – são arquivos contendo dados primitivos e objetos de dados.

As operações que são realizadas nos arquivos são denominadas de “Operações de Entrada/Saída”.

Quase todas as instruções que manipulam arquivos devem estar contidas em um bloco **try/catch**, uma vez que vários tipos de exceções podem ser lançados. As exceções mais comumente lançadas são:

IOException

Indica que ocorreu algum problema na operação de E/S.

EOFException

Indica que está sendo feita uma operação de leitura além do fim do arquivo.

FileNotFoundException

Indica que o arquivo não foi localizado.

18.1 Identificando e localizando Arquivos:

O primeiro passo para você trabalhar com um arquivo é informar em seu aplicativo o nome e onde o arquivo está localizado. Estas operações são viabilizadas pela classe **File**, integrante do pacote `java.io.*`. A classe **File** não lança exceções uma vez que ela apenas obtém o caminho para o diretório ou o arquivo desejado. Cabe a você verificar se o diretório ou arquivo existe ou não.

A classe **File** possui os seguintes construtores:

<i>Construtor</i>	<i>Descrição</i>
File(StringPathName)	Cria um objeto File que faz referencia um específico path.
File(StringPathName, StringSubPath)	Cria um objeto File que referencia um path combinando um específico path com um subpath.
File(File, StringSubPath)	Cria um objeto File que referencia um path combinando um outro objeto File com um subpath.

Alguns exemplos de utilização da classe **File**:

1. Criando uma referência para um arquivo denominado “teste.txt” em um diretório local:

File arquivoLeitura = new File(“teste.txt”);

2. Criando uma referência para o arquivo “teste.txt” localizado no diretório “Aplicacoes”:

File arquivoLeitura = new File(“c:/Aplicações/teste.txt”);

3. Criando uma referência para o arquivo “teste.txt” localizado no diretório “Aplicacoes” (outra forma):

File arquivoLeitura = new File(“../Aplicações”, “teste.txt”);

A seguir são apresentados alguns dos principais métodos disponibilizados pela classe **File**.

<i>Método</i>	<i>Descrição</i>
exists()	Retorna <i>true</i> se o pathname existe.
isDirectory()	Retorna <i>true</i> se o pathname existe e é um diretório.
isFile()	Retorna <i>true</i> se o pathname existe e é um arquivo.
getName()	Retorna o nome do diretório ou do arquivo.
length()	Retorna o tamanho em bytes do arquivo
mkdir()	Cria um novo diretório para o objeto File.
delete()	Apaga o arquivo ou o diretório do objeto File.
getAbsolutePath()	Obtém o endereço absoluto do objeto File.

O código a seguir obtém algumas informações a respeito de determinado arquivo.

```
import java.io.*;

public class TestaFiles{

    public static void main(String[] args){

        File arquivo = new File("TestaFiles.java");

        if (arquivo.exists()){

            System.out.println("Nome do arquivo: " + arquivo.getName());

            System.out.println("Nome do path: " + arquivo.getPath());

            System.out.println("Nome do path absoluto: " + arquivo.getAbsolutePath());

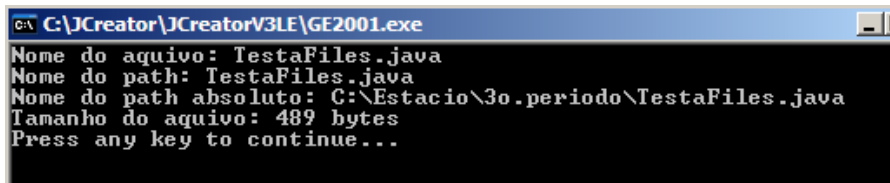
            System.out.println("Tamanho do arquivo: " + arquivo.length() + " bytes");

        }

    }

}
```

A figura abaixo apresenta o resultado da execução.



Observe que a instrução **if(arquivo.exists())** é fundamental para verificar se o

arquivo existe. Caso isto não seja feito a operação de manipulação dos dados do arquivo lançará exceções caso você queira realizar operações de leitura.

18.2 Trabalhando com Arquivos txt:

Inicialmente vamos tratar da gravação de dados em um arquivo do tipo **txt**. Mas, antes disto, vamos fazer algumas definições:

- **Stream**
É um fluxo de dados entre uma determinada localização e um arquivo. Para escrever um arquivo você gera um **outputstream**. Para ler um arquivo você gera um **inputstream**. São utilizados com disco, teclado, e aplicações que envolvem rede.
- **CharacterStream**
Para ler ou escrever um arquivo do tipo txt você utiliza um **characterstream**.
- **BinaryStream**
Para ler ou escrever um arquivo do tipo binário você utiliza um **binarystream**.

- **Buffer**

Área de memória onde os dados são colocados antes de serem transferidos do ou para os dispositivos. Minimiza o tempo de transferência. Seu uso é opcional, mas é fortemente recomendado

Para gravarmos dados existem as seguintes classes:

<i>Classe</i>	<i>Descrição</i>
PrintWriter(Writer)	Classe que escreve os dados para o characterstream
BufferedWriter(Writer)	Classe que cria o buffer que abrigará o characterstream.
FileWriter(File)	Classe que conecta um outputstream para um arquivo.
FileWriter(File, true)	Classe que conecta um outputstream ao final de um arquivo.

Para a realização de uma gravação eficiente de um arquivo txt as seguintes etapas precisam ser cumpridas:

1. Identificar o arquivo de destino (classe **File**).
2. Conectar o *outputstream* ao arquivo identificado no item anterior (classe **FileWriter**).
3. Criar o *buffer* que abrigará o *outputstream* (classe **BufferedWriter**).
4. Transferir o *outputstream* para o *buffer* (classe **PrintWriter**).

A classe **FileWriter** lança um **IOException**.

O segmento de código a seguir mostra como utilizar estas classes.

```
import java.io.*;

public class TestaFiles{

    public static void main(String[] args){

        File arquivo = new File("teste.txt");
        try{
            FileWriter conecta = new FileWriter(arquivo);
            BufferedWriter buffer = new BufferedWriter(conecta);
            PrintWriter imprime = new PrintWriter(buffer);

        }
        catch(IOException e){}
```

```

    }
}

```

Neste código todo o ambiente já está pronto para você iniciar a impressão, falta apenas apresentarmos os métodos da classe **PrintWriter**.

<i>Método</i>	<i>Descrição</i>
print(argumento)	Escreve o argumento no arquivo destino.
println(argumento)	Escreve o argumento no arquivo destino colocando uma identificação ao final do argumento.
flush()	Libera o stream para escrita imediatamente.
close()	Libera o stream para escrita e fecha o arquivo.

Os seguintes argumentos são aceitos: boolean, char, char[], String, object, int, long, double e float.

No próximo exemplo o nosso código já gravará dados no arquivo.

```

import java.io.*;

public class TestaFiles{

    public static void main(String[] args){

        File arquivo = new File("teste.txt");
        try{
            FileWriter conecta = new FileWriter(arquivo);
            BufferedWriter buffer = new BufferedWriter(conecta);
            PrintWriter imprime = new PrintWriter(buffer);
            imprime.println("Estou");
            imprime.println("Aprendendo");
            imprime.println("Java");
            imprime.close();
        }
        catch(IOException e){}
    }
}

```

Para ler dados de arquivos ou outros dispositivos existem as seguintes classes:

<i>Classe</i>	<i>Descrição</i>
FileReader(File)	Classe que conecta um objeto a um arquivo.
BufferedReader(Reader)	Classe que cria o buffer que abrigará o characterstream de entrada.

O segmento de código abaixo apresenta a utilização destas classes para a leitura de um arquivo.

```
import java.io.*;

public class TestaFiles{

    public static void main(String[] args){

        File arquivo = new File("teste.txt");
        if (arquivo.exists()){
            try{
                FileReader conecta = new FileReader(arquivo);
                BufferedReader ledor = new BufferedReader(conecta);
            }
            catch(IOException e){}
        }
    }
}
```

A classe **BufferedReader** oferece os seguintes métodos para você realizar a leitura de registros oriundos de arquivos em disco ou outros dispositivos.

<i>Método</i>	<i>Descrição</i>
readLine()	Retorna um string com a linha de texto lido.
read()	Lê um único caractere e retorna como um tipo inteiro.
skip(long)	Tenta saltar um determinado número de caracteres retornando o valor real saltado.
close()	Fecha o inputstream liberando o buffer.

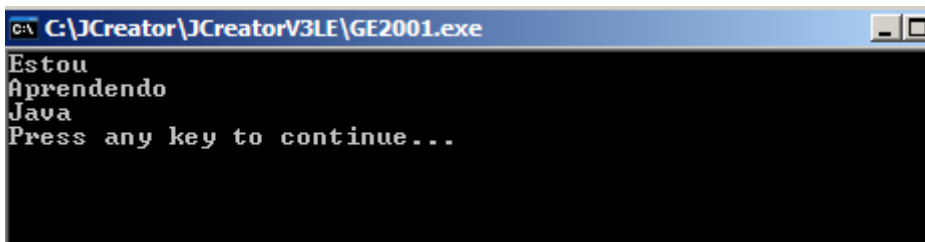
O código seguinte apresenta um exemplo de leitura de um arquivo residente em disco.

```
import java.io.*;
public class TestaFiles{
    public static void main(String[] args){
```

```

        File arquivo = new File("teste.txt");
        if (arquivo.exists()){
            try{
                FileReader conecta = new FileReader(arquivo);
                BufferedReader ledor = new BufferedReader(conecta);
                String registro = ledor.readLine();
                while(registro!=null){
                    System.out.println(registro);
                    registro=ledor.readLine();
                }
                ledor.close();
            }
            catch(IOException e){}
        }
    }
}

```



A figura ao lado apresenta o resultado da execução do código.

19 Estruturas de Dados:

Nas seções anteriores você teve contato com estruturas de dados de tamanho fixo, como os **arrays**, e também com estruturas de dados de tamanho variável, ou dinâmicas, como os vetores. Nesta seção serão vistas outras formas de estruturas de dados variáveis que são as listas vinculadas, as pilhas, as filas e as árvores binárias.

Antes de você aprender estas novas estruturas vamos apresentar um recurso interessante para resolver algumas questões ligadas a estas estruturas denominado recursividade.

19.1 Recursividade:

Um método ou procedimento é dito recursivo se for definido em termos de si próprio. Para este conceito ser melhor entendido vamos resolver o seguinte problema:

Problema:

Escreva uma classe contendo um método que receba como argumento um inteiro positivo N e retorne a soma de todos os números inteiros entre 0 e N.

Este programa teria o seguinte código:

```
public class TestaCalculadora{  
    public int somaNumeros(int quantidade){  
        int soma=0;  
        for (int i=0; i<= quantidade ;i++){  
            soma+=i;  
        }  
        return soma;  
    }  
}
```

A pergunta agora é: como este código poderia ser escrito não utilizando nenhuma estrutura de repetição do tipo *for*, *while* ou *do while* ? A resposta é: utilizando a recursividade onde o método deverá chamar a si próprio para resolver a questão.

O novo código ficaria assim:

```
public class TestaCalculadora{  
    public int somaNumeros(int quantidade){  
        if (quantidade == 1) return 1;  
        else  
            return (quantidade + somaNumeros(quantidade -1));  
    }  
}
```

Caso o valor de N seja igual a 3 teríamos a seguinte sequência de execução:

```
N==3  
    N==2  
        N== 1 return 1;  
    return 2+1;  
return 3+3;
```

Outro exemplo clássico na utilização da recursividade é no cálculo do fatorial de determinado número. Resolvendo convencionalmente temos o seguinte código:

```
public class TestaCalculadora{  
    public int calculaFatorial(int numero){  
        int fatorial = 1;  
        for (int i=numero; i>=1; i--){
```

```
        fatorial *=i;
    }
    return fatorial;
}
```

Utilizando a recursividade o novo código seria:

```
public class TestaCalculadora{

    public int calculaFatorial(int numero){

        if (numero == 0) return 1;
        else
            return (numero * calculaFatorial(numero - 1));
    }
}
```

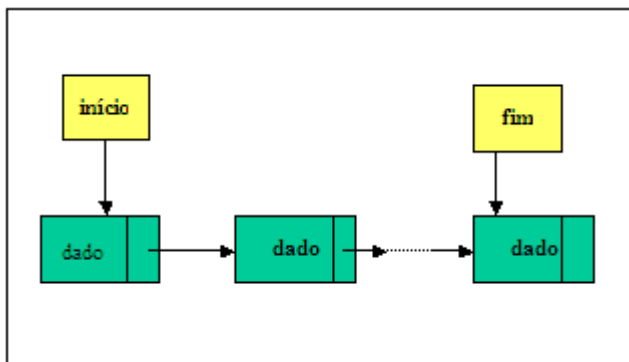
19.2 Listas Vinculadas:

Uma lista vinculada ou ligada é uma coleção de nós conectados por endereços de referência. Cada nó contém o dado de interesse e o endereço do nó seguinte. O último nó da lista contém um endereço *null* uma vez que não referencia nenhum outro nó.

Os dados contidos em um nó podem ser de qualquer tipo e uma lista vinculada não possui um número fixo de elementos.

Uma lista vinculada é um caso específico de uma lista linear, qual seja, os dados são colocados em seqüência.

A figura abaixo representa graficamente uma lista vinculada. Esta lista é chamada de **lista vinculada individualmente**. Quando um nó de uma lista referencia um nó próximo da lista e um nó anterior esta lista é chamada de lista duplamente vinculada. Neste tipo de lista o último nó apontará para o primeiro e para o nó anterior.



19.3 Pilhas:

Uma pilha é uma forma limitada de lista vinculada uma vez que novos nós somente podem ser inseridos ou removidos da parte superior da pilha. O último elemento armazenado em uma pilha deve ser configurado como *null* para indicar que é o fim da pilha. Uma pilha também é um caso específico de uma lista linear.

O conteúdo de uma pilha pode variar dinamicamente e a linguagem Java oferece a classe **Stack**, integrante do pacote `java.util`, para a sua implementação.

As operações definidas para uma pilha incluem:

1. Verificar se a pilha está vazia.
2. Inserir um elemento na pilha (empilhar ou “*push*”), no lado do topo.
3. Remover um elemento da pilha (desempilhar ou “*pop*”), do lado do topo.

Uma vez que o último elemento que entrou na pilha será o primeiro a sair, a pilha é também conhecida como uma estrutura do tipo LIFO (*Last In First Out*).

Podemos citar como exemplo o conjunto de pratos empilhados em um restaurante de comida a quilo. Os primeiros pratos colocados sobre a mesa serão os últimos a serem retirados pelos consumidores.

19.4 Filas:

As filas são as estruturas de dados mais populares em nosso cotidiano. Quando você se dirige a um caixa de uma agência bancária depara-se com um clássico exemplo de fila. Neste tipo de estrutura os nós são removidos somente da cabeça da fila e novos nós são inseridos apenas ao fim da fila. Uma fila também é um caso específico de uma lista linear.

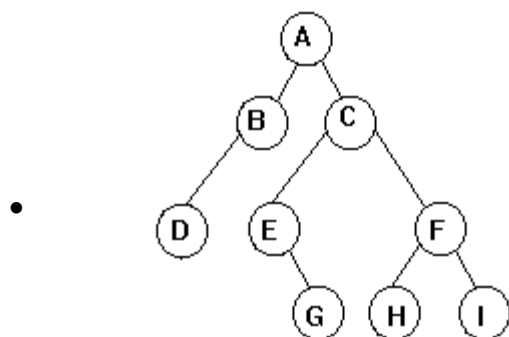
A linguagem Java, a partir da versão 1.5, oferece a interface **Queue** e a classe **PriorityQueue**, ambas integrantes do pacote `java.util`, para a implementação de filas.

Como exemplo de filas podemos citar os serviços dirigidos para uma impressora. Os serviços enviados por último também serão os últimos a serem impressos.

19.5 Árvores Binárias:

Diferentemente das estruturas anteriores as árvores binárias não são lineares. As árvores binárias também caracterizam-se por serem bidimensionais.

A figura abaixo representa graficamente uma árvore binária.



Uma Árvore Binária A é um conjunto finito de elementos denominados nós ou vértices, tal que:

- Se $A = 0$ a árvore é dita vazia.

Existe um nó especial r , chamado raiz de A , os nós restantes podem ser divididos em dois subconjuntos distintos, *Arel* e *Ard*, que são as sub-árvores esquerda e direita de r , respectivamente que, por sua vez, também

são árvores binárias.

Os filhos de um nó específico são chamados de irmãos e um nó sem filhos é chamado de nó-folha. Da figura acima podemos afirmar que o nó A é o nó-raiz e os nós B e C são irmãos assim como os nós E e F. Os nós D, G, H, e I são nós-folhas já que não possuem filhos.

20 Coleções:

Uma coleção pode ser definida como uma estrutura de dados que armazena referências para outros objetos sendo que, normalmente, estes objetos possuem o mesmo tipo.

As coleções possuem interfaces que definem todas as operações que podem ser realizadas nestas coleções. A tabela abaixo apresenta algumas destas interfaces e seus propósitos.

<i>Interface</i>	<i>Descrição</i>
Collection	É a interface mãe.
Set	Derivada de Collection. Não contém elementos duplicatas.
List	Derivada de Collection. Pode conter elementos duplicatas.
Map	Derivada de Collection. Associa chaves a valores.
Queue	Derivada de Collection. Implementa uma fila do tipo FIFO.

20.1 ArrayList:

Um **ArrayList** tem funcionamento análogo a um **Vector** pois ambos possuem a capacidade de armazenar *arrays* objetos. A principal diferença existente é que o acesso ao **Vector** é sincronizado, qual seja, qualquer método que acesse o conteúdo de um **Vector** o fará exclusivamente. Um **ArrayList**, de outra forma, não é sincronizado. Uma vez que a sincronização pode afetar a performance utilize sempre **ArrayList** caso você não precise sincronizar o acesso aos objetos.

Internamente, tanto o **ArrayList** quanto o **Vector** são um *array*. Quando você insere um elemento em um ou em outro o *array* irá se expandir, caso a capacidade inicial tenha se esgotado. Nesta situação, um **Vector**, por valor padrão, dobrará o seu tamanho enquanto um **ArrayList** aumentará cinquenta por cento, portanto é sempre recomendável dimensionar bem a capacidade inicial tanto do **ArrayList** quanto do **Vector** para evitar o desperdício de memória.

Tanto o **ArrayList** quanto o **Vector** são adequados para obter qualquer elemento de uma posição específica ou para adicionar ou remover elementos ao seu final. Entretanto, adicionar ou remover elementos de qualquer outra posição é mais custoso, uma vez que o índice do elemento removido deve ser preenchido por outro elemento, podendo provocar sucessivos deslocamentos.

A tabela abaixo apresenta alguns métodos da classe **ArrayList**.

<i>Método</i>	<i>Descrição</i>
add(objeto)	Adiciona um objeto ao ArrayList.
remove(índice)	Remove um objeto em determinado índice.
remove(objeto)	Remove o objeto.
contains(objeto)	Retorna <i>true</i> se o objeto existe.
isEmpty()	Retorna <i>true</i> se o ArrayList não possui elementos.
indexOf(objeto)	Retorna o índice do objeto procurado ou -1 se não encontrado.
size()	Retorna o número de elementos no ArrayList.
get(índice)	Retorna o objeto de determinado índice.

Para utilizar a classe **ArrayList** é necessário importar o pacote **import java.util.ArrayList** e para instanciá-la as seguintes maneiras podem ser utilizadas:

ArrayList nome-do-Array = new ArrayList (); ou

```
ArrayList nome-do-Array = new ArrayList(capacidade); ou
```

```
ArrayList<objeto> nome-do-Array = new ArrayList <objeto>(); ou
```

```
ArrayList<objeto> nome-do-Array = new ArrayList <objeto>(capacidade);
```

Se nenhum argumento for passado para o construtor da classe **ArrayList**, é assumido um valor padrão de tamanho igual a dez e um incremento de capacidade padrão igual a zero.

Para que os tipos numéricos possam ser armazenados em um **ArrayList** é necessário que antes eles sejam convertidos para objetos. Para realizar esta conversão existem duas classes **Integer** e **Double**. A classe **Integer** é constituída dos métodos **parseInt()** e **toString()** e aplica-se para os tipos definidos como **int** e como **long** já a classe **Double** é constituída dos métodos **parseDouble()** e **toString()** se aplica para os tipos definidos como **double**. Os métodos **parseInt()** e **parseDouble()** convertem um objeto **String** para o tipo numérico correspondente enquanto o método **toString()** converte o tipo numérico para um objeto **String**.

Para converter uma *string* para um inteiro escrevemos, por exemplo:

```
int quantidade = Integer.parseInt(quantidadeString); ou
```

```
double quantidade = Double.parseDouble(quantidadeString);
```

Para converter um número inteiro para uma *string* escrevemos:

```
String quantidadeString = Integer.toString(quantidade); ou
```

```
String quantidadeString = Double.toString(quantidade);
```

No próximo exemplo um **ArrayList** é criado e são adicionados dois objetos. São apresentados: o tamanho do **ArrayList**; a sua capacidade inicial; o conteúdo do objeto armazenado na primeira posição do vetor.

```
import java.util. ArrayList;
```

```
public class Teste{
```

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

```
        ArrayList<Object> array = new ArrayList<Object>(5); // Array com 5  
                                                    elementos
```

```
        array.add("Java") ;
```

```
        array.add(new Integer(2)); // Aqui é adicionado um inteiro
```

```
        System.out.printf("%s%d\n","Tamanho: ", array.size());
```

```

        System.out.printf("%s%s\n", "Primeiro Elemento: ", array.get(0));
    }
}

```



A figura ao lado apresenta o resultado da execução da classe **Teste**. O segundo objeto incluído foi um tipo primitivo inteiro que foi transformado em um objeto através da instanciação da classe **Integer**.

No próximo exemplo um objeto substitui um outro, a capacidade de armazenamento do vetor é igualada ao número de objetos adicionados e é verificada se a substituição ocorreu.

```

import java.util. ArrayList;

public class Teste{

    public static void main(String[] args){

        ArrayList array = new ArrayList(12);

        array.add("anna") ;

        array.add(new Integer(2)); // Aqui é incluído um inteiro

        array.set(0,"rose");

        array.trimToSize();

        System.out.printf("%s%d\n", "Tamanho: ", array.size());

        System.out.printf("%s%s\n", "Primeiro Elemento: ", array.get(0));

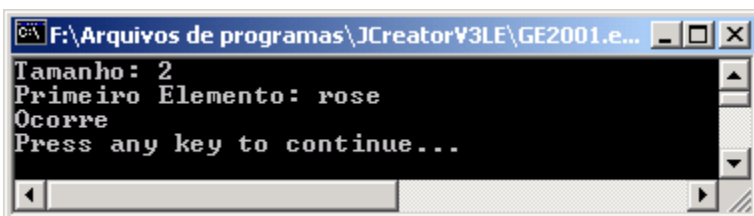
        if (array.contains("rose")) System.out.printf("%s\n", "Ocorre");

    }

}

```

A figura ao lado apresenta a execução da classe **Testa** onde pode ser observada a nova capacidade do **ArrayList** e o objeto substituto.



E se você quisesse classificar em ordem ascendente os objetos armazenados em um **ArrayList**? Você iria se decepcionar uma vez que a classe **ArrayList** não possui um método específico para realizar tal tarefa. A solução é a utilização de uma nova classe denominada **Collections** que possui o método *sort()*.

No código abaixo é criado um **ArrayList** com um string de nomes que é passado para o método *sort()* da classe **Collections**, e, em seguida, o conteúdo do *array* é impresso.

```
import java.util. ArrayList;

import java.util. Collections;

public class Teste{

    public static void main(String[] args){

        ArrayList<String> array = new ArrayList<String>();

        array.add("anna") ;

        array.add("rose");

        array.add("joao") ;

        array.add("andre") ;

        array.add("maria") ;

        Teste testa = new Teste();

        testa.chamaSort(array);

    }

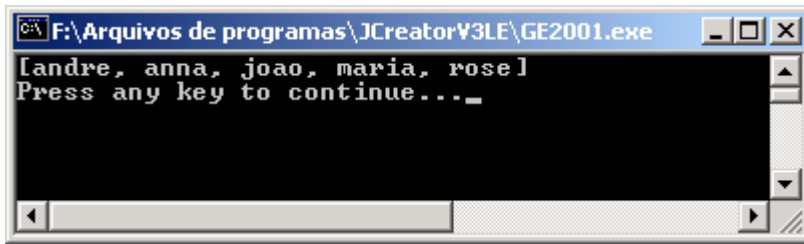
    public void chamaSort(ArrayList array){

        Collections.sort(array);

        System.out.printf("%s\n", array);

    }

}
```



A figura ao lado apresenta o resultado da execução do código. Observe que os nomes do **ArrayList** ficaram ordenados.

20.2 LinkedList:

A classe **LinkedList** fornece métodos que permitem obter, remover e inserir elementos tanto no início quanto no fim de uma lista. Conseqüentemente, a classe **LinkedList** torna-se apropriada para você criar estruturas de dados do tipo pilha, fila ou lista encadeada.

Todas as operações que envolvem a **LinkedList** são assíncronas, portanto, caso você necessite de sincronismo terá que implementá-lo explicitamente em seu código.

Para utilizar a classe **LinkedList** é necessário importar o pacote **import java.util. LinkedList** e para instanciá-la as seguintes maneiras podem ser utilizadas:

LinkedList nome-da-lista = new LinkedList (); ou

LinkedList <objeto> nome-do-Array = new ArrayList <objeto>();

A tabela abaixo apresenta alguns dos métodos da classe **LinkedList**.

<i>Método</i>	<i>Descrição</i>
add(objeto)	Adiciona um objeto ao fim da lista.
add(índice, objeto)	Adiciona um objeto em determinada posição da lista.
addFirst(objeto)	Adiciona um objeto ao início da lista.
addLast(objeto)	Adiciona um objeto ao fim da lista.
clear()	Remove todos os elementos da lista.
contains(objeto)	Verifica se a lista contém o objeto.
element()	Obtém um elemento da lista sem removê-lo.
get(índice)	Retorna o objeto de determinado índice.
getFirst()	Obtém o primeiro elemento da lista.
getLast()	Obtém o último elemento da lista.
indexOf(objeto)	Indica o índice da 1ª. ocorrência do objeto ou -1 se não encontrar.
lastIndexOf(objeto)	Indica o índice da última ocorrência do objeto ou -1 se não encontrar.
remove()	Remove o primeiro elemento da lista.
remove(índice)	Remove o elemento localizado por índice.
remove(objeto)	Remove um objeto específico.
removeFirst()	Remove o 1º. objeto da lista.
removeLast()	Remove o último objeto da lista.
set(índice, objeto)	Substitui o elemento em índice pelo objeto.
size()	Retorna o número de elementos na lista.

No próximo exemplo uma **LinkedList** é criada, como no primeiro exemplo do **ArrayList**, e são adicionados dois objetos. São apresentados: o tamanho da **LinkedList**; a sua capacidade inicial; o conteúdo do objeto armazenado na primeira posição da fila utilizando os métodos *get()* e *getFirst()*.

```
import java.util.LinkedList;

public class TestaFiles{

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



```

        LinkedList<Object> lista = new LinkedList<Object>();

        lista.add("anna");

        lista.add(new Integer(2)); // Aqui é incluído um inteiro

        System.out.printf("%s%d\n", "Tamanho: ", lista.size());

        System.out.printf("%s%s\n", "Primeiro Elemento: ", lista.get(0));

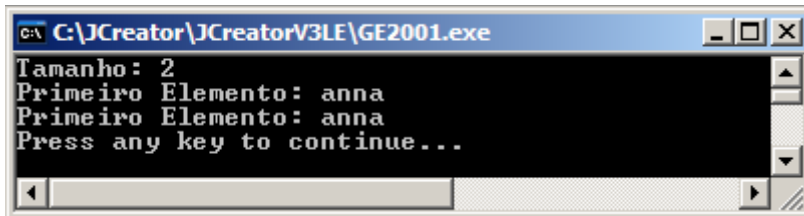
        System.out.printf("%s%s\n", "Primeiro Elemento: ", lista.getFirst());

        if (lista.contains("rose")) System.out.printf("%s\n", "Ocorre");

    }

}

```



A figura ao lado apresenta o resultado da execução.

Da mesma forma que o **ArrayList** a **LinkedList** não possui o método **sort()** então, aqui também teremos que utilizar a classe **Collections**. No código abaixo você verá que é criado uma **LinkedList** com um string de nomes que é passado para o método **sort()** da classe **Collections**, e, em seguida, o conteúdo da lista é impresso.

```

import java.util.LinkedList;

import java.util. Collections;

public class TestaFiles{

    public static void main(String[] args){

        LinkedList<String> lista = new LinkedList<String>();

        lista.add("anna");

        lista.add("rose");

        lista.add("joao");

        lista.add("andre");

        lista.add("maria");
    }
}

```

```
TestaFiles testa = new TestaFiles();

testa.chamaSort(lista);

}

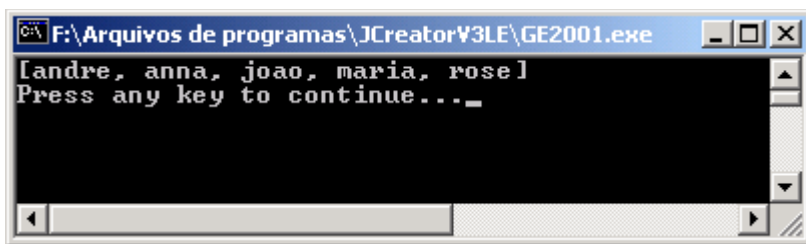
public void chamaSort(LinkedList lista){

    Collections.sort(lista);

    System.out.printf("%s\n", lista);

}

}
```



A figura ao lado apresenta o resultado da execução.

20.3 Classe Collections:

A classe **Collections** oferece vários métodos estáticos que auxiliam na manipulação das listas aprendidas. Estes métodos implementam alguns algoritmos que são de extrema utilidade no cotidiano computacional, possibilitando que você utilize soluções já prontas, testadas e que funcionam corretamente.

A tabela abaixo apresenta uma descrição sumária de cada um destes métodos:

<i>Método</i>	<i>Descrição</i>
sort	Classifica os elementos de uma lista.
binarySearch	Localiza um objeto em uma lista ordenada.
reverse	Inverte os elementos de uma lista.
shuffle	Embaralha os elementos de uma lista
copy	Copia as referências de uma lista para outra lista.
min	Retorna o menor elemento de uma coleção.
max	Retorna o maior elemento de uma coleção
addAll	Copia todos os elementos de um <i>array</i> para uma coleção
frequency	Informa quantos elementos na coleção são iguais a um elemento específico.
disjoint	Verifica se duas coleções não possuem nenhum elemento em comum.

20.3.1 sort() e reverse():

O métodos *sort()* e *reverse()* são dois dos métodos mais utilizados na computação comercial. Você sempre irá se deparar com situações em que a ordenação das informações será um processo obrigatório: seja a classificação dos nomes dos alunos de uma turma ou os salários dos funcionários de uma determinada empresa.

No código abaixo é apresentado um exemplo de uso destes métodos em que os elementos de uma lista são classificados em ordem crescente e ordem decrescente.

```
import java.util.LinkedList;

import java.util. Collections;

public class TestaSortReverse{

    public static void main(String[] args){

        LinkedList<String> lista = new LinkedList<String>();

        lista.add("anna") ;

        lista.add("rose");

        lista.add("joao") ;

        lista.add("andre") ;
```

```
        lista.add("maria") ;

        TestaSortReverse testa = new TestaSortReverse ();

        testa.chamaSort(lista);

        testa.chamaReverse(lista);

    }

    public void chamaSort(LinkedList lista){

        Collections.sort(lista);

        System.out.printf("%s\n", lista);

    }

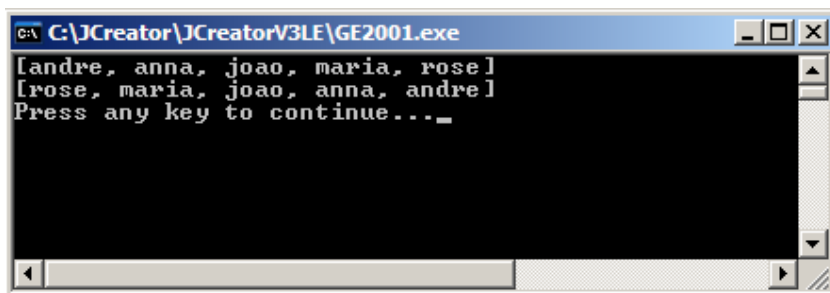
    public void chamaReverse(LinkedList lista){

        Collections.reverse(lista);

        System.out.printf("%s\n", lista);

    }

}
```



A figura ao lado apresenta o resultado da execução dos dois métodos.

Uma outra maneira de se realizar uma classificação é implementar a interface **Comparator**. A interface obriga que você desenvolva o método *compare()* que possui a seguinte sintaxe:

int compare(Object obj1, Object obj2)

Este método deve prover o seguinte retorno para quem o está invocando:

Se $\text{obj1} = \text{obj2}$ então retorna o inteiro 0.

Se $\text{obj1} > \text{obj2}$ então retorna o inteiro +1.

Se $\text{obj1} < \text{obj2}$ então retorna o inteiro -1.

Caso você queira classificar em ordem decendente basta inverter os sinais dos valores unitários, conforme será visto em exemplo próximo.

No código abaixo é apresentado um exemplo de uso da interface **Comparator**. Observe que a classe que a implementa é fornecida como um argumento da classe **Collections()**.

```
import java.util.*;

public class TestaFiles {

    public static void main(String args[]) {

        ArrayList lista = new ArrayList();

        lista.add(new Integer(-200));

        lista.add(new Integer(100));

        lista.add(new Integer(400));

        lista.add(new Integer(-300));

        lista.add(new Integer(1000));

        Collections.sort(lista, new Compara());

        System.out.println(lista);

    }

}

class Compara implements Comparator {

    public int compare(Object obj1, Object obj2) {

        int i1 = ((Integer)obj1).intValue();

        int i2 = ((Integer)obj2).intValue();
```

```

        if (i1 == i2) return 0;

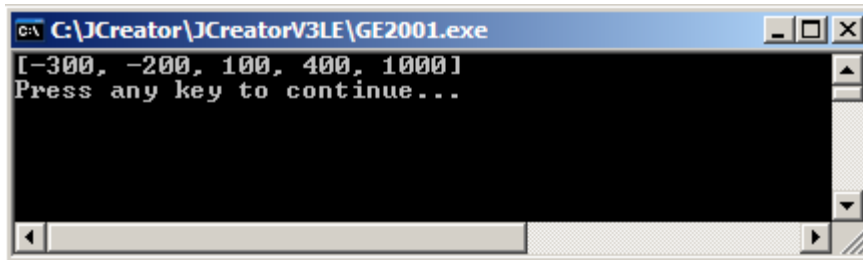
        else if (i1 < i2) return -1;

        return 1;

    }

}

```



A figura ao lado apresenta o resultado da execução do código.

Modificando o método *compare()* para a forma abaixo, você fará uma classificação em ordem decrescente.

```

class Compara implements Comparator {

    public int compare(Object obj1, Object obj2) {

        int i1 = ((Integer)obj1).intValue();

        int i2 = ((Integer)obj2).intValue();

        if (i1 == i2) return 0;

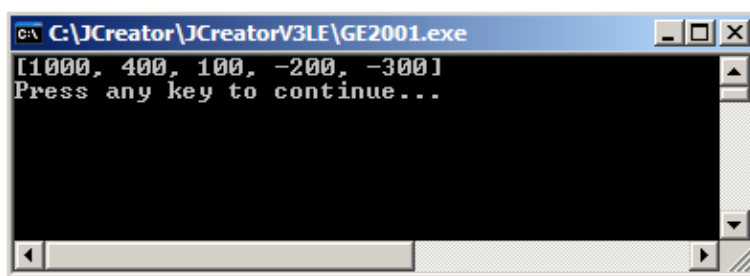
        else if (i1 < i2) return 1;

        return - 1;

    }

}

```



A figura ao lado apresenta o novo resultado da execução.

20.3.2 *shuffle()*:

O método *shuffle()* promove o embaralhamento dos elementos de uma lista. Como esta operação utiliza técnicas de randomização este método torna-se muito útil na implementação de alguns jogos bem populares como cartas ou bingos.

No código abaixo são gerados sequencialmente cinquenta e dois números inteiros que são submetidos ao método *shuffle()*. São realizadas duas execuções consecutivas da mesma classe para que você observe os resultados diferentes de embaralhamento que são produzidos.

```
import java.util.ArrayList;

import java.util.Collections;

public class TestaFiles {

    public static void main(String[] args) {

        ArrayList<Integer> lista = new ArrayList<Integer>();

        for (int i=1; i<=52; i++){

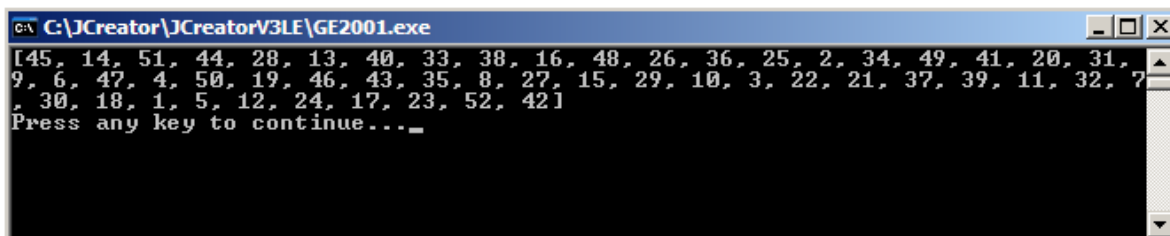
            lista.add(new Integer(i)); }

        Collections.shuffle(lista);

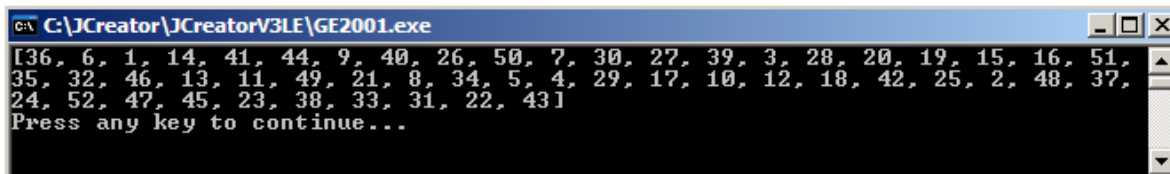
        System.out.println(lista);

    }

}
```



1ª execução.



2ª execução.

20.3.3 *max()* e *min()*:

Os métodos *max()* e *min()* retornam, respectivamente, os valores máximo e mínimo de determinada lista. No código abaixo são apresentadas as maiores e menores notas de determinada turma.

```
import java.util.ArrayList;

import java.util.Collections;

public class TestaFiles {

    public static void main(String[] args) {

        ArrayList<Double> lista = new ArrayList<Double>();

        lista.add(new Double(8));

        lista.add(new Double(10));

        lista.add(new Double(7.5));

        lista.add(new Double(2));

        lista.add(new Double(1.5));

        lista.add(new Double(9));

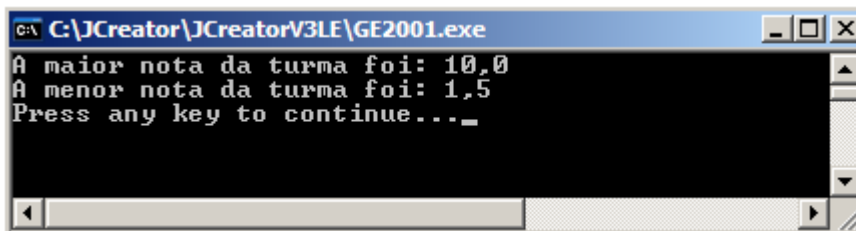
        lista.add(new Double(10));

        System.out.printf("%s%.1f\n", "A maior nota da turma foi: ", Collections.max(lista));

        System.out.printf("%s%.1f\n", "A menor nota da turma foi: ", Collections.min(lista));

    }

}
```



A figura ao lado apresenta o resultado da execução do código.

20.3.3 *frequency()*:

O método *frequency()* permite investigar com que frequência determinado objeto aparece em determinada lista.

No código abaixo são verificadas quantas notas dez e quantas notas zero ocorreram em determinado teste.

```
import java.util.ArrayList;

import java.util.Collections;

public class Frequencia {

    public static void main(String[] args) {

        ArrayList<Double> lista = new ArrayList<Double>();

        lista.add(new Double(8));

        lista.add(new Double(10));

        lista.add(new Double(7.5));

        lista.add(new Double(2));

        lista.add(new Double(0));

        lista.add(new Double(9));

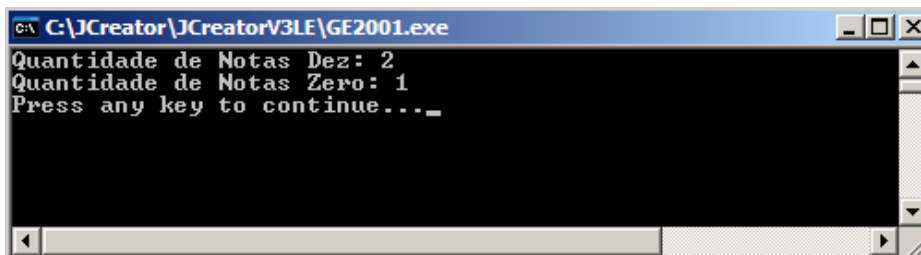
        lista.add(new Double(10));

        System.out.printf("%s%d\n", "Quantidade de Notas Dez: ", Collections.frequency(lista, new Double(10)));

        System.out.printf("%s%d\n", "Quantidade de Notas Zero: ", Collections.frequency(lista, new Double(0)));

    }

}
```



A figura ao lado apresenta o resultado da execução do código.

20.3.4 copy():

Com o método ***copy()*** você poderá copiar o conteúdo de uma lista para outra, sendo que a lista destino deverá possuir, no mínimo, o mesmo número de elementos que a lista de origem. Caso a lista de saída possua mais elementos que a lista de entrada eles serão preservados durante o processo de cópia.

No código abaixo são criadas uma lista origem, de nome entrada, com seis elementos, e uma lista destino, de nome saída, com sete elementos. Em seguida, a cópia é realizada.

```
import java.util.*;

public class Copiar {

    public static void main(String[] args) {

        ArrayList<String> entrada = new ArrayList<String>();

        ArrayList<String> saida = new ArrayList<String>();

        entrada.add("Ana");

        entrada.add("Rose");

        entrada.add("Solange");

        entrada.add("Marise");

        entrada.add("Raquel");

        entrada.add("Laila");

        saida.add("Kiko");

        saida.add("Claudio");

        saida.add("Pedro");

        saida.add("Guilherme");

        saida.add("Daniel");

        saida.add("Marcelo");

        saida.add("Rafael");

        Collections.copy(saida, entrada);

        System.out.println(saida);

        System.out.println( entrada);
```

```

    }
}

```

```

C:\JCreator\JCreatorV3LE\GE2001.exe
[Ana, Rose, Solange, Marise, Raquel, Laila, Rafael]
Press any key to continue...

```

A figura ao lado ilustra a execução do código. Observe que o nome “Rafael” foi preservado na lista destino.

20.3.5 *fill()*:

O método *fill()* é utilizado para preencher uma lista com um valor específico. As listas já devem estar previamente preenchidas para a operação ser realizada.

No código a seguir uma determinada lista é preenchida com a string “Ângélica”.

```

import java.util.*;

public class Preenche {

    public static void main(String[] args) {

        ArrayList<String> entrada = new ArrayList<String>();

        entrada.add("Ana");

        entrada.add("Rose");

        entrada.add("Solange");

        entrada.add("Marise");

        entrada.add("Raquel");

        entrada.add("Laila");

        Collections.fill(entrada, "Angelica");

        System.out.println( entrada);

    }

}

```

```

C:\JCreator\JCreatorV3LE\GE2001.exe
[Angelica, Angelica, Angelica, Angelica, Angelica, Angelica]
Press any key to continue...

```

A figura ao lado ilustra o preenchimento da lista.

20.3.6 ***binarySearch()***:

O algoritmo de pesquisa binária é, por sua eficiência, muito utilizado quando você pretende localizar algum elemento em uma lista já previamente classificada. Neste caso, a classe ***Collections*** oferece o método ***binarySearch()***. Este método retorna, caso o elemento pesquisado seja encontrado, o índice da posição deste elemento. Caso o elemento não seja encontrado o método retorna um valor negativo. Se existirem mais de um elemento com o mesmo valor o método não garante qual dos índices será retornado.

No código abaixo é criada uma lista com vários nomes e são pesquisados alguns deles.

```
import java.util.*;

public class TestaFiles {

    public static void main(String[] args) {

        ArrayList<String> entrada = new ArrayList<String>();

        entrada.add("Ana");

        entrada.add("Rose");

        entrada.add("Laila");

        entrada.add("Solange");

        entrada.add("Marise");

        entrada.add("Raquel");

        entrada.add("Laila");

        Collections.sort(entrada);

        System.out.printf("%s\n", entrada);

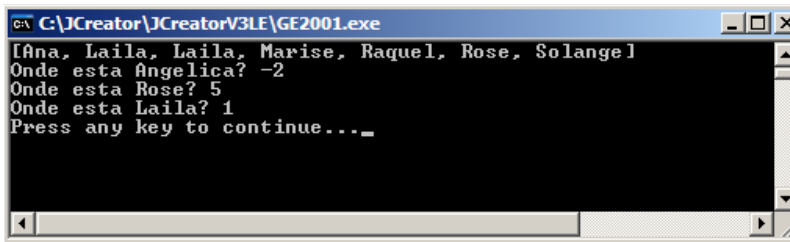
        System.out.printf("%s%d\n", "Onde esta Angelica?",
        Collections.binarySearch(entrada, "Angelica"));

        System.out.printf("%s%d\n", "Onde esta Rose? ",
        Collections.binarySearch(entrada, "Rose"));

        System.out.printf("%s%d\n", "Onde esta Laila? ",
        Collections.binarySearch(entrada, "Laila"));

    }
}
```

```
}
```



A figura ao lado ilustra a execução do código. Observe que antes de a lista ser pesquisada ela é ordenada pelo método *sort()*.

20.3.7 *disjoint()*:

O método *disjoint()* verifica se duas listas possuem todos os elementos com valores diferentes entre si, caso isto seja verdadeiro ele retorna o valor booleano *true*, caso contrário é retornado o valor booleano *false*.

No código a seguir são criadas três listas que são comparadas duas a duas.

```
import java.util.*;

public class NaoExiste {

    public static void main(String[] args) {

        ArrayList<String> lista1 = new ArrayList<String>();

        ArrayList<String> lista2 = new ArrayList<String>();

        ArrayList<String> lista3 = new ArrayList<String>();

        lista1.add("Ana");

        lista1.add("Rose");

        lista1.add("Laila");

        lista2.add("Solange");

        lista2.add("Marise");

        lista2.add("Raquel");

        lista3.add("Laila");

        lista3.add("Maria");

        lista3.add("Eduarda");
```

```

        lista3.add("Fernanda");

        lista3.add("Joana");

        System.out.printf("%s%B\n", "Lista1 com Lista2 ", Collections.disjoint(lista1,
        lista2));

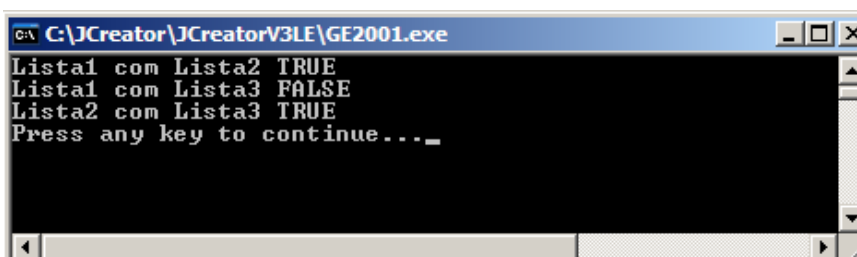
        System.out.printf("%s%B\n", "Lista1 com Lista3 ", Collections.disjoint(lista1,
        lista3));

        System.out.printf("%s%B\n", "Lista2 com Lista3 ", Collections.disjoint(lista2,
        lista3));

    }

}

```



A figura ao lado ilustra a execução do código. Observe que as listas não precisam estar previamente classificadas e nem precisam ter a mesma

dimensão.

20.4 Classe Stack:

A classe **Stack**, integrante do pacote `java.util`, herda da classe **Vector** e implementa a estrutura de dados do tipo pilha. Os métodos apresentados a seguir são oferecidos pela classe **Stack**:

<i>Método</i>	<i>Descrição</i>
<code>pop()</code>	Remove um objeto do início da pilha.
<code>push(objeto)</code>	Adiciona um objeto na pilha.
<code>isEmpty()</code>	Verifica se a pilha está vazia.
<code>peek()</code>	Retorna o elemento do topo da pilha sem removê-lo.

Caso você tente retirar um elemento de uma pilha vazia será lançada a exceção do tipo **EmptyStackException**.

O código a seguir insere nomes em uma pilha que é esvaziada até atingir o seu final.

```
import java.util.*;

public class Pilha {

    public static void main(String[] args) {

        Stack<String> pilha = new Stack<String>();

        pilha.push("Maria");

        pilha.push("Eduarda");

        pilha.push("Raquel");

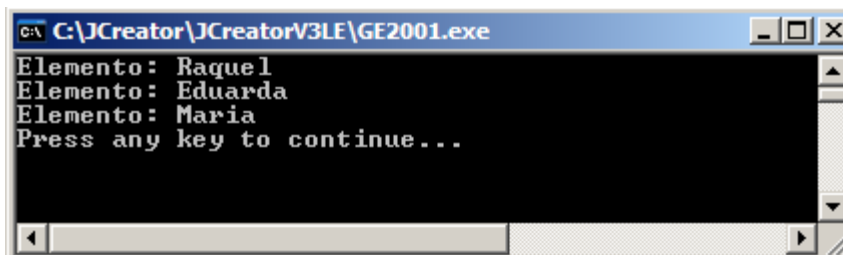
        while(!pilha.isEmpty()){

            System.out.printf("%s%s\n", "Elemento: ", pilha.pop());

        }

    }

}
```



A figura ao lado ilustra a execução do código. Observe que os nomes são apresentados na ordem inversa em que foram

inseridos, caracterizando uma estrutura de dados do tipo pilha.

20.5 Um novo for:

Na versão 1.5 do Java o time da Sun implementou uma nova funcionalidade para a estrutura de repetição **for**, voltada para facilitar a pesquisa em *arrays* e nas diversas listas existentes. Abaixo é apresentada a sintaxe deste **for** aprimorado:

for (parâmetro: nome_do_array ou nome_da_lista)

O parâmetro especifica o tipo e o nome da variável em que estamos interessados em recuperar, seja no *array* ou na lista.

No código abaixo é criada uma lista e através do comando ***for*** os seus elementos são recuperados e impressos.

```
import java.util.*;

public class NovoFor {

    public static void main(String[] args) {

        ArrayList<String> lista = new ArrayList<String>();

        lista.add("Maria");

        lista.add("Eduarda");

        lista.add("Raquel");

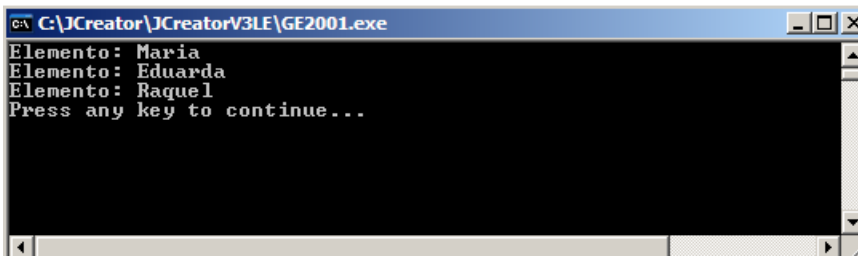
        for(String nome:lista){

            System.out.printf("%s%s\n", "Elemento: ", nome);

        }

    }

}
```



A figura ao lado ilustra a execução do código, onde pode ser constatado que esta outra forma de escrevermos o ***for***

ficou bem mais atraente.

20.6 Mapas:

Os mapas têm como propósito estabelecer uma relação chave valor. As chaves não podem ser duplicadas diferentemente dos valores que podem ser iguais. Na linguagem Java

os mapas são constituídos de três classes que implementam a interface **Map**: **HashMap**, **Hashtable**, e **TreeMap**. Veremos aqui as classes **HashMap** e **Hashtable**.

20.6.1 Classes *HashMap* e *Hashtable*:

As classes **HashMap** e **Hashtable** são muito parecidas em suas funcionalidades. A principal diferença é que o acesso a classe **HashMap** é feito de forma assíncrona enquanto que na classe **Hashtable** o acesso é síncrono. Outra característica destas classes é que ambas suportam valores nulos.

As seguintes formas de construção são suportadas:

- para a **HashMap**

HashMap<Tipo-da-Chave, Tipo-do-Valor> nome-do-objeto = new HashMap<Tipo-da-Chave, Tipo-do-Valor>()

- para a **Hashtable**

Hashtable<Tipo-da-Chave, Tipo-do-Valor> nome-do-objeto = new Hashtable<Tipo-da-Chave, Tipo-do-Valor>()

As classes **HashMap** e **Hashtable** possuem os seguintes métodos em destaque:

<i>Método</i>	<i>Descrição</i>
<code>clear()</code>	Limpa todo o conteúdo do mapa.
<code>containsKey(objeto chave)</code>	Retorna <i>true</i> se o mapa contém a chave especificada.
<code>containsValue(objeto valor)</code>	Retorna <i>true</i> se o mapa contém o valor especificado.
<code>get(objeto chave)</code>	Retorna o valor mapeado pela chave.
<code>isEmpty()</code>	Retorna <i>true</i> se o mapa estiver vazio.
<code>put(chave, valor)</code>	Associa um valor para determinada chave.
<code>remove(objeto chave)</code>	Remove o mapeamento de determinada chave.
<code>size()</code>	Informa o número de relacionamentos no mapeamento.
<code>values()</code>	Retorna uma coleção de valores do mapeamento.

No código a seguir um mapa é preenchido com as matrículas e as respectivas notas dos alunos de determinada turma. Em seguida são feitos alguns testes de verificação no mapeamento.

```
import java.util.*;

public class TestaFiles {

    public static void main(String[] args) {

        HashMap<Integer, Double> lista = new HashMap<Integer, Double>();

        lista.put(20051, 9.0);

        lista.put(20052, 7.5);

        lista.put(20055, 8.0);

        lista.put(20054, 9.5);

        lista.put(20053, 10.0);

        System.out.println(lista);

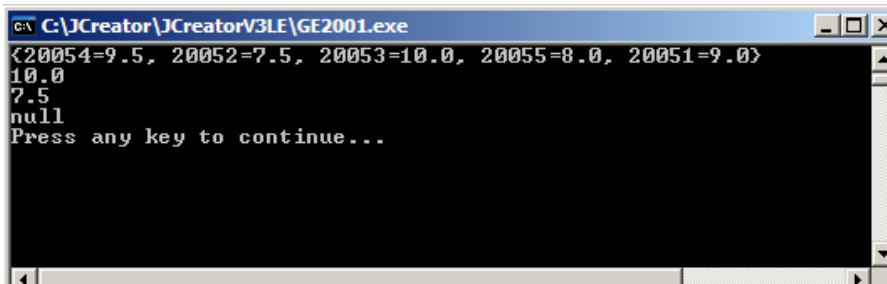
        System.out.println(lista.get(20053));

        System.out.println(lista.remove(20052));

        System.out.println(lista.get(20052));

    }

}
```



A figura ao lado ilustra o resultado da execução do código. É apresentado, também, o conteúdo da matrícula 20053 e removido o conteúdo da matrícula 20052.

21. Sockets:

Uma das maneiras existentes para você se comunicar com outra máquina é utilizar uma conexão do tipo *socket*. Para realizar esta conexão você precisa saber duas coisas

sobre a máquina com que você deseja se comunicar: quem ela é e em qual porta ela estará ouvindo você. Estas duas informações são representadas, respectivamente, pelo endereço IP e pelo número da porta TCP.

A máquina que fica esperando a conexão pode ser denominada de máquina-servidora e a máquina que estabelecerá a conexão como esta máquina-servidora é denominada máquina-cliente.

Então, os seguintes passos precisam ser seguidos para o estabelecimento de uma conexão entre duas máquinas:

- É iniciada na máquina servidora a aplicação que fica “ouvindo” o meio, esperando as mensagens cheguem através de determinada porta.
- A máquina cliente tenta se conectar com a máquina servidora informando o endereço IP e a porta.
- A máquina servidora aceita o pedido de comunicação e estabelece a conexão.
- A máquina cliente envia uma mensagem para a máquina servidora.
- A máquina cliente recebe uma mensagem resposta da máquina servidora.

Obviamente, a máquina servidora para enviar a resposta de volta para a máquina cliente também precisará saber o endereço IP e a porta desta máquina. Então podemos concluir que em uma conexão do tipo *socket* todas as máquinas envolvidas precisam ter informações sobre os endereços IP e sobre as portas.

Existem duas classes na linguagem Java que tratam destas conexões: a classe **Socket** e a classe **ServerSocket**.

A classe **ServerSocket** possui o seguinte construtor:

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

Como o nome já deixa claro esta classe é para ser inserida na aplicação da máquina servidora. Esta classe possui o método *accept()* que possibilitará à máquina servidora não bloquear a porta caso outra conexão esteja chegando. Com este método outra porta é automaticamente disponibilizada para a máquina servidora poder falar com a máquina cliente, permitindo, desta forma, conexões concomitantes.

A classe **Socket** possui um dos seguintes construtores:

```
Socket conecta = new Socket(endereçoIP, porta)
```

Existe um total de 65536 portas (0 a 65535), sendo que algumas destas portas já estão previamente definidas para algumas aplicações tais como: Telnet(porta 23), SMTP(porta 25), FTP(porta 20), HTTP(porta 80), POP3(porta 110) e HTTPS(porta 443).

À nossa disposição estão as portas de numeração acima de 1024. A utilização de outras portas necessitará de autorização especial do administrador do sistema em que a aplicação será executada.

A sua aplicação ficará ouvindo o meio sempre a partir de uma porta, portanto ela ficará sendo a “dona” desta porta enquanto estiver ativa. A aplicação, explicitamente, diz qual porta utilizará.

Para escrever dados para uma conexão do tipo **Socket** é utilizada a classe **PrintWriter** e para ler os dados são utilizadas as classes **InputStreamReader**, que estabelecerá a ponte com o **Socket**, e **BufferedReader**, que efetivamente lerá os dados.

Você precisa, então, das seguintes instruções para ler caracteres de uma máquina:

- **InputStreamReader** ponte = new **InputStreamReader**(conecta.getInputStream());
- **BufferedReader** ler = new **BufferedReader**(ponte);
- String mensagem = ler.readLine();

Para enviar dados para uma outra máquina as instruções são:

- **PrintWriter** escreve = new **PrintWriter**(conecta.getOutputStream());
- escreve.println(“mensagem a enviar #1”);
- escreve.println(“mensagem a enviar #2”);
- escreve.println(“mensagem a enviar #3”);

Observe que você precisará utilizar a classe **Socket** tanto na máquina servidora quanto na máquina cliente uma vez que ambas escrevem e/ou lêem em um processo de comunicação.

Em nosso primeiro exemplo são codificadas duas classes **SocketServidor** e **SocketCliente**.

A classe **SocketServidor** realiza as seguintes tarefas:

- Indica que será uma aplicação servidora.
- Fica ouvindo o meio através da porta 4000.

- Fica testando se alguma máquina cliente quer se conectar.
- Se alguma máquina cliente se conectou, envia uma mensagem.

A classe **SocketCliente** realiza as seguintes tarefas:

- Tenta se conectar à máquina servidora através do endereço IP 127.0.0.1, porta 4000.
- Caso conectado lê os dados oriundos da máquina servidora.
- Imprime os dados lidos.

Código da classe SocketServidor:

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

public class SocketServidor {

    public void manda(){

        System.out.println("Servidor ouvindo pela porta 4000");

        try{

            ServerSocket servidor = new ServerSocket(4000);

            for(;;){

                Socket conecta = servidor.accept();

                PrintWriter escreve = new PrintWriter(conecta.getOutputStream());

                String mensagem = "Falando pela rede";

                escreve.println(mensagem);

                escreve.close();

            }

        }

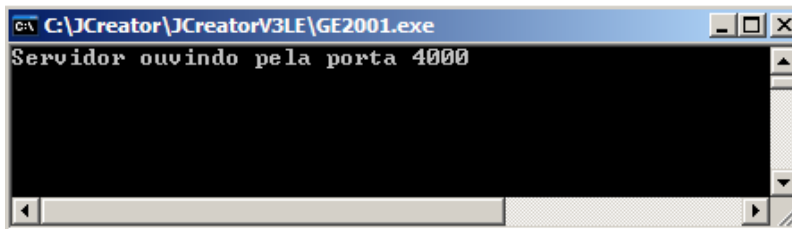
        catch(IOException e){

            System.out.println("Não foi");}

    }

}
```

```
    }  
  
    public static void main(String[] args){  
  
        SocketServidor dispara = new SocketServidor();  
  
        dispara.manda();  
  
    }  
}
```

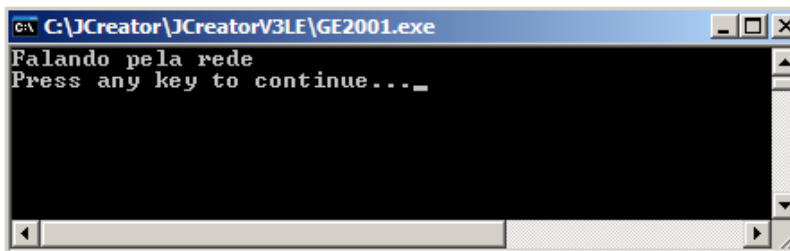


A figura ao lado ilustra a execução da classe `ServidorSocket`.

Código da classe `SocketCliente`:

```
import java.io.*;  
  
import java.net.*;  
  
public class SocketCliente {  
  
    public void manda(){  
  
        try{  
  
            Socket cliente = new Socket("127.0.0.1", 4000);  
  
            InputStreamReader ponte = new  
                InputStreamReader(cliente.getInputStream());  
  
            BufferedReader ler = new BufferedReader(ponte);  
  
            String mensagem = ler.readLine();  
  
            ler.close();  
  
            System.out.println(mensagem);  
  
        }  
    }  
}
```

```
        catch(IOException e){  
            System.out.println("Não recebeu");  
        }  
  
    public static void main(String[] args){  
        SocketCliente dispara = new SocketCliente();  
        dispara.manda();  
    }  
}
```



O resultado da execução da classe SocketCliente está representado na figura ao lado.