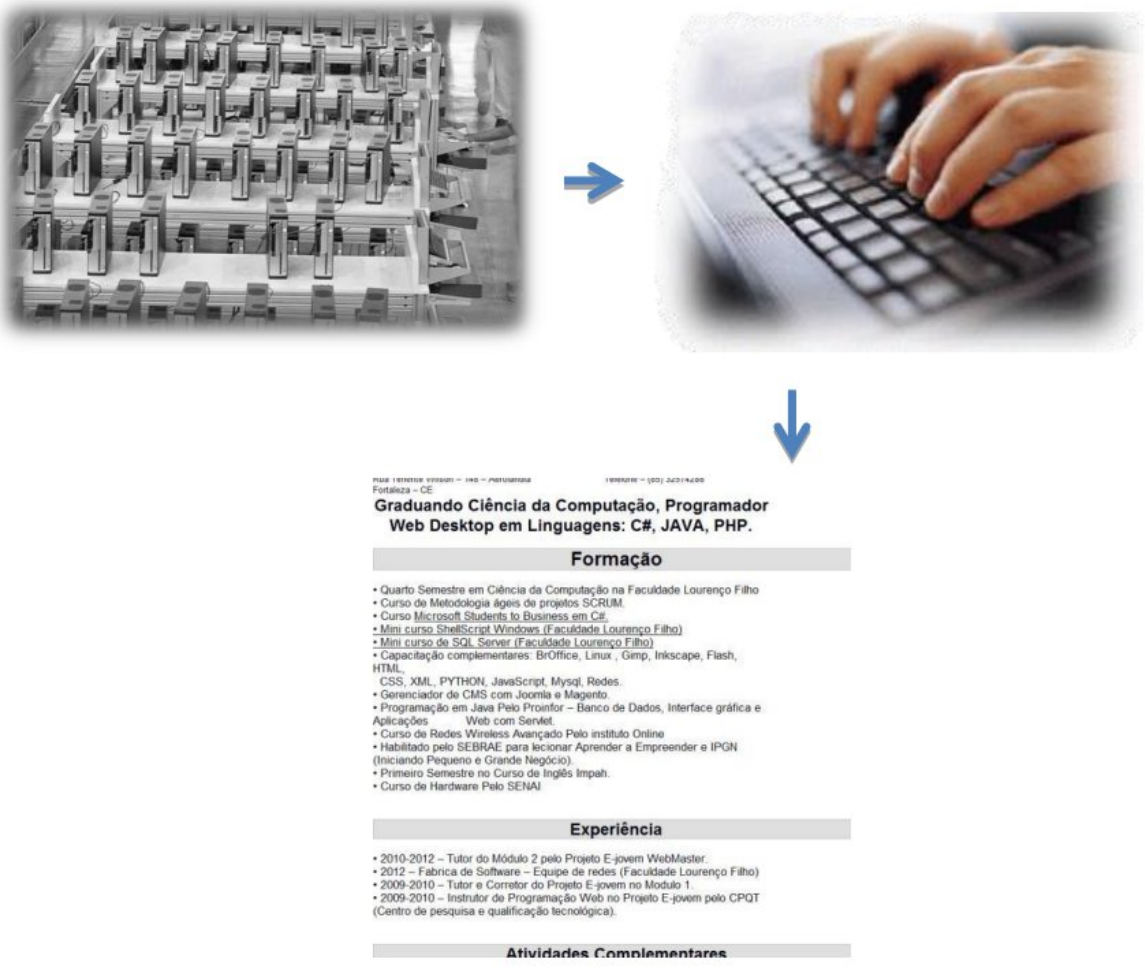
7.0 POO com Java

7.1 Introdução

Programação Orientada a Objetos se constitui em um conjunto de regras de programação. Estudamos anteriormente como programar de forma estruturada, iniciávamos a execução do código e finalizando todos os processos na mesma classe. Porém sem saber estávamos usando algumas regras de POO (Programação Orientada a Objetos) como, por exemplo, usando a classe "System" ou "JOptionPane". Essas classes não foram criadas por nós, porém elas foram reaproveitadas em nosso código com objetivo de resolver os nossos algoritmos. Vamos fazer uma comparação com o dia - a - dia para entendermos melhor.

Imagine que você trabalha com digitação de textos. Você não precisa saber como o computador foi criado, nem como o Software que usa para digitar o texto foi desenvolvido. O seu objetivo é usar desses recursos e criar um novo produto final. Um relatório de texto.



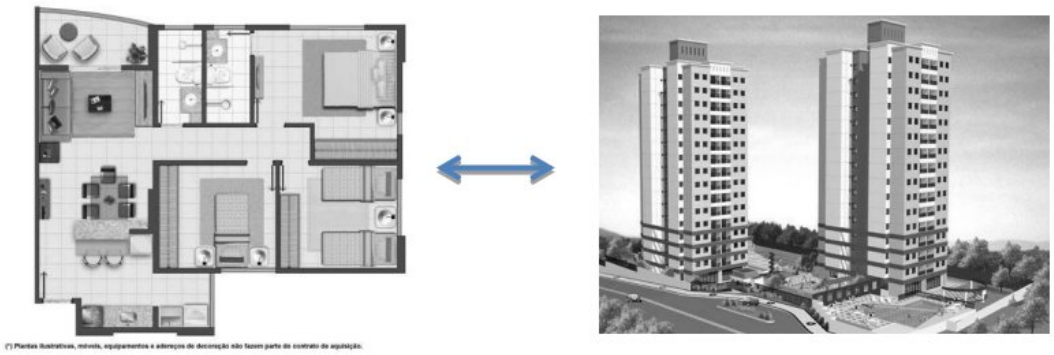
A Programação orientada a objeto é uma metodologia que os programadores utilizam com objetivo de atingir alguns requisitos importantes na construção de um bom programa como:

- **Organizar o código:** Com o nosso código dividido, podemos facilmente dar manutenção ou encontrar algum erro mais fa cilidade.
- **Reutilização do código:** Não precisamos reescrever todo o código, quando necessitamos dele novamente.
- **Manter Padrões:** Utilização de um mesmo padrão conceitual durante todo o processo de criação do código.

Até o final do curso iremos aprender e tra balhar nessa metodologia. Então, bastante atenção nesse e nos próximos capítulos.

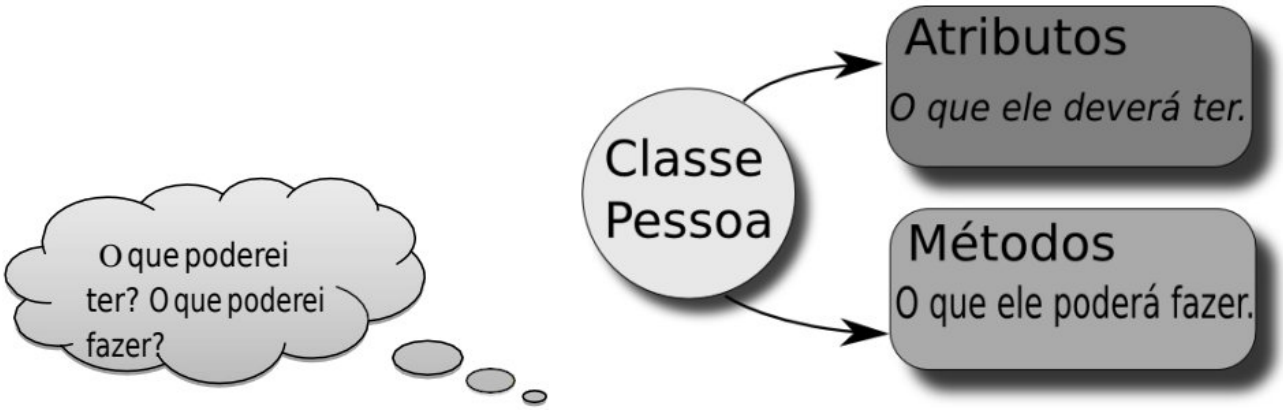
Classe, Atributos e Métodos.

Como estamos trabalhando com POO (Programação Orientada a Objetos), o nosso objetivo final é construir um Objeto que tenha suas características. Um Objeto é criado a partir de uma "Classe". Ela irá servir como modelo para criação do Objeto. Vamos fazer uma comparação com o dia- a -dia para entendermos melhor.



Um prédio não é construído de qualquer forma, o pedreiro irá construir o prédio de acordo com as instruções da planta. Tamanho do banheiro, quantos quartos, porta de entrada. A planta do prédio seria uma "Classe" ele serve como referencia na construção do prédio que é um "Objeto".

Iremos aprender como criar uma "Classe", ou seja, referencia para futuros "Objetos". Imagine que queremos criar um "Pessoa". Para poder criar uma "Pessoa" teremos antes que responder duas perguntas básicas.



- Anote tudo que uma pessoa pode ter, como nome, data de nascimento, RG, CPF, etc . Para a resposta da primeira pergunta, nós temos o que chamamos de " Atributos da classe Pessoa"
- Anote Tudo que uma pessoa pode fazer, ou seja, suas ações, como acordar, dormir, falar o seu nome, etc. Para a resposta da segunda pergunta nós temos o que chamamos de "Métodos da classe Pessoa"

Com o conceito de Classe, atributos e métodos. Podemos agora criar uma classe em código Java, veja no próximo tópico como iremos criá-la.

7.2 Construindo uma Classe.

Para criar uma classe, clicaremos com o botão direito do mouse no pacote do Projeto > novo > Classe Java.

Agora construímos a nossa classe, mas o que significa este "public" antes dos atributos e métodos e o que

```
1  //Pacote da classe
2  package treina;
3  //Construção da Classe
4  public class Pessoa {
5      //Atributos da classe
6      public String nome;
7      public String rg;
8      public String dNascimento;
9      public String cpf;
10
11     //Métodos da classe.
12     public void acordar(){
13         System.out.println("Acordando");
14     }
15
16     public void dormir(){
17         System.out.println("dormindo");
18     }
19
20     public void falarNome(){
21         System.out.println("Meu nome é"+ this.nome);
22     }
23
```

CLASSE

ATRIBUTOS

MÉTODOS

significa o “void”? Para entender melhor, temos que aprender sobre “Encapsulamento” e “tipo de retorno de dados”. Assunto que iremos ver no próximo tópico.

7.3 Encapsulamento.

Encapsulamento é uma forma de proteção do nosso código, existem informações que queremos que outras classes vejam e outras não. Imagine a sua conta de e-mail. A conta do seu usuário poderá ser vista por outras pessoas, pois ele irá identificar o seu e-mail dos demais, porém a sua senha é algo “privado” onde só você poderá ter acesso a ela. Da mesma forma que queremos mostrar ou proteger os nossos dados, uma classe também tem as mesmas necessidades e podemos defini-las através do encapsulamento.

- public - ou publico, qualquer classe pode ter acesso)
- private - ou privado, apenas os métodos da própria classe pode manipular o atributo)
- protected - ou protegido, pode ser acessado apenas pela própria classe ou pelas suas subclasses).

Assim podemos alterar o “public” da nossa classe por outro valor de encapsulamento, dependendo da necessidade de proteção dos nossos atributos ou métodos. Por boas práticas de programação, os atrib utos da nossa classe Pessoa estarão como “private”.

7.4 Retorno

Os métodos podem realizar uma ação interna e podem influenciar em outros métodos através do seu Retorno. Ele pode representar os tipos primitivos de dados que conhecemos como: int, string, double, float, chat, boolean por exemplo, ou pode retornar vazio, simbolizado por “void”. Os métodos que tiver retorno deverá ter dentro do seu corpo um “return” que irá representar o resultado final da minha operação.

Veja o exemplo abaixo:

```
20      public String falarNome(){
21          String minhaFrase = "Meu nome é"+ this.nome;
22
23          return minhaFrase;
24        }
```

Neste exemplo alteramos o método “falarNome” da classe pessoa. O tipo de retorno desse método é String. Então além da classe imprimir a frase, ele deverá guardar a String em uma variável chamada “minhaFrase” e retornar ela para ser usada em outras classes na linha 23.

Construtores, métodos seteres e geteres

CONSTRUTORES

São métodos especiais, que são acionados no momento da construção da classe, ela deverá ter o mesmo nome da Classe criada. Imagine que para que uma pessoa possa existir, ele precise inicialmente de um nome e data de nascimento. Então podemos construir o nosso construtor da seguinte forma:

```
11      public Pessoa(String nome,String dNascimento){
12          this.nome = nome;
13          this.dNascimento = dNascimento;
14      }
```

Nesse código o nosso construtor é responsável por alimentar os atributos nome e data de nascimento da pessoa. também garantimos que ao momento da construção do objeto, uma "Pessoa" não poderá existir se não tiver um nome e uma data de nascimento.

Mas como eu poderei alterar o meu CPF, RG, e como poderei mostrar esses valores de atributos? Assunto que veremos no próximo tópico.

MÉTODOS SETERES E GETERES

Os métodos nomeados como "set" e "get" são responsáveis respectivamente por alterar um atributo e mostrar um atributo da minha classe. Para que o NetBeans possa gerar esse código, iremos digitar: ALT+ INSERT > INSERIR CÓDIGO e marcar os atributos, "nome", "cpf", "rg".

Como eu poderei usar o meu construtor, meus métodos do tipo "set" e "get"?

Este assunto veremos no próximo tópico.

7.5 Estanciando (Construindo um Objeto)

Agora estamos prontos para construir o nosso objeto usando a nossa classe "Pessoa" como referencia. Então, iremos voltar à classe que tenha o método main e realizar a seguinte operação:

```
1   package treina;
2
3   public class Main {
4
5       public static void main(String[] args) {
6
7       Pessoa x = new Pessoa("Fabricio", "17/02/1987");
8
9       }
10
11  }
```

Agora estou estanciando o meu objeto e de acordo com meu construtor, eu preciso obrigatoriamente de um nome = “Fabrício” e uma data de nascimento = “17/02/87” todos eles do tipo String. Vamos agora usar os métodos da nossa classe criada.

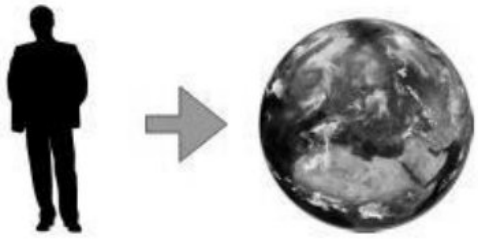
```
7       Pessoa x = new Pessoa("Fabricio", "17/02/1987");
8
9       x.setCpf("888888888-888");
10      x.setRg("5555555555555");
11
12     System.out.println(x.falarNome());
13     System.out.println("Meu RG - " + x.getRg());
14     System.out.println("Meu CPF - " +x.getCpf());
15
16      }
```

Dessa forma podemos modificar e acessar qualquer variável dependendo das nossas necessidades. Todas as informações do “Fabrício” estão dentro de uma variável do tipo “Pessoa” chamada de “x”. Se eu quiser criar uma nova pessoa, posso chamá -la de qualquer outro valor.

7.6 Variáveis Estáticas

Variáveis estáticas são valores que se for alterado em um objeto de uma determinada classe todos os objetos da estanciados da mesma classe, automaticamente terá o mesmo valor Ele normalmente é usado quando todas as classes tem algo em comum. Imagine a seguinte situação.

A Classe “Pessoa” terá um atributo chamado: ondeVivo = “Planeta Terra”



Todas as pessoas moram em um planeta que conhecemos por “Planeta Terra”. Porém pesquisas mais recentes mostram que o nome do nosso planeta deveria ser chamado de “Planeta Água”, pois nele encontra-se mais água do que a terra.

Se tivermos 300 objetos estanciados do tipo pessoa, como alterar ondeVivo = “Planeta Água” em todos os objetos em uma única instrução sem precisar percorrer todos eles?

Basta criar uma variável estática no atributo “ondeVivo”, assim se alteramos em 1 dos objetos essa variável, todas elas sofrerão a mesma alteração. Veja como criá -lo:

```
//Atributo estático criado na classe Pessoa
private static String ondeVivo = "Planeta Terra";

//Método set para alterar a variável estática
public void setOndeVivo(String ondeVivo) {
    Pessoa.ondeVivo = ondeVivo;
  }
//Método get para a variável estática
public String getOndeVivo() {
    return Pessoa.ondeVivo;
  }
```

Concluímos então que as variáveis estáticas não pertencem especificamente a nenhuma instancia de objeto e sim uma referencia da classe. O que explica a substituição do termo "this" para o nome da classe que contém a variável estática, no exemplo acima alteramos o termo "this" no método setOndeVivo pelo nome da classe que contém a variável estática (Pessoa).

7.7 Métodos Estáticos

Os Métodos Estáticos se caracterizam por ser um referencia de método da Classe, sem a necessidade de instanciarmos um objeto para utilizá -lo. Se quisermos, por exemplo, alertar ou mostrar o habitat das pessoas sem precisar construir um objeto, podemos realizar essa ação apenas adicionando a palavra reservada static nos métodos, dessa forma:

```
//Atributo estático criado na classe Pessoa
private static String ondeVivo = "Planeta Terra";

//Método ESTÁTICO set para alterar a variável estática
public static void setOndeVivo(String ondeVivo) {
    Pessoa.ondeVivo = ondeVivo;
  }
//Método ESTÁTICO get para a variável estática
public static String getOndeVivo() {
    return ondeVivo;
  }
```

Obs.: Métodos estáticos aceitam apenas variáveis estáticas.

Para que possamos utilizar os métodos estáticos precisamos inserir o nome da classe com o método estático criado.

```
public static void main(String[] args) {
    Pessoa.setOndeVivo("Planeta Água");
    System.out.println("Vivemos no : "+Pessoa.getOndeVivo());
}
```

8.0 Herança e Interface

8.1 Introdução

Estudamos como estruturar as nossas classes, agora precisamos gerar uma interação entre classes criadas. Para isso no conceito de POO também temos que saber Interface e herança. Iremos aprender cada uma delas, seu conceito e como implementá-las.

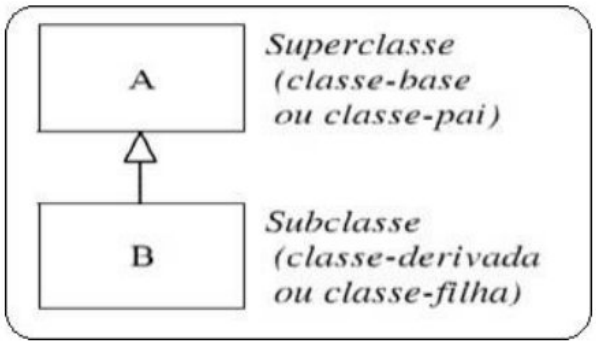
8.2 Herança

É fundamental para Java e outras linguagens de programação orientadas a objetos, o conceito de herança, que permite ao código definido em uma classe ser reutilizado em outras classes.
Sintaxe de herança em Java:

```
class NomeDaClasseASerCriada extends NomeDaClasseASerHerdada
```

Em Java, pode ser definida uma superclasse geral (mais abstrata), e depois estendê-la com subclasses mais específicas. A superclasse não sabe nada sobre as classes que herdam dela, mas todas as subclasses precisam declarar explicitamente a relação de herança. Uma superclasse recebe automaticamente as variáveis de instâncias acessíveis e os métodos definidos pela superclasse, mas é

também livre para substituir métodos da superclasse para definir comportamento mais específico. Os objetos de uma mesma classe podem tirar proveito de relações de herança, onde objetos recém-criados absorvem características de outros já existentes e adicionam -se novas características.



8.3 Object: A superclasse cósmica do Java

A class Object: A classe Object é o último ancestral – toda classe em Java estende de Object. Entretanto, nunca será preciso gerar sua herança diretamente:

A superclasse terminal Object é automaticamente inclusa se nenhuma superclasse for mencionada explicitamente. Pelo fato de todas as classes em Java estenderem Object, é importante familiarizar -se com os serviços fornecidos pela classe Object. Remeteremos ao leitor a pesquisarem em artigos na internet ou até mesmo ver a documentação on-line que lista os vários métodos da classe Object, mas que só aparecem quando lidamos com threads (linhas de execução). Você pode usar variáveis do tipo Object para referir-se a objetos de qualquer tipo.

É claro que uma variável do tipo Object somente será útil como um marcador genérico para valores arbitrários. Para fazer qualquer coisa específica com o valor, você precisa ter algum conhecimento sobre o tipo original e, então, aplicar uma conversão: Em Java, somente os tipos primitivos (números, caracteres e valores lógicos) não são objetos. Todos os tipos de arrays, independentemente de serem arrays de objetos ou arrays de tipos primitivos, são tipos de classes que estendem a classe Object.

Super: A palavra super seguida imediatamente de parênteses chama o construtor da superclasse de mesma assinatura, deve ser a primeira instrução dentro de um construtor da subclasse.

Super-classe: É uma classe da qual se herda outras classes. A cláusula opcional estende em uma declaração de classe especifica a superclasse diretamente da classe atual. Uma classe é dito ser uma subclasse direta da classe, que se estende. A superclasse direta é a classe a partir de cuja aplicação à

implementação da classe atual é derivada. A cláusula extends não deve aparecer na definição da classe java.lang.Object,pois é primordial a classe e não tem nenhuma superclasse direta. Se a declaração de classe para qualquer outra classe não tem nenhuma cláusula extends, então a classe tem a classe java.lang.Object como implícita superclasse direta.

Sub-classes: É uma classe que herda de uma classe ou de uma interface. Uma subclasse herda estado e comportamento de todos os seus antepassados. A superclasse o termo refere-se ancestral direto de uma classe, assim como todas as suas classes ascendentes. Essa hierarquia de classes segue uma apresentação na forma de árvore, em que a raiz é a superclasse de todo o sistema. Cada nível abaixo dessa superclasse do sistema acrescenta funções específicas.

Exemplo 01: Neste exemplo vamos mostrar como proceder com esse poderoso recurso do Java o qual chamamos de herança simples.
Código da superclasse Conta.

```
class Conta {                /**nome da classe*/
    long numero;             /**atributo da classe*/
    float saldo;             /**atributo da classe*/
    String cpf;              /**atributo da classe*/

    public Conta(){          /**construtor da classe*/
    }
    public Conta(long numero,String cpf){   /**construtor com argumentos*/
      this.numero = numero;
      this.cpf = cpf;
    }
    public void depositar(float valor){   /**método da classe*/
      this.saldo = saldo+valor;
    }
    public boolean sacar(float valor){    /**método da classe*/
      this.saldo = saldo-valor;
      return true;
    }
}
```

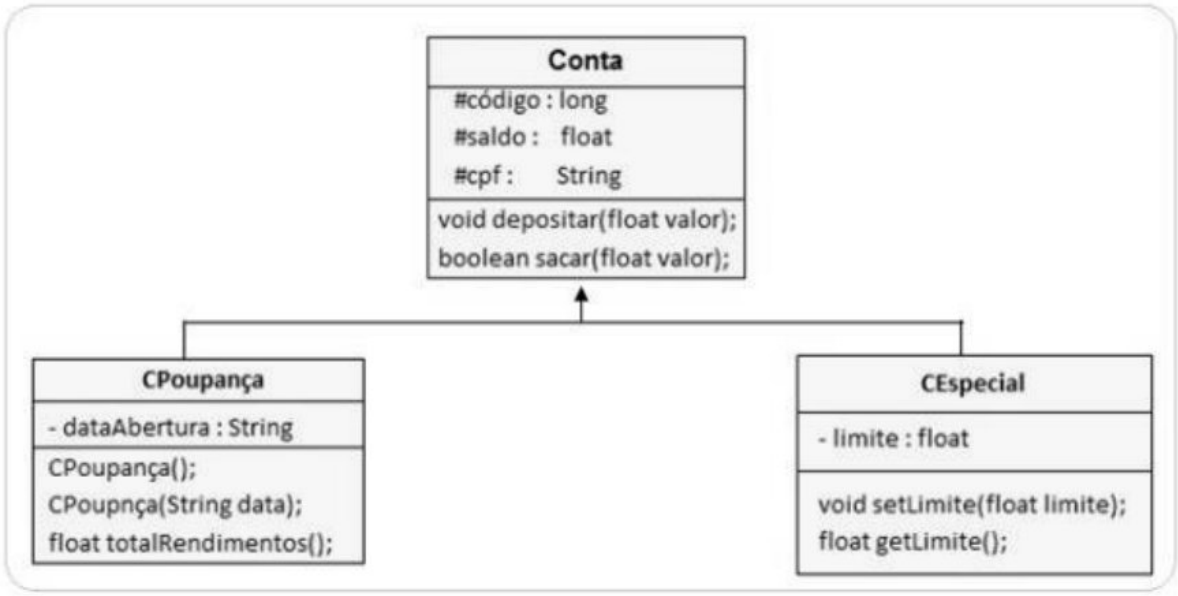
Código da subclasse **CPoupanca** que estende da superclasse **Conta**.

```
class CPaupanca extends Conta {     /**A classe CPoupança que herda de Conta*/
   private String dataAbertura;    /**atributo da classe CPoupança*/

   public CPaupanca() {            /**construtor da classe CPoupança*/
    }

   public CPaupanca(String data) {  /**construtor com argumentos*/
     super(7238, "2398273182-29");  /**chamada ao construtor da siuperclasse*/
     this.dataAbertura = data;
    }

   public float totalRendimentos(){
     return super.saldo = saldo*0.46f;
   }
}
```

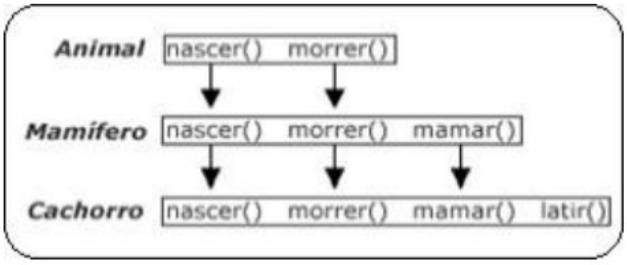
Código da subclasse **CEspecial** que estende da superclasse **Conta**.

```
class CEspecial extends Conta{  /**A classe CEspecial que herda de Conta*/
   private float limite;        /**atributo da classe CEspecial*/

   public CEspecial(){          /**construtor da classe CEspecial*/
     super(29183,"7342523839-89"); /**chamada ao construtor da superclasse*/
     limite = 510.36f;
   }
   public void setLimite(float limite){  /**método da classe CEspecial*/
     this.limite = limite;
   }
   public float getLimite(){             /**método da classe CEspecial*/
    return this.limite;
   }
}
```

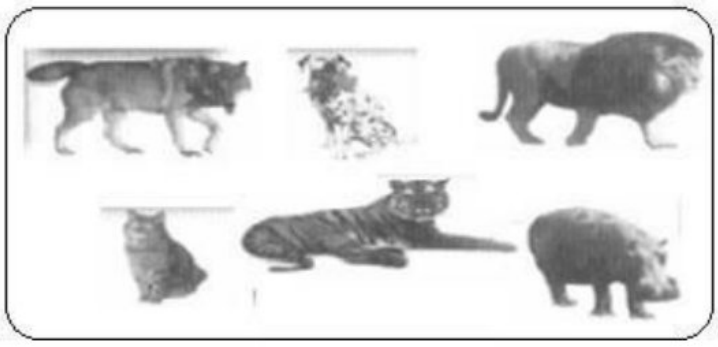
Diagrama de classes representando a herança entre as classes citadas acima.



Nota: Java permite que uma classe herde apenas as características de uma única classe, ou seja, não pode haver heranças múltiplas. Porém, é permitido heranças em cadeias, por exemplo: se a classe Mamífero herda a classe Animal, quando fizermos a classe Cachorro herdar a classe Mamífero, a classe Cachorro também herdará as características da classe Animal.



Como estamos tratando de herança de classes, toda classe tem seu método construtor. Portanto, se estamos trabalhando com duas classes, temos dois métodos construtores. Para acessarmos o método construtor da classe que está sendo herdada usamos o super().

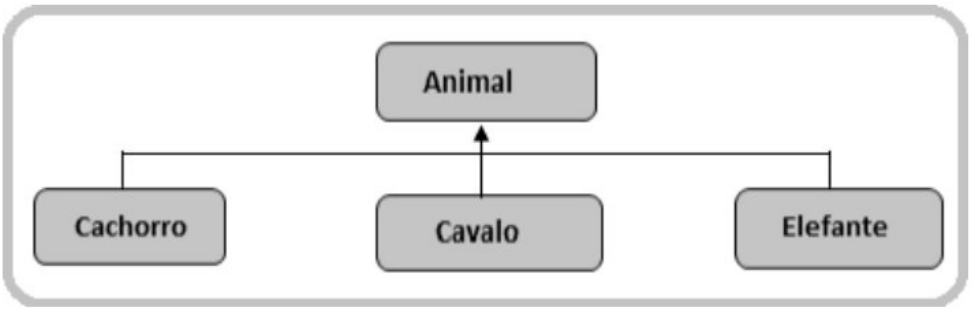


Exemplo 02:

Para demonstrar como definir o relacionamento de herança, considera-se o contexto de ter que desenvolver um programa que simule o comportamento de vários tipos de animais como mostrado na figura ao lado em um determinado ambiente.

Inicialmente, apenas um grupo de animais estará presente neste ambiente e cada animal deve ser representado por um objeto. Além disso, os animais movem -se no ambiente ao seu modo e podem fazer um conjunto de coisas. O programa deverá prever que novos tipos de animais poderão ser incluídos no ambiente.

O primeiro passo é observar cada animal e definir o que cada objeto do tipo animal tem em comum no que diz respeito aos atributos e comportamentos (métodos). Deve-se definir também como os tipos de animais se relacionam. Inicialmente, o programa deve simular o comportamento dos animais ilustrados na fi gura, ou seja, de um leão, de um lobo, de um gato, de um hipopótamo, de um tigre e de um cachorro.



O segundo passo consiste em projetar a superclasse, ou seja, a classe que representa o estado e o comportamento em comum a todos os animais. Para este exemplo, como todos os objetos são animais, a superclasse foi denominada como Animal e as variáveis e métodos em comuns a todos os animais foram atribuídos a ela, como ilustrado na Figura. Neste exemplo, cinco variáveis foram definidas: (o tipo de comida que o animal come), (o nível de fome do animal), (representação da altura e largura do espaço que o animal vagará ao seu redor) e (as coordenadas X e Y do animal no espaço). Além disso, quatro métodos foram definidos para definir o comportamento dos animais: (comportamento do animal ao fazer algum ruído), (comportamento 57 do animal ao comer), (comportamento do animal dormindo) e (comportamento do animal quando não está nem dormindo nem comendo, provavelmente vagueando no ambiente).

O terceiro passo é o de decidir se alguma subclasse precisa de comportamentos (métodos) específicos ao seu tipo de subclasse. Analisando a classe Animal, pode -se extrair que os métodos devem ter implementações diferentes em cada classe de animais, afinal, cada tipo de animal tem comportamento

distinto ao comer e fazer ruídos. Assim, esses métodos da superclasse devem ser sobrescritos nas subclasses, ou seja, redefinidos. Os métodos não foram escolhidos para serem redefinidos por achar que esses comportamentos podem ser generalizados a todos os tipos de animais.

CONSTRUTORES DA SUBCLASSE

```
class MinhaClasse{
   private String nome;      /**a variável nome está encapsulada*/
   private String email;     /**a variável email está encapsulada*/
   private int salario;      /**a variável salario está encapsulada*/

}
```

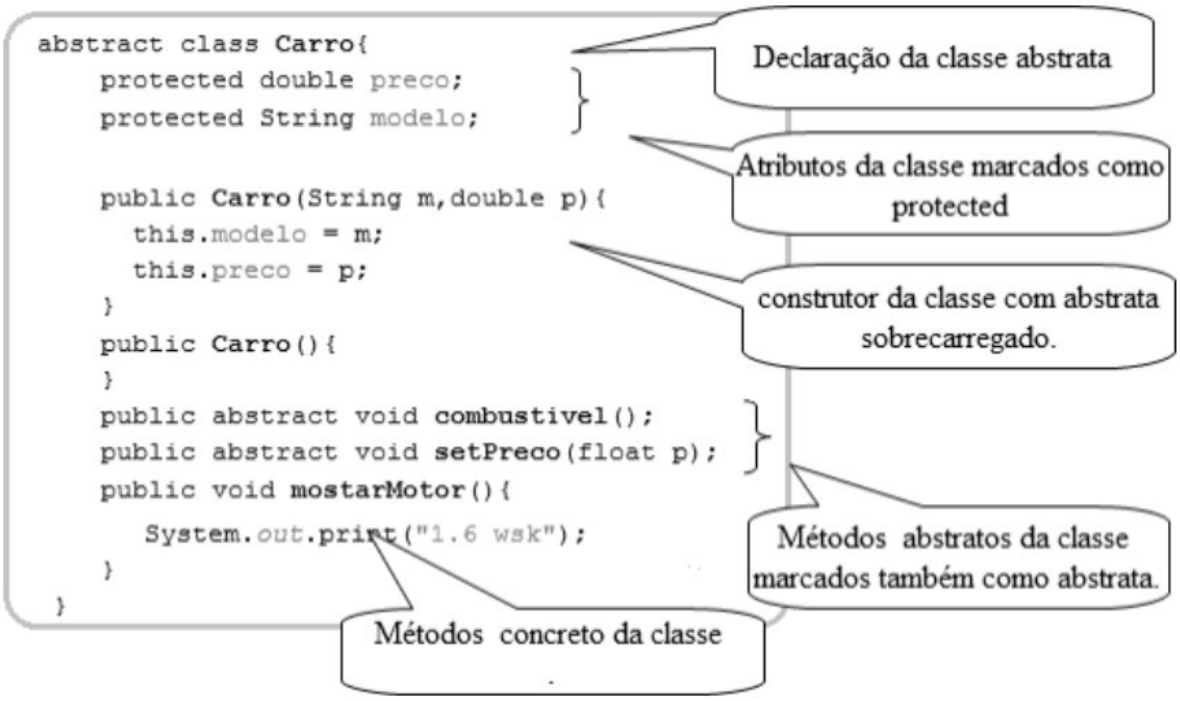
Os construtores da superclasse não são herdados pela subclasse, logo, esta deverá ter seus próprios construtores para inicializar seus atributos. Todavia, ao instanciar um objeto da subclasse, devemos inicializar aqueles atributos que ela está herdando, e o fazemos chamando algum construtor da superclasse, usando a chamada super(...).

8.3.1 Classe Abstrata

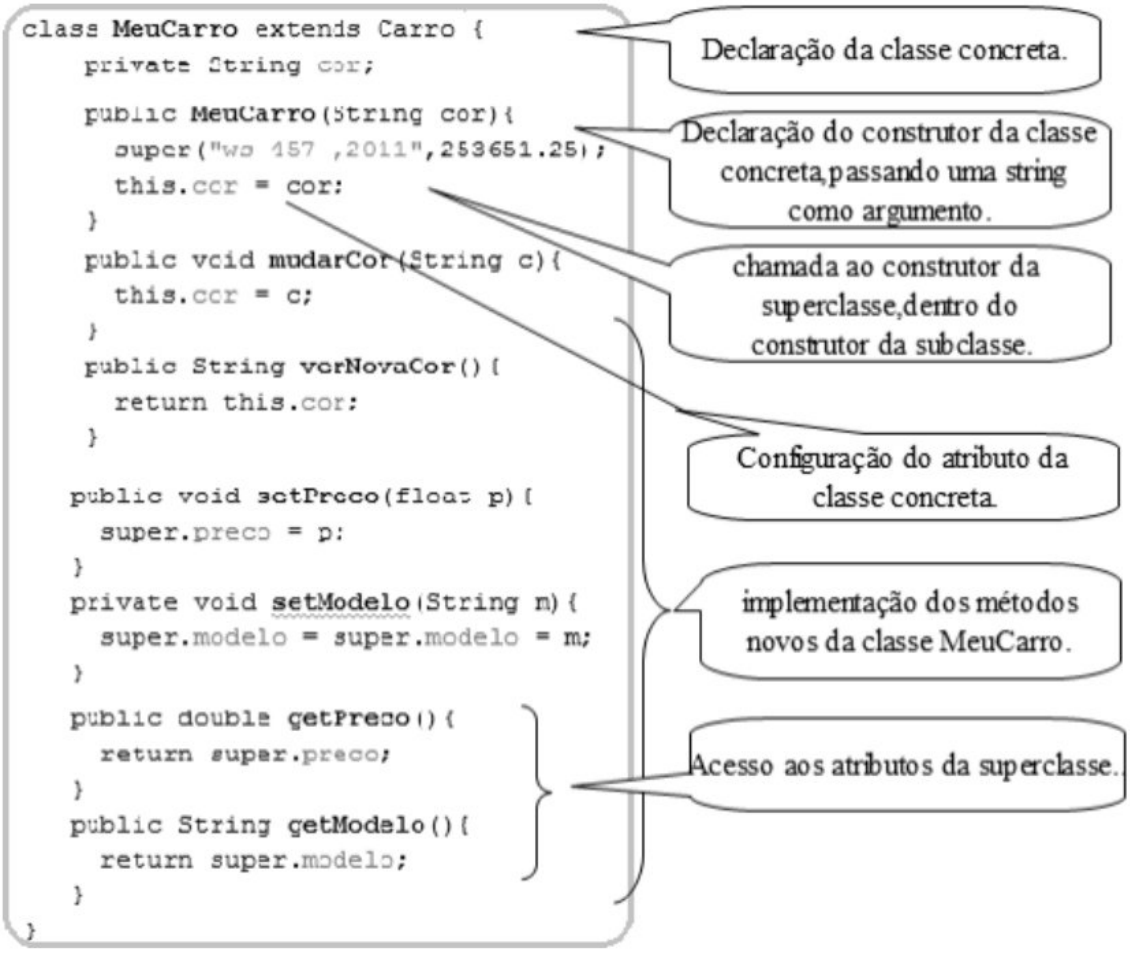
Para entender o conceito de herança, precisamos primeiro entender No Português, a palavra “Abstrata” refere-se a algo que não existe fisicamente, então se convertermos essa expressão em Java podemos dizer que Classe Abstrata é uma classe que não pode ser um Objeto. Assim ela precisará ser herdada ou implementada, no próximo tópico iremos ver como funciona em um exemplo de código.

Imagine que você tenha uma classe Carros que tenha métodos genéricos comuns a todos os veículos. Mas você não quer que alguém de fato crie um objeto carro genérico, abstrato. Como se inicializaria o seu estado? De que cor ele seria? Quantos assentos? Ou, mais importante, de que forma ele se comportaria? Em outras palavras, como os métodos seriam implementados? Temos certeza de que o dono de uma BMW lhe diria que o carro dele é capaz de fazer coisas que o GOL apenas sonha em fazer.

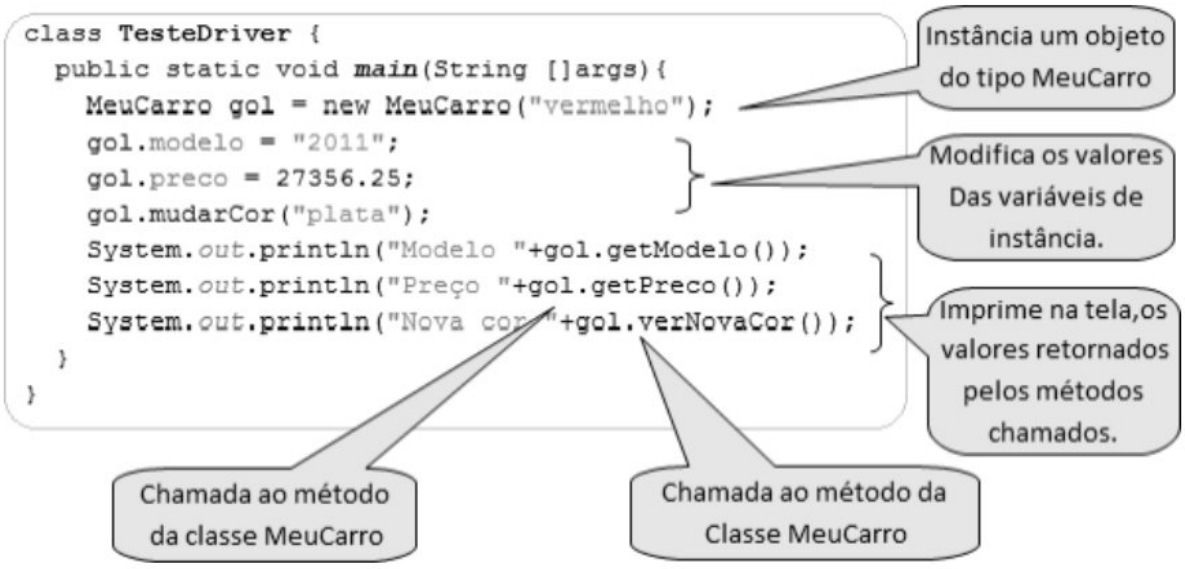
A implementação pode ser dessa forma mostrada acima, como também pode ser como segue no exemplo abaixo:



Dessa forma podemos dizer que “MeuCarro” que poderá ser um Objeto, herda de “Carro” que é uma classe abstrata.



Na Classe que contém o método Main, nós temos:



8.4 Interface

Interface é uma espécie de superclasse 100% abstrata que define os métodos que uma subclasse deve suportar, mas não como esse suporte deve ser implementado. Isso porque ao criar uma interface, estará definindo um contrato com o que a classe pode fazer, sem mencionar nada sobre como o fará. Uma interface é um contrato. Qualquer tipo de classe (concreta) e de qualquer árvore de herança pode implementar uma interface.

Exemplo 01:

```
interface Funcionario{  /**declara uma interface*/
}
```

Exemplo 02:

```
public abstract interface Funcionario{  /**declara uma interface*/
}
```

Exemplo 03:

```
public abstract interface Funcionario{       /**declara uma interface*/
   public abstract void receberSalario();    /**declara um método abstrato*/
   public abstract void modificaSalario();   /**declara um método abstrato*/
}
```

Regras para as interfaces:

- Uma interface em Java é uma coleção de métodos abstratos que pode ser implementada por uma classe.
- Uma interface na pode implementar outra interface ou classe.
- Os tipos de interface podem ser usados polimorficamente
- Os métodos de uma interface não devem ser estáticos.
- Uma interface deve ser declarada com a palavra -chave interface.
- Todos os métodos de uma interface são implicitamente public e abstract. As interfaces só podem ser declaradas. Uma interface pode estender uma ou mais interfaces diferentes.

Todas as variáveis definidas em uma interface devem ser public, static e final.

As interfaces só podem ser declarar constantes e não variáveis de instância.

9.0 Herança (Polimorfismo)

9.1 Introdução

Neste capitulo veremos como funciona a interação com classes em Polimorfismo. Usamos esse recurso quando queremos modificar métodos herdados para isso temos que aprender o conceito e prática de sobrecarga e sobrescrita. Veremos estes assuntos no próximo tópico.

Sobrecarga

Usamos sobrecarregar de métodos quando temos um determinado método com o mesmo nome, porém com assinaturas diferentes, sou seja, parâmetros diferentes, dessa forma quando usamos um determinado método com sobrecarga podemos escolher qual método sobrecarregado será usado de acordo com o nú mero e tipo de dados dos parâmetros oferecidos. Veja um exemplo abaixo:

```
1  package treina;
2  public class Animal {
3
4      public void comer(){
5          System.out.println("comendo..");
6      }
7
8      public void comer(boolean carneEstragada){
9          if(carneEstragada == true){
10             System.out.println("Procurar outro alimento");
11         }else{
12             System.out.println("comendo..");
13         }
14     }
15 }
16
```

Na Classe animal criamos dois métodos chamados comer, o método que não contém nenhum parâmetro, o animal apenas mostra a ação de comer, estado da carne. O nosso segundo método, também se chama comer, porém ele tem um assinatura diferente, ele recebe como parâmetro um valor boleano que se refere ao estado da carne, dessa forma o animal pode simplesmente comer ou verificar o tipo de alimento antes de se alimentar.

9.2 SobreEscrita

Usamos sobrescrita de métodos quando temos que modificar um método herdado da classe pai, para sobrescrever os métodos teremos que criar o mesmo método herdado na classe filha com o nome assinaturas igual. Levando em consideração a classe Animal criada no tópico anterior, veja no exemplo abaixo

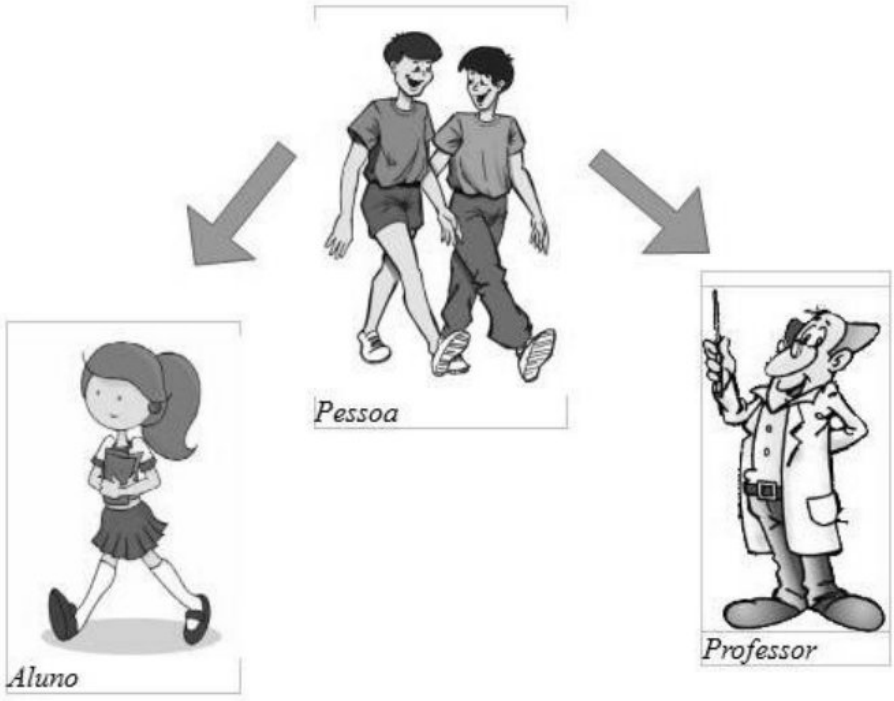
```
1  package treina;
2
3  public class Cachorro extends Animal{
4      @Override
5      public void comer(){
6          System.out.println("ração...");
7      }
8
9  }
```

No exemplo anterior, o cachorro já herda o método comer, porém o animal come qualquer alimento, este nosso cachorro não comer qualquer alimento, ele só come ração. O termo "@Override" no exemplo, não é obrigatório, apenas para avisar a IDE que este é um método de sobrescrita.

Polimorfismo

Podemos dizer que polimorfismo é quanto temos métodos de sobrescritas distintos entre classes filhas que referencia um mesmo Pai.

Veja o exemplo abaixo.



Note as roupas que eles usam. A pessoa por padrão tem um estilo de roupa, porém dependendo da situação, temos que usar roupas específicas para cada situação. Um professor também é uma pessoa, porém do tipo "Professor" e ele precisa de um tipo específico de uniforme para dar suas aulas. O aluno também é uma Pessoa, porém ele precisa de um fardamento adequado para assistir as aulas. Assim nós temos a classe "Professor" e "Aluno" herdando de "Pessoa" Porém cada uma delas tem métodos

diferentes quando nos referimos como eles se vestem. Vamos ver como ficaria em código Java o nosso Polimorfismo.

Classe Pai:

```
1  package treina;
2  public class Pessoa {
3
4      public void vestirRoupa(){
5          System.out.println("vestinho uma roupa leve para um dia de sol");
6      }
7
8  }
```

Classe Filha de "Pessoa" que sobrescreve método vestirRoupa.

```
1  package treina;
2  public class Professor extends Pessoa{
3      @Override
4      public void vestirRoupa(){
5          System.out.println("Vestindo uniforme de Professor");
6      }
7
8  }
```

Classe Filha de "Pessoa" que sobrescreve método vestirRoupa.

```
1  package treina;
2
3  public class Aluno extends Pessoa{
4      @Override
5          public void vestirRoupa(){
6              System.out.println("vestido fardamento do Aluno");
7      }
8
```

10.0 Tratamento de Exceções

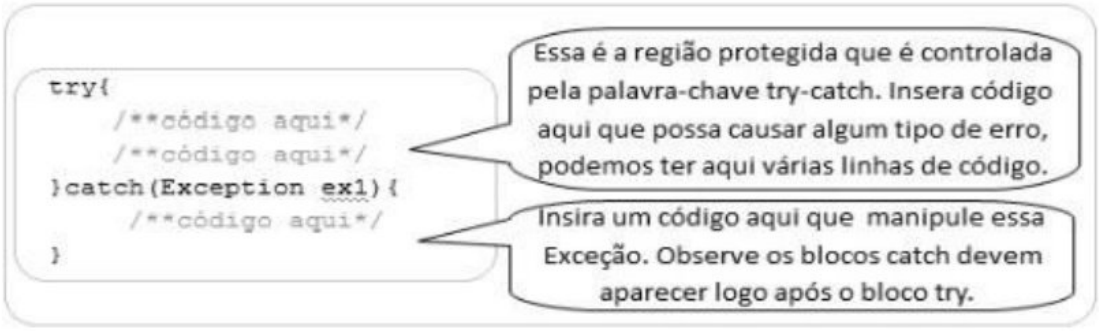
10.1 Capturando uma exceção usando o bloco (try e catch)

A linguagem Java fornece aos desenvolvedores um mecanismo sofisticado para manipulação de erros que produz códigos de manipulação eficientes e organizados: a manipulação de exceções. A manipulação de exceções permite que os desenvolvedores detectem erros facilmente sem escrever um código especial para testar valores retornados. Permite também manter o código de manipulação de exceções nitidamente separado do código que gerará a exceção. Além disso, permite que o mesmo código de manipulação de exceções lide com as diferentes exceções possíveis.

A classe Throwable é a única classe cujas instâncias diretas ou indiretas podem ser usadas para serem lançadas. A classe Throwable é pré -definida na linguagem Java, juntamente com várias outras classes definindo um grande número de categorias de erro. Os programadores Java, por convenção, caso queiram criar novas exceções devem estender a classe Exception que é uma subclasse de Throwable. Duas classes descendentes da classe Throwable podem ser consideradas especiais: a classe Error e a classe RuntimeException.

A linguagem Java não obriga o programador a tratar os lançamentos de exceção envolvendo instâncias da classe Error, da classe RuntimeException e das classes delas derivadas, todos os outros lançamentos de exceção devem ser tratados pelo programador. Capturando uma exceção com o bloco try e catch.

Vejamos a sintaxe.



O termo exceções significa „"condição excepcional"", e é uma ocorrência que altera o fluxo normal do programa. Várias coisas podem levar a exceções, incluindo falhas do hardware, exaustão de recurso e os famosos erros. Quando isso acontece, ou seja quando ocorre um evento excepcional ocorre em Java, diz se que uma exceção será lançada.

Exemplo 01:

```
public void calc(){
  try{
      int x = 13;
      int y = 0;
      if(x>y){
        float div = (x/y);
        System.out.println("Divisão: "+div);
      }
  }catch(Exception ex1){
        System.out.println("Ocorreu um erro");
  }
}
```

Nota: O método calc se for chamado com y igual zero (0), gerará um erro e uma exceção será lançada, esse erro poderá ser sinalizado no bloco try-catch, então a execução do método pulará para o bloco de código catch, onde será impressa uma mensagem de erro.

Exemplo 02:

Caso 01: Neste exemplo vamos chamar o método converteStr(), passando uma string numérica, que no nosso caso é 8 e o resultado é a conversão da string em um número de ponto flutuante, daí a divisão é feita e a execução do programa acontece sem nenhum problema.

```
public class MinhaExcecao {
   public void converteStr(String n){
      try{
          float x = 13.8f;
          float y = Float.parseFloat(n);
          float div = (x/y);
          System.out.println("Divisão: "+div);

      }catch(Exception ex1){
            System.out.println("Não é uma String numérica");
      }
   }
```

Caso 02: Neste mesmo exemplo vamos chamar agora o método converteStr(), passando uma string não numérica, que no nosso caso é a caractere 'n' e o resultado é o lançamento de uma exceção, jogando o fluxo de execução do programa para o bloco catch, onde imprime uma mensagem na no console.

```
public static void main(String []args){
   new MinhaExcecao().converteStr("n");
}
```

FINALLY

Finally: É o trecho de código final. A função básica de finally é sempre executar seu bloco de dados mesmo que uma exceção seja lançada. É muito útil para liberar recursos do sistema quando utilizamos, por exemplo, conexões de banco de dados e abertura de buffer para leitura ou escrita de arquivos.

```
class Tratamento {
    public static void main(String[] args) {
        try {
            int x = 12;
            int y = 0;
            float z = (x / y);
            System.out.println("Tratando exceções em java" + z);
        } catch (Exception ex) {
            ex.printStackTrace();
        } finally {
            System.out.println("Esse bloco eh executado.");
        }
    }
}
```

Resultado do código acima:

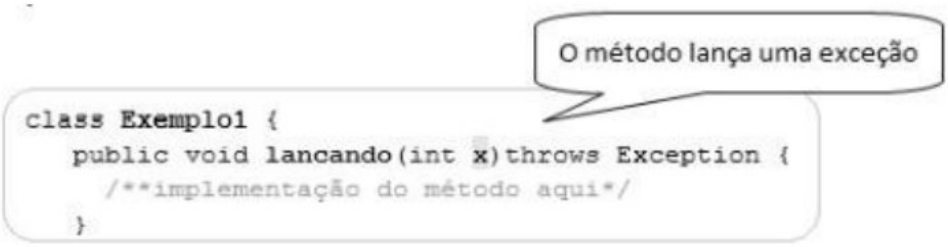
```
Saída - projeto_apostila (run)                    Tarefas
run:
Esse bloco eh executado.
java.lang.ArithmeticException: / by zero
        at br.com.apostila.Tratamento.main(PassagemPorReferencia.java:44)
CONSTRUÍDO COM SUCESSO (tempo total: 0 segundos)
```

LANÇADO UMA EXCEÇÃO COM O THROWS

O objeto de exceção é uma instância da classe Exception ou de uma de uma de suas subclasses. A cláusula catch usa, como parâmetro, uma instância de objeto de um tipo derivado da classe Exception. A linguagem Java exige que todo método capture a exceção verificada que pode vir a lançar ou então

declare que lançará a exceção. A declaração da exceção faz parte da interface pública do método. Na declaração de uma exceção que poderá ser lançada, a palavra-chave throws é usada em uma definição do método, junto com uma lista de todas as exceções verificadas que poderão ser lançadas.

Exemplo 01:

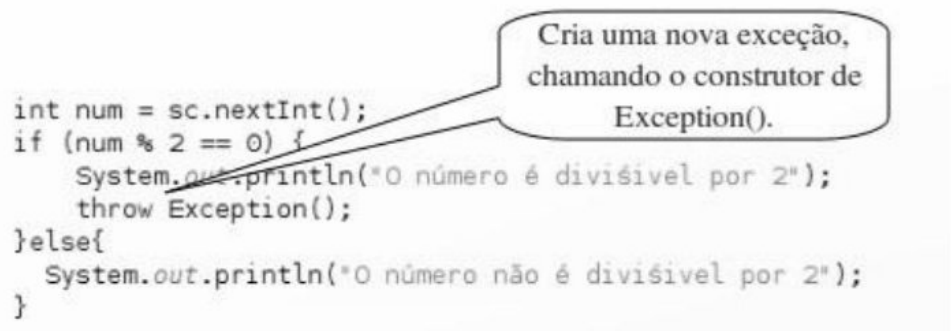


Exemplo 02:

```
class Exemplo2 {
  public static void main(String []args) throws Error{
     if(args[0]==null){
        System.out.println("Lança uma Exceção!");
      }else{
         System.out.println(args[0]);
      }
   }
}
```

CRIANDO UMA EXCEÇÃO COM THROW

A máquina Virtual Java lança referências para instâncias de maneira implícita utilizando as classes pré-definidas. O programador pode utilizar o comando throw de forma explícita. Quando o fluxo de controle atinge o comando throw <expressão>, a expressão é avaliada. Esta expressão corresponde em geral à criação de um objeto e resulta numa referência, p. ex. throw new ErroDoTipoX();. A partir daí o fluxo de controle será desviado para uma cláusula catch apropriada de algum comando try-catch.



O fluxo de controle segue a cadeia dinâmica dos registros de ativação das invocações dos métodos, ou

seja a execução de um método pode terminar (i) por que o fluxo de controle atingiu o final do método (return implícito, somente no caso d e métodos do tipo void!), (ii) porque o fluxo de controle atingiu um comando return, ou (iii) porque foi executado um throw implícito ou explícito que não foi apanhado por um comando try -catch daquele método. A procura por um try-catch apropriado é propagada até o ponto em que há um retorno para o método main, neste caso a execução do programa será finalizada, com mensagem de erro provida pela MVJ dizendo que uma exceção foi lançada sem que fosse apanhada. E deverá ser mostrado algo como:

```
Saída - projeto_apostila (run)
run:
Entre com um número
8
Exception in thread "main" java.lang.NullPointerException
O número é divisível por 2
Trata a Minha Exceção!
        at br.com.teste.NovaExcecao.criarExcecao(NovaExcecao.java:13)
        at br.com.teste.TesteDriver.main(TesteDriver.java:7)
Java Result: 1
CONSTRUÍDO COM SUCESSO (tempo total: 4 segundos)
```

Exemplos de exceções já definidas no pacote java.lang:

- **ArithmeticException:** indica situações de erros em processamento aritmético, tal como uma divisão inteira por 0. A divisão de um valor real por 0 não gera uma exceção (o resultado é o valor infinito).
- **NumberFormatException:** indica que tentou-se a conversão de uma string para um formato numérico,mas seu conteúdo não representava adequadamente um número para aquele formato. É uma subclasse de IllegalArgumentException;
- **IndexOutOfBounds:** indica a tentativa de acesso a um elemento de um agregado aquém ou além dos limites válidos. É a superclasse de ArrayIndexOutOfBoundsException, para arranjos, e de StringIndexOutOfBounds, para strings.
- **NullPointerException:** indica que a aplicação tentou usar uma referência a um objeto que não foi ainda definida.
- **ClassNotFoundException:** indica que a máquina virtual Java tentou carregar uma classe mas não foi possível encontrá-la durante a execução da aplicação.

11.0 Criando Diagrama UML

Introdução

Neste capítulo iremos aprender como, instalar e criar uma diagramação de classes usando o Netbeans para a linguagem Java. UML significa: "Unified Modeling Language", ou seja "Linguagem de Modelagem unificada". Modelar nosso código com objetivo de documentar e visualizar melhor o projeto.

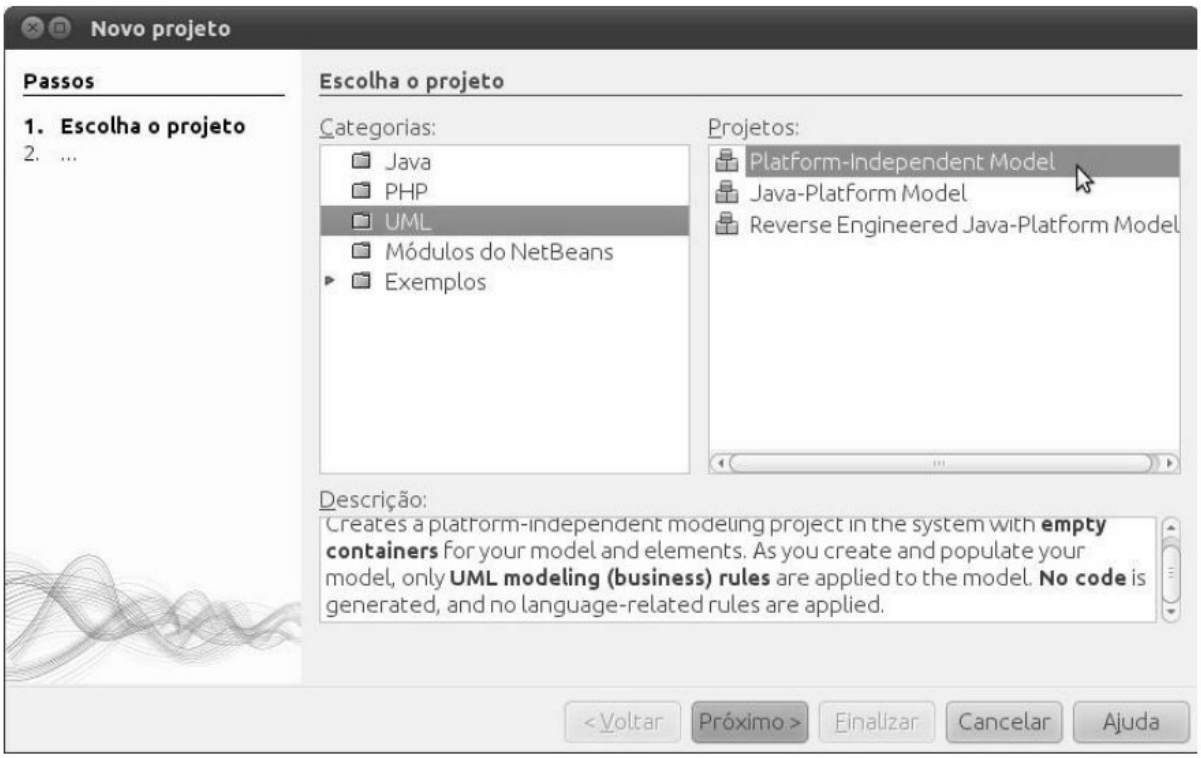
Para realizar a instalação é simples, basta ir em Ferramentas > Plugins e buscar por UML. Este plugin está disponível livremente a partir da versão 6.7.

Antes de começarmos a trabalhar com eles, precisamos lembrar quais os tipos de diagramas que podemos desenvolver e qual a utilidade de cada uma dentro de um projeto de qualquer escala, visto no começo da apostila.

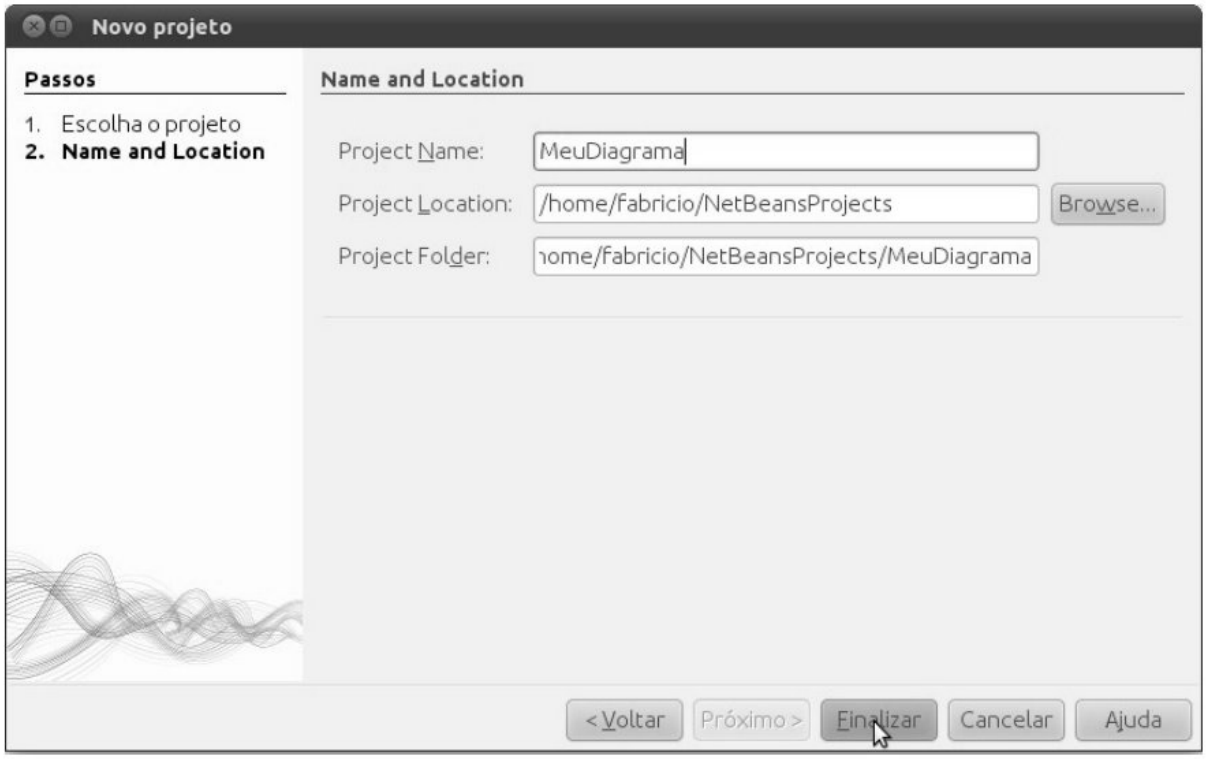
Logo iremos aprender como exemplo a trabalhar com Diagrama de Classe, pois esse diagrama é bem próximo da criação e estruturação de classes e herança entre as classes criadas.

Abrindo o Projeto na IDE Netbeans.

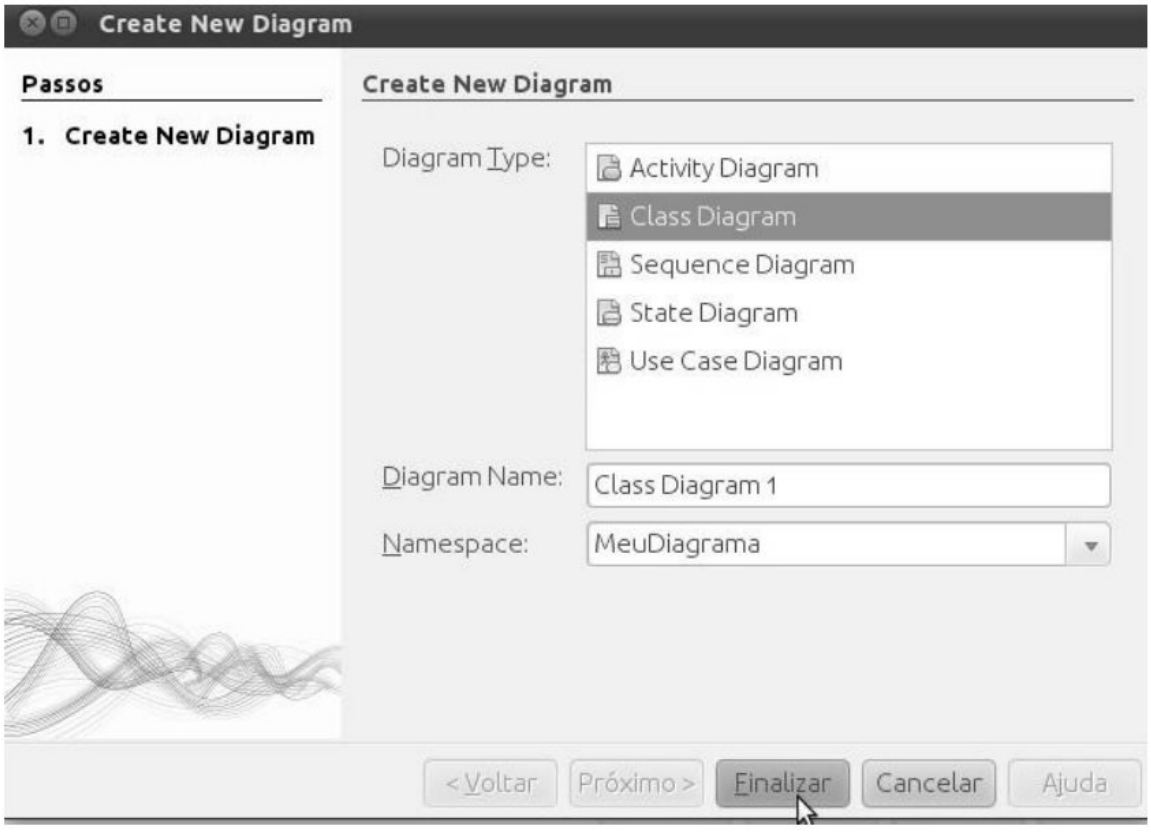
Criando um Novo Projeto Diagrama.



Nomeando Projeto de Diagrama.

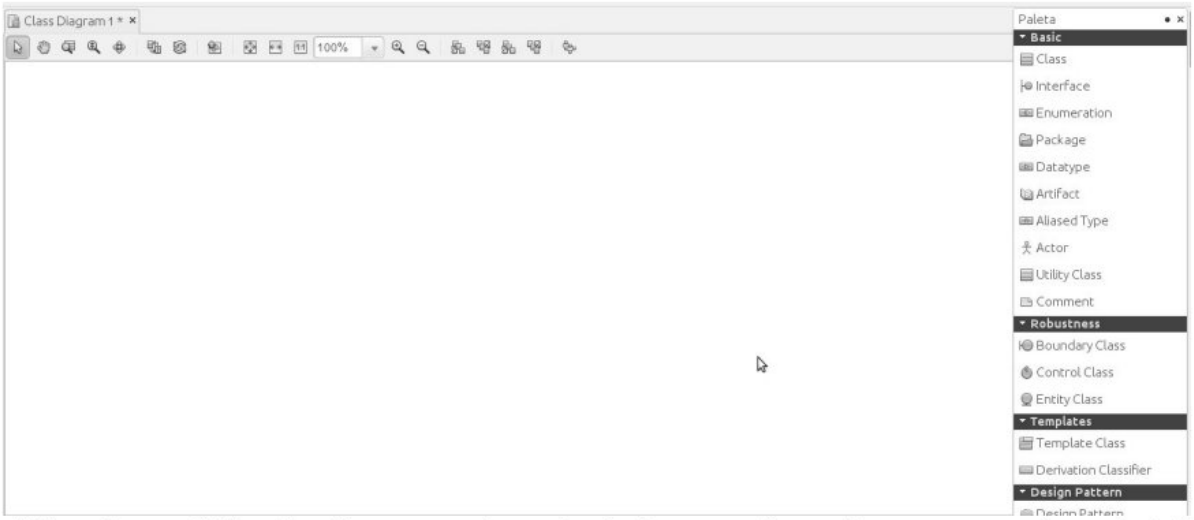


Escolhendo Diagrama de Classe.



Entrando no Diagrama criado, podemos visualizar no Lado direto da IDE Netbeans, um botão chamado "Paleta" ele será responsável abrir uma aba onde podemos montar os pacotes de classe.

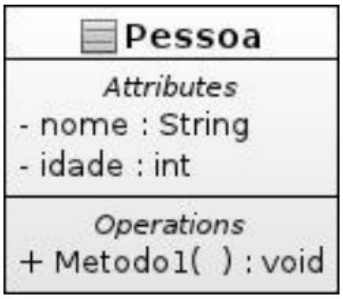
Veja a interface da aplicação:



Na direita Clicando em "Class", clique no palco principal que será gerado uma nova classe indefinida. Para alterar a Classe, basta clicar em Create Atributes para criar os atributos , e Create Operation para criar os métodos, lembre-se que para o diagrama, primeiro vem o nome da variável ou método, em seguida vem o seu tipo:

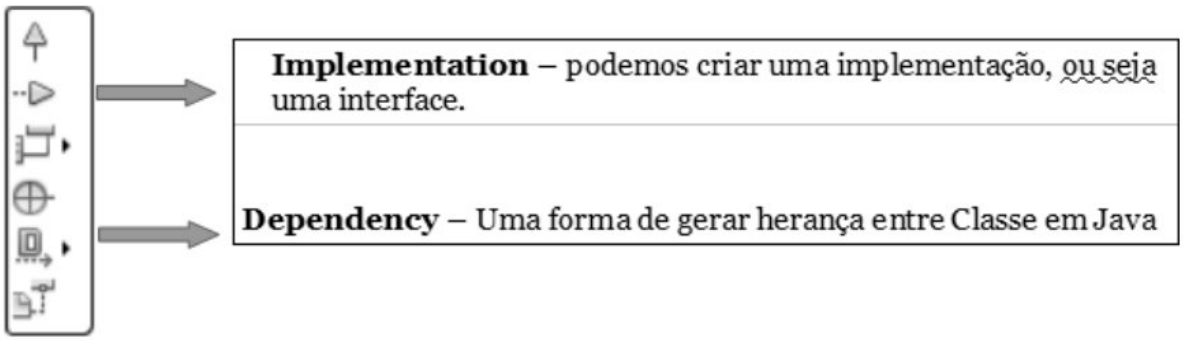
Exemplo: nome : String

Veja como criar uma classe simples com o tipo, pessoa:



Dessa forma podemos criar qualquer classe com os seus atributos e métodos de forma a poder visualizar melhor a estrutura do projeto.

Para criar interface ou Herança, basta clicarmos em uma classe ao qual você quer montar uma relação e aparecerá o seguinte menu:



12.0 Fila e Pilha

Iremos estudar Alguns estilos de agrupamento e organização de dados, onde podemos listar, retirar ou colocar elementos desse grupo de dados de forma organizada, levando em consideração a sua posição no grupo, para isso podemos estudar conceitos de Fila e Pilha para essas aplicações, veremos nos próximos tópicos como trabalhar com cada uma delas.

12.1 Fila

Imagine que você tem uma Fila de atendimento em um Banco. O que podemos observar em uma Fila?

- A fila poderá está vazia, onde o caixa não irá ter que trabalhar.
- A fila poderá está cheia, onde não poderá entrar ninguém mais na fila até que pelo menos saia uma pessoa dela.
- Quando alguém novo chega, ele vai para o final da fila.
- A pessoa que é atendida é sempre o primeiro em uma fila e logo depois de ser atendido, ele sai da fila, dando lugar a outras pessoas alterando a ordem de chamada de todo mundo. Quem era o terceiro, passa a ser o segundo, quem era o segundo passa a ser primeiro na fila.

Tudo o que acontece em uma fila normal também pode ser implementado na Programação Java.

IMPLEMENTANDO FILA EM JAVA

Para que possamos implementar uma fila em Java temos que usar os métodos da interface "Queue" da classe "LinkedList" Dessa forma:

```
Queue fila = new LinkedList();
```

O Objeto "fila" será responsável por conter método ao qual poderemos incluir, retirar, e mostrar o próximo elemento da fila, veja na tabela abaixo os seus métodos e suas funções:

Método	Função
element(Object);	Mostra o primeiro elemento da fila sem removê-lo
offer();	Insere um novo elemento da fila
poll();	Retorna e retira o próximo elemento da fila. Retorna "null" se a fila estiver vazia

Veja a implementação abaixo:

```
4  import java.util.LinkedList;
5  import java.util.Queue;
6
7
8  public class Main {
9
10     public static void main(String[] args) {
11         Queue fila = new LinkedList();
12
13         fila.offer("5");
14         fila.offer("6");
15         fila.offer("1");
16         fila.offer("7");
17         fila.offer("3");
18
19         System.out.println(fila.poll());
20         System.out.println(fila.poll());
21         System.out.println(fila.poll());
22
23         fila.offer("10");
24         fila.offer("1");
25
26         System.out.println(fila.poll());
27
28     }
```

Os elementos nesse exemplo estão inseridos na seguinte ordem:

5	6	1	7	3

Retirando os 3 primeiros elementos da Fila com o método "Poll", temos:

7	3

Inserindo o Elemento 10 e 1 na Fila usando o método "offer", temos:

7	3	10	1

Sendo assim, se usarmos o método "Poll" mais 5 vezes, estes elementos seriam retornados e retirados da fila nessa ordem mantendo a fila vazia. Quando a Fila estiver vazia, o comando "Poll" passará a retornar "null".

Vários sistemas usam esse recurso, como sistema de atendimento ao cliente, um sistema de fila de impressora onde os primeiros documentos a serem impressos terão prioridade em relação aos outros documentos que serão impresso em seguida. Pesquise por mais implementações usando o recurso de Fila.

12.2 Pilha

Imagine uma Pilha de Pratos a ser lavados. O que podemos observar?

- A pilha poderá está vazia, onde não precisamos lavar nenhum prato.
- A pilha poderá está cheia, onde não poderá entrar nenhum prato a mais, até ser lavado algum.
- O prato que chegar na pilha estará logo acima dos demais.
- O prato a ser lavado é justamente o de cima, ou seja, o ultimo prato da pilha, então os últimos pratos a entrarem na pilha, serão os primeiro a ser lavados, logo que eles forem lavados eles sairão da pilha, dando espaço para os pratos de baixo a ser lavados.

Claramente podemos perceber o velho ditado "Os últimos serão os primeiros" e esta regra também poderá ser implementado em JAVA.

IMPLEMENTANDO PILHA EM JAVA

Para que possamos implementar uma pilha em Java temos que usar a Classe Stack, dessa forma:

```
Stack pilha = new Stack();
```

O Objeto "pilha" será responsável por conter método ao qual poderemos incluir, retirar, e mostrar o

ultimo elemento da pilha, veja na tabela abaixo os seus métodos e suas funções:

Método	Função
empty()	Verifica se a pilha está vazia retornando um Boolean
push();	Insere um novo elemento da pilha
pop();	Retorna e retira o ultimo elemento da pilha.
peek();	Retorna o ultimo elemento da pilha sem retirá-lo.

Veja na prática:

```
1  package treina;
2  import java.util.Stack;
3
4
5  public class Main {
6
7      public static void main(String[] args) {
8          Stack pilha = new Stack();
9
10         pilha.push("5");
11         pilha.push("6");
12         pilha.push("1");
13         pilha.push("7");
14         pilha.push("3");
15
16         System.out.println(pilha.pop());
17         System.out.println(pilha.pop());
18         System.out.println(pilha.pop());
19
20         pilha.push("10");
21         pilha.push("1");
22
23         System.out.println(pilha.pop());
24
```

Os elementos nesse exemplo estão inseridos na seguinte ordem:

5	6	1	7	3

Retirando os 3 primeiros elementos da pilha com o método “pop”, temos:

5	6

Inserindo o Elemento 10 e 1 na Pilha usando o método “push”, temos:

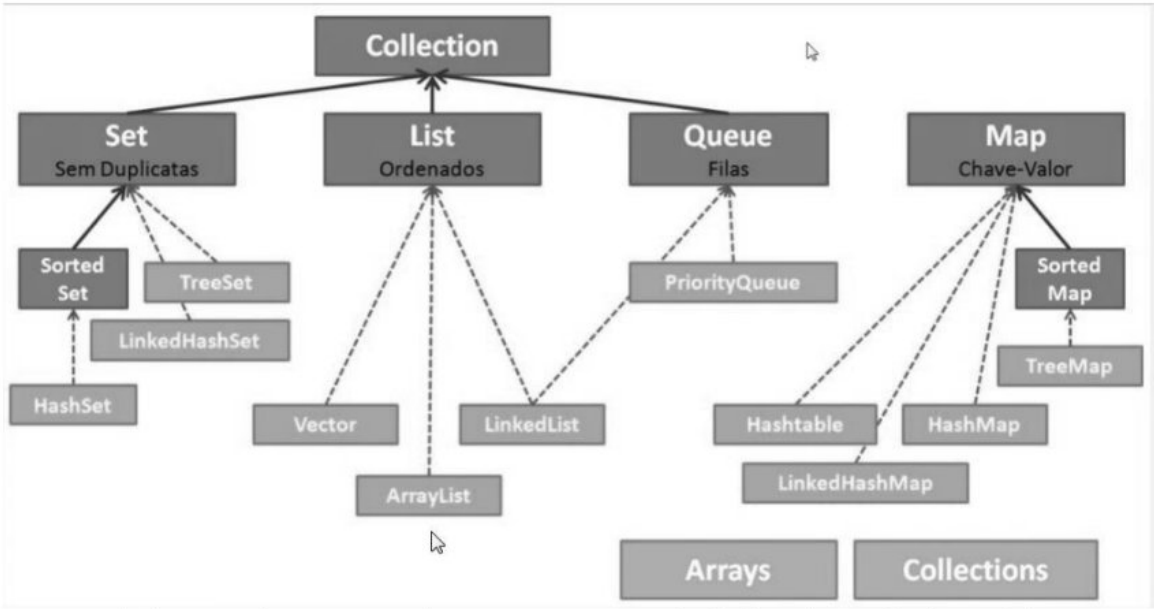
5	6	10	1

Sendo assim, se usarmos o método “pop” mais quatro vezes, esses elementos seriam retornados e retirados da pilha na ordem inversa mantendo a pilha vazia. O Comando “pop” não retorna null quando a pilha estiver vazia, então recomenda -se sempre verificar se contém elementos na pilha usando o método “empty”.

Vários sistemas usam esse recurso, um deles é bem usado por nós sem percebermos, imagine que está digitando um texto e precisa “desfazer” uma operação. Cada elemento de “Input” é posto em uma pilha e quando usamos o “CTRL+Z” é como usar um “pop” na pilha de retirando apenas o ultimo elemento digitado. Pesquise por mais implementações usando o recurso de Pilha.

13.0 Coleções: List e Map

Introdução a Coleções/Collections



Aprendemos em capítulos anteriores que existe uma estrutura de dados chamada Array. Essa estrutura de dados ela foi criada para guardar na memória do computador dados primitivos ou objetos do mesmo tipo, de maneira organizada (bloco de dados contínuo), em que cada informação pode ser acessada por índices (index),

A estrutura de dados Array é muito inflexível, é preciso definir seu tamanho assim que nós a criamos. E isso pode gerar um problema para nós programadores Java.

Vamos analisar a seguinte situação:

Criaremos um Array para guardar os nomes de uma lista de 10 alunos do Curso

de Java:

```
10          //cria um Array de String de tamanho 10.
11          String[] alunosNomes = new String[10];
```

Mas se no meio do meu programa eu precisar acrescentar a minha lista mais 20 alunos. Como o tamanho do Array alunosNomes foi definido com 10 posições então não seria possível adicionar os 20 novos alunos a essa estrutura, teríamos que tratar isso. Uma das alternativas para fazer isso é acrescentar o seguinte código:

```
13          //cria um Array de String de tamanho 20.
14           String[] alunosNomes2 = new String[20];
15
16           //copia os dados do do Array alunosNomes para o Array alunosNomes2.
           for(int i = 0 ; i < 10 ; i++){
18              alunosNomes2[i] = alunosNomes[i];
19          }
```

Como facilitar isso?

Como usar o conceito de Array, porém com um pouco mais de flexibilidade?

Coleções podem ser a saída para este problema. Uma das formas de pensar nas coleções é como Arrays turbinadas, incrementadas com várias funcionalidades e regras.

A definição formal de Coleções é: estruturas de dados utilizadas para a rmazenar e manipular informações, ou seja, são objetos que representam um grupo de objetos.

Esse capítulo tem como objetivo abordar as principais funcionalidades e regras de duas das principais coleções do Java: List e Map.

13.1 Collection List ou Coleção Lista

Uma lista é uma coleção de elementos arrumados numa ordem linear, isto é, onde cada elemento tem um antecessor (exceto o primeiro) e um sucessor (exceto o último). Normalmente implementada como "Array" ou "Lista Encadeada". A Lista pode ser mantida ordenada ou não e as operações mais importantes de uma coleção do tipo Lista são:

Adição de elementos

- Adicionar um objeto em qualquer lugar da lista, fornecendo o índice desejado;

Remoção de elementos

- Remover um objeto presente em qualquer lugar da lista, fornec endo o índice desejado;

Acesso aos elementos

- Obter o elemento de qualquer posição da lista, fornecendo o índice desejado;
- Iterar sobre os elementos;

Pesquisa de elementos

- Descobrir se certo elemento está na lista;
- Descobrir o índice de certo elemento na lista (onde está);

Indagar sobre atributos

- Obter o número de elementos da coleção;

Em Java existem três classes que trabalham com a Coleção do tipo Lista ou List elas se encontram no pacote/package java.util . São elas:

- Vector
- ArrayList
- LinkedList

Escolhemos explanar nesse capítulo a classe Java mais utilizada para implementar listas que é a classe ArrayList.

ArrayList

Methods

Modifier and Type	Method and Description
boolean	**add**(E e) Appends the specified element to the end of this list.
void	**add**(int index, E element) Inserts the specified element at the specified position in this list.
boolean	**addAll**(Collection<? extends E> c) Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's Iterator.
boolean	**addAll**(int index, Collection<? extends E> c) Inserts all of the elements in the specified collection into this list, starting at the specified position.
void	**clear**() Removes all of the elements from this list.
Object	**clone**() Returns a shallow copy of this ArrayList instance.
boolean	**contains**(Object o) Returns true if this list contains the specified element.
void	**ensureCapacity**(int minCapacity) Increases the capacity of this ArrayList instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument.
E	**get**(int index) Returns the element at the specified position in this list.
int	**indexOf**(Object o) Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
boolean	**isEmpty**() Returns true if this list contains no elements.
Iterator<E>	**iterator**() Returns an iterator over the elements in this list in proper sequence.
int	**lastIndexOf**(Object o) Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.
ListIterator<E>	**listIterator**() Returns a list iterator over the elements in this list (in proper sequence).
ListIterator<E>	**listIterator**(int index) Returns a list iterator over the elements in this list (in proper sequence), starting at the specified position in the list.
E	**remove**(int index) Removes the element at the specified position in this list.
boolean	**remove**(Object o) Removes the first occurrence of the specified element from this list, if it is present.
boolean	**removeAll**(Collection<?> c) Removes from this list all of its elements that are contained in the specified collection.
protected void	**removeRange**(int fromIndex, int toIndex) Removes from this list all of the elements whose index is between fromIndex, inclusive, and toIndex, exclusive.
boolean	**retainAll**(Collection<?> c) Retains only the elements in this list that are contained in the specified collection.
E	**set**(int index, E element) Replaces the element at the specified position in this list with the specified element.
int	**size**() Returns the number of elements in this list.
List<E>	**subList**(int fromIndex, int toIndex) Returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive.
Object[]	**toArray**() Returns an array containing all of the elements in this list in proper sequence (from first to last element).
<T> T[]	**toArray**(T[] a) Returns an array containing all of the elements in this list in proper sequence (from first to last element); the runtime type of the returned array is that of the specified array.
void	**trimToSize**() Trims the capacity of this ArrayList instance to be the list's current size.

Acessando a documentação do Java 7 podemos visualizar todas as especificações dessa classe, usada para programar coleções do tipo lista. Abaixo tabela com todos os métodos da classe ArrayList:
Desses métodos listados na documentação do Java 7 iremos estuda r os métodos principais, os quais fazem as ações de adição de elementos, remoção de elementos, acesso aos elementos, pesquisa de elementos e indagar sobre atributos de uma determinada lista.

CRIANDO UMA LISTA EM JAVA USANDO A CLASSE ARRAYLIST

Para criar uma lista usando a Classe ArrayList do Java, escreva em seu programa a seguinte linha de código:

```
15      ArrayList<tipo> nome_da_lista  =  new ArrayList<> ();
```

tipo: tipo do objeto que você quer guarda na lista. Ex: Integer, String, Carro, Pessoa, etc,
Exemplo:

```
14
15      //cria uma lista de dados do tipo String de nome alunosNomes
16      ArrayList<String> alunosNomes  =  new ArrayList<String> ();
```

Na versão 7 do Java você também pode criar uma lista de ssa forma:

```
17      //cria uma lista de dados do tipo String de nome alunosNomes
18      ArrayList<String> alunosNomes  =  new ArrayList<> ();
```

Todas os códigos que trabalharem com listas usando a classe ArrayList precisam importar a classe ArrayList. Isso pode ser feito adicionando a seguinte instrução no cabeçalho da classe:
import java.util.ArrayList;

ADICIONANDO ELEMENTOS EM UMA LISTA

Agora que você já criou seu objeto Lista você pode adicionar elementos dentro dela. As Listas quando criadas elas não possuem nenhum elemento, ou seja, elas são Listas Vazias.
Na Lista, diferente do Array, você não especifica seu tamanho, ela cresce de forma dinâmica à medida que você acrescenta dados a ela.
Vamos aprender como adicionar elementos a uma Lista, para isso iremos usar o método add() que pertence a classe ArrayList.

```
13          nome_da_lista.add(dado);
```

Exemplo:

```
17        //cria uma lista de dados do tipo String de nome alunosNomes
18       ArrayList<String> alunosNomes  =  new ArrayList<> ();
19
20       //adicionando o nome de alunos
21       alunosNomes.add("Fabricio Rosal");
22       alunosNomes.add("Jonas Júnior");
23       alunosNomes.add("Linus Torvalds");
```

O método add(), usado da forma mostrada acima ele se mpre coloca o novo dado adicionado à lista na ultima posição da mesma, mas podemos utilizá-lo de outra maneira, de tal forma que podemos escolher a posição em que queremos adicionar o novo elemento.

Vamos ver como fica essa outra forma de usar método add( ).

```
15          nome_da_lista.add(posição, dado);
```

Exemplo:

```
17       //cria uma lista de dados do tipo int de nome anos
18           ArrayList<Integer> anos  =  new ArrayList<> ();
19
20       //adicionando anos a lista
21           anos.add(2000);
22           anos.add(2001);
23           anos.add(2003);
24       /**
25        * adicionando o ano de 2002 na posição 2 da lista, o novo dado será
26        * posicionado entre 2001 e 2003
27        */
28           anos.add(2,2002);
```

Para visualizar todos os lados da lista na forma de uma String use o método toString(), desta forma:

```
16
17          //esse método retorna todos os dados da lista em forma de String
18            nome_da_lista.toString();
```

Vamos aplicar esse método no exemplo em que criamos uma lista de anos, ficará desta maneira:

```
17      //cria uma lista de dados do tipo int de nome anos
18          ArrayList<Integer> anos  =  new ArrayList<> ();
19
20      //adicionando anos a lista
21          anos.add(2000);
22          anos.add(2001);
23          anos.add(2003);
24      /**
25       * adicionando o ano de 2002 na posição 2 da lista, o novo dado será
26       * posicionado entre 2001 e 2003
27       */
28          anos.add(2,2002);
29
30      //imprimindo os dados contidos na lista anos, utilizando o método toString()
31      System.out.print(anos.toString());
```

Saída do programa:

```
Saída - CursoJavaUdt (run)
run:
[2000, 2001, 2002, 2003]CONSTRUÍDO COM SUCESSO (tempo total: 0 segundos)
```

REMOVENDO ELEMENTOS DE UMA LISTA

Para remover um dado da lista utilizamos o método remove() da classe ArrayList. O método remove() funciona da seguinte forma:

```
20          nome_da_lista.remove(posição);
```

Vamos aplicar esse método no exemplo em que criamos uma lista de anos, ficará desta maneira:

Outra forma de usar o método remove() é passando para o método o objeto que você quer remover.

```
16
17      //cria uma lista de dados do tipo int de nome anos
18          ArrayList<Integer> anos  =  new ArrayList<> ();
19
20      //adicionando anos a lista
21          anos.add(2000);
22          anos.add(2001);
23          anos.add(2003);
24      /**
25       * adicionando o ano de 2002 na posição 2 da lista, o novo dado será
26       * posicionado entre 2001 e 2003
27       */
28          anos.add(2,2002);
29
30        /**
31         * remover dado da posição 1 da lista anos, o dado 2001 será removido
32         * e todos os dados após ele avançaram uma posição
33         */
34              anos.remove(1);
35
36      //imprimindo os dados contidos na lista anos, utilizando o método toString()
37              System.out.print(anos.toString());
```

Ficando desta forma:

```
23          nome_da_lista.remove(objeto);
```

Exemplo:

```
17       //cria uma lista de dados do tipo String de nome alunosNomes
18      ArrayList<String> alunosNomes  =  new ArrayList<> ();
19
20      //adicionando o nome de alunos
21      alunosNomes.add("Fabricio Rosal");
22      alunosNomes.add("Jonas Júnior");
23      alunosNomes.add("Linus Torvalds");
24
25
26      //removendo o objeto "Jonas Júnior" da lista
27      alunosNomes.remove("Jonas Júnior");
28
29      //imprimindo os dados contidos na lista anos, utilizando o método toString()
30      System.out.print(alunosNomes.toString());
```

Saída do programa sem o método remove():

```
Saída - CursoJavaUdt (run)                                    Tarefas
run:
[Fabricio Rosal, Jonas Júnior, Linus Torvalds]CONSTRUÍDO COM SUCESSO (tempo total: 0 segundos)
```

Saída do programa aplicando o método remove():

```
Saída - CursoJavaUdt (run)                                    Tarefas
run:
[Fabricio Rosal, Linus Torvalds]CONSTRUÍDO COM SUCESSO (tempo total: 0 segundos)
```

ACESSANDO ELEMENTOS DA LISTA

Para acessar um valor de uma determinada lista usamos o método get() da classe ArrayList dessa forma:

Exemplo:

```
32      //acessa o valor no índice 1 da lista de nome alunosNomes
33      alunosNomes.get(1);
```

Temos métodos na classe ArrayList para auxiliar na interação da lista, ou seja, percorrer a lista acessando os seus valores em todas as direções possíveis.

Esses métodos precisam, além da lista que iram auxiliar a percorre, um objeto do tipo ListIterator onde esse objeto será como um leitor de posições da lista que pode percorrer a lista em todos os sentidos possíveis.

Vamos entender melhor com o seguinte exemplo:

Vamos criar uma lista em Java, usando a classe ArrayList, para guardar o nome das linhas de ônibus de Fortaleza.

```
16
17          /*cria uma lista de dados do tipo String para armazenar nomes de
18           * linhas de ônibus de Fortaleza
19           */
20          ArrayList<String> nomeLinhaOnibus  =  new ArrayList<> ();
21          //adicionando o nome de linhas de ônibus de Fortaleza
22          nomeLinhaOnibus.add(" Parajana");
23          nomeLinhaOnibus.add(" Guajeru 2");
24          nomeLinhaOnibus.add(" Circular 1");
25          nomeLinhaOnibus.add(" Parangaba Nautico");
```

Agora vamos criar um objeto do tipo ListIterator, onde esse objeto será como um leitor de posições da lista que pode ser deslocado para frente e para trás ao longo de uma lista. Para criar um objeto do tipo ListIterator precisamos importar a classe que cria esse objeto para o nosso programa, pois esse objeto não pode ser instanciado pela classe ArrayList. A importação da classe é feita adicionando a seguinte instrução no cabeçalho de sua classe:

import java.util.ListIterator;

Após importar o método podemos criar nosso objeto do tipo ListIterator da seguinte maneira:

```
17
18          /*cria uma lista de dados do tipo String para armazenar nomes de
19           * linhas de ônibus de Fortaleza
             */
18          /*cria uma lista de dados do tipo String para armazenar nomes de
19           * linhas de ônibus de Fortaleza
20           */
21          ArrayList<String> nomeLinhaOnibus  =  new ArrayList<> ();
22          //adicionando o nome de linhas de ônibus de Fortaleza
23          nomeLinhaOnibus.add(" Parajana");
24          nomeLinhaOnibus.add(" Guajeru 2");
25          nomeLinhaOnibus.add(" Circular 1");
26          nomeLinhaOnibus.add(" Parangaba Nautico");
27
28          //criando o leitor de posições da lista
29          ListIterator<String> percorreLista  =  nomeLinhaOnibus.listIterator();
30
31
32  //percorre a lista imprimindo todos os seus elementos e seus respectivos índices
33      while(percorreLista.hasNext()){
34
35          //pega o índice do elemento sucessor ao elemento lido pelo percorreLista
36          int indice = percorreLista.nextIndex();
37          System.out.print(indice);
38
39          /**
40           * faz o percorreLista avançar para o próximo elemento da lista
41           * nomeLinhaOnibus
42           */
43          String nome = percorreLista.next();
44          System.out.println(nome);
45      }
```

Por fim, podemos usar os métodos da classe ArrayList que nos ajudam a percorrer uma lista, da seguinte forma:

Saída do programa:

```
Saída - CursoJavaUdt (run)
run:
0 Parajana
1 Guajeru 2
2 Circular 1
3 Parangaba Nautico
CONSTRUÍDO COM SUCESSO (tempo total: 0 segundos)
```

De forma semelhante ao método hasNext() e o método next() podemos ainda usar em nossos programas Java os seguintes métodos:

Métodos para auxiliar percorrendo uma Lista usando a classe ArrayList

- hasNext(): retorna true se o elemento lido pelo percorredorLista tem sucessor, caso contrário

retorna false;

- hasPrevious(): retorna true se o elemento lido pelo percorredorLista tem antecessor, caso contrário retorna false;
- next(): faz o percorreLista avançar para a posição seguinte da lista e retornar o novo valor lido;
- previous(): faz o percorreLista retroceder para a posição anterior da lista e retornar o novo valor lido;
- nextIndex(): retorna o índice do elemento sucessor ao elemento l ido pelo percorreLista;
- previousIndex(): retorna o índice do elemento antecessor ao elemento lido pelo percorreLista;

PESQUISANDO ELEMENTOS DA LISTA

Pesquisar um elemento na lista é bastante simples. Vamos usar o método indexOf() que se encontra na classe ArrayList para pesquisa em uma lista. Esse método retorna o valor do índice do objeto procurado se ele existir na lista, caso contrário retorna -1.

```
27            nome_da_lista .indexOf(dado)
```

Exemplo:

```
16
17      //cria uma lista de dados do tipo String para armazenar nomes de frutas
18      ArrayList<String> frutas  =  new ArrayList<> ();
19      //adicionando o nome frutas a lista
20      frutas.add("banana");
21      frutas.add("acerola");
22      frutas.add("goiaba");
23      frutas.add("abacaxi");
24      /**
25       * verificar se existe a fruta "goiaba" e "laranja" se existir imprime o
26       * índice do objeto procurado, caso contrário imprime -1
27       */
28      System.out.println(frutas.indexOf("goiaba"));
29      System.out.println(frutas.indexOf("laranaja"));
```

Saída do programa:

```
Saída - CursoJavaUdt (run)
run:
2
-1
CONSTRUÍDO COM SUCESSO (tempo total: 0 segundos)
```

13.2 Collection Map ou Coleção Mapa

Um mapa armazena pares (chave, valor) chama dos itens, essas chaves e valores podem ser de qualquer tipo, portanto, a chave é utilizada para achar um elemento rapidamente. Estruturas especiais são usadas para que a pesquisa seja rápida, como "Tabela Hash" ou "Árvore". Os dados armazenados nos mapas podem ser mantidos ordenados ou não (com respeito às chaves).

A coleção do tipo Mapa difere das Listas por diversas características, as principais são:

- as estruturas de dados que as implementam;
- na coleção do tipo Mapa podemos escolher o tipo e o valor q ue terão as chaves(índices), através delas teremos acesso aos dados. As chaves (índices) e dados dos Mapas podem ser de qualquer tipo, inclusive um objeto. Já nas Listas somente os valores armazenados podem ser de qualquer tipo, seus índices são sempre números inteiros indexados a partir do zero de forma automática, à medida que vamos adicionando elementos à coleção.

As operações mais importantes de uma coleção do tipo Mapa são:

Adição de elementos

- Adicionar um item no mapa (fornecendo chave e valor)

Remoção de elementos

- Remover um item com chave dada

Acesso aos elementos

- Iterar sobre os itens

Pesquisa de elementos

- Descobrir se um elemento com chave dada está na coleção

Indagar sobre atributos

- Obter o número de elementos
- Observe que o acesso à coleção sempre é feita conhecendo a chave

Vamos utilizar a classe HashMap para trabalhar com coleções do tipo mapa ou map.

13.3 HashMap

A classe HashMap é uma implementação da interface Map do pacote java.util, e possibilita trabalhar com mapeamento de objetos no esque ma chave/valor, ou seja, informada a chave, resgato o valor.

Acessand a documentação do Java 7 podemos visualizar todas as especificações dessa classe, usada para programar coleções do tipo mapa/map. Abaixo tabela com todos os métodos da classe HashMap:

Methods

Modifier and Type	Method and Description
void	**clear**() Removes all of the mappings from this map.
Object	**clone**() Returns a shallow copy of this HashMap instance: the keys and values themselves are not cloned.
boolean	**containsKey**(Object key) Returns true if this map contains a mapping for the specified key.
boolean	**containsValue**(Object value) Returns true if this map maps one or more keys to the specified value.
Set<Map.Entry<K,V>>	**entrySet**() Returns a Set view of the mappings contained in this map.
V	**get**(Object key) Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
boolean	**isEmpty**() Returns true if this map contains no key-value mappings.
Set<K>	**keySet**() Returns a Set view of the keys contained in this map.
V	**put**(K key, V value) Associates the specified value with the specified key in this map.
void	**putAll**(Map<? extends K,? extends V> m) Copies all of the mappings from the specified map to this map.
V	**remove**(Object key) Removes the mapping for the specified key from this map if present.
int	**size**() Returns the number of key-value mappings in this map.
Collection<V>	**values**() Returns a **Collection** view of the values contained in this map.

Vamos explanar os principais métodos da classe HashMap, métodos que nos permitirá criar, adicionar, remover, selecionar e pesquisar itens em uma coleção do tipo Map.

CRIANDO UMA COLEÇÃO DO TIPO MAPA USANDO A CLASSE HASHMAP

Para criar um mapa usando a Classe HashMap do Java, escreva em seu programa a seguinte linha de código:

```
29        HashMap< tipo_da_chave , tipo_do_valor> nome_do_mapa = new HashMap<>();
```

tipo_da_chave: coloque o tipo que terá a chave/índice do seu Map, essa chave será usada para acessar os valores guardados no Map;

tipo_do_valor: coloque o tipo que terá os valores que serão guardados em seu Map;

Pronto, dessa forma já instanciamos uma coleção do tipo Map usando a classe HashMap.

Agora temos que inserir elementos a coleção do tipo Map , pois só criamos o Map, mas não há elementos dentro dele.

Lembre se de importar a Classe HashMap para seu programa, já que estamos usando-a para implementar e trabalhar com Coleções do tipo Mapa. Podemos importar a classe colocando a seguinte instrução no

cabeçalho do programa o qual estamos escrevendo:

import java.util.HashMap;

ADICIONANDO ELEMENTOS EM UMA COLEÇÃO DO TIPO MAPA

Após criar o seu Map, podemos adicionar um elemento usando o método put() da Classe HashMap da seguinte forma:

```
31          nome_do_mapa.put(objeto_chave,objet_valor);
```

objeto_chave: valor do objeto que será a chave de acesso para os valores do mapa;
objeto_valor: valor do objeto que será guardado no mapa;

Exemplo:
Vamos criar um Map para guardar estados e suas respectivas capitais. Os
estados serão as chaves e as capitais serão os valores guardados no Mapa, ambos serão do tipo String.

```
17          HashMap<String,String> estadoCapital = new HashMap<>();
```

Agora vamos adicionar os valores ao nosso mapa usando o método put, da seguinte forma:

```
19          estadoCapital.put("Ceara", "Fortaleza");
20          estadoCapital.put("Rio Grande do Norte", "Natal");
21          estadoCapital.put("Bahia", "Salvador");
```

Vamos usar o método toString() para visualizar como estão os dados dentro do nosso mapa, da seguinte forma:

```
23          System.out.println(estadoCapital.toString());
```

Saída do Programa:

```
Saída - CursoJavaUdt (run)
run:
{Ceara=Fortaleza, Bahia=Salvador, Rio Grande do Norte=Natal}
CONSTRUÍDO COM SUCESSO (tempo total: 1 segundo)
```

REMOVENDO ELEMENTOS EM UMA COLEÇÃO DO TIPO MAPA

Para remover elementos de uma coleção do tipo Mapa usando os métodos da Classe HashMap usamos o método remove().

```
33          nome_do_mapa.remove(objeto_chave);
```

objeto_chave: chave de acesso do elemento a ser removido do mapa;

O método remove() retorna uma String com o valor removido do mapa ou retorna o valor null se não for achado nenhum valor associado a chave passada no argumento do método.

Exemplo:

```
17          HashMap<String,String> estadoCapital = new HashMap<>();
18
19          estadoCapital.put("Ceara", "Fortaleza");
20          estadoCapital.put("Rio Grande do Norte", "Natal");
21          estadoCapital.put("Bahia", "Salvador");
22
23           //imprime o mapa antes da remoção
24          System.out.println(estadoCapital.toString());
25
26          /**
27           * procura o valor a ser removido passando a chave como argumento
28           * retorna o valor que está sendo removido do Mapa
29           */
30          String retorno = estadoCapital.remove("Ceara");
31
32          //imprime valor removido do mapa
33          System.out.println(retorno);
34
35          //imprime como ficou o mapa apos a remocao
36          System.out.println(estadoCapital.toString());
```

Saída do programa:

```
Saída - CursoJavaUdt (run)
run:
{Ceara=Fortaleza, Bahia=Salvador, Rio Grande do Norte=Natal}
Fortaleza
{Bahia=Salvador, Rio Grande do Norte=Natal}
CONSTRUÍDO COM SUCESSO (tempo total: 0 segundos)
```

SELECIONANDO/PESQUISANDO ELEMENTOS EM UMA COLEÇÃO DO TIPO MAPA

O método de seleção get() da classe HashMap é o mesmo método usado para se pesquisar um valor dentro de um mapa. Esse método funciona da seguinte forma:

```
35          nome_do_mapa.get(objeto_chave);
```

objeto_chave: chave de acesso do elemento a ser selecionado/pesquisado do mapa;

Exemplo:

```
18
19          HashMap<String,String> estadoCapital = new HashMap<>();
20
21          estadoCapital.put("Ceara", "Fortaleza");
22          estadoCapital.put("Rio Grande do Norte", "Natal");
23          estadoCapital.put("Bahia", "Salvador");
24
25          /**pesquisa/seleciona valor em um mapa e retorna o valor caso ache,
26           * senão retorna null
27           */
28          String retorno = estadoCapital.get("Baa");
29          System.out.println(retorno);
```

Saída do programa:

```
Saída - CursoJavaUdt (run)
run:
null
CONSTRUÍDO COM SUCESSO (tempo total: 0 segundos)
```

NÚMERO DE ELEMENTOS EM UMA COLEÇÃO DO TIPO MAPA

Para sabermos o número de elementos contido em um mapa podemos utilizar o método size(), esse método pode ser usado de maneira semelhante com coleções do tipo Lista usando a classe ArrayList.

O método size() funciona da seguinte forma:

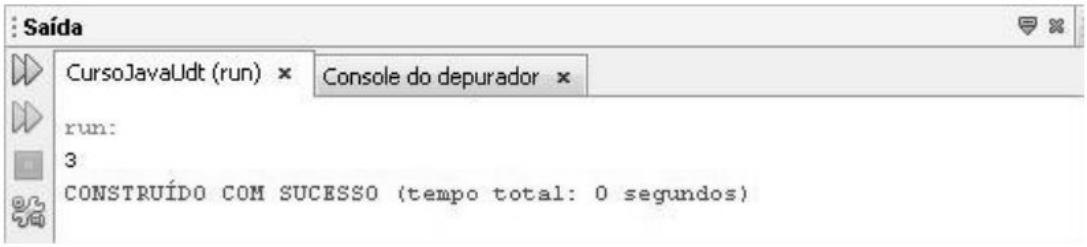
```
37              nome_do_mapa.size();
```

A chamada ao método size() nos retorna um valor inteiro que é a quantidade de elementos de um mapa.

Exemplo:

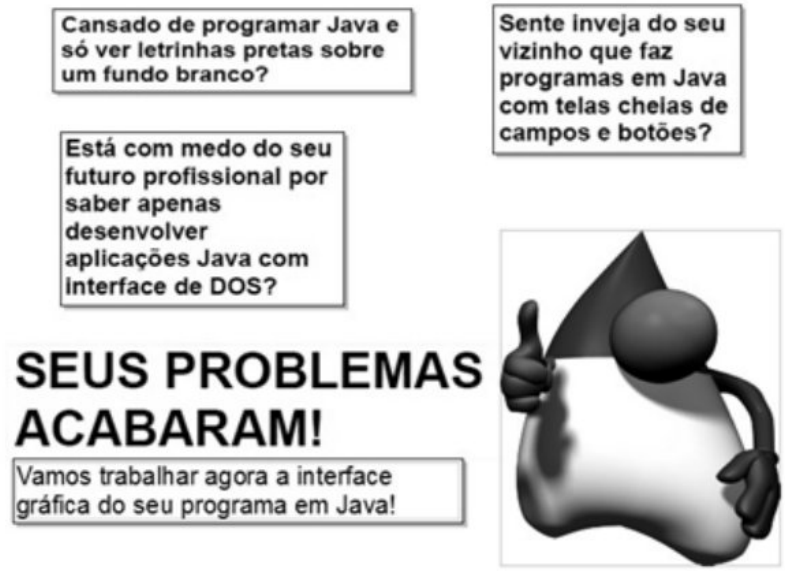
```
17
18          HashMap<String,String> estadoCapital = new HashMap<>();
19
20          estadoCapital.put("Ceara", "Fortaleza");
21          estadoCapital.put("Rio Grande do Norte", "Natal");
22          estadoCapital.put("Bahia", "Salvador");
23
24          //quantidade elementos do mapa estadoCapital
25          int quantidadeItensMapa = estadoCapital.size();
26          System.out.println(quantidadeItensMapa);
```

Saída do programa:



14.0 Interface Gráfica com usuário - GUI

A Interface Gráfica com o Usuário, também conhecido como GUI - Graphical User Interface, em Java, é feita através de bibliotecas de classes, sendo que a primeira a surgir foi a AWT(Abstract Window Toolkit). A AWT surgiu já na vers ão 1.0, mas se tornou confiável a partir da versão 1.1. A maneira como as classes dessa biblioteca trabalham garante a criação dos elementos da interface de usuário seguindo o comportamento destinado às ferramentas GUI nativas de cada plataforma (Windows, Mac, Solaris, ...).
Alguns exemplos destes elementos são: botões, listas, menus, componentes de textos, contêineres (janelas e barras de menus), caixas de diálogo para abrir ou salvar arquivos, além de elementos para manipulação de imagens, fontes e cores.



A portabilidade de plataforma funcionava bem em aplicações simples, mas aplicações que envolviam elementos mais complexos, como menus e barras de rolagem, por exemplo, apresentavam diferenças de comportamento conforme a plataforma. O que aconteceu foi que as aplicações visuais feitas em Java não se pareciam, e nem tinham as mesmas funcionalidades, com as aplicações convencionais de cada plataforma.
A partir da versão 2 do Java, a JFC (Java Foundation Classes) apresentou novos recursos para a construção da GUI das aplicações, o que melhorou muito os problemas de portabilidade. São eles:

- Java 2D: novas funções para desenhos e gráficos.
- Drag & Drop: clicar, arrastar, copiar e colar.
- Swing: biblioteca de classes extensão da AWT, onde são apresentados novos componentes de interface e o que é conhecido por look and feel, que é uma adaptação perfeita da GUI ao sistema operacional específico de desenvolvimento.

É bom salientar que o Swing não substitui o AWT, mas é o kit de ferramentas GUI mais utilizado para desenvolvimento de aplicações visuais. O AWT continua existindo, mantendo a mesma arquitetura criada para o Java versão 1.1.

O Swing possui muito mais recursos, além de garantir maior portabilidade e, em boa parte dos casos, é mais fácil de usar. Isso não significa que ou se utiliza AWT ou se utiliza Swing, normalmente o que acontece é que elementos das duas bibliotecas são utilizados conjuntamente nas aplicações.

14.1 Bibliotecas AWT e SWING

São as fontes das classes que utilizaremos a partir daqui. Nelas moram os componentes gráficos para Java, e por isso devemos importá-las no início de nossas classes gerenciadoras de telas:

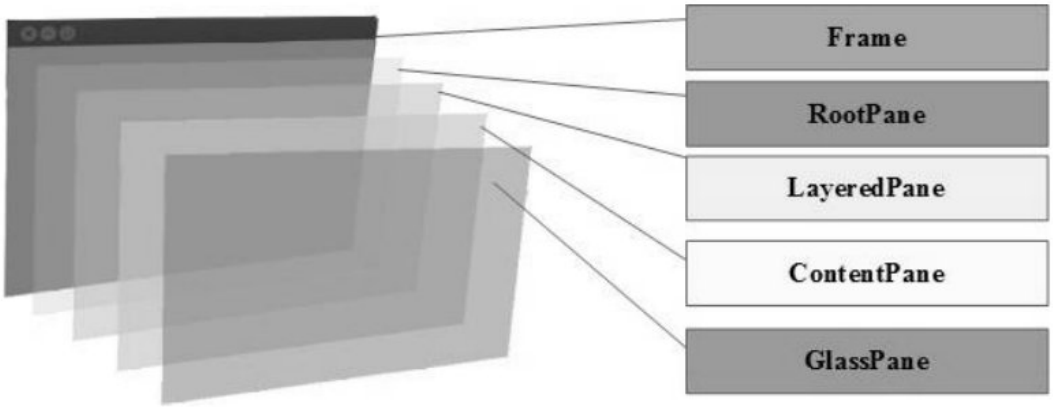
- import javax.swing.*;
- import java.awt.*;

14.2 JFrame

Pode parecer estranho para quem vem de outras linguagens, como o Delphi ou o VB, mas em Java, cada tipo de componente gráfico é uma classe com atributos e métodos próprios. E para utilizá-las, devemos instanciá-las.

CAMADAS JFRAME

Antes que possamos criar nossa primeira ja nela precisamos conhecer como funciona a hierarquia dos elementos que são responsáveis pela visualização gráfica.



Onde:

RootPane – Representa uma camada que irá gerenciar todas as camadas internas.

LayeredPane – Contém uma região para Barra de Menu e o ContentPane

ContentPane – Contém todos os Componentes como um campo de texto,campo de seleção de opções, etc.**GlassPane –** Camada que por padrão é invisível, porém pode ser usado para mostrar algum componente por cima das demais, como uma imagem.

Agora que sabemos como funciona um Jframe iremos criar nossa primeira janela em Java.

Nossa primeira Janela com JFrame

A primeira e mais simples chama-se JFrame, e cria telas retangulares sem nada dentro, um espaço onde trabalharemos logo depois.

```
6   public class classetela {
7       public static void main (String[]args) {
8           JFrame tela = new JFrame("Primeira tela");
9           tela.setBounds(100,100,400,300);
10          tela.getContentPane().setBackground(Color.RED);
11          tela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
12          tela.setLayout(null);
13          tela.setVisible(true);
14      }
15  }
```

O método SetBounds trabalha com a localização e tamanho da tela, sendo a seguinte ordem: distância do canto esquerdo da tela, distância do topo da tela, largura e altura.

tela.getContentPane().setBackground(Color.RED);

Aqui, invocamos o método que verifica qual é o campo d e Content do JFrame, ou seja, a região onde não está o menu nem as bordas, e muda sua cor a partir do método set Background. Eu usei uma biblioteca de cores, a Color, e defini a janela como vermelha; Poderia ter usado azul, cinza, verde, etc.

tela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

Aqui, eu determino qual a ação a executar quando este JFrame for fechado. No caso, ele fecha o programa todo, num efeito de BREAK. A instrução DISPOSE_ON_CLOSE apenas levará o JFrame a fechar sem derrubar o programa todo.

tela.setLayout(null);

Aqui, escolhi um Layout para o meu JFrame. Ou melhor, escolhi que seja nenhum. Existem Layouts prontos para JFrames, onde os botões ficam dispostos em forma de fileira, ou nos cantos, ou expandidos. A maioria tem um resultado grotesco. Se quisermos desenhar nossa tela com liberdade para fazê-la bonita, temos que escolher o NULL;

tela.setVisible(true);

Depois de preparado, o JFrame pode já ficar visível. Este passo vem no final de toda a sua montagem.

Agora vamos ver como ficou nossa janela.

14.3 Criando uma classe com propriedades JFrame

No tópico anterior aprendemos como construir uma Janela usando as propriedades do JFRAME diretamente. Para a nossa janela criada, o método main tinha a responsabilidade direta de criar uma janela. Para essa situação, se quiséssemos criar uma nova janela tínhamos que chamar novamente a Classe JFrame e atribuir todas as características novamente linha a linha. Podemos construir uma janela de uma forma melhor criando um próprio modelo com métodos, encapsulamento, herança e os demais recursos aprendidos sobre orientação a objetos. Para criar o nosso modelo de JFrame precisamos primeiro criar uma classe e herdá-la de JFRAME dessa forma:

```
import javax.swing.JFrame;

public class MinhaJanela extends JFrame{

}
```

Assim nossa classe chamada "MinhaJanela" herda todas as propr iedades de JFrame através da classe Pai.

```
import javax.swing.JFrame;

public class MinhaJanela extends JFrame{

    public MinhaJanela(String titulo){
        super(titulo);
    }
}
```

Podemos definir um padrão para qualquer janela criada partindo de “MinhaJanela”, vamos agora definir que toda janela nossa terá obrigatoriamente um título. Basta inserirmos um construtor e redirecionar esse valor para classe que tem a responsabilidade de receber o título de uma Janela.

No construtor podemos também definir qualquer outra propriedade na sua chamada, como coordenadas, cor do plano de fundo, procedimento utilizado quando for fechada a janela, layout, visibilidade, etc. Veja o Exemplo abaixo ainda no mesmo construtor:

```
public MinhaJanela(String titulo){
    super(titulo);
    super.setBounds(100, 100, 400, 300);
    super.getContentPane().setBackground(Color.GRAY);
    super.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    super.setLayout(null);
    super.setVisible(true);
}
```

Agora indicamos que qualquer instancia de “MinhaJanela” terá as mesmas propriedades, sendo diferenciadas apenas pelo seu Título.
Vamos instanciar no método main a classe que acabamos de definir:

```
public static void main(String args[]){
  MinhaJanela jan = new MinhaJanela("Janela personalizada 1");
}
```

Temos então o mesmo resultado do tópico anterior, perceba que se quisermos criar uma 2 janela por exemplo, não precisamos definir todas as propriedades novamente, basta instanciar uma nova “MinhaJanela” definindo um segundo título, assim teremos uma nova janela.
Usando herança, podemos através da nossa classe definida, acessar os elementos do Jframe baseado no seu nível de encapsulamento. No Exemplo abaixo usaremos o método setLocation() para alterar a localização da nosssa janela. Esse método não foi criado por nós, mas foi definido pelas classes Pai nas definições de herança da Classe Jframe.

```
public static void main(String args[]){
  MinhaJanela jan = new MinhaJanela("Janela personalizada 1");
  MinhaJanela jan2 = new MinhaJanela("Janela personalizada 2");
  jan.setLocation(100, 100);
  jan2.setLocation(600, 100);
}
```

Temos então o seguinte resultado final:

Iremos usar o exemplo da Classe Minha Janela para criar os componentes nos próximos tópicos.

14.4 JLabel

É um pequeno componente com o qual colocamos pedaços de texto sem nenhuma função, a não ser a de indicar para quê servem os campos de uma tela qualquer.

```
public class MinhaJanela extends JFrame{

    //Criando um Atribuito do tipo JLabel
    private JLabel lblNome;

    public MinhaJanela(String titulo){
        super(titulo);
        super.setLayout(null);

        //Instanciado o JLabel
        this.lblNome = new JLabel("Frase indicativa");

        //Definindo posição do JLabel instanciado.
        this.lblNome.setBounds(100, 100, 200, 30);

        //Adicioandno JLabel na Janela
        super.getContentPane().add(this.lblNome);
        //Defininções da janela
        super.setBounds(100, 100, 400, 300);
        super.getContentPane().setBackground(Color.GRAY);
        super.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        super.setVisible(true);
    }
}
```

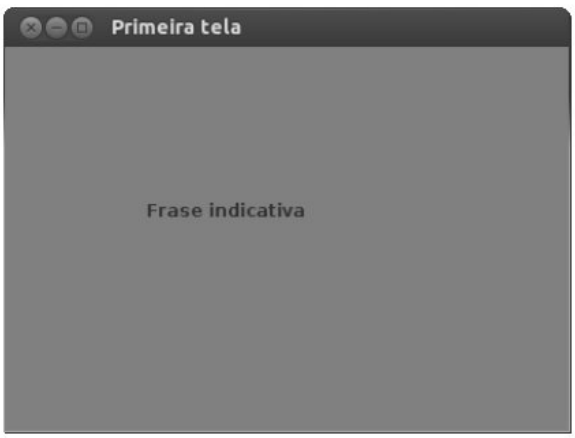
O código acima não apresenta grandes novidades. Apenas instanciamos o Jlabel , dando-lhe um texto qualquer para exibir. Depois, demos o mesmo setBounds nele, em seguida fizemos a ação realmente revolucionária:

super.getContentPane().add(texto);

Adicionamos o objeto “texto” na tela usando super.getContentPane(). Note que o setBounds do “texto” passa a usar como bordas as bordas do JFrame. E fica contido nele.

O JLabel pode ter desenhos por dentro, mas veremos isso mais adiante, quando tratarmos do Icon, outro tipo de objeto Java.

O resultado de nossa janela é esse:



14.5 JButton

O JButton é, finalmente, o botão. Colocamos ele nas telas com inúmeras funções. Estas funções serão tratadas mais adiante, quando chegarmos às aulas sobre Eventos. Por enquanto, basta-nos desenhar os botões.

O JButton poderá ser instanciado de três formas diferentes:

JButton() – aqui, teremos depois que rodar um método setText("algum texto") para dar-lhe um conteúdo(um conteúdo que aparece para o usuário).

JButton(String) – aqui, escrevemos algum texto para aparecer no botão e o usuário saber do que se trata.

JButton(icon) – neste caso, adicionamos uma figura ao botão para dar -lhe uma aparência personalizada.

Veja este código:

```
public class MinhaJanela extends JFrame{
    //Criando um Atribuito do tipo JLabel
    private JLabel lblNome;
    //Criando um Atributo do tipo JButton
    private JButton btnOk;

    public MinhaJanela(String titulo){
        super(titulo);
        super.setLayout(null);
        //Instanciado o JLabel
        this.lblNome = new JLabel("Frase indicativa");
        //Definindo posição do JLabel instanciado.
        this.lblNome.setBounds(100, 100, 200, 30);
        //Instanciado botao.
        this.btnOk = new JButton("Ok");
        //Definindo posiçao do JButton instanciado.
        this.btnOk.setBounds(100, 200, 100, 30);
        //Adicioanando JLabel  e JButton na Janela
        super.getContentPane().add(this.lblNome);
        super.getContentPane().add(this.btnOk);
        //Defininções da janela
        super.setBounds(100, 100, 400, 300);
        super.getContentPane().setBackground(Color.GRAY);
        super.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        super.setVisible(true);
    }
}
```

No exemplo acima, usamos praticamente a mesma tela criada antes, porém adicionamos as rotinas de criação de um botão. Primeiro, instanciam os a classe JButton e usamos os métodos já conhecidos setBounds para definir suas coordenadas.

A nossa janela ficará assim:

14.6 JTextField

O JTextField é uma caixa utilizada para que o usuário possa escrever um texto de uma linha só. Trata-se de um componente bastante simples, que tem basicamente as mesmas especificações do que acabamos de ver nos componentes anteriores.

Enriquecendo o código anterior, adicionando um JTextField:

```
public class MinhaJanela extends JFrame{

    private JLabel lblNome;
    private JButton btnOk;
    //Criando um atributo do Tipo JTextField
    private JTextField txtNome;

    public MinhaJanela(String titulo) {
        super(titulo);
        super.setLayout(null);
        this.lblNome = new JLabel("Frase indicativa");
        this.lblNome.setBounds(100,100, 200, 30);

        this.btnOk = new JButton();
        this.btnOk.setBounds(100, 200, 100, 30);
        this.btnOk.setText("Ok");
        //Instanciando uma caixa de texto
        this.txtNome = new JTextField();
        //Definindo coordenadas
        this.txtNome.setBounds(100, 130, 150, 20);

        super.getContentPane().add(this.lblNome);
        //Adicionando no Painel
        super.getRootPane().add(this.txtNome);
        super.getContentPane().add(this.btnOk);

        super.setBounds(100, 100, 400, 300);
        super.getContentPane().setBackground(Color.GRAY);
        super.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        super.setVisible(true);
    }
} }
```

O tratamento do texto de um JTextField dar-se-á por meio de dois métodos bastante simples:

getText() – servirá para capturar o quê o usuário digitou no campo.

setText() – servirá para determinarmos que texto deve ir no campo. Muito útil quando se abre um registro de um banco de dados e o campo JTextField deve receber um campo de registro para alteração.

A nossa janela ficará assim após acrescentar o JTextField:

14.6 Icon

A classe Icon tem um uso bastante simples, mas muito importante. Com ela, podemos carregar uma imagem (devidamente criada em um arquivo JPG ou PNG), para dentro do programa, utilizando esta imagem como quisermos.

Primeiro temos que inserir uma imagem no seu projeto, crie uma pasta dentro do seu projeto chamada "img" e nela coloque qualquer imagem, em seguida insera uma imagem chamada "logo.png";

Em sua Janela faça o seguinte código:

```
public class JanEventoActionListener extends JFrame{

    private final Container cont;
    private JLabel lblIcone;

    public JanEventoActionListener(String titulo){
        super(titulo);
        super.setLayout(new FlowLayout());
        this.cont = super.getContentPane();
        //Capturando imagem e repassando para uma label
        try {
            URL url = super.getClass().getResource("/img/logo.png");
            Icon i = new ImageIcon(url);
            this.lblIcone = new JLabel(i);
        } catch (Exception e) {
            JOptionPane.showMessageDialog(null, "Erro na imagem ");
        }

       this.cont.add(this.lblIcone);
        this.confJanela();
    }

     public void confJanela(){...8 linhas }

}
```

No exemplo acima carregamos as informações da Imagem que importamos, em seguida partindo da construção de uma ImageIcon passando como argumento url indicada, capturamos apenas as informações de Icon para ser utilizada nos nossos componentes, alguns componentes aceitam Icon como argumento no construtor, como Jbutton, Jpanel, etc.

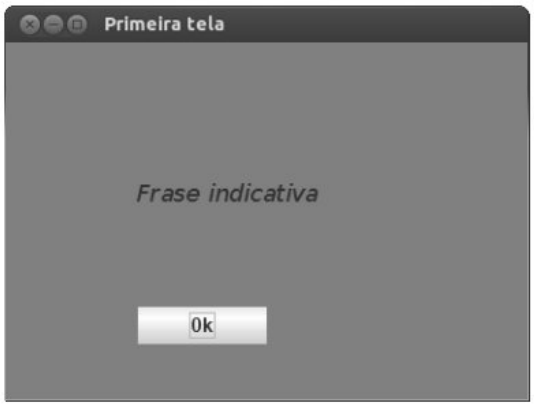
14.7 Editando fonte com o setFont()

Podemos formatar as fontes de nossos botões e rótulos.

Vamos usar o método setFont dessa forma:

```
//Alterando fonte da label para tipo Arial, italic com tamanho 18
this.lblNome.setFont(new Font("Arial", Font.ITALIC , 18));

//Alterando fonte do botão para tipo Courier, BOLD com tamanho 15
this.btnOk.setFont(new Font("Courier",Font.BOLD,15));
```

Nossa janela ficará assim:



14.8 JMenuBar, JMenu e JMenuItem

Estas classes são utilizadas sempre em conjunto, para criar um menu na parte superior do JFrame, e deixar seu software com aquela facilidade de uso que os usuários modernos exigem.

Aqui, começamos a trabalhar uma modalidade de programação que costumo chamar informalmente de "empilhamento de objetos", pois os objetos só funcionam quando colocamos um dentro do outro em uma espécie de pirâmide.

O objeto que serve de base é o JMenuBar, adicionad o diretamente ao JFrame, e sobre ele vão todas as coisas. Ele é a barra clara no alto do JFrame, dentro da qual os menus ficarão.

Depois, temos os JMenu, que ficam dentro da MenuBar e podem, inclusive, ficar um dentro do outro. Dentro dos JMenu, ficam os JMenuItem, que têm uma função cada um. Para ativar suas funções, utilizaremos os Listeners, mais adiante. Deixemos isso para outro capítulo.

```
public class JavaSwing extends JFrame {
    //Criando Atribuito de JMenuBar
    private final JMenuBar barraMenu;
    private final JMenu mnArquivo,mnSalvar;
    private final JMenuItem mniNovo,mniAjuda,mniMod1,mniMod2;
    public JavaSwing(String titulo) {
        super(titulo);
        super.setLayout(null);
        //Criando barra de Menu Principal
        this.barraMenu = new JMenuBar();
        //Criando os Menus
        this.mnArquivo = new JMenu("Arquivo");
        this.mnSalvar = new JMenu("Salvar como..");
        //Criando Itens de Menus
        this.mniNovo = new JMenuItem("Novo");
        this.mniAjuda = new JMenuItem("Ajuda");
        this.mniMod1 = new JMenuItem("Modo 1");
        this.mniMod2 = new JMenuItem("Modo 2");
```

Criando uma janela, um JMenuBar(container de menus), Menus(JMenu) e por fim itens de menus(JMenuItem).

Agora vamos adicionar o JMenuBar dentro da janela, depois adicionar os JMenus dentro do JmenuBar, e por fim adicionar dentro do JMenu itens de menus(JMenuItem).

```
        //Adicionando Itens de Menu no Menu Salvar
        this.mnSalvar.add(this.mniMod1);
        this.mnSalvar.add(this.mniMod2);
        //Adicionando Menus e Itens de Menu no Menu de Arquivo
        this.mnArquivo.add(this.mniNovo);
        this.mnArquivo.add(this.mnSalvar);
        //Adicionando Menu e Itens de Menu na Barra Principal
        this.barraMenu.add(this.mnArquivo);
        this.barraMenu.add(this.mniAjuda);

        //definindo coordenadas
        this.barraMenu.setBounds(0, 0, 400, 20);
        //ADicionando Barra Principal na camada de Barra de Menu
        super.getLayeredPane().add(this.barraMenu);
        this.confJanela();
    }
```

A ultima linha de código representa um método que será responsável pelas configurações da janela Principal, veja a baixo a sua implementação.

```
    //Método que irá definir a janela principal.
    private void confJanela(){
        //Uma outra forma de alterar o tamanho de um Janela ou Componente
        super.setSize(400, 300);
        //Referencia de coordenada centralizada no monitor
        super.setLocationRelativeTo(null);
        super.getContentPane().setBackground(Color.GRAY);
        super.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        super.setVisible(true);
    }
}
```

Agora vamos ver como ficou nossa janela:

14.9 JCheckBox

Uma JCheckBox é um daqueles componentes em que vemos uma caixinha quadrada com uma frase ao lado, e podemos marcar ou desmarcar a caixinha quadrada.

As CheckBoxes não são usadas para fazer escolhas com apenas uma opção válida, podendo ser todas marcadas ou todas desmarcadas livremente, desde que o programador não coloque restrições no próprio código.

Veja como trabalhar com JCheckBox no exemplo de código abaixo:

```
public class JanPreferencia extends JFrame{
    //Definindo Atributos JCheckBox
    private final JCheckBox cbPizza,cbRefri;

    public JanPreferencia(String title) {
        super(title);
        super.setLayout(null);
        //Instanciando JCheckBox
        this.cbPizza = new JCheckBox("Pizza");
        this.cbRefri = new JCheckBox("Refrigerante");
        //Definindo coordenadas
        this.cbPizza.setBounds(10, 10, 150, 20);
        this.cbRefri.setBounds(10, 35, 150, 20);
        //Inserinido na janela Principal.
        this.getContentPane().add(this.cbPizza);
        this.getContentPane().add(this.cbRefri);
        //Chamada ao método
        this.confJanela();
    }
    //Método responsável pelas defininções da Janela
     private void confJanela(){...8 linhas }
}
```

Depois, para termos um retorno do valor de cada JCheckBox, utilizamos o método isSelected(), que retorna um valor booleano.

Assim:

boolean x = cbPizza.isSelected();

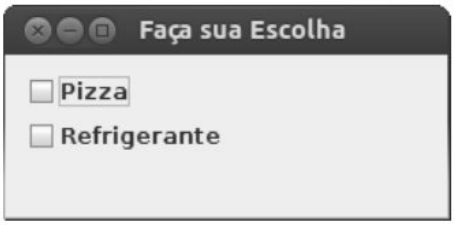
Ou ainda:

if (cbPizza .isSelected()) {

- Funções a serem executadas.

}

Veja como ficou nossa janela:



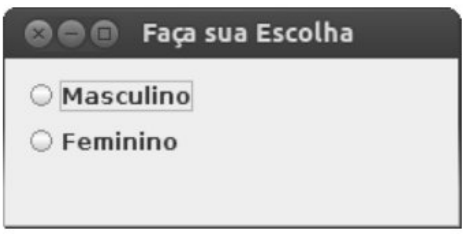
14.10 JRadioButton

O JRadioButton funciona apenas quando pensamos nele inserido em um grupo. Eu posso colocar quantas opções quiser, formando um menu, e se todos os meus JRadioButtons estiverem no mesmo grupo, o usuário só poderá marcar um deles. Ao marcar um, os outros perdem a marcação. Para isso, temos que adicionar também um ButtonGroup.

```
public class JanPreferencia extends JFrame{
    //Definindo Atributos JRadioButton
    private final JRadioButton rbMaculino,rbFeminino;
    //Definindo grupo de botões
    private final ButtonGroup bgSexo;

    public JanPreferencia(String title) {
        super(title);
        super.setLayout(null);
        //Instanciando JCheckBox
        this.rbMaculino = new JRadioButton("Masculino");
        this.rbFeminino = new JRadioButton("Feminino");
        //Definindo coordenadas
        this.rbMaculino.setBounds(10, 10, 150, 20);
        this.rbFeminino.setBounds(10, 35, 150, 20);
        //Agrupando JRadioButtons Criados.
        this.bgSexo = new ButtonGroup();
        this.bgSexo.add(this.rbMaculino);
        this.bgSexo.add(this.rbFeminino);
        //Inserinido na janela Principal.
        this.getContentPane().add(this.rbMaculino);
        this.getContentPane().add(this.rbFeminino);
        //Chamada ao método
        this.confJanela();
    }
    //Método responsável pelas defininções da Janela
     private void confJanela(){...8 linhas }
}
```

Veja o código abaixo e saiba como trabalhar com JRadioButton. Veja como ficou nossa janela:

JComboBox



A JComboBox desenha uma caixinha parecida com a JTextField, só que ao invés de termos um texto a escrever, temos uma seta com a qual abrimos uma fileira de possibilidades de preenchimento para a caixa. Veja como trabalhar como trabalhar com JcomboBox, observando o exemplo abaixo:

```
public class JanPreferencia extends JFrame{
    //Deinindo Atributos
    private final JLabel lblCidade;
    private final JComboBox comb;


    public JanPreferencia(String title) {
        super(title);
        super.setLayout(null);
        //Instanciando e Definindo tamanho e posição do JLabel
        this.lblCidade = new JLabel("Cidade :");
        this.lblCidade.setBounds(10, 10, 60, 20);
        //Instanciando e adicionando elementos no JComboBox
        this.comb = new JComboBox();
        this.comb.addItem("Tabuleiro do Norte");
        this.comb.addItem("Fortaleza");
        this.comb.addItem("Sobral");
        this.comb.addItem("Quixadá");
        // Definindo tamanho e posição do JComboBox
        this.comb.setBounds(80, 10, 160, 20);
        //Adicionando elementos na Janela Principal
        super.getContentPane().add(this.lblCidade);
        super.getContentPane().add(this.comb);
        //Chamada ao método
        this.confJanela();
    }
    //Método responsável pelas defininções da Janela
     private void confJanela(){...8 linhas }
}
```

Para obter o retorno das informações, temos uma série de métodos que nos dizem o que, afinal, o usuário escolheu. Os mais usados são os seguintes:

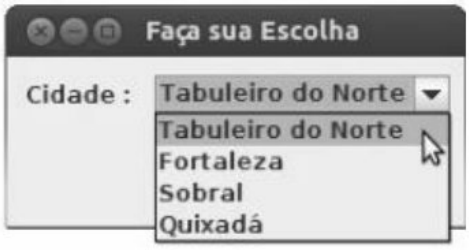
comb.getSelectedItem()

Este retorna o texto que está na caixa do Combo. Poderá ser usado especialmente para quando colocamos o setEditable em TRUE, ou seja, quando o usuário pode escrever na caixa. O retorno deste método é uma String, com o texto da caixa.

comb.getSelectedIndex();

Este método retorna o índice do item selecionado (como se as opções fossem um vetor).

Veja abaixo o resultado:



Existem mais componentes Swing, além dos mostrados até aqui. Foram mostrados os principais componentes. Caso você precise ou deseje conhecer mais componentes Swing acesse:

http://docs.oracle.com/javase/7/docs/api/

15.0 Gerenciadores de Layout

Gerenciamento de layout (Layout Management) é o processo de determinar o tamanho e a posição dos componentes na janela gráfica do programa. É ele o responsável por determinar onde o componente irá ficar como irá ficar, qual o comportamento a ser tomado na hora de redimensionar a tela.

Por padrão o Java vem com cinco gerenciadores de layout: BorderLayout, BoxLayout, FlowLayout, GridBagLayout, GridLayout e um adicional chamado CardLayout. São os gerenciadores de layout que determinam como os componentes ir ão ficar em seu programa, portanto um bom conhecimento sobre o funcionamento dos mesmos faz -se necessário para ter domínio de programação gráfica em Java.

Geralmente, quando chamamos o método "add" para adicionar um componente na tela, ele será posicionado de acordo com o gerenciador de layout previamente configurado. A maior parte simplesmente vai adicionando os componentes com base na ordem em que foram programados, porém alguns outros, como o BorderLayout, necessitam que você informe a posição relativa no container.

A escolha do gerenciador de layout depende muito das necessidades do programa. Por padrão todos os objetos Containers são configurados para usar o FlowLayout, enquanto um JFrame e JDialog já usam o BorderLayout como padrão.

Em ambos os casos é usado o método "setLayout" para especificar um novo layout manager, sendo necessário passar para este método o tipo que desejamos. Abaixo há um exemplo para cada tipo de gerenciador de layout, ficando assim fácil para vermos como cada um se comporta.

15.1 BorderLayout

BorderLayout trabalha como algumas coordenadas predefinidas , como norte, sul, leste, oeste e centro. Para poder usar esse recurso temos que antes usar o método setLayout que antes era definido como “null” para uma nova instancia de Border Layout.

Veja o Exemplo abaixo.

```
public class JanPreferencia extends JFrame{
    //Definindo atributos
    private final JButton btn1,btn2,btn3,btn4,btn5;
    private Container contPane;

    public JanPreferencia(String title) {
        super(title);
        //Definindo layout como Border Layout
        super.setLayout(new BorderLayout());
        //Resgatando referencia de container da Janela Principal.
        this.contPane = super.getContentPane();
        this.btn1 = new JButton("Botão 1");
        this.btn2 = new JButton("Botão 2");
        this.btn3 = new JButton("Botão 3");
        this.btn4 = new JButton("Botão 4");
        this.btn5 = new JButton("Botão 5");

        //Definindo Coordenadas BorderLayout
        this.contPane.add(this.btn1,BorderLayout.NORTH);
        this.contPane.add(this.btn2,BorderLayout.CENTER);
        this.contPane.add(this.btn3,BorderLayout.WEST);
        this.contPane.add(this.btn4,BorderLayout.SOUTH);
        this.contPane.add(this.btn5,BorderLayout.EAST);


        this.confJanela();
    }
    //Método responsável pelas defininções da Janela
     private void confJanela(){...8 linhas }
}
```

Veja como ficou nossa janela:

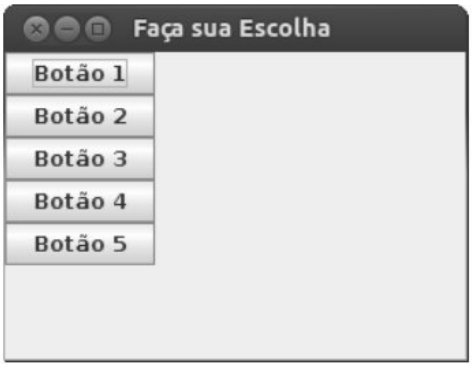
15.2 BoxLayout

Um BoxLayout coloca os componentes em uma única linha ou coluna, permitindo ainda que você especifique o alinhamento deles. Para que um BoxLayout possa existir temos que passar como argumento no seu construtor o container que será aplicado e um alinhamento que poderá ser horizontal ou vertical,

representado pelas constantes estáticas da classe BoxLayout : X_AXIS (para alinhamento horizontal dos meus componentes) e Y_AXIS (para alinhamento horizontal).

```
public class JanPreferencia extends JFrame{
    //Definindo atributos
    private final JButton btn1,btn2,btn3,btn4,btn5;
    private Container contPane;

    public JanPreferencia(String title) {
        super(title);
        //Resgatando referencia de container da Janela Principal.
        this.contPane = super.getContentPane();
        //Definindo layout como BoxLayout
        super.setLayout(new BoxLayout(this.contPane, BoxLayout.Y_AXIS));
        //Instanciando botões da Janela.
        this.btn1 = new JButton("Botão 1");
        this.btn2 = new JButton("Botão 2");
        this.btn3 = new JButton("Botão 3");
        this.btn4 = new JButton("Botão 4");
        this.btn5 = new JButton("Botão 5");

        //Adicionando elementos em coordenadas predefinidas pelo BoxLayout
        this.contPane.add(this.btn1);
        this.contPane.add(this.btn2);
        this.contPane.add(this.btn3);
        this.contPane.add(this.btn4);
        this.contPane.add(this.btn5);
        this.confJanela();
    }
    //Método responsável pelas defininções da Janela
     private void confJanela(){...8 linhas }
}
```

Nossa janela ficará dessa forma:



15.3 FlowLayout

Como dito anteriormente, um FlowLayout é o gerenciador padrão de todo Container, caso não seja especificado outro. FlowLayout posiciona os componentes lado a lado. Ela se diferencia de um BoxLyout pois seus componentes podem se alinha horizontalmente e verticalmente baseando-se nas dimensões do container. Veja o exemplo de sua aplicação.

```
public class JanPreferencia extends JFrame{
    //Definindo atributos
    private final JButton btn1,btn2,btn3,btn4,btn5;
    private Container contPane;

    public JanPreferencia(String title) {
        super(title);
        //Resgatando referencia de container da Janela Principal.
        this.contPane = super.getContentPane();
        //Definindo layout como FlowLayout
        super.setLayout(new FlowLayout());
        //Instanciando botões da Janela.
        this.btn1 = new JButton("Botão 1");
        this.btn2 = new JButton("Botão 2");
        this.btn3 = new JButton("Botão 3");
        this.btn4 = new JButton("Botão 4");
        this.btn5 = new JButton("Botão 5");

        //Adicionando componentes no layout definido
        this.contPane.add(this.btn1);
        this.contPane.add(this.btn2);
        this.contPane.add(this.btn3);
        this.contPane.add(this.btn4);
        this.contPane.add(this.btn5);

        this.confJanela();
    }
    //Método responsável pelas defininções da Janela
     private void confJanela(){...8 linhas }
}
```

Veja como ficou nossa janela:

15.4 GridLayout

Um GridLayout atua como uma grade de uma planilha de cálculo, colocando os componentes em linhas e colunas pré-determinadas e deixando os componentes com o mesmo tamanho, para isso precisamos definir um limite de linhas e colunas para esse layout. Veja isso no exemplo abaixo:

```
public class JanPreferencia extends JFrame{
    //Definindo atributos
    private final JButton btn1,btn2,btn3,btn4,btn5;
    private Container contPane;

    public JanPreferencia(String title) {
        super(title);
        //Resgatando referencia de container da Janela Principal.
        this.contPane = super.getContentPane();
        //Definindo layout como GridLayout
        super.setLayout(new GridLayout(3,2));
        //Instanciando botões da Janela.
        this.btn1 = new JButton("Botão 1");
        this.btn2 = new JButton("Botão 2");
        this.btn3 = new JButton("Botão 3");
        this.btn4 = new JButton("Botão 4");
        this.btn5 = new JButton("Botão 5");

        //Adicionando Componentes em posições já definidas
        this.contPane.add(this.btn1);
        this.contPane.add(this.btn2);
        this.contPane.add(this.btn3);
        this.contPane.add(this.btn4);
        this.contPane.add(this.btn5);

        this.confJanela();
    }
    //Método responsável pelas defininções da Janela
     private void confJanela(){...8 linhas }
}
```

Veja como ficou nossa janela:

15.5 GridBabLayout

O mais flexível gerenciador de layout é o GridBagLayout. Ele permite colocar componentes em grades de

colunas, sendo possível um componente ocupar mais de uma coluna ao mesmo tempo. As linhas também não precisam necessariamente ter os mesmos tamanhos, ou seja, você pode configurar diferentes larguras e alturas de acordo com a necessidade. Eis o exemplo abaixo.

```
public class JanPreferencia extends JFrame{
    //Definindo atributos
    private final JButton btn1,btn2,btn3,btn4,btn5;
    private GridBagConstraints c;
    private Container contPane;

    public JanPreferencia(String title) {
        super(title);
        //Resgatando referencia de container da Janela Principal.
        this.contPane = super.getContentPane();
        //Definindo layout como GridBagLayout verticalmente
        super.setLayout(new GridBagLayout());
        //Instanciando botões da Janela.
        this.btn1 = new JButton("Botão 1");
        this.btn2 = new JButton("Botão 2");
        this.btn3 = new JButton("Botão 3");
        this.btn4 = new JButton("Botão 4");
        this.btn5 = new JButton("Botão 5");
        //Setando Coordenadas com GridBagLayout
        this.contPane.add(this.btn1,this.definindoLoc(0, 1));
        this.contPane.add(this.btn2,this.definindoLoc(1, 2));
        this.contPane.add(this.btn3,this.definindoLoc(2, 3));
        this.contPane.add(this.btn4,this.definindoLoc(0, 4));
        this.contPane.add(this.btn5,this.definindoLoc(0, 5));
        this.confJanela();
    }
    //Método que irá definir as coordenadas com gridBagLayout
    private GridBagConstraints definindoLoc(int codX, int codY){
        this.c = new GridBagConstraints();
        this.c.gridx = codX;
        this.c.gridy = codY;
        return this.c;
    }
    //Método responsável pelas defininções da Janela
     private void confJanela(){...8 linhas }
}
```

Veja como ficou nossa janela:

Logicamente é possível fazer muito mais com os gerenci adores de layout, há várias maneiras de configurar e utilizar. Para obter mais informações sobre os métodos disponíveis para cada um consulte a API, disponível em:

http://docs.oracle.com/javase/7/docs/api/

16.0 Eventos (Listeners)

Eventos em Java nada mais é que métodos especiais que são realizados baseados em alguma ação de um usuário seja ele, um clique, um digitar de uma tecla, ao passar o mouse, etc.. Iremos ver nos próximos tópicos como utilizar esses métodos especiais em nossos projetos.

ActionListener

O ActionListener. Trata-se de uma classe que detecta se houve uma ação em uma referencia a que ela está associada (um botão, por exemplo) e que, dependendo da ação, executará uma função. ActionListener é representado por uma interface, então para que a minha classe possa realizar tratamento de eventos, a primeira implementação que iremos fazer é dar um "implements" na classe que queremos ter essa funcionalidade , vejamos o exemplo abaixo:

```
public class JanEventoActionListener extends JFrame implements ActionListener{
    private final Container cont;
    private final JButton btnOk;
    private final JTextField txtNome;
```

Iremos também criar uma caixa de texto e um botão para utilizar um evento que irá capturar o que foi digitado e imprimir na tela.

A Implementação ActionListener irá obrigar a minha classe a conter um método chamado, actionPerformed, com um parâmetro do tipo "ActionEvent". Veja a implementação abaixo.

```
@Override
public void actionPerformed(ActionEvent e) {

}
```

Agora iremos construir nosso layout:

```
public JanEventoActionListener(String titulo){
    super(titulo);
    super.setLayout(new FlowLayout());
    this.cont = super.getContentPane();

    this.btnOk = new JButton("Ok");
    this.btnOk.addActionListener(this);
    this.btnOk.setActionCommand("Ok");

    this.txtNome = new JTextField(8);
    this.cont.add(this.txtNome);

    this.cont.add(this.btnOk);
    this.confJanela();
```

Os métodos que iremos utilizar para criar o evento são, **addActionLIstener** e **setActionCommand**, eles serão responsáveis por redirecionar e enviar um comando para o método que nós criamos **actionPerformed**.

addActionListener: Método que irá receber uma classe que irá conter uma implementação actionListener, no nosso caso, nossa própria classe terá esse método, o motivo do argumento "this" no exemplo anterior.

SetActionCommand: Método que irá enviar uma instrução para o método actionPerformed, com esse recurso, podemos por exemplo criar outros outros botões enviando outros comandos para o mesmo método realizando assim outras funções em uma mesma janela. No exemplo acima utilizamos um comando que definimos como "OK".

Iremos agora ver como irá iremos implementar o método actionPerformed: O processo é bem simples, caso o ActionCommand do evento "e" seja igual a "OK" (comando que definimos usando setActionCommand), uma caixa com uma mensagem aparecerá na tela informando o nome digitado.

```
//Método que será acionado no clique do botão.
@Override
public void actionPerformed(ActionEvent e) {
 if(e.getActionCommand().equals("Ok")){
     String txtDigitado = this.txtNome.getText();
     JOptionPane.showMessageDialog(null, "Texto Digitado = "+txtDigitado);
 }
}
```

Se para modificar um comando utilizamos o método setActionCommando , para que possamos resgatar o que foi modificado usaremos os método getActionCommando e compararemos com o comando que queremos acionar.

Agora iremos capturar o que foi digitado na caixa de texto, o exemplo acima mostra que chamamos o atributo txtNome que foi definindo com JtextField e utilizar o método getText() que tem como retorno uma String, essa String será capturada impressa na tela no momento do clique do botão.

Vejamos o resultado dessa operação:

Importante: Antes de passa para o próximo tópico, tenha certeza que entendeu todos os processos envolvidos, pois os próximos eventos serão implementações semelhantes.

16.1 FocusListener

O FocusListener se parece com o ActionListener em seu modo de construção, mas serve para detectar quando o foco (ou seja, o cursor, a seleção, não o ponteiro do mouse) está sobre um objeto ou não.

Ele possui dois métodos que precisam ser implementados: o focusGained e o focusLost, um para quando o objeto ganha foco e outro para quando perde.

Para ativar o método, usamos addFocusListener. Usando esse recurso iremos mostrar um exemplo de alteração da cor de um plano de fundo baseado em um foco em uma caixa de texto. Veja o exemplo abaixo:

```
public class JanEventoFocusListener extends JFrame implements FocusListener{

    private final Container cont;
    private final JButton btnOk;
    private final JTextField txtNome;

    public JanEventoFocusListener(String titulo){
        super(titulo);
        super.setLayout(new FlowLayout());
        this.cont = super.getContentPane();

        this.btnOk = new JButton("Ok");

        this.txtNome = new JTextField(8);
        this.txtNome.addFocusListener(this);
        this.cont.add(this.txtNome);

        this.cont.add(this.btnOk);
        this.confJanela();
    }

     @Override
    public void focusGained(FocusEvent e) {
        this.txtNome.setBackground(new Color(200, 250, 200));
    }

    @Override
    public void focusLost(FocusEvent e) {
        this.txtNome.setBackground(Color.WHITE);
    }


     public void confJanela(){...8 linhas }

}
```

O resultado Final:

16.2 KeyListener

O KeyListener é uma interface que serve para que possamos coordenar as ações do usuário com o teclado quando ele está escrevendo em algum espaço ou utilizando alguma função. Posso atribuir KeyListener a qualquer objeto gráfico do J ava e detectar quando e o que o usuário está teclando.

A implementação KeyListener obriga a ter 3 métodos :

- **keyTyped:** Será ativado ao pressionar um botão caractere ou numérico do teclado.
- **keyPressed:** Será ativado ao segurar um botão, como SHIFT, CTRL, ALT etc.
- **keyReleased:** Será ativo ao deixar de pressionar o botão.

Para ativar o método, usamos addKeyListener. No exemplo abaixo alteramos a cor de plano de fundo da caixa de texto para cada uma das ações realizadas pelo usuário.

```
public class JanEventoKeyListener extends JFrame implements KeyListener{

    private final Container cont;
    private final JButton btnOk;
    private final JTextField txtNome;

    public JanEventoKeyListener(String titulo){
        super(titulo);
        super.setLayout(new FlowLayout());
        this.cont = super.getContentPane();

        this.btnOk = new JButton("Ok");

        this.txtNome = new JTextField(8);
        this.txtNome.addKeyListener(this);
        this.cont.add(this.txtNome);

        this.cont.add(this.btnOk);
        this.confJanela();
    }

    @Override
    public void keyTyped(KeyEvent e) {
       this.txtNome.setBackground(Color.GRAY);
    }

    @Override
    public void keyPressed(KeyEvent e) {
        this.txtNome.setBackground(Color.YELLOW);
    }

    @Override
    public void keyReleased(KeyEvent e) {
        this.txtNome.setBackground(Color.WHITE);
    }

    public void confJanela(){...8 linhas }

}
```

NOTA: Pode-se detectar qual o botão apertado pelo usuário explorando a variável “KeyEvent e”, que retornará o código da tecla apertada com o método: getKeyChar().

16.3 MouseListener

O MouseListener, como seu nome obviamente revela, serve para tratarmos eventos relacionados ao uso do mouse.

```
public class JanEventoMouseListener extends JFrame implements MouseListener{

    private final Container cont;

    public JanEventoMouseListener(String titulo){
        super(titulo);
        super.setLayout(null);
        this.cont = super.getContentPane();
        this.cont.addMouseListener(this);
        this.confJanela();
    }
```

A implementação MouseListener obriga a ter 5 métodos :

- mouseClicked: Será ativado no clique simples do mouse.
- mousePressed: Será ativado ao segurar o clique no mouse.
- mouseReleased: Será ativado ao deixar o clique do mouse.
- mouseEntered : Será ativado ao entrar na região do componente pelo mouse.
- mouseExited: Será ativa ao deixar a região do componente pelo mouse.

Para ativar o método, usamos **addMouseListener**. O código abaixo mostra como alterar o plano de fundo do Container usando event os do mouse Veja a implementações dos métodos abaixo:

```
@Override
public void mouseClicked(MouseEvent e) {
   this.cont.setBackground(Color.GREEN);
}

@Override
public void mousePressed(MouseEvent e) {
    this.cont.setBackground(Color.BLUE);

}

@Override
public void mouseReleased(MouseEvent e) {
    this.cont.setBackground(Color.RED);
}

@Override
public void mouseEntered(MouseEvent e) {
     this.cont.setBackground(Color.WHITE);
}

@Override
public void mouseExited(MouseEvent e) {
     this.cont.setBackground(Color.BLACK);
}
```

Nós podemos saber qual é a localização do ponteiro do mouse utilizando dois métodos:

getX() – retorna a posição do mouse no eixo X (horizontal) dentro do objeto a que estamos atribuindo o MouseListener.

getY() – Faz o mesmo, só que com a posição do eixo Y (vertical).