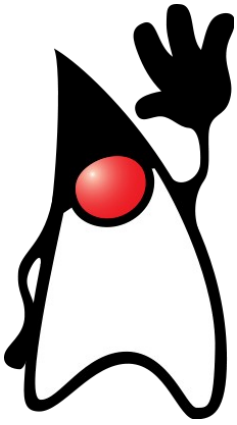
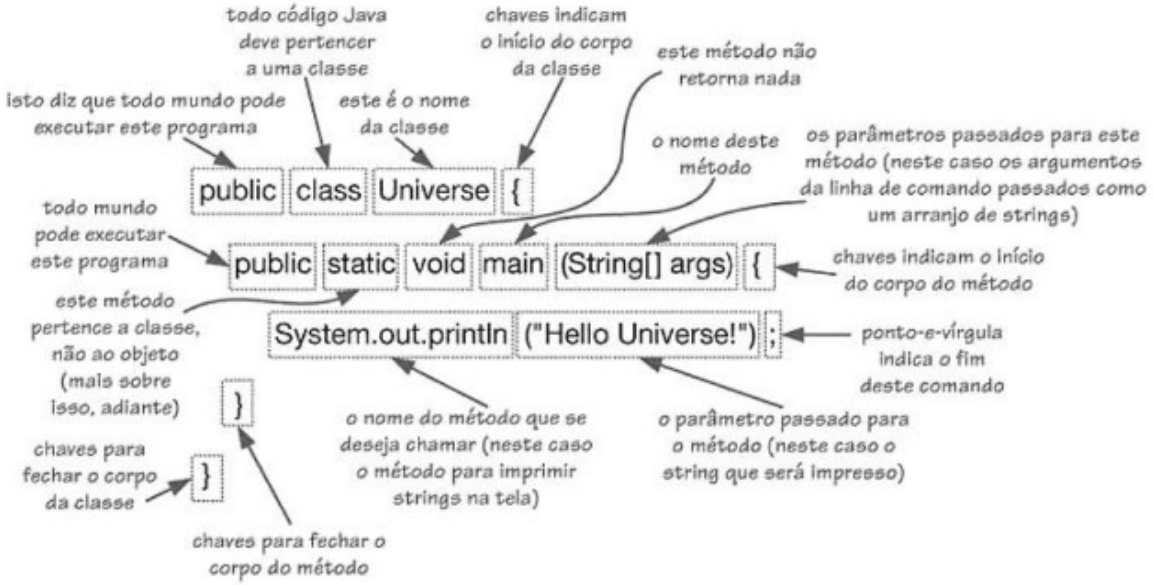
A linguagem Java foi idealizada para ser usada em pequenos dispositivos, como tvs, video-cassetes, aspiradores, liquidificadores e outros. Apesar disso a linguagem teve seu lançamento focado no uso em clientes web (browsers) para rodar pequenas aplicações (applets). Hoje em dia esse não é o grande mercado do Java: apesar de ter sido idealizado com um propósito e lançado com outro, o Java ganhou destaque no lado do servidor (JEE).



Duke, o mascote do Java

Java utiliza do conceito de máquina virtual, onde existe, entre o sistema operacional e a aplicação, uma camada extra responsável por "traduzir" - mas não apenas isso - o que sua aplicação deseja fazer para as respectivas chamadas do sistema operacional onde ela está rodando no momento.
Sua aplicação roda sem nenhum envolvimento com o sistema operacional! Sempre conversando apenas com a ***Java Virtual Machine (JVM).***
Essa camada, a máquina virtual, não entende código java (arquivos ***.java***), ela entende um código de máquina específico. Esse código de máquina é gerado por um compilador java, como o ***javac***, e é conhecido por "**bytecode**", diferente das linguagens sem máquina virtual, vai servir para diferentes sistemas operacionais, já que ele vai ser "traduzido" pela JVM.
Depois de compilar, o bytecode foi gerado. Quando o sistema operacional listar os arquivos contidos no diretório atual, você poderá ver que um arquivo **.class** foi gerado, com o mesmo nome da sua classe Java.

Sintaxe básica e métodos principais da linguagem JAVA:



Variáveis primitivas

Em Java, toda variável tem um tipo que não pode ser mudado uma vez que declarado: *tipoDaVaríavel nomeDaVariável;*

- **Inteiro** (*int*). Ex: int idade; int quantidade = 0;
- **Real** (*float* ou *double*). Ex.: float valor; float salario = 3056.05f; double pagamento = 345.09;
- **Literal** (*char* ou *String*). Ex. char sexo = 'f'; char = op; String nome = "Ana"; String email;
- **Boolean** *(*true ou false). Ex. boolean ativo = true; boolean ligado;

Tipo	**Tamanho**
Boolean	1bit
Byte	1byte
Short	2bytes
Char	2bytes
Int	4bytes
Float	4bytes
Long	8bytes

Operadores matemáticos

Você pode usar os operadores +, -, / e * para operar com números, sendo eles responsáveis pela adição, subtração, divisão e multiplicação, respectivamente.
Além desses operadores básicos, há o operador % (módulo) que nada mais mais é que o resto de

uma divisão inteira. Veja alguns exemplos: *int um = 5 % 2;*

Controle de fluxo

If/else (Se/senão)

A sintaxe do if no Java é a seguinte :

```
if (condicaoBooleana) {
        codigo;
}
```

Uma condição booleana é qualquer expressão que retorne *true* ou *false*. Para isso, você pode usar os operadores <, >, <=, >= e outros. Um exemplo:

```
int idade = 15;
if (idade < 18) {
System.out.println("Não pode entrar");
}
```

Além disso, você pode usar a cláusula else para indicar o comportamento que deve ser executado no caso da expressão booleana ser falsa:

```
int idade = 15;
if (idade < 18) {
        System.out.println("Não pode entrar");
}
else {
        System.out.println("Pode entrar");
}
```

Você pode concatenar expressões booleanas através dos operadores lógicos "E" e "OU". O "E" é representado pelo & e o "OU" é representado pelo |.

```
int idade = 15;
boolean amigoDoDono = true;
if (idade < 18 & amigoDoDono == false) {
        System.out.println("Não pode entrar");
}
else {
        System.out.println("Pode entrar");
}
```

Esse código poderia ainda ficar mais legível, utilizando-se o operador de negação, o !. Esse operador transforma uma expressão booleana de false para true e vice versa.

```
int idade = 15;
boolean amigoDoDono = true;
if (idade < 18 & !amigoDoDono) {
        System.out.println("Não pode entrar");
}
else {
        System.out.println("Pode entrar");
}
```

Repare que o trecho amigoDoDono == false virou !amigoDoDono. Eles têm o mesmo valor.
O que acontece é que os operadores && e || funcionam como seus operadores irmãos (& e |), porém eles funcionam da maneira mais rápida possível, quando percebem que a resposta não mudará mais, eles param de verificar as outras condições booleanas. Por isso eles são chamados de operadores de curto circuito (short circuit operators).

Laço

While (Enquanto)

O while é um comando usado para fazer um laço (loop), isto é, repetir um trecho de código algumas vezes. A idéia é que esse trecho de código seja repetido enquanto uma determinada condição

permanecer verdadeira.

```
int idade = 15;
while(idade < 18) {
        // espera ele crescer
        idade = idade + 1;
}
```

O trecho dentro do bloco do while será executado até o momento em que a condição idade < 18 passe a ser falsa. E isso ocorrerá exatamente no momento em que idade == 18, o que fará imprimir 18.

do.. While (faça..Enquanto)

A sintaxe do laço *do .. while* em Java, é a seguinte

```
do
{
        //esse código será executado pelo menos uma vez
} while( condição );
```

Seu uso é bem simples e intuitivo, visto que o laço while nós já conhecemos.
O 'do' do inglês quer dizer 'faça'. Ou seja, esse laço nos diz 'faça isso enquanto 'condição' for verdadeira'

For (Para)

Outro comando de loop extremamente utilizado é o for. A idéia é a mesma do while, fazer um trecho de código ser repetido enquanto uma condição continuar verdadeira. Mas além disso, o for isola também um espaço para inicialização de variáveis e o modificador dessas variáveis. Isso faz com que fique mais legível as variáveis que são relacionadas ao loop:

```
for (inicializacao; condicao; incremento) {
codigo;
}
```

Um exemplo é o a seguir:

```
for (int i = 0; i < 10; i = i + 1) {
        System.out.println("olá!");
}
```

Repare que esse for poderia ser trocado por:

```
int i = 0;
while (i < 10) {
        i = i + 1;
        System.out.println("olá!");
}
```

Porém, o código do for indica claramente que a variável i serve em especial para controlar a quantidade de laços executados. Quando usar o for? Quando usar o while? Depende do gosto e da ocasião.

Pós e Pré incremento ++ , --

Em "i = i + 1" poderemos substituir por i++ quando isolado, porém, em alguns casos, temos o seguinte:

```
int i = 5;
int x = i++;
```

Qual é o valor de x? O de i, após essa linha, é 6.
O operador ++, quando vem a frente da variável, retorna o valor antigo, e incrementa (pós incremento), fazendo x valer 5.

Se você tivesse usado o ++ antes da variável (pré incremento), o resultado seria 6, como segue:

```
int i = 5;
int x = ++i;
```

Casting e Conversão de Valores

Veja o Exemplo abaixo:

```
int i = 5;
double d2 = i;
```

O código acima compila sem problemas, já que um double pode guardar um número com ou sem ponto flutuante. Todos os inteiros representados por uma variável do tipo int podem ser guardados em uma variável double, então não existem problemas no código acima.
Ás vezes, precisamos que um número quebrado seja arredondado e armazenado num número inteiro. Para fazer isso sem que haja o erro de compilação, é preciso ordenar que o número quebrado seja moldado (casted) como um número inteiro. Esse processo recebe o nome de ***casting***.

```
double d3 = 3.14;
int i = (int) d3;
```

O casting foi feito para moldar a variável d3 como um int. O valor dela agora é 3. O mesmo ocorre entre valores *int* e *long*.
Mas o *casting* não pode ser aplicados em todos os caso. Ai devemos **converter os valores/conteúdos** com as funções **parse** e **valueof,** precedidos de seus tipos finais. Como por exemplo de Texto para Inteiro ou Double, ou de Inteiro ou Double para String. Ex.

```
String frase = "34";
int valor1 = Integer.parseInt(frase);
double valor2 = Double.parseDouble(frase);
String valorTexto = String.valueOf(valor1 + valor2);
```

Orientação à objetos (O.O.)

Orientação à objetos é uma maneira de programar que ajuda na organização e resolve muitos problemas enfrentados pela programação procedural (Estruturada).
A Orientação a objetos vai te ajudar em muito em se organizar e escrever menos, além de concentrar as responsabilidades nos pontos certos, flexibilizando sua aplicação.
Outra enorme vantagem, de onde você realmente vai economizar montanhas de código, é o *polimorfismo*, que veremos posteriormente.

Classes

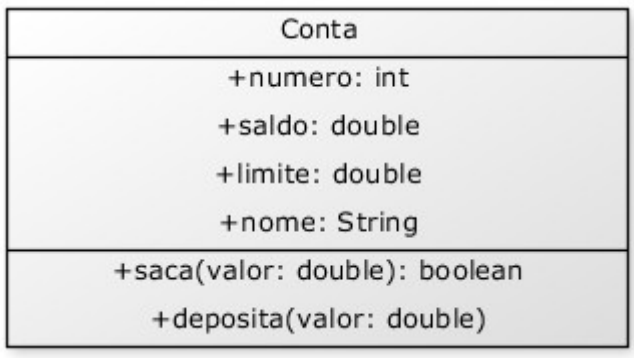
Representa a generalização de alguma informação, juntamente com as funcionalidades que deve ter um grupo de objetos, que existem no mundo real. Ex.: *Class Aluno, Class Conta;*

Atributos

Quando uma variável é declarada diretamente dentro do escopo da classe, é chamada de variável de objeto, ou atributo.
Ex. int numero; double saldo;

Métodos

São operações que serão realizadas em métodos (ações do objeto) definidos na própria Classe . Por exemplo, para realizar a operação de depósito e saque *(deposita(double valor)* e *boolean saca(double valor))*



Exemplo de Classes

Objetos: Em programação orientada a objetos, um objeto passa a existir a partir de uma ***instância***

de um "molde" (classe). Sempre que usarmos a palavra chave **new** , estamos construindo (instanciando) um objeto. A classe define o comportamento do objeto, usando atributos (propriedades) e métodos (ações).Exemplo:

*Conta conta = **new** Conta(); //Onde "Conta" é a Classe, "conta" o objetos e "new" a ordem de criação do objeto com base na classe;*

Modificadores

Em programação orientada a objetos, modificador de acesso, também chamado de visão de método ou ainda visão de atributo, é a palavra-chave que define um atributo, método ou classe como ***público*** (ou ***public "+"***, qualquer classe pode ter acesso), ***privado*** (ou ***private "-"***, apenas os métodos da própria classe pode manipular o atributo) ou ***protegido*** (ou ***protected "#"***, pode ser acessado apenas pela própria classe ou pelas suas subclasses).

Geralmente, utiliza-se modificadores de acesso para privar os atributos do acesso direto (tornando-os privados) e implementa-se métodos públicos que acessam e alteram os atributos. Tal prática pode ser chamada de *encapsulamento*.

Construtores

Quando o *new* é chamado, ele executa o construtor da classe. O construtor da classe é um bloco declarado com o mesmo nome que a classe.

```
//Declarando Conta
public class Conta{
        double saldo;
        //Construtor
        public Conta(double valor){
                saldo = valor;
        }
}

//Instanciando Conta
Conta conta = new Conta(100);
```

Encapsulamento (criar uma cápsula)

Consiste na separação de aspectos internos e externos de um objeto. Este mecanismo é utilizado amplamente para impedir o acesso direto ao estado de um objeto (seus atributos), disponibilizando externamente os métodos que acessam (***getters***) e alteram (***setters***) estes estados. Exemplo: você não precisa conhecer os detalhes dos circuitos de um telefone para utilizá-lo. A carcaça do telefone encapsula esses detalhes, provendo a você uma interface mais amigável. Exemplo:

```
public class Conta{
        private double saldo;
        //Sets e Gets
        public void setSaldo( double valor){
                this.saldo = valor;      //<- O termo "this" refere-se a esta classe/objeto.
        }
        public double getSaldo(){
                return saldo;   //<- O termo "return" determina qual será o conteúdo de retorno;
        }
```

Polimorfismo (Múltiplas Formas)

Polimorfismo é a capacidade de um objeto poder ser referenciado de várias formas. (cuidado, polimorfismo não quer dizer que o objeto fica se transformando, muito pelo contrário, um objeto nasce de um tipo e morre daquele tipo, o que pode mudar é a maneira como nos referimos a ele).

Sobrecarga: duas funções/métodos com o mesmo nome mas assinaturas

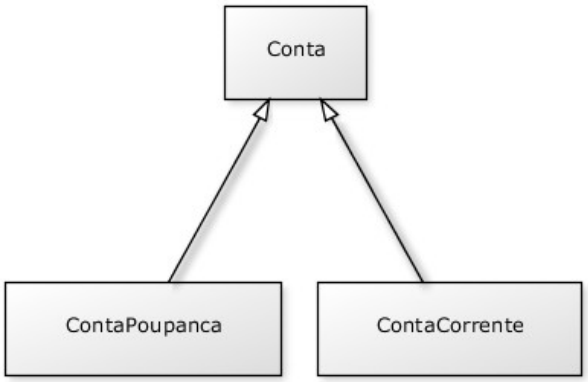
Exemplo:

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

```
                int num1 = 10, num2 = 20;
                double num3 = 1.2, num4 = 3.5;
                System.out.println(soma(num1, num2));
                System.out.println(soma(num3, num4));
        }

        static int soma(int a, int b) { // método "static" (estático), pois não será instanciado
                return a+b;
        }

        static double soma(double a, double b) {
                return a+b;
        }
}
```

Herança (ou generalização)

É o mecanismo pelo qual uma classe (sub-classe) pode estender outra classe (super-classe), aproveitando seus comportamentos (métodos) e variáveis possíveis (atributos). Um exemplo de herança: Mamífero é super-classe de Humano. Ou seja, um Humano é um mamífero. Há herança múltipla quando uma sub-classe possui mais de uma super-classe.



Exemplo de Herança

```
public class ContaCorrente extends Conta {
        public void atualiza(double taxa) {
                this.saldo += this.saldo * taxa * 2;
        }
        public void deposita(double valor) {
                this.saldo += valor - 0.10;
        }
}
```

Classes Abstratas

Usamos a palavra chave ***abstract*** para impedir que ela possa ser instanciada. Esse é o efeito direto de se usar o modificador ***abstract*** na declaração de uma classe.

```
        abstract class Funcionario {
                protected double salario;
                public double getBonificacao() {
                        return this.salario * 1.2;
                }
                // outros atributos e métodos comuns a todos Funcionarios
        }
```

Pacotes (ou Namespaces)

São referências para organização lógica de classes e interfaces;

Interface

É um contrato entre a classe e o mundo externo. Quando uma classe implementa uma interface, ela está comprometida a fornecer o comportamento publicado pela interface.

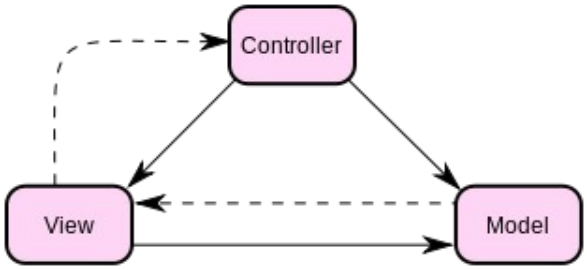
```
        //Declara uma interface
        interface Autenticavel {
                boolean autentica(int senha);
        }

        //instanciando uma interface
```

```
class Gerente extends Funcionario implements Autenticavel {
        private int senha;
        // outros atributos e métodos
        public boolean autentica(int senha) {
                if(this.senha != senha) {
                return false;
        }
        // pode fazer outras possíveis verificações, como saber se esse
        // departamento do gerente tem acesso ao Sistema
        return true;
        }
}
```

Try e Catch (Tratamento de Exceções e Erros)
Umas das utilidades proporcionadas pela orientação a objetos de Java é a facilidade em tratar possíveis erros de execução chamados de exceções.
Sempre que um método de alguma classe é passível de causar algum erro, então, podemos usar o método de tentativa - o ***try.***
Tudo que estiver dentro do bloco ***try*** será executado até que alguma exceção seja lançada, ou seja, até que algo dê errado.
Quando uma exceção é lançada, ela sempre deve ser capturada. O trabalho de captura da exceção é executado pelo bloco ***catch***.
Um bloco ***try*** pode possuir vários blocos de ***catch***, dependendo do número de exceções que podem ser lançadas por uma classe ou método.
O bloco catch obtém o erro criando uma instância da exceção. Portanto, a sintaxe do bloco try catch é:

```
try {
  // código a ser executado
} catch (ClasseDeExceção instânciaDaExceção) {
  // tratamento da exceção
}
```

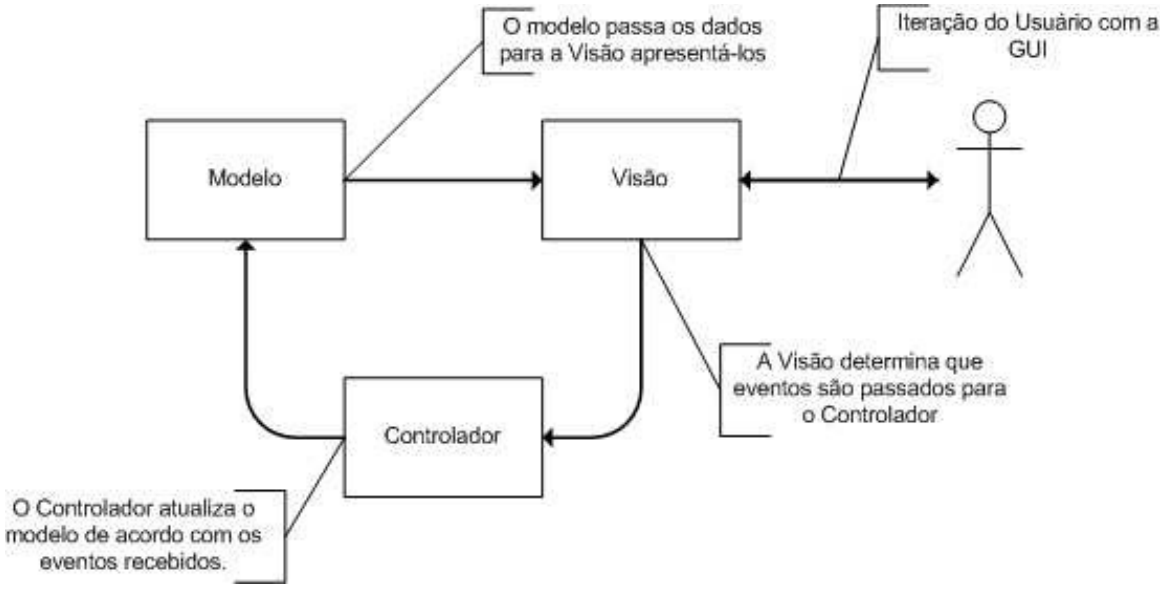
MVC (Model-View-Controller em português Modelo-Visão-Controlador)
É um padrão de arquitetura de software (design pattern) que separa a representação da informação da interação do usuário com ele. O modelo (model) consiste nos dados da aplicação, regras de negócios, lógica e funções. Uma visão (view) pode ser qualquer saída de representação dos dados, como uma tabela ou um diagrama. É possível ter várias visões do mesmo dado, como um gráfico de barras para gerenciamento e uma visão tabular para contadores. O controlador (controller) faz a mediação da entrada, convertendo-a em comandos para o modelo ou visão. As ideias centrais por trás do MVC são a reusabilidade de código e separação de conceitos.



Model ou Modelo: pensamos em regras de negócio, estamos pensando no Modelo da aplicação. Basicamente é isto. No Model podemos ter validações, acesso a banco, acesso à arquivos, cálculos, etc.
O usuário, por exemplo, coloca um produto em um carrinho de compras e é no Model que faremos o cálculo final do pedido(descontos, juros), que validaremos a conta do usuário, que calcularemos o frete, que validaremos o endereço e por aí vai.
Mas como o usuário seleciona um produto em um carrinho de compras por exemplo? Como insere um Endereço para a entrega? Na View!

View ou Visão: A View só existe por um único motivo: Mostrar dados! Na prática isso nunca em muitas vezes não acontece, mas deveria! Você que já desenvolveu em Delphi/VB com toda certeza já viu regras em um botão. Você que já desenvolveu em Java também já viu centenas de regras nas páginas JSP ou espalhadas em Servlets que criam as páginas. Isso é feio! O dia que precisarmos mudar de JSP para JSF ou GWT simplesmente não mudaremos!
Mas e agora? Entendido o conceito básico do Model, como a View passa os valores digitados/selecionados para o Model? Passa direto? Não!

Controller ou Controlador: Aqui temos um maestro! Temos regras de negócio no Controller? Não! Temos visualização no Controller? Não! O Controller simplesmente delega para o Model as solicitações da View. O Controller é burro no sentido de regras de negócio da aplicação. Ele é responsável por saber quem está pedindo algo e a quem enviará este algo! O Controller conhece a View e conhece o Model mas o Model não conhece a View, porém a View observa o Model e este avisa quando seus dados foram atualizados, para a View mostrá-los.



Resumo…
Além de dividir a aplicação em três tipos de componentes, o desenho MVC define as interações entre eles.

- **Um controlador (controller)** pode enviar comandos para sua visão associada para alterar a apresentação da visão do modelo (por exemplo, percorrendo um documento). Ele também pode enviar comandos para o modelo para atualizar o estado do modelo (por exemplo, editando um documento).
- **Um modelo (model)** notifica suas visões e controladores associados quando há uma mudança em seu estado. Esta notificação permite que as visões produzam saídas atualizadas e que os controladores alterem o conjunto de comandos disponíveis. Uma implementação passiva do MVC monta estas notificações, devido a aplicação não necessitar delas ou a plataforma de software não suportá-las.
- **A visão (view)** solicita do modelo, através do controlador, a informação que ela necessita para gerar uma representação de saída.

Array e ArrayList
O Java, por padrão, possui uma série de recursos prontos (APIs) para que possamos tratar de estrutura de dados, também chamados de coleções (collections).

Podemos dizer que ArrayList é uma classe para coleções. Uma classe genérica (generic classes), para ser mais exato.
Coleções mesmo, de qualquer tipo de 'coisa', desde que seja um objeto.
Você pode criar seus objetos - através de uma classe - e agrupá-los através de ArrayList e realizar, nessa coleção, várias operações, como: adicionar e retirar elementos, ordená-los, procurar por um elemento específico, apagar um elemento específico, limpar o ArrayList dentre outras possibilidades.

Como declarar e usar ArrayList em Java
Importe:

import java.util.ArrayList;

Por ser um tipo diferente, sua sintaxe é um pouco diferente do que você já viu até então:

ArrayList<Objeto> nomeDoArrayList = new ArrayList< Objeto >();

No exemplo a seguir, vamos usar um ArrayList de String para trabalhar com o nome de várias Bandas de música:

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

Exemplo de uso do ArrayList
Após declarar a ArrayList 'bandas' que armazenará Strings, vamos adicionar alguns nomes.
Primeiro adicionamos a banda "Rush":

bandas.add("Rush");

Existe um método do ArrayList chamado 'toArray()' que coloca todos os elementos de um ArrayList em um Array.
Ou seja: bandas.toArray() é um Array!

Porém, já vimos que existe um método 'toString' da classe Arrays que retorna uma String com os elementos de um Array. Vamos usar esse método parar exibir todos os elementos do ArrayList, que transformamos em Array através do método 'toArray()':

Arrays.toString(bandas.toArray());

Vamos adicionar a segunda banda, "Beatles" e imprimir, usando o mesmo método.
Note que quando usamos 'add', sempre adicionamos o elemento pro fim da ArrayList.
Confirme isso agora, vendo que a banda "Iron Maiden" ficará depois de "Beatles".
Vamos pegar o primeiro elemento, o elemento '0', através do método 'get':

bandas.get(0);

Note que é a banda "Rush", pois ela foi a primeira a ser adicionada.
Vamos adicionar o "Tiririca" na posição do "Rush", ou seja, na posição '0':

bandas.add(0,"Tiririca");
ou
bandas.add(bandas.indexOf("Rush"), "Tiririca");

Pois o método 'indexOf' retorna o índice em que ocorre "Rush".
Para saber o tamanho que tem seu ArrayList, basta usar o método 'size()':

bandas.size();

Feito isso, rapidamente remova o "Tiririca", pois alguém pode ver.
Para tal, use o método 'remove':

bandas.remove("Tiririca");

Ok. Não quer mais brincar de ArrayList? Remova tudo

bandas.clear();

Código Exemplo:

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

public class arrayLists{

   public static void main(String[] args){
        ArrayList<String> bandas = new ArrayList<String> ();

        bandas.add("Rush");
        System.out.print( "Adicionando a banda Rush: " );
        System.out.println( Arrays.toString( bandas.toArray() ) );

        bandas.add("Beatles");
        System.out.print( "Adicionando a banda Beatles: " );
        System.out.println( Arrays.toString( bandas.toArray() ) );

        bandas.add("Iron Maiden");
        System.out.print( "Adicionando a banda Iron Maiden: " );
        System.out.println( Arrays.toString( bandas.toArray() ) );

        System.out.print( "Quem está na índice 0: " );
        System.out.println( bandas.get(0) );

        System.out.print( "Adicionando Tiririca onde estava o Rush: " );
        bandas.add( bandas.indexOf("Rush"), "Tiririca");
        System.out.println( Arrays.toString( bandas.toArray() ) );

        System.out.print( "Número de elementos na lista: " );
        System.out.println( bandas.size() );

        System.out.print( "Removendo o Tiririca: " );
        bandas.remove("Tiririca");
        System.out.println( Arrays.toString( bandas.toArray() ) );

        System.out.print( "Removendo tudo: " );
        bandas.clear();
        System.out.println( Arrays.toString( bandas.toArray() ) );
      }

}
```

Saída:

```
Adicionando a banda Rush: [Rush]
Adicionando a banda Beatles: [Rush, Beatles]
Adicionando a banda Iron Maiden: [Rush, Beatles, Iron Maiden]
Quem está na índice 0: Rush
Adicionando Tiririca onde estava o Rush: [Tiririca, Rush, Beatles, Iron Maiden]
Número de elementos na lista: 4
Removendo o Tiririca: [Rush, Beatles, Iron Maiden]
Removendo tudo: []
```