

## Capítulo 01

### **O que é o Java ?**

O Java é:

- \* Uma linguagem de programação;
- \* Um ambiente de desenvolvimento;
- \* Um ambiente de aplicação;

O Java é resultado de uma busca por uma linguagem de programação que pudesse fornecer uma ligação com o C++, mas com segurança.

Os primeiros objetivos alcançados com desenvolvimento desta nova linguagem foram:

- \* Criação de uma linguagem orientada a objetos;
- \* Fornecimento de um ambiente de desenvolvimento por dois motivos:
  - \* Velocidade no desenvolvimento - eliminando o ciclo de compilar-linkar-carregar-testar;
  - \* Portabilidade do Código - com um interpretador que especifica a forma do nível do sistema operacional. ( Pode rodar em qualquer tipo de sistema operacional);
- \* Não tem acesso a ponteiros do sistema operacional;
- \* Fornece dinamismo durante a manutenção de programas;

### **Garbage Collection (Libera uma coleção)**

O Java não segura áreas de memória que não estão sendo utilizadas, isto porque ele tem uma alocação dinâmica de memória em tempo de execução.

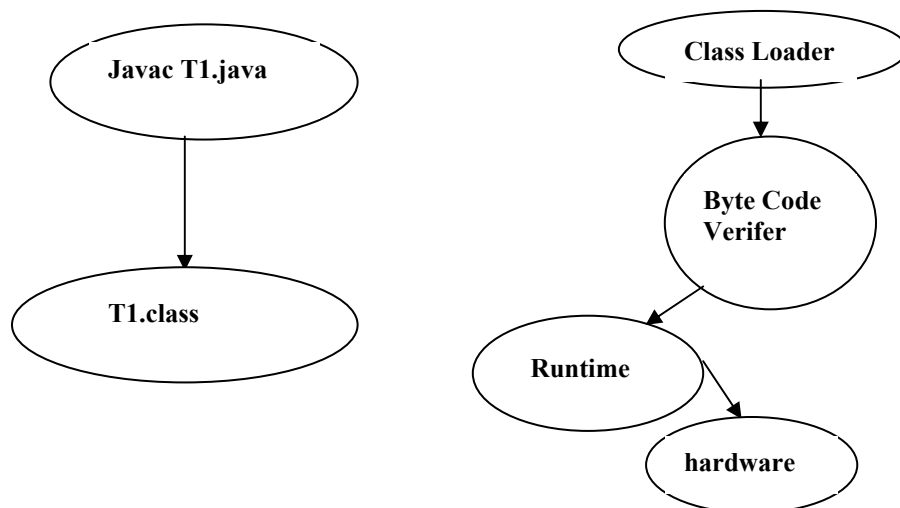
No C e C++ (e em outras linguagens) o programa desenvolvido é responsável pela alocação e desalocação da memória.

Durante o ciclo de execução do programa o Java verifica se as variáveis de memória estão sendo utilizadas, caso não estejam o Java libera automaticamente esta área que não está sendo utilizada.

## Segurança no Código

Os arquivos do Java são compilados e são convertidos de arquivos texto para um formato que contém blocos independentes de bytes codes (Código Intermediário).

Em tempo de execução estes bytes codes são carregados, são verificados através do Byte Code Verifier (uma espécie de segurança), passam a seguir para o interpretador e são executados. Caso este código seja acionado diversas vezes, existe um passo chamado JIT Code Generator, que elimina o utilização por demasia do tráfego da rede.



Abra o Notepad e crie o seguinte programa. Salve-o como **Prog0101.java**

```
class Prog0101
{
    public static void main (String arg [])
    {
        int a = 5, b = 10;
        a = a + 5;
        System.out.println("Meu Primeiro Progama");
        System.out.println(" O valor da variavel a = " + a);
    }
}
```

Após terminar o programa compile-o e execute-o:

```
C:\.....\Javac Progr0101.java
```

```
C:\.....\ Java Prog0101
```

## Capítulo 02

### **Comentários**

Estes são os três tipos permitidos de comentários nos programas feitos em Java:

```
// comentário de uma linha
/* comentário de uma ou mais linhas */
/** comentário de documentação */ (Arquivos de documentação)
```

### **Ponto e vírgula, Blocos e o espaço em branco**

- \* No java, os comandos são terminados com o sinal de ponto e vírgula (;)
- \* Um bloco tem início e tem o seu fim representados pelo uso das chaves {};
- \* O uso do espaço em branco permite uma melhor visualização dos comandos e em consequencia facilita a sua manutenção.

### **Identificadores**

Na linguagem Java um identificador é startado com uma letra, underscore ( \_ ), ou sinal de dólar (\$), e existe uma diferenciação entre letras maiúsculas e minúsculas:

Identificadores válidos:

```
* identifier
* userName
* User_name
* _sys_var1
*$change
```

### **Tipos Básicos no Java**

No Java existem oitos tipos básicos e um tipo especial.

#### *Tipo Lógico*

- boolean: on e off; true e false ou yes e no.

#### *Tipo Textual*

- char e String

Um caracter simples usa a representação do tipo char. O tipo char representa na forma Unicode um caracter de 16-bit.

O literal do tipo char pode ser representado com o uso do (' ').

'\n' – nova linha  
'\r' – enter  
'\u????' – especifica um caracter Unicode o qual é representado na forma Hexadecimal.  
'\t' – tabulação  
'\ ' - \  
'" ' - ""

O tipo String, como não é primitivo, é usado para representar uma seqüência de caracteres.

#### Palavras Reservadas

abstract	do	implements	private	throw
boolean	double	import	protected	throws
break	else	instanceof	public	transient
byte	extends	int	return	true
case	false	interface	short	try
catch	final	long	static	void
char	finally	native	super	volatile
class	float	new	switch	while
continue	for	null	synchronized	
default	if	package	this	

#### Tipo Integral – byte, short, int e long

Existem quatro tipos de integral:

Tamanho da Integral	Nome ou Tipo	Espaço
8 bits	byte	$-2^7 \dots 2^7 - 1$
16 bits	short	$-2^{15} \dots 2^{15} - 1$
32 bits	int	$-2^{31} \dots 2^{31} - 1$
64 bits	long	$-2^{63} \dots 2^{63} - 1$

### *Tipo Ponto Flutuante*

Uma variável do tipo ponto flutuante pode ser declarada usando a palavra *float* ou *double*.

3.14	Um ponto flutuante simples;
6.02E23	Um valor de ponto flutuante largo;
2.718F	Um valor de ponto flutuante simples;
123.4E+306D	Um valor de ponto flutuante usando o tipo <i>double</i> .

### **Convenção de Código no Java**

*Class* - Nomes de classes podem ser maiúsculas ou minúsculas ou misturado (maiúsculas e minúsculas), **mas por convenção o nome das classes começam por letra maiúsculas.**;

*Interfaces* - Nomes de Interfaces suportam nomes iguais aos das classes;

*Métodos* - Nomes de métodos podem ser verbos, podendo misturar entre maiúsculas e minúsculas, **sendo entretando a primeira letra maiúscula**;

*Constantes* - Nomes de constantes podem ser maiúsculas, minúsculas, misturadas, separadas com underscores.

*Variáveis* - Todas as instancias, classes e variáveis globais suportam maiúsculas e minúsculas.

*Controles de Estruturas*- Convencionou-se o uso de { } (chaves);

*Espaços* - Convencionou-se o uso de quatro espaços para identações;

*Comentários* - Use os comentários para explicar os segmentos de código que não são obvios.

## Exercícios:

### Exercício 01

```
class Prog0201
{
    public static void main (String arg [])
    {   int a = 5, b = 10;
        a = a + 5;
        // b = b*2;
        System.out.println("Valor da variavel a: " + a);
        System.out.println(" Valor da variavel b: " + b);
    }
}
```

### Exercício 02

```
class Prog0202
{
    public static void main (String arg [])
    {   int a = 5, b = 10;
        a = a + 5;
        b = b*2;
        System.out.println("Valor da variavel a: " + a);
        System.out.println(" Valor da variavel b: " + b);
    }
}
```

## Capítulo 03

### ***Variáveis e Tempo de vida***

Você tem dois meios para descrever variáveis: usando o tipo simples de ligação *int* e *float* ou usando tipos de classes definidas pelo programa. Você pode declarar variáveis de duas formas, uma dentro de um método e a outra dentro da classe a qual este método está incluído.

### ***Inicialização de variáveis***

No Java não é permitido o uso de variáveis indefinidas.

Variáveis definidas dentro do método são chamadas de variáveis automáticas, locais, temporárias ou estáticas e devem ser inicializadas antes do uso.

Quando um objeto é criado, as variáveis membro são inicializadas com os seguintes valores em tempo de alocação:

<b>Tipo de variável</b>	<b>Valor inicial</b>	<b>Tamanho</b>
byte	0	8 bits
short	0	16 bits
int	0	32 bits
long	0L	64 bits
float	0.0f	32 bits
Double	0.0d	64 bits
Char	'\u0000' (Null)	64 bits
Boolean	false	

### ***Operadores***

No Java os operadores são muito similares ao estilo e funcionalidade de outras linguagens como por exemplo o C e o C++.

Pré-incremento:

```
x = 10;  
x = x + 1;  
O valor da variável x é 11
```

ou

```
x = 10;  
++x
```

O valor da variável x é 11.

Pós-Incremento:

```
x = 10;  
x = x + 1;  
O valor da variável x é 11
```

ou

```
x = 10;  
x++;  
O valor da variável x é 11.
```

Diferença entre o Pré-Incremento e o Pós-Incremento:

```
x = 10  
++x => neste exato momento a variável a vale 11
```

```
x = 10  
x++ => neste exato momento a variável x vale 10
```

Separadores:

. [ ] ( ) ; ,

Operadores:

Operadores	Descrição
==	Igualdade
!=	Negação
+ - * /	Aritméticos
&&	e
	ou

### **Concatenação**

O operador + é utilizado para concatenar objetos do tipo String, produzindo uma nova String:

```
String PrimeiroNome = "Antonio";  
String SegundoNome = "Carlos";  
String Nome = PrimeiroNome + SegundoNome;
```

### **Casting ( Conversão de tipos )**

A linguagem Java não suporta conversões arbitrárias de tipos de variáveis. Você deve explicitar a conversão entre tipos de variáveis.

Exemplo:

```
long bigval = 6; // Operação válida
int smallval = 99L; // Operação inválida porque são de tipos diferentes

float z = 12.414F; // Operação válida
float zp = 12.414; // Operação inválida porque esta tentando atribuir um valor
double.
```

Convertendo

```
Long bigValue = 99L;
Int squashed = (int)(bigValue);
```

### **Controle de Fluxos**

Declarações de Fluxos

#### ***If, else***

```
if (expressão) // expressão cujo retorno é um valor do tipo boolean
{ Declarações ou blocos }
else // caso a condição anterior não seja satisfeita
{ Declarações ou blocos }
```

#### ***switch***

```
switch (expressão) // Esta expressão deve ser do tipo int ou char
{
    case cond01:
        declarações;
        break; // usado para sair do case.
    case cond02:
        declarações;
        break;
    case cond03:
        declarações;
        break;
}
```

#### ***for Loops***

```
for (expr_inicial; condição_boolean; incremento)
{
    Declarações ou blocos;
}
```

### ***while Loops***

```
while(condição_boolean)
{
    Declarações ou blocos;
}
```

### ***do Loops***

```
do
{
    Declarações ou blocos;
}
while(condição_boolean);
```

## Exercícios

### Exercício 01

```
public class Prog0301
{
    public static void main(String arg[ ])
    {
        int x = (int)(Math.random()*100);
        int z ;
        int y ;

        if (x > 50)
        {
            y = 9;
        }
        z = y + x;
    }
}
```

### Exercício 02

```
public class Prog0302
{
    public static void main(String arg[ ])
    {
        int x = (int)(Math.random()*100);
        int z = 0;
        int y = 0;

        if (x > 50)
        {
            y = 9;
        }
        else
        {
            System.out.println("O valor de x é menor que 50");
            y = (int)(Math.random() * (3.14));
        }
        z = y + x;
        System.out.println("O valor da variável x = " + x);
        System.out.println("O valor da variável y = " + y);
    }
}
```

```
        System.out.println("O valor da variável z = " + z);
    }
}
```

#### Exercício 03

```
public class Prog0303
{
    public static void main(String arg[])
    {
        int valor = (int)(Math.random()*5);
        switch(valor)
        {
            case 0:
                System.out.println("Primeira Opção (Valor igual a zero)");
                break;
            case 1:
                System.out.println("Segunda Opção (Valor igual a um)");
                break;
            default:
                System.out.println("Outras Opções (Valor maior que um)");
                break;
        }
    }
}
```

#### Exercício 04

```
public class Prog0304
{
    public static void main(String arg[])
    {
        int valor = (int)(Math.random()*5);
        while(valor <> 0)
        {
            valor = (int)(Math.random()*5);
            System.out.println("Valor igual a " + valor);
        }
    }
}
```

### ***Declaração de Arrays***

```
char s [ ];  
Point p [ ];
```

Em Java um Array é uma classe.

### ***Criando um Array***

Você pode criar arrays, ligando-o a todos os objetos, usando a palavra new, da seguinte forma:

```
s = new char[20];  
p = new Point[100];
```

```
String names[ ];  
names = new String[4];  
names[0]= "Georgina";  
names[1]="Jen";  
names[2]="Simon";  
names[3]= "Tom";
```

ou

```
String names[ ];  
names = new String[4];  
String names [ ] = { "Georgina", "Jean", "Simon", "Tom"};
```

### ***Arrays Multi-dimensionais***

Java não possui arrays multi-dimensionais, mas ele permite declarar um array que é baseado em um outro array.

```
int twoDim [ ] [ ] = new int [4] [ ] ;  
twoDim[0] = new int [5] ;  
twoDim[1] = new int [5] ;
```

## Exercícios

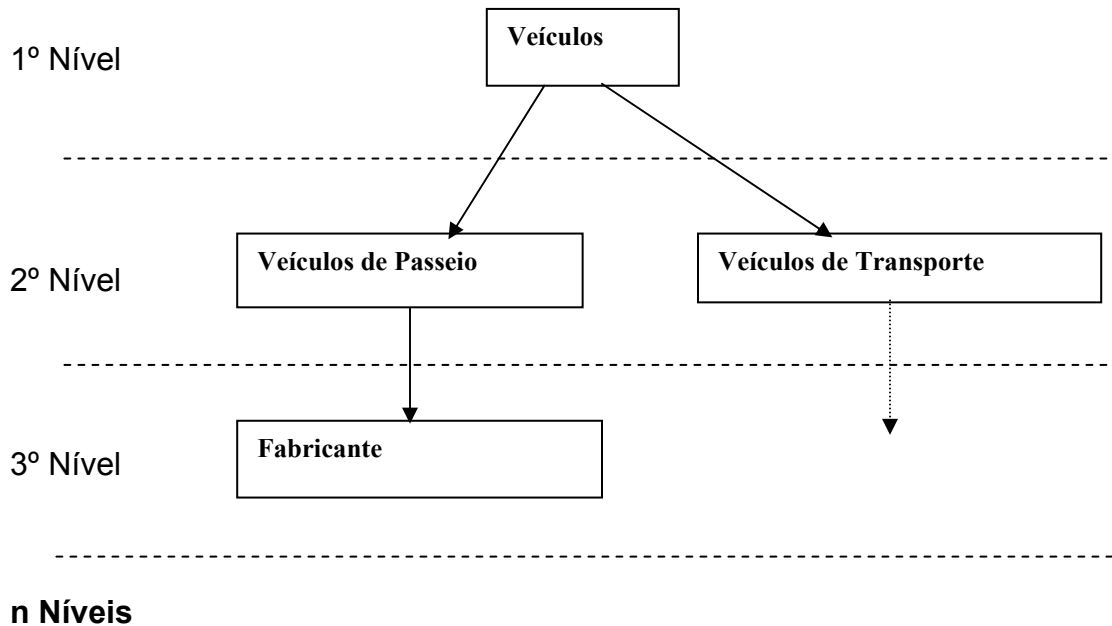
```
classe Prog0401
{
    thisArray int;
    thatArray int;
    public static void main (String args [ ])
    {
        int[ ] thisArray = {1,2,3,4,5,6,7,8,9,10}
        for (int i =0; i < thisArray.length; i++)
        {
            System.out.println( " Conteúdo do Array " + thisArray[i]);
        }
    }
}
```

```
class Prog0402 {

    public static void main(String arg[ ])
    {
        int A[] = new int[3];
        A[0] = 50;
        A[1] = 100;
        A[2] = 150;
        System.out.println("Tamanho do vetor = " + A.length);

        int conta;

        for(conta = 0; conta < A.length ; conta++)
        {System.out.println("indice = " + conta + " valor= " + A[conta] );}
    }
}
```



### ***Abstração de Tipos de Dados***

Quando itens de dados são compostos para tipos de dados, semelhante a uma data, você pode definir um número de bits de programas que especifica a operação do tipo de dados.

Em Java você pode criar uma associação entre o tipo data e a operação tomorrow a seguir:

```
public class Date {  
    private int day, month, year;  
    public void tomorrow( )  
    { // código que incrementa o dia  
    }  
}
```

### ***Definição de Métodos***

Em Java, métodos são definidos usando uma aproximação que é muito similar à usada em outras linguagens, como por exemplo C e C++. A declaração é feita da seguinte forma:

< modifiers > < tipo de retorno > < nome > ( < lista de argumentos > ) < bloco >

< modifiers > -> segmento que possui os diferentes tipos de modificações incluindo *public*, *protected* e *private*.

< tipo de retorno > -> indica o tipo de retorno do método.

< nome > -> nome que identifica o método.

< lista de argumentos > -> todos os valores que serão passados como argumentos.

```
public void addDays (int days)
```

### **Passagem de Valores**

Em Java o único argumento passado é “by-value”; este é um argumento *may not be changed* do método chamado. Quando um objeto é criado, é passado um argumento para o método e o valor deste argumento é uma referência do objeto. O conteúdo do objeto passível de alteração é chamado do método, mas o objeto referenciado jamais é alterado.

### **A Referência This**

É aplicado a métodos não estáticos.

O Java associa automaticamente a todas as variáveis e métodos referenciados com a palavra *this*. Por isso, na maioria dos casos torna-se redundante o uso em todas as variáveis da palavra *this*.

Existem casos em se faz necessário o uso da palavra *this*. Por exemplo, você pode necessitar chamar apenas uma parte do método passando uma instância do argumento do objeto. (Chamar um classe de forma localizada);

```
    Birthday bDay = new Birthday(this);
```

### **Ocultando Dados**

Usando a palavra *private* na declaração de *day*, *month* e *year* na classe *Date*, você impossibilitará o acesso a estes membros de um código fora desta classe. Você não terá permissão para atribuir valores, mas poderá comparar valores.

### **Encapsulamento**

É uma proteção adicional dos dados do objeto de possíveis modificações impróprias, forçando o acesso a um nível mais baixo para tratamento do dados.

### **Sobrescrevendo Métodos**

O Java permite que você tenha métodos com o mesmo nome, mas com assinaturas diferentes. Isto permite a reusabilidade dos nomes dos métodos.

```
public void print( int i )  
public void print( float f )  
public void print( String s)
```

Quando você escreve um código para chamar um desses método, o método a ser chamado será o que coincidir com tipos de dados da lista de parâmetros.

### **Construtores**

O mecanismo de inicialização do Java é automático, ou seja se não inicializarmos um construtor, o Java o inicializará automaticamente.

Mas existem casos que se faz necessário a declaração explícita dos construtores.

Para escrever um método que chama um construtor, você deve seguir duas regras:

- 1ª O nome do método precisa ser igual ao nome da classe.
- 2ª Não deve retornar um tipo declarado para o método.

### **Subclasses (Relacionadas)**

Na classe Pai você deve declarar os objetos comun a todos, e nos sub-níveis (Subclasses), você declara as particularidades:

```
public class Employee {  
  
    private String name;  
    private Date hireDate;  
    private Date dateOfBirth;  
    private String jobTitle;  
    private int grade;  
  
    . . . . .
```

```
}
```

### ***Subclasses (Extensões)***

Em linguagens de orientação a objetos, um mecanismo especial é fornecido para que permita ao programa definir classes e termos previstas na definição de outras classes. Esta arquitetura em Java usa a palavra `extends`.

```
public class Employee {  
  
    private String name;  
    private Date hireDate;  
    private Date dateOfBirth;  
    private String jobTitle;  
    private int grade;  
  
}  
  
public class Manager extends Employee {  
  
    private String departament;  
    private Employee [ ] subordinates;  
  
    .....  
}
```

### ***Herança Simple***

Em Java não existe herança múltipla.  
Em Java os Construtores não são herdados.

Java permite que uma classe estenda uma outra classe, com isso esta classe herda as características da outra classe.

### ***Polimorfismo***

A idéia de polimorfismo é a de muitas formas, onde eu posso utilizar uma classe de diversas maneiras e formas possíveis.

```
public class Employee extends Object
```

and

```
public class Manager extends Employee
```

### ***Argumentos de Métodos e Coleções Heterogêneas***

Usando esta aproximação você pode escrever métodos que aceitam um objeto genérico. O uso do polimorfismo fornece uma série de facilidades.

```
public TaxRate findTaxRate( Employee e) {  
  
    Manager m = new Manager( );  
    .....  
  
    TaxRate t = findTaxRate(m);
```

Isto é possível porque um Gerente é um empregado.

Uma coleção heterogênea é uma coleção de coisas diferentes. Em linguagem orientada a objetos, você pode criar uma coleção de coisas que tem uma classe de antepassados comuns. Assim nós poderemos fazer isso.

```
Employee [ ] staff = new Employeee [ 1024 ];  
staff[ 0 ] = new Manager ( );  
staff[ 1 ] = new Employee ( );
```

E assim sucessivamente nos podemos escrever um método de tipos que põe empregados ordenados por idade ou em ordem salarial sem ter que se preocupar com a ordem de inserção.

### ***O Operador instanceof***

Fornecer o dado que você adquiriu através da passagem de parâmetros.

Caso você receba um object por referência do tipo Employee, esta referência poderia não ser mostrado para Manager. Se você quiser testar isso use instanceof.

```
public void method(Employee e) {  
    if (e instanceof Manager) {  
  
    }  
    else if ( e instanceof Contractor) {  
  
    }  
    else {
```

```
}  
}
```

### **Objetos Lançados**

Em circunstâncias onde você recebeu uma referência para uma classe pai, e você determinou que o objeto é de fato uma subdivisão de classe particular usando o operador de instanceof, você pode restabelecer a funcionalidade completa do objeto lançado.

### **Sobre-escrevendo Métodos**

O Java permite que você declare métodos na classe pai e não desenvolva nenhuma lógica dentro desses métodos, permitindo com isso que o desenvolvimento desses métodos venha a ser trabalhados dentro das sub-classes posteriores.

```
Class Funcionario( )
```

```
ler( )
```

```
Class Motorista( )
```

```
ler ( )
```

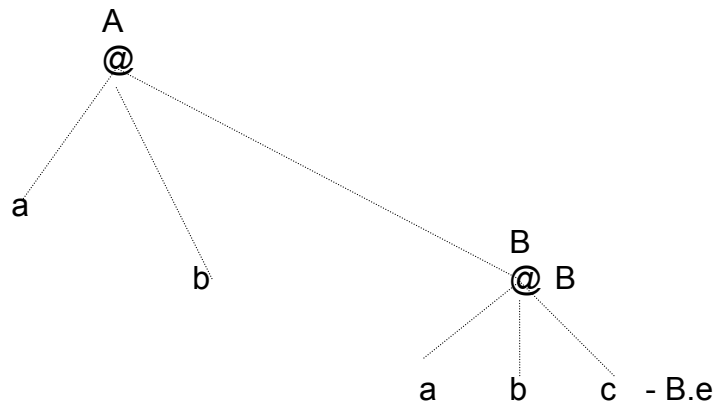
```
Super.ler( ) -> Referencia o método da classe pai
```

### **Comparando Java com outras Linguagens**

Em C++ você pode adquirir este comportamento, desde que você marque o método como virtual na fonte.

### **Classes se agrupando - Pacotes**

Java provê o mecanismo de pacotes como um modo de se agrupar classes relacionadas. Tão longe, todos nossos exemplos pertencem à falta ou pacotes não mencionados.



onde:

@ - Pacote -  
letra (a, b ...) - classe

### ***A declaração import***

Em Java, quando você quer usar instalações de pacotes, você usa a declaração de importação para contar para o compilador onde achar as classe que você vai usar.

A declaração de importação (import) deve preceder a declaração de todas as classes.

## **Exercícios**

```
class Prog0501
{
    public static void main(String arg[ ])
    {
        int a, b=10, c=3;
        a = func(b,b+c); //chamada de funcao. O retorno é passado para a
variável a
        System.out.println("valor de a = " + a);
    }

    // A definicao da funcao esta logo abaixo

    public static int func(int x, int y)
    {
        int ret;
        ret = x+y;
        return ret;
    }
}
```

### ***O que é uma Exception ?***

Em Java, a classe de Exceção define condições de erro moderados que seus programas podem encontrar. Em vez de você deixar o programa terminar, você pode escrever um código para tratar as exceções e continuar a execução do programa.

A linguagem Java implementa C++ nomeando exceções para poder construir um código elástico. Quando um erro acontece em seu programa, o código que acha o erro pode “disparar” uma exceção. Dispara uma exceção é o processo de sinalizar o processo corrente atual de que um erro aconteceu. Você enlata a captura da exceção e quando possível recupera a condução das próximas rotinas.

### ***Manipulação de Exceptions***

#### Declarações try e catch

Uma maneira de manipular possíveis erros, é usando as declarações try e catch. A declaração try indica a parte do código aonde poderá ocorrer uma exception, sendo que para isso você deverá delimitar esta parte do código com o uso de chaves. Na declaração catch você coloca o código a ser executado caso venha a ocorrer uma exception.

```
try {  
    // código que pode ocasionar uma exception  
}  
catch{  
    // tratamento do erro  
}
```

#### Declaração finally

A declaração finally é utilizada para definir o bloco que irá ser executado tendo ou não uma exception, isto após o uso da declaração de try e catch.

```
try {  
    // código que pode ocasionar uma exception  
}  
catch{  
    // tratamento do erro  
}  
finally {  
    // código  
}
```

Exceptions mais comuns

**ArithmeticException** - `int i = 12 / 0`

**NullPointerException** - ocorre quando utilizo um objeto que não foi instanciado.

**NegativeArraySizeException** - ocorre quando é atribuído um valor nulo para um array.

**ArrayIndexOutOfBoundsException** - ocorre quando tento acessar um elemento do array que não existe.

### ***Categorias de Exceptions***

Há três grandes categorias de exceções em Java. De fato, a classe `Java.lang.Throwable` age como uma classe pai, para que todos os objetos disparados possam ser pegos nas exceptions.

Deve-se evitar usar a classe `Throwable`, procure usar uma das três classes descritas à seguir:

- \* Erro - indica um problema sério de recuperação difícil, se não impossível;
- \* `RuntimeException` - problema ocorrido durante a implementação;
- \* Outra exceção - indica uma dificuldade durante a implementação que pode acontecer razoavelmente por causa de efeitos ambientais e pode se manipulado.

### ***Declare ou Manipule a sua Exception***

Na construção de um código em Java, o programador deve prever métodos para tratar possíveis erros. Existem duas maneiras de o programador satisfazer esta exigência. A primeira é com o uso da declaração `try` e `catch`, como foi visto anteriormente, e a segunda maneira é indicando que a exceção não é dirigida para este método, sendo então jogado para o método chamador.

```
public void troublesome( ) throws IOException
```

## Exercícios

```
class Prog0601
{
    public static void main (String args[ ])
    {
        int i = 0;
        int scap = 0;
        String greetings [ ] = { "Hello word" , "No, I mean it!" , "HELLO WORLD!"};
        While (i < 4)
        {
            try
            {
                System.out.println(greetings[i]);
            }
            catch(ArrayIndexOutOfBoundsException e)
            {
                scap++;
                System.out.println("Valor do Índice foi refixado " + scap);
                if (scap < 5)
                {
                    i = -1;
                }
            }
            catch(Exception e)
            {
                System.out.println(e.toString());
            }
            finally
            {
                System.out.println("Esta mensagem será sempre impressa.");
            }
            i++;
        }
    }
}
```

### ***O Pacote java.awt***

No pacote java.awt contém as classes para a geração de componentes Gui. Os componentes Gui geralmente são de aspectos visíveis, como botões ou labels.

### ***Posição dos componentes***

A posição dos componentes no container é determinado pelo gerente de layout. O tamanho dos componentes é também de responsabilidade do gerente de layout.

### ***O método main()***

É responsável pela criação da instância de objetos e pela inicialização do método go().

### ***O new Frame***

Cria uma instância da classe java.awt.Frame. O Frame no java é o nível mais alto da janela, aonde ficam por exemplo o barra de títulos.

### ***O setLayout()***

Cria uma instância do fluxo do gerente de layout, aonde é colocado o Frame.

### ***O new Button()***

Cria uma instância da classe java.awt.Button. O botão é o local na janela aonde é empurrado para que se tenha uma ação.

### ***O add()***

Este método adiciona elementos (botões, caixas de texto, etc.) ao frame.

### ***O pack()***

Este método fixa o tamanho dos elementos no frame.

### **O *setVisible()***

Este método torna os elementos visíveis no frame.

### **O *Flow Layout Manager***

O flow layout para posicionar componentes em uma determinada linha ou conjunto de linhas. A cada instante uma nova linha fica cheia e uma nova linha é começada.

### **O *Border Layout Manager***

É mais complexo do que o Flow Layout Manager, pois fornece componentes que possibilitam manusear o posicionamento do elemento dentro do container.

### **O *Grid Layout Manager***

Você pode criar linha e colunas dentro de seu container.

### **O *CardLayout***

Permite você criar duas interfaces semelhantes a botões, sendo entretanto o painel completo.

### ***Frame***

É uma armação aonde são inseridos diversos elementos como: botões, labels, etc.

### ***Panels***

São recipientes aonde são inseridos diversos elementos como: botões, labels, etc.

## Exercícios

### Exercício 01

```
import java.awt.*;
class Prog0701
{
    public static void main (String [] arg)
    {
        Frame f = new Frame();
        Button b1 = new Button();
        Button b2 = new Button();
        Button b3 = new Button();
        Button b4 = new Button();
        f.setLayout(new FlowLayout());
        f.add(b1);
        f.add(b2);
        f.add(b3);
        f.add(b4);
        f.setSize(200,200);
        f.setVisible(true);
    }
}
```

### Exercício 02

```
import java.awt.*;
class Prog0702
{
    public static void main (String [] arg)
    {
        Frame f = new Frame();
        Button b1 = new Button();
        Button b2 = new Button();
        Button b3 = new Button();
        Button b4 = new Button();
        Button b5 = new Button();
        Panel p = new Panel();
        p.setLayout(new FlowLayout());
        f.setLayout(new BorderLayout());
        p.add(b1);
        p.add(b2);
        f.add("North",p);
        f.add("Center",b3);
        f.add("West",b4);
        f.add("East",b5);
        f.setSize(200,200);
    }
}
```

```
        f.setVisible(true);
    }
}
```

### Exercício 03

```
import java.awt.*;
class Prog0703 extends Frame
{
    String x = "Para voltar, ola mundo ";

    //construtor

    Prog0703(String x)
    {
        super(x);
    }
    public void paint (Graphics g)
    {
        g.setColor(Color.red);
        g.drawString(x,50,50);
    }
    public static void main (String arg [])
    {
        // instancia o frame com um título
        Prog0703 w = new Prog0703("Janela do Prog0703 ");
        w.setSize(200,200);
        w.setVisible(true);
    }
}
```

## Capítulo 07

### **O Pacote *java.awt***

No pacote *java.awt* contém as classes para a geração de componentes Gui.

Os componentes Gui geralmente são de aspectos visíveis, como botões ou labels.

### **Posição dos componentes**

A posição dos componentes no container é determinado pelo gerente de layout. O tamanho dos componentes é também de responsabilidade do gerente de layout.

### **O método *main()***

É responsável pela criação da instância de objetos e pela inicialização do método *go()*.

### ***new Frame***

Cria uma instância da classe *java.awt.Frame*. O *Frame* no *java* é o nível mais alto da janela, aonde ficam por exemplo o barra de títulos.

### ***setLayout()***

Cria uma instância do fluxo do gerente de layout, aonde é colocado o *Frame*.

### ***new Button()***

Cria uma instância da classe *java.awt.Button*. O botão é o local na janela aonde é empurrado para que se tenha uma ação.

### ***add()***

Este método adiciona elementos (botões, caixas de texto, etc.) ao *frame*.

### ***pack()***

Este método fixa o tamanho dos elementos no frame.

### ***setVisible()***

Este método torna os elementos visíveis no frame.

### ***Flow Layout Manager***

O flow layout para posicionar componentes em uma determinada linha ou conjunto de linhas. A cada instante uma nova linha fica cheia e uma nova linha é começada.

### ***Border Layout Manager***

É mais complexo do que o Flow Layout Manager, pois fornece componentes que possibilitam manusear o posicionamento do elemento dentro do container.

### ***Grid Layout Manager***

Você pode criar linha e colunas dentro de seu container.

### ***CardLayout***

Permite você criar duas interfaces semelhantes a botões, sendo entretanto o painel completo.

### ***Frame***

É uma armação aonde são inseridos diversos elementos como: botões, labels, etc.

### ***Panels***

São recipientes aonde são inseridos diversos elementos como: botões, labels, etc.

## **Exercícios**

### Exercício 01

```
import java.awt.*;
import java.awt.event.*;

class Prog0701 extends Frame implements ActionListener{
    String x;

    public void paint(Graphics g)
    {if (x != null)
     g.drawString(x,100,100);
    }

    public void actionPerformed(ActionEvent e)
    { x="TRATEI O EVENTO DE BOTAO";
      repaint();
    }

    public static void main (String arg [])
    {Prog0701 f = new Prog0701();
     Button B = new Button();
     f.setLayout(new FlowLayout());
     f.add(B);
     B.addActionListener(f);
     f.setSize(300,300);
     f.show();
    }
}
```

### Exercício 02

```
import java.awt.*;
import java.awt.event.*;

class Prog0702 extends Frame implements ActionListener{
    String x;
    Button b;
    Button c;

    public void paint(Graphics g)
    {if (x != null)
     g.drawString(x,100,100);
    }
}
```

```

public Prog0702()
{ setLayout(new FlowLayout());
  b = new Button();
  c = new Button();
  add(b);
  add(c);
  b.addActionListener(this);
  c.addActionListener(this);
  setSize(300,300);
  show();
}

```

### Exercício 03

```

import java.awt.*;
import java.awt.event.*;

class Prog0703 extends Frame implements ItemListener{
  Checkbox b;
  Checkbox c;

  public Prog0703()
  { setLayout(new FlowLayout());
    b = new Checkbox("one",null,true);
    c = new Checkbox("two");
    add(b);
    add(c);
    b.addItemListener(this);
    c.addItemListener(this);
    setSize(300,300);
    show();
  }

  public void itemStateChanged(ItemEvent e)
  { if (e.getSource().equals(b))
    { System.out.println("Cliquei no botao b");
      System.out.println("O estado do botao b e: " + b.getState() );
    }
    else
    if (e.getSource().equals(c))
    { System.out.println("Cliquei no botao c");
      System.out.println("O estado do botao c e: " + c.getState() );
    }
  }

  public static void main (String arg [])

```

```
{  
  new Prog0703();  
}
```

## Capítulo 08

### **O que é um evento?**

Quando o usuário executa uma ação à interface de usuário, isto causa um evento a ser emitido. Eventos são objetos que descrevem o que aconteceu. Vários tipos diferentes de classes de evento existem para descrever categorias gerais diferentes de ação de usuário.

### **Fontes de evento**

Uma fonte de evento (ao nível de interface de usuário) é o resultado de alguma ação de usuário em um componente de AWT. Por exemplo, um trinco de rato em um componente de botão gerará (fonte) um `ActionEvent`.

### **Manipuladores de evento**

Quando um evento acontece, o evento é recebido pelo componente com o que o usuário interagiu; por exemplo, o botão, slider, campo de texto, e assim por diante. Um manipulador de evento é um método que recebe o objeto de Evento de forma que o programa pode processar a interação do usuário.

### **Como são processados Eventos**

Entre JDK 1.0 e JDK 1.1, houve mudanças significantes do modo que são recebidos eventos e são processados.

### **JDK 1.0 Contra JDK 1.1 Modelo de Evento**

JDK 1.0 usa um modelo de evento de hierarquia, e JDK 1.1 usa um modelo de evento de delegação. Enquanto este curso cobre o JDK 1.1, é importante reconhecer como estes dois modelos de evento comparam.

### **Modelo de hierarquia (JDK 1.0)**

O modelo de evento de hierarquia é baseado em hierarquia de retenção. Eventos que não são dirigidos ao componente continuarão propagando ao recipiente do componente automaticamente.

### **Modelo de delegação (JDK 1.1)**

O JDK 1.1 introduziu um modelo de evento novo, chamou-o de evento de delegação. No modelo de evento de delegação, são enviados eventos a componente, onde cada componente registrar uma rotina de manipulação de eventos (chamado de ouvinte) para receber o evento. Deste modo o manipulador de eventos pode estar em uma classe separada do componente. A manipulação do evento é delegada então à classe separada.

### **Vantagens do Modelo de delegação**

- \* Não são dirigidos eventos acidentalmente; no modelo de hierarquia é possível para um evento propagar a um recipiente e é dirigido a um nível que não é esperado.
- \* É possível criar filtro (adaptador) de classes para classificar ações de evento.
- \* O modelo de delegação é muito melhor para a distribuição de trabalho entre classes.
- \* O modelo de evento novo provê apoio por feijões de Java.

### ***Desvantagens do Modelo de delegação***

- \* É mais complexo, pelo menos inicialmente, entender.
- \* A migração do JDK 1.0 para o JDK 1.1 não é fácil.
- Embora o JDK atual apóia o JDK 1.0, os modelos podem ser misturados.

## Capítulo 09 – Os Componentes da Biblioteca AWT

\* Reconheça os componentes chaves da biblioteca AWT.

\* Use os componentes da AWT para construir interfaces de usuário para programas reais.

- Controle as cores e mananciais usados por um componente de AWT

### Instalações do AWT

O AWT provê uma variedade larga de instalações standards.

### Buttons

É uma das interfaces básicas, que é ativa após ser precionada.

### Checkboxes

Checkboxe fornece uma interface simples, aonde as opções possíveis são “on/off”.

### CheckboxGroups

CheckboxGroups é um tipo de Checkboxe especial, diferenciando do anterior pelo sinal de seleção, no caso uma bolinha, e aonde apenas uma opção poderá ser marcada.

### Choice

O Choice fornece uma seleção simples dentro de uma lista de opções.

### Canvas

Canvas é uma tela de fundo (papel de parede) aonde irão ser adicionados componentes.

### Label

Label é um objeto simples que mostra uma linha simples de texto.

### TextField

TextField é um objeto simples que permite a digitação dentro dele.

### TextArea

TextArea é um objeto mais completo que permite a digitação de várias linhas dentro dele.

## List

O List é um tipo de Choice, mas que permite a seleção múltipla dentro da lista de opções.

## Exercícios

```
import java.awt.*;
import java.awt.event.*;

class Prog 0801 extends Frame implements ItemListener{
    CheckboxGroup cgb = new CheckboxGroup();
    Checkbox one = new Checkbox("one",cgb,false);
    Checkbox two = new Checkbox("two",cgb,false);
    Checkbox three = new Checkbox("three",cgb,true);

    public Prog0801()
    { setLayout(new FlowLayout());
      add(one);
      add(two);
      add(three);
      one.addItemListener(this);
      two.addItemListener(this);
      three.addItemListener(this);
      setSize(300,300);
      show();
    }

    public void itemStateChanged(ItemEvent e)
    { if (e.getSource().equals(one))
      { System.out.println("Cliquei no botao one");
        System.out.println("O estado do botao one e: " + one.getState() );
      }
      else
      if (e.getSource().equals(two))
      { System.out.println("Cliquei no botao two");
        System.out.println("O estado do botao two e: " + two.getState() );
      }
      if (e.getSource().equals(three))
      { System.out.println("Cliquei no botao three");
        System.out.println("O estado do botao three e: " + three.getState() );
      }
    }

    public static void main (String arg [])
    {
      new Prog0801();
    }
}
```

Exercício 02

```

import java.awt.*;
import java.awt.event.*;

class Prog0802 extends Frame implements ItemListener{
    Choice cgb = new Choice();

    public Prog0802()
    { setLayout(new FlowLayout());
      cgb.addItem("one");
      cgb.addItem("two");
      cgb.addItemListener(this);
      add(cgb);
      setSize(300,300);
      show();
    }

    public void itemStateChanged(ItemEvent e)
    { if (e.getSource().equals(cgb))
      System.out.println(cgb.getSelectedItem());
    }

    public static void main (String arg [])
    {
      new Prog0802();
    }
}

```

### Exercício 03

```

import java.awt.*;
import java.awt.event.*;

class Prog0803 extends Frame implements ActionListener{
    TextField b;

    public Prog0803()
    { setLayout(new FlowLayout());
      b = new TextField(20);
      add(b);
      b.addActionListener(this);
      setSize(300,300);
      show();
    }

    public void actionPerformed(ActionEvent e)
    { if (e.getSource().equals(b))
      { System.out.println(b.getText()); }
    }
}

```

```
    }  
  
    public static void main (String arg [])  
    {  
        new Prog0803();  
    }  
}
```

#### Exercício 04

```
import java.awt.*;  
import java.awt.event.*;  
  
class Prog0804 extends Frame implements ActionListener{  
    TextArea b;  
    Button a;  
  
    public Prog0804()  
    { setLayout(new FlowLayout());  
      a = new Button();  
      b = new TextArea(5,5);  
      add(a);  
      add(b);  
      a.addActionListener(this);  
      setSize(300,300);  
      show();  
    }  
}
```

## Capítulo 09

### **Frame**

Frame é uma armação aonde serão inseridos objetos.

### **Panel**

Panel é um tipo de container aonde serão inseridos objetos.

### **Dialog**

Um Dialog é semelhante ao Frame, nisso é uma janela parada grátis com algumas decorações.

Não são feitos Dialog normalmente visível para o usuário quando eles são criados primeiro.

### **FileDialog**

FileDialog é um tipo de implementação que permite ao usuário selecionar pastas dentro de uma lista de opções.

### **ScrollPane**

ScrollPane são as barras de rolagens de um panel.

### **Menus**

Menu é uma série de opções disponível.

Um fator importante da criação de menus no Java é a respeito do Help. O help pode ser deslocado para ficar o mais a direita possível dos demais itens.

O MenuBar é um tipo de menu horizontal.

O MenuItem são os itens do menu.

### **CheckboxMenuItem**

É uma variação dos tipos de menu, aonde o usuário seleciona uma das opções e esta opção selecionada fica marcada com “checkbox”.

### **PopupMenu**

É um tipo de menu acionado a partir de qualquer posição na tela.

## **Controlando Aspectos Visuais**

### **Colors**

Existem dois métodos para controlar as cores:

- SetForeground()
- SetBackground()

Fonts

É utilizado para controlar os tipos de fontes.

### **Printing**

```
Frame f = new Frame("Print test")
```

```
...
```

## **Exercícios**

### Exercício 01

```
import java.awt.*;
import java.awt.event.*;

class Prog0901 extends Frame implements ActionListener{
    List lst = new List(4,false);

    public Prog0901()
    { setLayout(new FlowLayout());
      lst.add("a");
      lst.add("b");
      lst.add("c");
      lst.add("d");
      lst.add("e");
      lst.add("f");

      lst.addActionListener(this);
      add(lst);
      setSize(300,300);
      show();
    }

    public void actionPerformed(ActionEvent e)
    { if (e.getSource().equals(lst))
      System.out.println(lst.getSelectedItem());
    }

    public static void main (String arg [])
    {
      new Prog0901();
    }
}
```

### Exercício 02

```
import java.awt.*;
import java.awt.event.*;

class Prog0902 extends Frame implements ActionListener{
    Button a = new Button();
    String vet[];
    List lst = new List(4,true);

    public Prog0902()
```

```

{ setLayout(new FlowLayout());
  lst.add("a");
  lst.add("b");
  lst.add("c");
  lst.add("d");
  lst.add("e");
  lst.add("f");
  a.addActionListener(this);
  add(lst);
  add(a);
  setSize(300,300);
  show();
}

public void actionPerformed(ActionEvent e)
{ if (e.getSource().equals(a))
  vet=lst.getSelectedItems();
  for (int i=0; i < vet.length;i++)
  {
    System.out.println(vet[i]);
  }
}

public static void main (String arg [])
{
  new Prog0902();
}
}

```

### Exercício 03

```

import java.awt.*;
import java.awt.event.*;

class Prog0903 extends Frame implements WindowListener
{
  Prog0903()
  { addWindowListener(this);
    setSize(300,300);
    setVisible(true);
  }
  public void windowActivated(WindowEvent e)
  {
  }
  public void windowClosed(WindowEvent e)
  {
  }
}

```

```

public void windowClosing(WindowEvent e)
{
    System.exit(0);
}
public void windowDeactivated(WindowEvent e)
{
}
public void windowDeiconified(WindowEvent e)
{
}
public void windowIconified(WindowEvent e)
{
}
public void windowOpened(WindowEvent e)
{
}
static public void main(String agr[])
{ new Prog0903();
}
}

```

#### Exercício 04

```

import java.awt.*;
class Prog0904 {
    public static void main (String [] arg)
    { Frame f = new Frame();
      Button b1 = new Button();
      Button b2 = new Button();
      Button b3 = new Button();
      Button b4 = new Button();
      f.setLayout(new FlowLayout());
      f.add(b1);
      f.add(b2);
      f.add(b3);
      f.add(b4);
      f.setSize(200,200);
      f.setVisible(true);
    }
}

```

#### Exercício 05

```

import java.awt.*;
import java.awt.event.*;

class Prog0905 extends Frame implements ActionListener{

```

```

MenuBar mb = new MenuBar();
Menu m1 = new Menu("File");
Menu m2 = new Menu("Edit");
Menu m3 = new Menu("Help");

MenuItem mi11 = new MenuItem("Save");
MenuItem mi12 = new MenuItem("Open");
MenuItem mi13 = new MenuItem("Exit");

public Prog0905()
{ setLayout(new FlowLayout());
  m1.add(mi11);
  m1.add(mi12);
  m1.addSeparator();
  m1.add(mi13);

  mb.add(m1);
  mb.add(m2);
  mb.add(m3);

  mi11.addActionListener(this);
  mi12.addActionListener(this);
  mi13.addActionListener(this);

  setMenuBar(mb);

  setSize(300,300);
  show();
}

public void actionPerformed(ActionEvent e)
{ if (e.getSource().equals(mi11))
  { System.out.println("Item Seleccionado: mi11 " );
  }
}

public static void main (String arg [])
{
  new Prog0905();
}
}

```

## Capítulo 10

### O que é um Applet?

Um applet é um pedaço de código de Java que corre em um ambiente de browser. Difere de uma aplicação do modo que é executado. Uma aplicação é começada quando seu método main() é chamado. Através de contraste, o ciclo de vida de um applet é um pouco mais complexo. Este módulo examina como um applet é corrido, como carregar isto no browser, e como escrever o código para um applet.

### *Carregando um Applet*

Você tem que criar um arquivo de HTML que conta para o browser o que carregar e como correr isto. Você então "aponta" o browser ao URL que especifica aquele arquivo de HTML.

### Segurança e Restrições nos Applet

Existem Seguranças e Restrições nos Applets porque applets são pedaços de código que são carregados em cima de arames, eles representam um prospecto perigoso; imagine se alguém escrever um programa malicioso que lê seu arquivo de contra-senha e envia isto em cima da Internet?

O modo que Java previne isto está contido na classe de SecurityManager que controla acesso para quase toda chamada de sistema-nível no Java Máquina Virtual (JDK).

Portanto Applets não faz chamadas ao seu sistema operacional.

### Métodos de Applet chaves

Em uma aplicação, no programa é entrado ao método main(). Em um applet, porém, este não é o caso. O primeiro código que um applet executa é o código definido para sua inicialização, e seu construtor.

Depois que o construtor é completado, o browser chama um método no applet chamado init (). O método init () executar inicialização básica do applet . Depois de init () é completado, o browser chama outro método chamado start ().

### Applet Display

Você pode puxar sobre a exibição de um applet criando um método paint(). O método paint() é chamado pelo ambiente de browser sempre que a exibição do applet precise ser refrescado. Por exemplo, isto acontece quando a janela de browser é aparecida depois de ser minimizado.

### **Exercícios**

#### Exercício 01

```

import java.awt.*;
import java.applet.*;

public class Prog1001 extends Applet
{
    public void init()
    {
        System.out.println("Executando o Init");
    }
    public void start()
    {
        System.out.println("Executando o Star");
    }
    public void stop()
    {
        System.out.println("Executando o Stop");
    }
    public void destroy()
    {
        System.out.println("Executando o destroy");
    }
}

```

## Exercício 02

```

import java.awt.event.*;
import java.awt.*;
import java.applet.*;

public class Prog1002 extends Applet implements Runnable,ActionListener
{
    TextField b1,b2;
    String x;
    Thread k;

    public void init()

    {
        setLayout(new FlowLayout());
        b1 = new TextField(10);
        b2 = new TextField(10);
        b1.addActionListener(this);
        b2.addActionListener(this);
        add(b1);
        add(b2);
        k = new Thread(this);
        k.start();
    }
}

```

```

}

public void paint(Graphics g)
{
    if ( x != null)
        g.drawString(x,50,50);
}

public void run()

{
    int cont = 0;
    while(true)
        { cont = cont + 1;
          b1.setText(""+cont);}
}

public void stop()

{
    k.stop();
}

public void start()

{
    k.resume();
}

public void actionPerformed(ActionEvent e)

{
    x = b2.getText();
    repaint();
}

}

```

## Capítulo 11

## Exercícios

```
import java.util.*;
public class Prog1101 extends Vector
{
    public Prog1101()
    {
        super(1,1);
    }
    public void addInt(int i)
    {
        addElement(new Integer(i));
    }
    public void addFloat(float f)
    {
        addElement(new Float(f));
    }
    public void addString(String s)
    {
        addElement(s);
    }
    public void addCharArray(char a [])
    {
        addElement(a);
    }
    public void printVector()
    {
        Object o;
        int length = size();
        System.out.println("Numero de elementos do vetor " + length + " e eles
sao: ");
        for (int i = 0; i < length; i++)
        {
            o = elementAt(i);
            if (o instanceof char [])
            {
                System.out.println(String.valueOf((char[]) o));
            }
            else
                System.out.println(o.toString());
        }
    }
    public static void main(String args [])
    {
        Prog1101 v = new Prog1101();
        int digit = 5;
        float real = 3.14f;
    }
}
```

```
char letters[] = {'a', 'b', 'c', 'd'};
String s = new String("Hi there");
v.addInt(digit);
v.addFloat(real);
v.addString(s);
v.addCharArray(letters);
v.printVector();
}
}
```

### ***Derrame Fundamentos***

Um fluxo ou é uma fonte de bytes ou um destino para bytes. A ordem é significativa. Por exemplo, um programa queira ler de um teclado pode usar um fluxo, a ordem de entrada dos dados será a ordem do fluxo.

### ***Introduza Métodos de Fluxo***

Estes três métodos provêm acesso para os dados do tubo. O método `lido` simples devolve um tipo `int` que ou contém um byte lido do fluxo ou `-1` que indica o fim de condição de arquivo.

### ***Método do Fluxo de produção***

Estes métodos escrevem ao fluxo de produção. Como com a entrada, você tentar escrever dados no bloco prático maior. `close ()`  
Deveriam ser fechados fluxos de produção quando você terminou com eles. Novamente, se você tem uma pilha e fecha o topo um, isto fecha o resto dos fluxos. `flush ()`

### ***Os leitores e Escritores***

Unicode

Java usa Unicode por representar fios e caracteres, e a versão 16 bits provê pedaço de fluxos para permitir tratar caráter similares. Esta versão é chamada de os leitores e escritoras, e como com fluxos, uma variedade deles está disponível no pacote `java.io`.

Byte e Conversões de Caráter

Através de falta, se você constrói um leitor simplesmente ou o escritor conectou a um fluxo, então as regras de conversão mudarão entre bytes que usam o caráter de plataforma de falta que codifica e Unicode.

O Leitor de Buffered e Escritor

Porque convertendo entre formatos é como outras operações de I/O, eficazmente executadas em pedaços grossos grandes, que geralmente é uma idéia boa para encadear um `BufferedReader` ou `BufferedWriter` sobre o fim de um `InputStreamReader` ou `OutputStreamWrite`.

### ***Arquivos***

Antes de você querer executar operações de I/O em um arquivo, você tem que obter informação básica sobre aquele arquivo. A classe de Arquivo provê várias utilidades para lidar com arquivos e obter informação básica sobre estes arquivos.

## **Exercícios**

### Exercício 01

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

```
class Prog1201 { public static void main(String arg[]) throws IOException
    { int b;
      int count=0;
      while( (b = System.in.read()) != (int)\n')
        { count++; System.out.println( (char) b);}
      System.out.println("contagem = " + count);
    }
}
```

### Exercício 02

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

```
class Prog1202 { public static void main(String arg[])
    { int b = (int)'F' ;
      System.out.write(b);
      System.out.flush();
    }
}
```

### Exercício 03

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

```
class Prog1203 { public static void main(String ar[]) throws IOException
    { String x = "Curso de Java";
      byte[] a = new byte[10];
      x.getBytes(0,7,a,0);
      System.out.write(a);
    }
}
```

#### Exercício 04

```
import java.io.*;

class Prog1204 { public static void main(String ar[]) throws IOException
    {int car;
    FileInputStream arq;
    try{ arq = new FileInputStream("teste.txt");
        while( (car = arq.read() ) !=-1 )
            System.out.write(car);
        }
    catch(FileNotFoundException e) { System.out.println("nao existe");
        }
    }
}
```

#### Exercício 05

```
import java.io.*;

class Prog1205 { public static void main(String ar[]) throws IOException
    { int car;
    FileOutputStream arq;
    try{ arq = new FileOutputStream("teste1.txt");
        while( (car = System.in.read()) != -1)
            arq.write(car);
            arq.flush( );
        }
    catch(FileNotFoundException e) { System.out.println("nao existe");}
    }
}
```

#### Exercício 06

```
import java.io.*;

class Prog1206 { public static void main(String arg[]) throws IOException
    {
    FileOutputStream arq1 = new FileOutputStream("E1.txt");
    FileOutputStream arq2 = new FileOutputStream("E2.txt");
    DataOutputStream d1 = new DataOutputStream(arq1);
    DataOutputStream d2 = new DataOutputStream(arq2);
    d1.writeByte((int) 'a');
    d1.writeByte((int) 'a');
    d1.writeByte((int) 'a');
    d2.writeChar((int) 'a');
    d2.writeChar((int) 'a');
    }
}
```

```
    d2.writeChar((int) 'a');
  }
}
```

#### Exercício 07

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

```
class Prog1207 { public static void main(String arg[]) throws IOException
  {
    FileOutputStream arq1 = new FileOutputStream("dado.bin");
    DataOutputStream d1 = new DataOutputStream(arq1);
    d1.writeInt(10);
    d1.writeInt(20);
    d1.writeInt(345);
    FileInputStream arq2 = new FileInputStream("dado.bin");
    DataInputStream d2 = new DataInputStream(arq2);

    try{ while(true) System.out.println(d2.readInt());}
    catch(EOFException e) { System.out.println("acabou o arquivo");}
  }
}
```

#### Exercício 08

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

```
class Prog1208 { public static void main(String arg[]) throws IOException
  { String str = "ABCDEF";
    FileOutputStream arq1 = new FileOutputStream("saida1");
    DataOutputStream d1 = new DataOutputStream(arq1);
    FileOutputStream arq2 = new FileOutputStream("saida2");
    DataOutputStream d2 = new DataOutputStream(arq2);
    d1.writeBytes(str);
    d2.writeChars(str);
  }
}
```

## Capítulo 13

## Capítulo 13 – Threads (Linhas)

### ***Threads e Threading no Java***

#### **O que são Linhas?**

Uma visão simplista de um computador é a que tem uma CPU que manipula os dados, uma memória ROM que contém o programa que o CPU executa, e RAM que segura os dados nos quais o programa opera. Nesta visão simples, um trabalho é executado de cada vez.

#### ***Thread e thread para referenciar a ideia de execução.***

#### **Três Partes de uma Linha**

Uma linha inclui três partes principais. Primeiro, há o CPU virtual. Segundo, há o código que esta CPU está executando. Terceiro, há os dados no que o código trabalha.

#### **Criando a Linha**

Demos uma olhada no modo que uma linha é criada, e discuta como os argumentos de construção são usados para prover o código e dados para a linha quando corre.

#### **Começando a Linha**

Embora nós criamos a linha, não começa correndo imediatamente. Para começar isto, nós usamos o método `start()`. Este método está na classe de Linha, tão determinado no nosso esboço prévio que nós simplesmente dizemos.

#### **Thread Sheduling**

Embora a linha se torna `runnable`, necessariamente não começa imediatamente. Claramente em uma máquina que de fato tem só uma CPU, pode estar fazendo só uma coisa de cada vez.

#### **Testando uma Linha**

Às vezes é possível para uma linha estar em um estado desconhecido. É possível inquirir se uma linha ainda é viável através do método `isAlive()`.

#### **Pondo Linhas em Cabo (Putting Threads on Hold)**

Vários mecanismos existem que podem parar a execução de uma linha temporariamente. Seguindo este tipo de suspensão, execução pode ser retomada como se nada tivesse interrompido sua execução, a linha aparece ter executado uma instrução muito lentamente simplesmente.

O método `sleep()` foi introduzido na primeira seção e foi usado para deter uma linha para um período de tempo.

Às vezes é apropriado suspender execução de uma linha indefinidamente em qual caso alguma outra linha será responsável para retomar a execução. Um par de métodos de Linha está disponível para isto. Eles são chamados `suspend()` e `resume()`.

O método `join()` causa a linha atual para esperar até a linha em qual o método `join()` é chamado termina.

#### **Outros modos para criar linhas**

Nós discutimos criando Linha por uso de uma classe separada que implementa `Runnable`. De fato, esta não é a única possível aproximação.

## Usando sincronizaram em Java

### O Problema

Imagine uma classe que representa uma pilha. Advertência que a classe não faz nenhum esforço para dirigir transbordamento ou underflow da pilha, e que a capacidade de pilha está bastante limitada. Estes aspectos não são pertinentes a nossa discussão.

### A Bandeira de Fechadura de Objeto (The Object Lock Flag)

Em Java, toda instância de qualquer objeto tem um flag (bandeira) associado com isto. Esta bandeira pode ser pensada de como uma " bandeira " de fechadura. Um palavra sincronizado é provido para permitir interação com esta bandeira.

### Reunindo isto

Como foi sugerido, o mecanismo sincronizador () só trabalha se o programador põe as chamadas nos lugares corretos.

### Paralisação completa

Em programas onde linhas múltiplas estão competindo para acesso a recursos múltiplos, pode haver uma possibilidade de uma condição conhecida como paralisação completa. Isto acontece quando uma linha está esperando por uma fechadura segurada por outra linha, mas a outra linha já está esperando por uma fechadura segurada pela primeira linha.

## ***Enfie Interação (Thread Interaction)***

### **O Problema**

Por que duas linhas poderiam precisar interagir?

### **A Solução**

Toda instância de objeto em Java tem duas tranças de linha associadas com isto. O primeiro é usado por linhas que querem obter a bandeira de fechadura, e foi discutido na seção. A segunda trança é usada para implementar os mecanismos de comunicação de `wait ()` e `notify ()`.

## Exercícios

### Exercício 01

```
class Prog1 extends Thread
{ int a = 0;
  public void run(){
    for(int r=0;r<7;r++)
      { a = a+1; System.out.println(getName()+" "+a);}
  }
}
```

```
class Prog2 extends Thread
{ int b = 0;
  public void run(){
    for(int r=0;r<7;r++)
      { b = b+1; System.out.println(getName()+" "+b);}
  }
}
```

```
class Prog1301 { public static void main(String arg[])
{
  Prog1 x = new Prog1();
  Prog2 y = new Prog2();
  x.setName("Paulo");
  y.setName("Mario");
  x.start(); y.start();
  for (int i =0; i<3; i++)
    System.out.println("executando o Main");
  /*try{x.join(); y.join();} catch(InterruptedException e){} */
}
}
```

### Exercício 02

```
class Prog3 extends Thread {
  public void run()
  { int a = 0;
    for(int r=0;r<7;r++)
      { a = a+1;
        System.out.println(getName()+" "+ a);
      }
  }
}
```

```
class Prog1302 { public static void main(String arg[])
```

```
{ Prog3 x = new Prog3();  
  Prog3 y = new Prog3();  
  x.setName("Paulo");  
  y.setName("Mario");  
  x.start();  
  y.start();  
  try{ x.join(); y.join();} catch(InterruptedException e){}  
}  
}
```

## **Rede com Java**

### **Sockets**

Sockets é o nome de um modelo de programação particular, para o vínculo de comunicação entre processos. Por causa da popularidade deste modelo de programa, o nome socket foi usada de novo em outras áreas inclusive Java.

### **Montando a Conexão**

Para montar a conexão, uma máquina tem que estar correndo um programa que está esperando por uma conexão, e o outro fim tem que tentar alcançar o primeiro. Isto é semelhante a um sistema de telefone; uma festa tem que fazer a ligação de fato, enquanto a outra festa tem que estar esperando pelo telefone quando aquela ligação é feita.

### **Enviando a Conexão**

Quando fazemos um telefone chamar, nos precisamos saber o número de telefone para discar. Quando fazemos uma conexão de cadeia, nos precisamos saber o endereço ou o nome da máquina distante.

### **Números de porto**

Números de porto em sistemas de TCP/IP são 16 números de pedaço e estão no alcance 0-65535. Em prática, número de porto abaixo de 1024 é reservado para serviços predefinidos, e você não os deveria usar a menos que você deseje comunicar com um desses serviços.

### **O Modelo de Rede no Java**

No idioma de Java, são implementadas conexões TCP/IP com classes no `java.pacote` líquido.

### **Sockets de UDP**

Onde TCP/IP é um protocolo conexão-orientado, o Usuário Protocolo de Datagram é um protocolo de "connectionless". Contudo, um modo simples e elegante para pensar nas diferenças entre estes dois protocolos é a diferença entre o telefone chamada e correio postal.

### **DatagramPacket**

DatagramPacket tem dois constructors: um para dados receptores e o outro por enviar dados.