

PROGRAMAÇÃO ORIENTADA A OBJETOS

Professor Manoel Carvalho Marques Neto
Email: manoelnetom@gmail.com

Abstração: do Assembly a POO

Uma viagem pela história da abstração nas linguagens de programação. (Texto copiado da Internet sem autor definido).

Janeiro/2003

E no Começo Havia somente Bits...

A história da programação foi gradualmente aumentando os níveis de granularidade. Muito antigamente os programadores manipulavam bits individuais. Então a linguagem Assembly foi inventada e os programadores começaram a escrever instruções que eram equivalentes a alguns bytes. A vantagem era clara: em vez de pensar em termos de insignificantes 1s e 0s, você poderia pensar em termos do que o computador estava fazendo em um nível funcional (mova este valor para aquela posição de memória, multiplique esses dois bytes).

Isto é chamado de aumento do nível de abstração. Cada vez que você aumenta o nível de abstração em uma linguagem de programação, você obtém mais programas (medido em termos de bits) com menos esforço. Você também altera a linguagem com a qual você se comunica com o computador de uma linguagem binária para algo mais próximo da forma com que nos comunicamos em Inglês.

Cada unidade do nível de abstração tem um contrato: a linguagem faz uma promessa exata do que o computador fará quando a unidade for executada. Para a seguinte instrução na linguagem Assembly:

LD (BC),A

a linguagem promete que moverá o valor do registrador de nome A para a posição na memória apontada pelos registradores B e C. Obviamente, este é apenas um pequeno pedaço do que você gostaria que o computador realmente fizesse, tal como "ser um processador de textos" ou "renderizar um frame de um vídeo game", mas é muito mais claro e mais fácil de usar do que isto:

00000010

Pode não parecer tão fácil assim de se lembrar de LD (BC), A, mas cada uma das letras tem um significado correto e relativamente fácil de lembrar: LD é a abreviação para LOAD; A, B e C referem a alguns registradores, e (BC) refere à forma de se fazer indireção na memória. 00000010 pode ser apenas sete 0s e um 1, mas a ordem é tão crítica quanto difícil de memorizar. Trocar dois dos bits para 00000100 significa INC B (incrementar o registrador B), o que é totalmente diferente.

Programação Estruturada

A linguagem Assembly tem uma característica reconhecidamente problemática chamada **goto**, onde o controle pode ser transferido de algum ponto no programa para qualquer outro ponto no programa. Isto torna o software incrivelmente difícil de se manter. É confuso depurar um pedaço de código, ou imaginar o que ele vai fazer, se você não puder imaginar qual o estado do computador quando o código começa. Este estado depende de onde o computador estava momentos antes de entrar naquele código. Com **goto**, poderia estar em

qualquer lugar em todo o programa. Os projetistas de linguagens de computadores queriam uma forma de limitar as formas que o código poderia ser executado.

Eles descobriram que muitos gotos eram usados de formas particulares. Algumas vezes eles eram usados em um laço: faça um conjunto de instruções, e então as faça novamente. Algumas vezes eles eram usados para fazer uma de duas opções, ou possível laço (faça todo este código em laço até que algum bit vire zero). Este é um dos primeiros exemplos da descoberta de um padrão de projeto (design pattern). E mais importante, eles conseguiram provar que você poderia fazer tudo usando apenas essas poucas instruções: o **goto** genérico não era necessário para nada.

Assim eles inventaram a programação estruturada. C é um exemplo de uma linguagem de programação estruturada, embora ela não tenha sido a primeira. A unidade básica (granularidade) de um programa C é uma expressão ou instrução, cada uma das quais é traduzida em uma série de instruções em linguagem Assembly. Assim como instruções em linguagem Assembly, instruções em linguagens estruturadas também têm contratos. Por exemplo, o contrato de

```
while(expression) {  
    statement1  
    statement2  
    ...  
}
```

é avaliar a expressão, e então realizar as instruções 1 e 2 (e assim por diante), e repetir isto enquanto a expressão for verdadeira. Isto é um contrato no sentido que o computador sempre fará isso. Todas as instruções serão feitas, na mesma ordem, toda vez, uma por vez, sempre (e apenas) até que a expressão se torne falsa.

A remoção do goto da linguagem de programação e substituí-lo com estruturas como for, while e if chama-se programação estruturada. As estruturas for, while e if representam o próximo nível de abstração acima da programação em Assembly.

Note que o contrato aqui é muito mais complicado do que o contrato do mnemônico da linguagem de montagem LD. Quanto mais alto for o nível de abstração, mais complexo os contratos são. Isto permite que você diga muito com pouca programação. Substituir mnemônicos compostos por letras por instruções de byte único pode ter sido a única vez na história da programação quando abstrações fizeram o programa mais comprido do que o código objeto resultante. Mas isto nem sempre é verdade: LD (BC),A tem tantas letras quanto a quantidade de bits de 00000010.

Uma importante característica do laço while é que você não pode pular para o meio dele: você entra nele apenas pelo topo. Com isto é fácil imaginar o que está acontecendo no programa quando ele entra no laço while. Se você quer imaginar o que o laço faz, você tem apenas que olhar para o código que vai dentro dele.

Alguns diriam que você as instruções continue, break e return geralmente simplificam o código. Geralmente é preferível evitar saltar para fora dos laços sempre que possível, mas não podemos tornar o código completamente obscuro só para evitar isto. Cada um faz da forma que achar melhor, mas geralmente um esforço economizado durante a implementação geralmente é gasto mais tarde quando voltamos para estender ou dar manutenção no código.

O Próximo Nível: Funções e Procedimentos

Instruções e código estruturado podem ser pensados em termos de operações da linguagem Assembly, em um nível mais alto de abstração. O próximo nível de abstração é agrupar instruções em unidades operacionais com contratos próprios. Instruções são juntadas para formar funções, procedimentos, sub-rotinas ou métodos como eles são chamados nas várias linguagens. O legal com funções é que elas limitam a quantidade de código para o qual você tem que olhar a fim de entender um pedaço de código.

```
/** Toma o array e retorna uma versão ordenada,  
 * remove duplicatas. Pode retornar o mesmo array, ou  
 * pode alocar um novo. Se não existem duplicatas, provavelmente  
 * retornará o array antigo. Se existem, terá que criar um novo. */  
int[] sort(int[] array)  
{  
    ... o corpo ...  
}
```

Você pode aprender muito sobre as funções sem mesmo ver o seu corpo. O nome sort, e o fato de ela receber um array de inteiros e retornar um (possivelmente diferente) array de inteiros, diz para você muito sobre o que a função supostamente faz. O resto do contrato é descrito no comentário, que fala sobre outras coisas tal como alocação de memória. Isto realmente é mais importante em C e C++ do que em Java, onde é freqüente o contrato expressar até quem é responsável por liberar a memória alocada na função.

A vida do programador que dará manutenção ao código fica muito mais simples porque o programa está dividido dentro dessas unidades funcionais. Uma regra comum é que uma função deveria ser apenas tão grande quanto uma tela de código. Isto torna possível visualizar de uma vez, uma completa, identificada e compreensível unidade do programa que você estará mantendo.

Os nomes das funções e procedimentos são uma parte crítica da abstração. Toma um pedaço de código e permite que você o referencie posteriormente com uma única palavra (ou uma coleção de palavras curtas, junteTudoDestaForma ou _algo_assim). Cada linha na função tem o objetivo de realizar o que o nome indica. Uma vez que o escopo da sua função cresce para além do que o nome representa você deve considerar quebrar a função em pedaços com nomes melhores. Se você começar a escrever um código como este:

```
void sortNamesAndSendEmail()  
{
```

```
// Ordena os nomes  
... Gasta 100 linhas para ordenar os nomes ...  
  
// Envia email  
... Gasta 500 linhas enviando o email ...  
  
}
```

é um bom indicador que é hora de começar a quebrar a função em pedaços. Provavelmente será melhor escrever duas funções:

```
sortNames()
```

```
sendEmail()
```

o que permite que você elimine o comprido e estranho nome de função `sortNamesAndSendEmail`.

Classes: Além da Programação Estruturada

Programação estruturada e funções de forma organizada resolveram alguns dos problemas de manutenção limitando a quantidade de código que você tem que olhar a fim de entender alguma dada linha. Entretanto, ainda havia uma forma em que distantes pedaços de código poderiam afetar uma linha de código em particular: estado.

O exemplo de ordenação mostrado anteriormente ordena apenas inteiros, o que não é um trabalho particularmente interessante. Por que você irá querer ordenar apenas uma lista de números? Muito provavelmente, você irá querer ser capaz de ordenar uma lista de objetos de algum tipo, com base em uma chave inteira. Ou, mais geralmente, você gostaria de ser capaz de ordenar com base em qualquer chave, desde que você possa comparar quaisquer dois objetos.

Mesmo antes da programação orientada a objetos, havia formas de agrupar blocos de dados em unidades funcionais. Em C, essas unidades eram chamadas de structs (estruturas). Contudo, não havia uma forma confiável de compará-las. Você precisava de algum nível de abstração um pouco mais alto do que o fornecido por estruturas que permitisse que você falasse qual de duas estruturas deveria vir primeiro. A abstração de agrupar código e suas definições de dados é chamada de classe na maioria das linguagens de programação.

C++ é uma linguagem orientada a objetos. Ela fornece um nível maior de abstração do que C. Em geral, níveis mais altos de abstração trazem uma queda na performance, e muitas pessoas criticam C++ pelo seu custo de performance quando comparado ao C.

Java objetiva um nível de abstração mais alto do que o encontrado em C++ por abstrair totalmente o acesso às posições na memória. Apesar de não ser a primeira linguagem a fazer isto (Lisp e Basic são algumas, entre as linguagens de programação de propósito geral), ela provavelmente teve a maior penetração no mercado.

E este nível de abstração também denigre a performance na maioria das vezes. Nem sempre, é claro. Uma vantagem para a abstração é que os tradutores intermediários podem realizar quaisquer otimizações que eles quiserem, desde que não violem os contratos. Quão maior for o programa, mais difícil será para você efetuar todas as otimizações por conta

própria e ainda manter-se em dia com os cronogramas. A quanto mais tempo uma linguagem existir, mais truques os escritores de compiladores aprendem sobre otimização. Cada vez mais, linguagens com níveis maiores de granularidade executam mais rapidamente do que aquelas com níveis menores. Não tem como você escrever um grande programa para um processador Pentium e torná-lo tão eficiente quanto o mesmo programa escrito em C; o pipeline sugará todos os seus ganhos de performance (mesmo se você saiba quais sejam eles).

Abstração: Custos e Benefícios

Em muitos projetos de manutenção de software, o custo da performance adicional de baixos níveis de abstração é muito mais alto do que o custo dos ciclos do computador que é muito mais alto do que o custo do ciclos de computador que seriam necessário para executar o programa. No posto de um programador de manutenção, seu tempo é extremamente caro. O tempo de seus usuários é mais caro ainda (e geralmente há mais deles do que você), portanto a exatidão do programa é a chave. Se os usuários perderem trabalho ou tempo esperando que o seu software fique correto, isto facilmente representa muito dinheiro.

Altos níveis de abstração levam a uma melhora na manutenção, simplesmente porque há menos código. E quanto menos código, menos você terá que ler para entendê-lo. Certamente há limites para isso. É melhor ter 50 linhas de código claro do que 10 linhas de total obscuridade. Em geral, contudo, usando níveis mais altos de abstração, você consegue melhorar a manutenibilidade.

É claro, há o lado ruim desses altos níveis de abstração: a performance. Quanto mais flexível for o programa, mais difícil será para otimizá-lo. Como um mantenedor, você terá que encontrar o equilíbrio correto. O velho ditado de C.A.R. Hoare que diz "Otimização prematura é a raiz para todo o mal" é particularmente aplicável à abstração. Escolher seus níveis de abstração apropriadamente, e otimizar aquelas partes que não podem ser feitas para funcionar no nível de abstração que você escolheu. O retorno está no tempo de programação, tanto em desenvolvimento quanto em manutenção, o que deixa os usuários felizes. Que é o objetivo, afinal de contas, não?

Programação Orientada a Objetos

A Programação Orientada a Objetos (POO), diferente de outros paradigmas, inclui tanto a fase de análise quanto a fase de projeto. Isso significa dizer que um programa orientado a objetos "bem feito" deve passar pelas duas etapas.

A análise é o estudo do domínio de um problema que leva a uma especificação de comportamentos observáveis externamente. Analisar significa obter as necessidades de um sistema e o que este precisa para satisfazer as necessidades do usuário. Em outras palavras, analisar é sinônimo de investigar.

A Análise Orientada a Objetos (AOO) é uma categoria de análise que é realizada para a elaboração de um software a ser implementado através de uma linguagem de programação orientada a objetos. Ela tem dois propósitos específicos:

- 1) Formalizar uma visão do mundo real dentro do qual o sistema será desenvolvido, estabelecendo os modelos (classes) que servirão como principais estruturas do sistema.

2) Formalizar a colaboração de um conjunto de modelos na execução do sistema proposto.

O projeto orientado a objetos é o processo de especificação das partes da construção de um programa. Durante o projeto são implementados, em uma linguagem de programação orientada a objetos, os objetos que farão parte do sistema.

Todo programa OO deve sempre começar a ser elaborado pela etapa de análise OO. Para que isso seja possível é pré-requisito um entendimento maior dos conceitos básicos do paradigma: Classes, Objetos, Abstração, Relações entre classes (Herança, Agregação, Composição e Dependência).

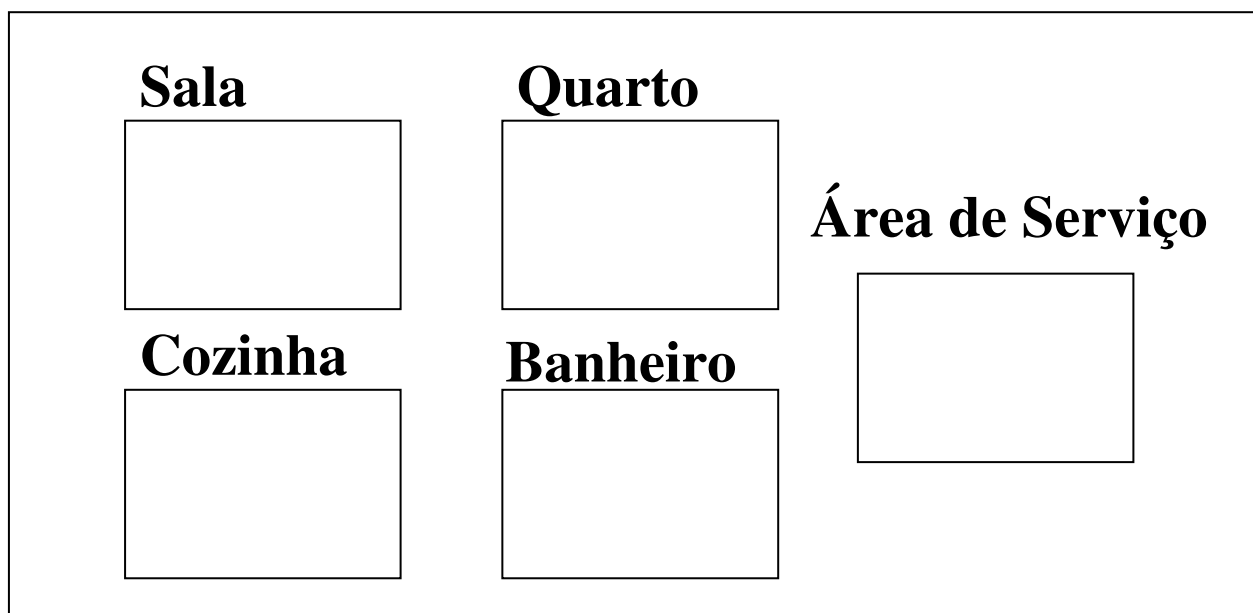
Classes

Existem várias definições com significados semelhantes sobre classes. Algumas usam um "linguajar" rebuscado que muitas vezes dificultam o seu entendimento. Vejamos um exemplo:

"Uma classe é uma descrição de um conjunto de objetos que compartilham os mesmos atributos (características), operações, relacionamentos e semântica".

Esse conceito descreve bem e muito formalmente o que é uma classe, porém parte do princípio que você já entende conceitos como objetos, atributos, operações e etc. Existem dois conceitos de classe que devem facilitar o seu entendimento. O primeiro deles faz uma relação direta com a área de arquitetura. Vejamos:

"Uma classe é um modelo para algo que existe no mundo real da mesma forma que a planta de um apartamento quarto e sala. Uma planta descreve o modelo de apartamentos que existe em um prédio como, por exemplo: Um AP tem Sala, Cozinha, Quarto, Banheiro e etc. Com uma planta são construídos diversos apartamentos diferentes(objetos do mundo real), mas que têm o mesmo modelo(classe)".



O outro conceito de classe é ligado diretamente ao conceito de “Tipo de Dados”. Uma linguagem de programação normalmente suporta tipos primitivos de dados e tipos não primitivos.

Um tipo de dado primitivo é aquele é conhecido a priori pelo compilador. Na linguagem pascal temos, por exemplo, os tipos primitivos longint, char, boolean e etc.

Um tipo de dado não primitivo é aquele criado pelo programador. Na linguagem C, por exemplo, a definição de uma *struct* é na verdade a definição de um tipo de dado não conhecido a princípio pelo compilador C.

```
struct Empregado {  
    int ID;  
    int idade;  
    float salario;  
    char departamento;  
}
```

Tanto em C quanto em pascal a definição de operações (procedimentos e funções) sobre um tipo não primitivo é feito separadamente da própria definição do tipo. Essa é a grande diferença para o paradigma OO. Em POO um tipo não primitivo de dados contempla tanto a definição dos campos como das funções ou procedimentos.

Uma classe pode então ser definida como “um tipo não primitivo de dados que contempla além dos campos (variáveis ou atributos) funcionalidades (comportamentos ou operações) de algo que se que descrever”. Em Java o mesmo tipo Empregado, com comportamentos associados, seria definido da seguinte maneira:

```
class Empregado{  
    int ID;  
    int idade;  
    float salario;  
    char departamento;  
  
    void calculaIdade(int anoNascimento){  
        this.idade=2006-anoNascimento;  
    }  
  
    int getIdade(){  
        return this.idade;  
    }  
}
```


OBJETOS

Um objeto é uma entidade concreta e independente que armazena dados, encapsula serviços, troca mensagens com outros objetos e é modelado para executar as funções do sistema. Um objeto é a “concretização” de uma classe. Vejamos o exemplo da classe apartamento citado na seção anterior:

A classe apartamento tem Sala, Cozinha, Quarto, Banheiro: Esse é o seu modelo. A partir desse modelo pode-se criar diversos apartamentos(ap101, ap102, ap 1301 e etc) que são diferentes mas seguem o mesmo modelo: todos são do tipo Apartamento.

Para criar um objeto em Java podemos usar a seguinte sintaxe:

```
Empregado emp = new Empregado();
```

Vejamos o que significa cada uma das partes do processo de criação de um objeto:

A palavra reservada *new* representa um dos operadores (ex: +, -, *, / , % e etc.) da linguagem Java. O seu objetivo é alocar espaço de memória para um objeto. Esse espaço representa um objeto e, como tudo que está alocado em memória, tem um tempo de vida definido. Daí o conceito que diz que um objeto é uma instância de uma classe.

O espaço na memória está organizado de acordo com modelo definido na classe. Para isso, além de usar a palavra reservada *new*, o processo de criação de objetos usa um comportamento (funcionalidade) especial presente em todas as classes: O Construtor.

Construtor é uma função especial para criar e “inicializar” novas instâncias da classe. Ela sempre tem o mesmo nome da Classe (ex: Classe Empregado -> Construtor Empregado()).

Cada objeto tem estado e comportamento. O estado é o valor guardado em cada uma de suas variáveis e o comportamento corresponde ao conjunto de métodos que ele pode executar. É comum a confusão de conceitos quando comparamos dois objetos.

Abstração

É o princípio de ignorar os aspectos de um assunto não relevantes para o propósito em questão, tornando possível uma concentração maior nos assuntos principais. Em termos de desenvolvimento de sistemas, isto significa concentrar-se no que um objeto é e faz antes de se decidir como ele será implementado.

O uso de abstração preserva a liberdade para tomar decisões de desenvolvimento ou de implementação apenas quando há um melhor entendimento do problema a ser resolvido. Abstração é o processo de observar uma entidade do mundo real e descrevê-la de acordo com as necessidades do software a ser implementado.

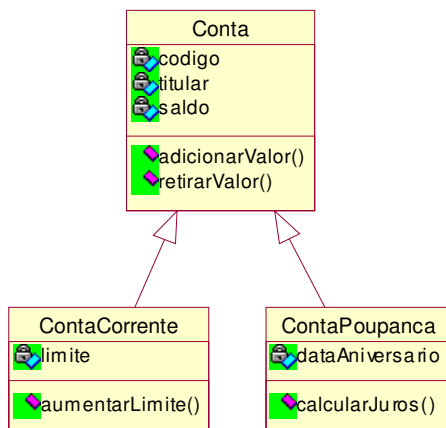
Encapsulamento

Encapsular é omitir informações pelo princípio de que uma determinada entidade esconde informações as quais são necessárias apenas à mesma. O encapsulamento previne manipulações incorretas de um objeto. O encapsulamento exhibe, através da Interface de utilização do objeto, quais as ações possíveis ao ele sem, no entanto, mostrar como estas foram implementadas.

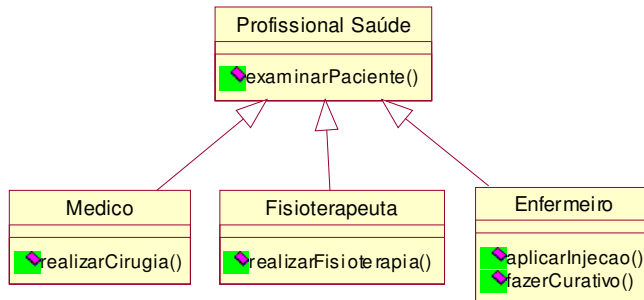
O uso de encapsulamento evita que um programa torne-se tão interdependente que uma pequena mudança tenha grandes efeitos colaterais. O uso de encapsulamento permite que a implementação de um objeto possa ser modificada sem afetar as aplicações que usam este objeto.

Herança

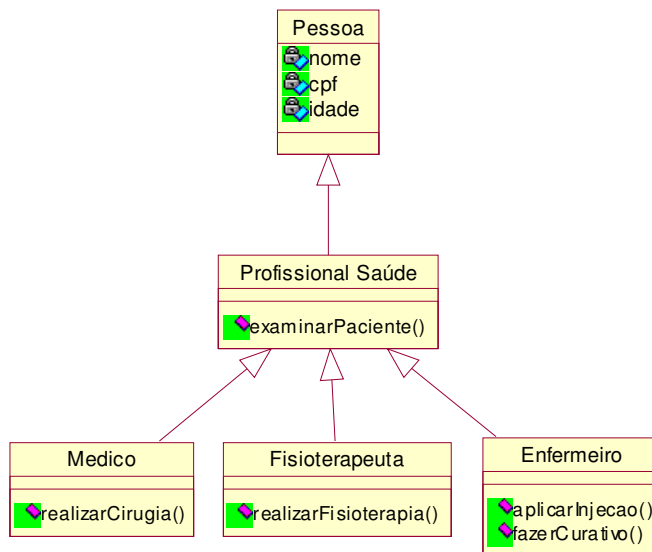
Herança é a capacidade de uma classe definir o seu comportamento e sua estrutura aproveitando definições de outra classe, normalmente conhecida como classe base ou classe pai. Note que na figura abaixo as subclasses herdam tudo o que a classe pai possui e acrescenta as suas características particulares:



Através do mecanismo de herança é possível definirmos classes genéricas que agreguem um conjunto de definições comuns a um grande número de objetos (Generalização), ou ainda, a partir destas especificações genéricas podemos construir novas classes, mais específicas, que acrescentem novas características e comportamentos aos já existentes (Especialização).



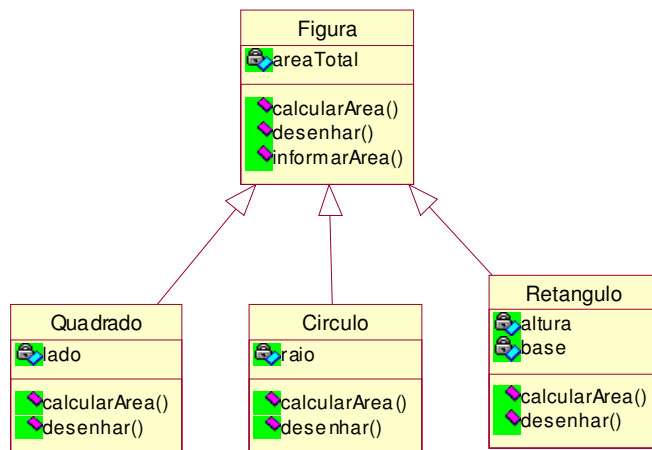
O conceito de especialização e generalização pode acontecer em níveis diferentes como exibido na figura abaixo.



Classes Abstratas

Classes abstratas são classes que não produzem instâncias. Elas agrupam características e comportamentos que serão herdados por outras classes. Fornecem padrões de comportamento que serão implementados nas suas subclasses. Podem ter métodos com implementação definida. No exemplo exibido abaixo a classe Figura é abstrata. Ela contém métodos abstratos (não implementados) e métodos concretos (implementados). Os métodos abstratos são aqueles cuja implementação depende das subclasses de Figura como, por exemplo, calcularArea(). O calculo da área de um circulo é diferente da de um quadrado!

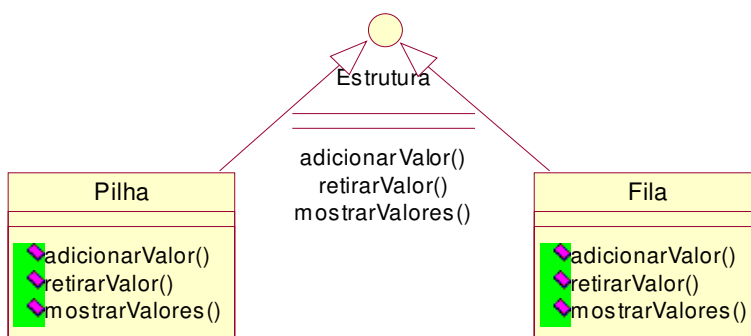
Os métodos abstratos de classes abstratas não podem ter um corpo, ou seja, não tem um algoritmo definido para executar o comportamento. São as classes que herdam da classe abstrata que fornecerão o comportamento a ser implementado.



Interfaces

As interfaces são estritamente modelos de comportamento. Elas não podem ser instanciadas, pois não produzem objetos. A relação existente entre as classes que implementam uma Interface e a Interface é uma relação do tipo "implementa os métodos de". Uma interface não precisa ter significado semântico. Se uma classe é definida por apenas métodos abstratos então é melhor defini-la como uma Interface.

Nenhum método de interface pode ter um corpo, ou seja, não tem um algoritmo definido para executar o comportamento. São as classes que implementarem a Interface que fornecerão o comportamento. Os métodos da Interface Estrutura são obrigatoriamente implementados em Pilha e Fila. Isso significa que Fila e Pilha têm comportamentos semanticamente iguais mas provavelmente com implementações diferentes.



Polimorfismo

É a possibilidade de se solicitar um serviço a um objeto, cuja execução vai depender do tipo de objeto instanciado. As classes Circulo, Retângulo e Quadrado são do tipo Figura. Todas elas possuem um método chamado desenhar() com os mesmo parâmetros de entrada e

saída e um método chamado `informarArea()`. O resultado destes métodos depende do tipo da Figura que receber a mensagem. Todas as classes que implementam as mesmas interfaces produzem comportamentos polimórficos.

O que é JAVA?

Java é uma linguagem de programação orientada a objetos desenvolvida pela Sun Microsystems. Modelada depois de C++, a linguagem Java foi projetada para ser pequena, simples e portátil a todas as plataformas e sistemas operacionais, tanto o código fonte como os binários. Esta portabilidade é obtida pelo fato da linguagem ser interpretada, ou seja, o compilador gera um código independente de máquina chamado byte-code. No momento da execução este byte-code é interpretado por uma máquina virtual instalado na máquina. Para portar Java para uma arquitetura Hardware /S.O. específica, basta instalar a máquina virtual (interpretador). Além de ser integrada à Internet, Java também é uma excelente linguagem para desenvolvimento de aplicações em geral. Dá suporte ao desenvolvimento de software em larga escala.

Criando uma APLICAÇÃO

Para começar, criaremos uma simples aplicação em Java: a clássica "Hello World!", o exemplo que todos os livros de linguagens usam. Como todas as linguagens de programação, o código fonte será criado em um editor de texto ASCII puro. No Unix alguns exemplos são emacs, pico, vi e outros. No Windows, notepad ou dosedit também servem. A seguir, o código da aplicação "Hello World!" (arquivo: HelloWorld.java):

```
class HelloWorld {  
    public static void main (String args[]) {  
        System.out.println("Hello World!");  
    }  
}
```

Para compilar a aplicação, basta digitar o comando:

```
javac HelloWorld.java
```

Este comando vai gerar o arquivo HelloWorld.class, que é o byte-code da aplicação. Para executar o byte-code basta digitar o comando:

```
java HelloWorld
```

Hoje em dia, a maioria dos editores de código Java facilita o processo de edição e compilação. Editores como NetBeans, Eclipse entre outros contemplam funcionalidades como documentação em tempo real auto complete de código além de construção de interfaces gráficas automáticas.

O BÁSICO

Variáveis e tipos de dados

Variáveis são alocações de memória nas quais podemos guardar dados. Elas têm um nome, tipo e valor. Toda vez que necessite usar de uma variável você precisa declará-la e só então poderá atribuir valores a mesma.

Declarando variáveis

As declarações de variáveis consistem de um tipo e um nome de variável: como segue o exemplo:

```
int idade;  
String nome;  
boolean existe;
```

Os nomes de variáveis podem começar com uma letra, um sublinhado "_", ou um cifrão (\$). Elas não podem começar com um número. Depois do primeiro caracter pode-se colocar qualquer letra ou número.

Tipos de variáveis

Toda variável deve possuir um tipo. Os tipos que uma variável pode assumir uma das três "coisas" a seguir:

- 1) Uma das oito primitivas básicas de tipos de dados
- 2) O nome de uma classe ou interface
- 3) Um array

Veremos mais sobre o uso de arrays e classes mais à frente. Os oito tipos de dados básicos são: inteiros, números de ponto-flutuante, caracteres e booleanos (verdadeiro ou falso).

Tipos Inteiros:

Tipo	Tamanho	Alcance
byte	8 bits	-128 até 127
short	16 bits	-32.768 até 32.767
int	32 bits	-2.147.483.648 até 2.147.483.647
long	64 bits	-9223372036854775808 até 9223372036854775807

Existem dois tipos de números de ponto-flutuante: float (32 bits, precisão simples) e double (64 bits, precisão dupla).

Atribuições a variáveis

Após declarada uma variável a atribuição é feita simplesmente usando o operador '=':

```
idade = 18;  
existe = true;
```

Comentários

Java possui três tipos de comentário, o /* e */ como no C e C++. Tudo que estiver entre os dois delimitadores são ignorados:

```
/* Este comentário ficará visível somente no código o compilador ignorará completamente  
este trecho entre os delimitadores  
*/
```

Duas barras (//) também podem ser usadas para se comentar uma linha:

```
int idade; // este comando declara a variável idade
```

E finalmente os comentários podem começar também com `/**` e terminar com `*/`. Este comentário é especial e é usado pelo javadoc e para gerar uma documentação API do código. Para aprender mais sobre o javadoc acesse a home page (<http://www.javasoft.com>).

Caracteres especiais

Caracter	Significado
<code>\n</code>	Nova Linha
<code>\t</code>	Tab
<code>\b</code>	Backspace
<code>\r</code>	Retorno do Carro
<code>\f</code>	"Formfeed" (avança página na impressora)
<code>\\</code>	Barra invertida
<code>\'</code>	Apóstrofe
<code>\"</code>	Aspas
<code>\ddd</code>	Octal
<code>\xdd</code>	Hexadecimal

Operadores Aritméticos

Operador	Significado	Exemplo	Resultado
<code>+</code>	soma	<code>3 + 4</code>	7
<code>-</code>	subtração	<code>5 - 7</code>	-2
<code>*</code>	multiplicação	<code>5 * 5</code>	25
<code>/</code>	divisão	<code>14 / 7</code>	2
<code>%</code>	módulo(mod)	<code>20 % 7</code>	6

Exemplo Aritmético:

```
class ArithmeticTest {
    public static void main ( Strings args[] ) {
        short x = 6;
        int y = 4;
        float a = 12.5f;
        float b = 7f;

        System.out.println ( "x é " + x + ", y é " + y );
        System.out.println ( "x + y = " + (x + y) );
        System.out.println ( "x - y = " + (x - y) );
        System.out.println ( "x / y = " + (x / y) );
        System.out.println ( "x % y = " + ( x % y ) );

        System.out.println ( "a é " + a + ", b é " + b );
        System.out.println ( " a / b = " + ( a / b ) );
    }
}
```

A saída do programa acima é :

```
x é 6, y é 4
x + y = 10
x - y = 2
x / y = 1
x % y = 2
a é 12.5, b é 7
a / b = 1.78571
```

Mais sobre atribuições

Variáveis podem atribuídas em forma de expressões como:

```
int x, y, z;
x = y = z = 0;
```

No exemplo as três variáveis recebem o valor 0;

Operadores de Atribuição

Expressão	Significado
$x += y$	$x = x + y$
$x -= y$	$x = x - y$
$x *= y$	$x = x * y$
$x /= y$	$x = x / y$

Incrementos e decrementos

Como no C e no C++ o Java também possui incrementadores e decrementadores :

```
y = x++;
y = --x;
```

As duas expressões dão resultados diferentes, pois existe uma diferença entre prefixo e sufixo. Quando se usa os operadores ($x++$ ou $x--$), y recebe o valor de x antes de x ser incrementado, e usando o prefixo ($++x$ ou $--x$) acontece o contrario, y recebe o valor incrementado de x.

Comparações

Java possui várias expressões para testar igualdade e magnitude. Todas as expressões retornam um valor booleano (true ou false).

Operadores de comparação

Operador	Significado	Exemplo
$==$	Igual	$x == 3$
$!=$	Diferente (Não igual)	$x != 3$
$<$	Menor que	$x < 3$
$>$	Maior que	$x > 3$
$<=$	Menor ou igual	$x <= 3$
$>=$	Maior ou igual	$x >= 3$

Operadores lógicos

Operador	Significado
&&	Operação lógica E (AND)
	Operação lógica OU (OR)
!	Negação lógica (NOT)
&	Comparação bit-a-bit E (AND)
	Comparação bit-a-bit OU (OR)
^	Comparação bit-a-bit OU-Exclusivo (XOR)
<<	Deslocamento à esquerda
>>	Deslocamento à direita
>>>	Deslocamento à direita com preenchimento de zeros
-	Complemento bit-a-bit
x <<= y	Atribuição com deslocamento a esquerda (x = x << y)
x >>= y	Atribuição com deslocamento a direita (x = x >> y)
x >>>= y	Atribuição com deslocamento a direita e com preenchimento de zeros (x = x >>> y)
x &= y	Atribuição AND (x = x & y)
x = y	Atribuição OR (x = x y)
x ^= y	Atribuição XOR (x = x ^ y)

ARRAYS, LOOPS E CONDICIONAIS

Arrays

Arrays em Java são diferentes do que em outras linguagens. Arrays em Java são objetos que podem ser passados e acoplados a outros objetos.

Arrays podem conter qualquer tipo de elemento valorado (tipos primitivos ou objetos), mas você não pode armazenar diferentes tipos em um simples array. Ou seja, você pode ter um array de inteiros, ou um array de strings, ou um array de array, mas você não pode ter um array que contenha ambos os objetos strings e inteiros.

A restrição acima descrita significa que os arrays implementados em Java são genéricos homogêneos, ou seja, um único array pode armazenar qualquer tipo de objeto com a restrição que todos sejam da mesma classe.

Declarando um Array

```
String difficult[];  
Point hits[];  
int temp[];
```

Outra alternativa de declaração

```
String[] difficult;  
Point[] hits;  
int[] temp;
```

Criando Objetos Arrays

Um dos caminhos é usar o operador new para criar uma nova instância de um array, por exemplo:

```
int[] temps = new int[99];
```

Quando voce cria um objeto array usando o operador new, todos os índices são inicializados para você (0 para arrays numéricos, falso para boolean, '\0' para caracteres, e NULL para objetos). Você também pode criar e inicializar um array ao mesmo tempo.

```
String[] chiles = { "jalapeno", "anaheim", "serrano", "jumbou", "thai"};
```

Cada um dos elementos internos deve ser do mesmo tipo e deve ser também do mesmo tipo que a variável que armazena o array. O exemplo acima cria um array de Strings chamado chiles que contém 5 elementos.

Acessando os Elementos do Array

Uma vez que você tem um array com valores iniciais, você pode testar e mudar os valores em cada índice de cada array.

Os arrays em Java sempre iniciam-se na posição 0 como no C++. Por exemplo:

```
String[] arr= new String[10];  
arr[10]="out";
```

Isto provoca um erro de compilação pois o índice 10 não existe, pois isto está fora das bordas do array.

```
arr[9] = "inside";
```

Esta operação de atribuição é válida e insere na posição 9 do array, a string "inside".

Arrays Multidimensionais

Java não suporta arrays multidimensionais. No entanto, você pode declarar e criar um array de arrays e usá-los como você faria no estilo-C.

```
int coords[][]= new int[12][12];  
coords[0][0] = 1;  
coords[0][1] = 2;
```

Condicionais

O condicional contém a palavra chave `if`, seguido por um teste booleano. Um opcional `else` como palavra chave pode ser executado no caso do teste ser falso, Exemplo:

```
if ( x < y)
    System.out.println(" x e menor do que y");
else
    System.out.println(" y e maior);
```

Nota técnica: A diferença entre o `if` em Java e C ou C++ é que o teste deve retornar um valor booleano (`true` ou `false`).

Bloco

Um bloco é definido por `{ }` e contém um grupo de outros blocos. Quando um novo bloco é criado um novo escopo local é aberto e permite a definição de variáveis locais. As variáveis definidas dentro de um bloco só podem ser vistas internamente a este, e são terminadas ou extintas no final da execução deste `{ }`.

```
void testblock(){
    int x = 10, w=1;

    if (x> w)
    { // inicio do bloco
        int y=50;
        System.out.println("dentro do bloco");
        System.out.println("x:" + x);
        System.out.println("y:" + y);
    } // final do bloco

    System.out.println("w:" + w);
    System.out.println("y:" + y); // erro variável não conhecida
}
```

O operador Condicional

Uma alternativa para o uso do `if` e `else` é um operador ternário condicional. Este operador ternário (`?:`), é chamado assim porque tem três termos como parâmetro.

Exemplo:

```
test ? trueresult : falseresult
int menor = x < y ? x : y ; // A variável menor recebe o valor do menor
entre x e y.
```

O switch

Um comum mecanismo para substituição de `ifs` que pode ser usado para um grupo de testes e ações junto a um simples agrupamento chama-se `switch`.

```

switch (teste){
    case valorum;
        resultum;
        break;

    case valordois;
        resultdois;
        break;

    case valortres:
        resulttres;
        break;

    default: defaultresult;
}

```

O valor é comparado com cada um dos casos relacionados. Se a combinação não for encontrada, o bloco default executado. O default é opcional, então caso este não esteja associado ao comando, o bloco do switch sem executar nada.

Looping For

O loop em Java tem esta sintaxe:

```

for(inicialização; teste; incremento) {
    bloco de comandos;
}

```

Você também pode incluir um comando simples, sendo assim não há necessidade da utilização de chaves. Exemplo:

```

String strArray[] = new String[10];
for ( i=0; i< strArray.length; i++)
    strArray[i]="";

```

Inicializa um array de 10 elementos com "";

Loop While

O while é usado para repetir um comando, ou um conjunto de comando enquanto a condição é verdadeira.

```

while (condição){
    bloco de comandos;
}

```

A condição é uma expressão booleana. Exemplo:

```

int count=0;
while( count < array1.length && array1[count]!=0){
    array2[count]=(float) array1[count++];
}

```

```
}
```

Loop Do

A principal diferença entre o `while` e o `do` é que o teste condicional no caso do `while` é feita antes de se executar o código interno ao loop. Desta forma, o que pode acontecer no `while` é que o loop pode não ser executado se a condição for `false`. Já no loop `do` o corpo do loop é executado pelo menos uma vez, pois o teste de permanência é executado no fim do loop.

```
do{  
    bodyOfLoop;  
} while(condition);
```

Definindo Classes

Para definir uma classe use a palavra chave `class` e o nome da classe. Exemplo:

```
class Minhaclasse{  
    ...  
}
```

Se esta classe é uma subclasse de outra classe, use `extends` para indicar a superclasse (HERANÇA). Exemplo:

```
class Minhaclasse extends SuperClasse{  
    ...  
}
```

Definindo Variáveis de Instância

As variáveis de instância, aparentemente, são declaradas e definidas quase exatamente da mesma forma que as variáveis locais, a principal diferença é que a alocação delas é na definição da classe. Exemplo:

```
class Bike extends Veículo {  
    String tipo;  
    int correia;  
    int pedal;  
    String marcha;  
}
```

Acessando Valores de Variáveis de Instância

Para que seja possível o acesso a variáveis de instância é necessário que se crie a instância da classe (objeto) cujos atributos (variáveis de instância) se quer acessar. Para cada instância vai existir uma cópia do atributo na memória. Ex:

```
Bike meu=new Bike();
```

Obtendo valores de uma variável de instância

```
String cambio = meu.marcha; // Onde meu é uma Bike
```

Escrevendo valores em uma variável de instância

```
meu.marcha = "shimano";
```

```
meu.pedal = 1;
```

Exemplo: acessando variáveis

```
import java.awt.Point; // classe que contém dois atributos (X e Y) do tipo int.
```

```
class TestePonto {  
  
    public static void main (String args[]) {  
        Point meuPonto = new Point(10,10);  
  
        System.out.println("X é " + meuPonto.x);  
        System.out.println("Y é " + meuPonto.y);  
  
        meuPonto.x = 5;  
        meuPonto.y = 15;  
        System.out.println("X é " + meuPonto.x);  
        System.out.println("Y é " + meuPonto.y);  
    }  
}
```

O resultado desse programa será:

```
X é 10  
Y é 10  
X é 5  
Y é 15
```

Constantes

Para declarar uma constante, use a palavra chave **final** antes da declaração da variável e inclua um valor inicial para esta variável. Exemplo:

```
final float pi=4.141592;  
final boolean debug=false;  
final int maxsize = 40000;
```

Variáveis de Classe

As variáveis de classe são boas para a comunicação entre os diferentes objetos da mesma classe, ou para manter travamento de estados globais sobre um conjunto de objetos.

Exemplo:

```
class MembroDaFamilia {  
  
    static String sobrenome; // há uma cópia por classe  
  
    String primeiroNome; // há uma cópia por instância  
    int idade;  
    ...  
}
```

Exemplo de uso:

```
MembroDaFamilia mainha = new MembroDaFamilia(); mainha.sobreNome = "Mendonca";  
MembroDaFamilia eu = new MembroDaFamilia();  
System.out.println(eu.sobrenome);
```

Será impresso na tela **Mendonça** apesar de, no objeto "eu", não modificarmos o atributo sobrenome. Mais à frente nós voltaremos a falar sobre a palavra reservada **static**.

Definição de Métodos

A definição dos métodos têm quatro partes básicas:

- O nome do método;
- O tipo objeto ou tipo primitivo de retorno;
- Uma lista de parâmetros;
- O corpo do método;

A definição básica de um método tem esta aparência:

```
tipoderetorno nomedometodo(tipo1 arg1, tipo2 arg2, ...){  
    ....  
}
```

Exemplo:

```
int[] makeRange(int lower, int upper) { ... }
```

A RangeClass classe:

```
class RangeClass{  
  
    int[] makeRange(int lower, int upper){  
        int arr[] = new int[ (upper - lower) + 1];  
  
        for (int i=0; i<arr.length;i++)  
            arr[i]=lower++;  
        return arr;  
    }  
}
```

```

public static void main(String arg[]){
    int theArray[];
    RangeClass theRange=new RangeClass();
    theArray= theRange.makeRange(1,10);

    System.out.print("The array: [ " );

    for ( int i=0; i < theArray.length; i++)
        System.out.print(theArray[i] + " ");

    System.out.println("]");
}
}

```

A saída do programa é :

The array: [1 2 3 4 5 6 7 8 9 10]

A palavra chave **this**

No corpo de uma definição de método, você pode querer referir-se ao objeto corrente-o objeto que o método foi chamado - para referir-se às variáveis de instância ou para passar o objeto corrente como um argumento para um outro método. Para este tipo de referência, você pode usar a palavra chave **this**.

```

class Pessoa {
    String nome;
    int idade;
    Pessoa ( String nome, int idade ) {
        this.nome = nome;
        this.idade = idade;
    }

    public void imprimeDados () {
        System.out.print ( "Nome: " + this.nome + " Idade: " + this.idade);
    }
}

```

Passando argumentos para Métodos

```

class PassByReference{
    int onetoZero(int arg[]){
        int count=0;

        for(int i=0; i< arg.length; i++){
            if(arg[i]==1){
                count++;
                arg[i]=0;
            }
        }
        return count;
    }
}

```



```

    }
}

public static void main (String arg[])
    int arr[]= { 1,3,4,5,1,1,7};
    PassByReference test = new PassByReference();
    int numOnes;

    System.out.print("Values of the array: [");
    for( int i=0; i < arr.length; i++){
        System.out.print(arr[i] + " ");
    }
    System.out.println("]");

    numOnes= test.onetoZero(arr);
    System.out.println("Number of Ones = " + numOnes);
    System.out.print("New values of the array: [ ");
    for( int i=0; i < arr.length; i++){
        System.out.print(arr[i] + " ");
    }
    System.out.println("]");
}

```

As saídas deste programa:

```

Values of the array: [ 1 3 4 5 1 1 7 ]
Number of Ones = 3
New values of the Array: [ 0 3 4 5 0 0 7]

```

MAIS SOBRE MÉTODOS

Polimorfismo ou Sobrecarga

Os métodos em Java podem ser sobrecarregados, ou seja, podem-se criar métodos com o mesmo nome, mas com diferentes assinaturas (parâmetros) e diferentes definições. Quando se chama um método em um objeto, o Java casa o nome do método, o número de argumentos e o tipo dos argumentos e escolhe qual a definição do método a executar.

Para criar um método sobrecarregado, é necessário criar diferentes definições de métodos na sua classe, todos com o mesmo nome, mas com diferentes parâmetros (número de argumentos ou tipos).

No exemplo a seguir veremos a definição da classe Retangulo, a qual define um retângulo plano. A classe Retangulo têm quatro variáveis para instanciar, as quais definem o canto superior esquerdo e o canto inferior direito do retângulo: x1, y1, x2 e y2.

```

class Retangulo {
    int x1 = 0;
    int y1 = 0;
    int x2 = 0;
    int y2 = 0;
}

```

Quando uma nova instância da classe Retangulo for criada, todas as suas variáveis são inicializadas com 0. Definindo um método construaRetang (): este método recebe quatro inteiros e faz um "resize" do retângulo de acordo com as novas coordenadas e retorna o

objeto retângulo resultante (note que os argumentos possuem o mesmo nome das variáveis instanciáveis, portanto deve-se usar o this para referenciá-las):

```
Retangulo construaRetang ( int x1, int y1, int x2, int y2 ) {  
    this.x1 = x1;  
    this.y1 = y1;  
    this.x2 = x2;  
    this.y2 = y2;  
    return this;  
}
```

Querendo-se definir as dimensões do retângulo de outra forma, por exemplo pode-se usar o objeto Point ao invés de coordenadas individuais. Faremos a sobrecarga do método construaRetang (), passando agora como parâmetro dois objetos Point:

```
Retangulo construaRetang (Point superiorEsquerdo, Point inferiorDireito) {  
    x1 = superiorEsquerdo.x;  
    y1 = superiorEsquerdo.y;  
    x2 = inferiorDireito.x;  
    y2 = inferiorDireito.y;  
    return this;  
}
```

Porém querendo-se definir um retângulo usando somente o canto superior esquerdo e uma largura e altura do retângulo pode-se ainda definir mais um método construaRetang ():

```
Retangulo construaRetang (Point superiorEsquerdo, int largura, int altura) {  
    x1 = superiorEsquerdo.x;  
    y1 = superiorEsquerdo.y;  
    x2 = (x1 + largura);  
    y2 = (y1 + altura);  
    return this;  
}
```

Para finalizar o exemplo mostra-se a seguir um método para imprimir as coordenadas do retângulo e um main para fazer o teste:

```
import java.awt.Point;
```

```
class Retangulo {  
    int x1 = 0;  
    int y1 = 0;  
    int x2 = 0;  
    int y2 = 0;
```

```
Retangulo construaRetang ( int x1, int y1, int x2, int y2 ) {  
    this.x1 = x1;  
    this.y1 = y1;  
    this.x2 = x2;  
    this.y2 = y2;  
    return this;  
}
```

```

Retangulo construaRetang (Point superiorEsquerdo, Point inferiorDireito) {
    x1 = superiorEsquerdo.x;
    y1 = superiorEsquerdo.y;
    x2 = inferiorDireito.x;
    y2 = inferiorDireito.y;
    return this;
}

Retangulo construaRetang (Point superiorEsquerdo, int largura, int altura) {
    x1 = superiorEsquerdo.x;
    y1 = superiorEsquerdo.y;
    x2 = (x1 + largura);
    y2 = (y1 + altura);
    return this;
}

void imprimaRetangulo () {
    System.out.print ( "Retângulo: < " + x1 + ", " + y1 );
    System.out.println ( ", " + x2 + ", " + y2 + ">");
}

public static void main ( String args[] ) {
    Retangulo retang = new Retangulo();

    System.out.println ( "Chamando construaRetang com coordenadas 25, 25, 50, 50 :"
);
    retang.construaRetang ( 25, 25, 50, 50 );
    retang.imprimaRetangulo ();
    System.out.println ( "-----");

    System.out.println ( "Chamando construaRetang com os pontos (10, 10) , (20, 20)
:" );
    retang.construaRetang ( new Point (10,10) , new Point (20, 20) );
    retang.imprimaRetangulo ();
    System.out.println ( "-----");

    System.out.println ( "Chamando construaRetang com os pontos (10, 10) , largura
(50) e altura (50) :" );
    retang.construaRetang ( new Point (10,10) , 50, 50);
    retang.imprimaRetangulo ();
    System.out.println ( "-----");
}
}

```

Métodos Construtores

Um método construtor é um tipo especial de método que determina como um objeto é inicializado quando ele é criado.

Diferente dos métodos normais um método construtor não pode ser chamado diretamente; ao invés disto os métodos construtores são chamados automaticamente pelo Java. No momento em que o objeto é instanciado, ou seja quando se usa new o Java faz três coisas:

- Aloca memória para o objeto
- Inicializa as variáveis daquela instância do objeto
- Chama o método construtor da classe

Construtores Básicos

Os construtores parecem muito com os métodos normais, com duas diferenças básicas:

- Construtores sempre têm o mesmo nome da classe
- Construtores não podem ter tipo de retorno

Exemplo:

```
class Pessoa {
    String nome;
    int idade;

    Pessoa (String n, int i) {
        nome = n;
        idade = i;
    }

    void printPessoa () {
        System.out.print ("Oi meu nome é : "+ nome);
        System.out.println (". Eu tenho : "+idade+ " anos");
    }

    public static void main ( String args[] ) {
        Pessoa p;
        p = new Pessoa ("Maria", 20);

        p.printPessoa();
    }
}
```

Polimorfismo de Construtores

Igual aos métodos normais os construtores também podem ter números variáveis de tipos e parâmetros. Por exemplo os métodos construaRetang() definidos na classe Retangulo seriam excelentes construtores para a mesma classe, pois eles estão justamente instanciando as variáveis. Segue o exemplo abaixo com as devidas alterações :

```
import java.awt.Point;
class Retangulo {
    int x1 = 0;
    int y1 = 0;
    int x2 = 0;
    int y2 = 0;
    Retangulo ( int x1, int y1, int x2, int y2 ) {
        this.x1 = x1;
        this.y1 = y1;
        this.x2 = x2;
        this.y2 = y2;
    }
}
```

```

Retangulo (Point superiorEsquerdo, Point inferiorDireito) {
    x1 = superiorEsquerdo.x;
    y1 = superiorEsquerdo.y;
    x2 = inferiorDireito.x;
    y2 = inferiorDireito.y;
}

Retangulo (Point superiorEsquerdo, int largura, int altura) {
    x1 = superiorEsquerdo.x;
    y1 = superiorEsquerdo.y;
    x2 = (x1 + largura);
    y2 = (y1 + altura);
}

void imprimaRetangulo () {
    System.out.print ( "Retângulo: < " + x1 + ", " + y1 );
    System.out.println ( ", " + x2 + ", " + y2 + ">");
}

public static void main ( String args[] ) {
    Retangulo retang;

    System.out.println ( "Retangulo com coordenadas 25, 25, 50, 50 :" );
    retang = new Retangulo (25, 25, 50, 50 );
    retang.imprimaRetangulo ();
    System.out.println ( "-----");

    System.out.println ( "Retangulo com os pontos (10, 10) , (20, 20) :" );
    retang = new Retangulo ( new Point (10,10) , new Point (20, 20) );
    retang.imprimaRetangulo ();
    System.out.println ( "-----");
    System.out.println ( "Retangulo com os pontos (10, 10) , largura (50) e altura (50)
:" );
    retang = new Retangulo ( new Point (10,10) , 50, 50);
    retang.imprimaRetangulo ();
    System.out.println ( "-----");
}
}

```

Métodos Destrutores

Os métodos destrutores são chamados logo antes do "coletor de lixo" passar e sua memória se liberada. O método destrutor é chamado de `finalize()` a classe `Object` define um método destrutor padrão, que não faz nada. Para criar um método destrutor para suas próprias classes basta sobrepor o método `finalize()` com o seguinte cabeçalho:

```

protected void finalize () {
    ...
}

```

Dentro do método `finalize` você pode colocar tudo que você precisa fazer para a limpeza do seu objeto.

Um pouco mais sobre a palavra reservada **STATIC**.

O texto abaixo foi obtido no site do G.U.J (grupo de usuários Java)

Quando e porquê usar static?

static é uma das palavras-chave do Java, e é também motivo de muita confusão e dúvidas entre o pessoal que está começando. Aliás, mesmo os mais experientes confundem-se às vezes em usá-la.

O método static mais famoso de todos é o main. É através dele que vimos nosso primeiro programa em Java nascer, e é sempre via main que nossos programas criam vida. Por definição da linguagem Java, o método main precisa necessariamente ter acesso public, ser static, não retornar coisa alguma (void) e receber como argumento um array de String (String args[]):

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

Entendendo static

Como regra geral, tenha isso em mente: dentro de métodos static somente é possível poder acessar outros métodos e variáveis que também sejam static. Dentro do método pode-se definir qualquer tipo de variável, static ou não. Caso seja necessário acessar algum método ou membro não-static, é necessário criar uma instância da classe e então chamar o que quiser. Já o contrário é um pouco diferente: dentro de membros não-static, é possível acessar tanto propriedades static quanto as não-static.

O fato de ser preciso primeiramente criar uma instância da classe para só então chamar algum método não-static ou acessar uma variável comum dentro de um método static deve-se ao fato que dentro dele não existe uma referência para o ponteiro this. O ponteiro this é utilizado para referenciar propriedades da classe em que estamos trabalhando. Por exemplo:

```
...
// Variável simples, para guardar o nome
private String nome;

// Algum método comum
public void meuMetodo()
{
    this.nome = "Fulano";
}
```

No exemplo acima, this.nome = "Fulano" diz que estamos atribuindo ao membro da classe chamado nome o valor "Fulano". O uso de this é automático, portando o exemplo acima poderia ser escrito simplesmente da forma:

```
...
// Variável simples, para guardar o nome
private String nome;

// Algum método comum
public void meuMetodo()
{
    nome = "Fulano";
}
```

Note que agora não usamos this, e funciona da mesma maneira.

Se o método meuMetodo fosse static, o código acima não funcionaria, pois como foi dito antes, métodos static não possuem this.

Ao contrário do que o nome soa, static não significa que a variável ou método será estática do ponto de vista que seu valor não pode mudar (**final** é usado para estes casos). A palavra static nos garante que somente haverá uma, e não mais que uma, referência para determinada variável ou método disponível em memória.

Em outras palavras, declarando alguma coisa como static, todas as instâncias da classe irão compartilhar a mesma cópia da variável ou método. Declarar algo como static também permite você acessar as coisas diretamente, ou seja, sem precisar criar uma instância da classe. Existe inclusive um Design Patter baseado no uso de static: Singleton.

Exemplificando

Para entender melhor tudo o que foi dito, nada melhor que alguns exemplos práticos para ver com os próprios olhos o funcionamento. O primeiro exemplo consiste em uma classe com 2 variáveis, uma static e outra não-static. A cada novo objeto criado, incrementados ambas variáveis e imprimimos o resultado na tela. Digite o seguinte código em um arquivo chamado "TesteStatic.java":

```
// TesteStatic.java
class Classe1
{
    // Variavel static
    public static int contador = 0;

    // Variavel nao-static
    public int outroContador = 0;

    public Classe1() {}

    // esse método Precisa ser static porque "contador" é static
    public static void incrementaContador()
    {
        contador++;

        System.out.println("contador agora é "+ contador);
    }

    public void incrementaOutroContador()
    {
        outroContador++;

        System.out.println("outroContador agora é "+ outroContador);
    }
}

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

```

{
    Classe1 c1 = new Classe1();
    c1.incrementaContador();
    c1.incrementaOutroContador();

    Classe1 c2 = new Classe1();
    c2.incrementaContador();
    c2.incrementaOutroContador();

    Classe1 c3 = new Classe1();
    c3.incrementaContador();
    c3.incrementaOutroContador();

    Classe1 c4 = new Classe1();
    c4.incrementaContador();
    c4.incrementaOutroContador();
}
}

```

A saída gerada por este programa será

```

contador agora é 1
outroContador agora é 1
contador agora é 2
outroContador agora é 1
contador agora é 3
outroContador agora é 1
contador agora é 4
outroContador agora é 1

```

Note que a variável "contador", que é static, não teve seu valor zerado a cada novo objeto criado da classe Classe1, mas sim incrementado, enquanto "outroContador", que é uma variável comum, ficou sempre em 1, pois a zeramos o valor no construtor da classe.

Acesso direto

Ao contrário de tipos de dados não-static, qualquer variável ou método static podem ser acessado diretamente, sem necessitar de uma instância da classe criada:

```

// TesteStatic2.java
class Classe2
{
    // Escreve alguma frase na tela
    public static void escreve(String msg)
    {
        System.out.println(msg);
    }

    // Retorna a multiplicação de dois números int
    public static int multiplica(int n1, int n2)
    {
        return (n1 * n2);
    }
}

```



```

// Construtor, apenas para mostrar que
// ele nem chega ser chamado
public Classe2()
{
    System.out.println("Construtor de Classe2");
}

}

public class TesteStatic2
{
    public static void main(String args[])
    {
        Classe2.escreve("Multiplicando 3 vezes 3:");
        int resultado = Classe2.multiplica(3, 3);
        Classe2.escreve("Resultado: "+ resultado);
    }
}

```

Rode este programa e repare no resultado. Veja que o construtor da classe não foi chamado, pois não aparece na tela a string "Construtor de Classe2". Repare também que não criamos instância alguma de Classe2 para chamar seus métodos. Caso os métodos escreve e multiplica não fossem static, seria necessário fazer

```

public class TesteStatic2
{
    public static void main(String args[])
    {
        Classe2 c2 = new Classe2();
        c2.escreve("Multiplicando 3 vezes 3:");
        int resultado = c2.multiplica(3, 3);
        c2.escreve("Resultado: "+ resultado);
    }
}

```

Note que o código acima funciona perfeitamente mesmo com os métodos static. Isso funciona porque apesar de podermos chamar diretamente as coisas quando elas são static, não é obrigatório, podendo perfeitamente ser criada uma instância da classe e então chamar os métodos.

O uso de static depende muito do caso, e conforme você vai pegando mais experiência, irá naturalmente identificar os lugares que precisam - ou que são mais convenientes - ao uso de tal palavra-chave.

Tratamento de Exceções em Java

Projetos Java raramente apresentam uma estratégia consistente e completa para tratamento de exceções. Geralmente, os desenvolvedores adicionam o mecanismo de tratamento de exceções como um improviso ou sem pensar muito sobre ele. Um processo de reengenharia durante o estágio de codificação pode transformar este descuido em um grande problema. Uma estratégia clara e detalhada de tratamento de erros e exceções dá

grande retorno na forma de um código robusto. O mecanismo de tratamento de exceções do Java oferece os seguintes benefícios:

- Separa a parte funcional do código de tratamento de erros através do uso de cláusulas try-catch.
- Permite um caminho claro para a propagação do erro. Se o método chamado encontra uma situação que ele não pode tratar, ele pode lançar uma exceção e deixar o método que o chamou tratar desta exceção.
- Ao responsabilizar o compilador por garantir que situações "excepcionais" têm seu tratamento antecipado, força uma codificação robusta.

A fim de desenvolver uma estratégia consistente para tratamento de exceções, examine essas questões que atormentam continuamente os desenvolvedores Java:

- Quais exceções devo usar?
- Quando eu devo usar exceções?
- Qual a melhor forma de usar exceções?

Quando temos que projetar APIs e aplicações que podem cruzar os limites do sistema ou serem implementadas por parceiros externos, essas questões apenas se intensificam.

Vamos nos aprofundar mais nos vários aspectos das exceções.

Que exceções eu devo usar?

Exceções podem ser de dois tipos:

- Exceções forçadas pelo compilador, ou exceções checadas.
- Exceções de runtime, ou exceções-não checadas.

Exceções forçadas pelo compilador (checadas) são instâncias da classe `Exception` ou de uma de suas subclasses -- excluindo a ramificação da `RuntimeException`. O compilador espera que todas as exceções checadas sejam apropriadamente tratadas. Exceções checadas devem ser declaradas em cláusulas `throws` do método que as lança -- assumindo, é claro, que ela não tenha sido pega dentro deste mesmo método. O método chamador deve cuidar destas exceções ou capturando-as ou declarando-as em sua cláusula `throws`. Assim, a criação de uma exceção checada força o programador a prestar atenção para a possibilidade de ela ser lançada. Um exemplo de uma exceção checada é `java.io.IOException`. Como o nome sugere, ela é lançada sempre que uma operação de entrada/saída é terminada de forma anormal. Examine o código a seguir:

```
try
{
    BufferedReader br = new BufferedReader(new FileReader("arquivo.txt"));
    String line = br.readLine();
}
catch(FileNotFoundException fnfe)
{
    System.out.println("Arquivo não encontrado.");
}
catch(IOException ioe)
{
    System.out.println("Não foi possível ler de arquivo.txt");
}
```

O construtor de `FileReader` lança uma `FileNotFoundException` (uma sub-classe de `IOException`) se o arquivo informado não é encontrado. Do contrário, se o arquivo existe,

mas por alguma razão o método `readLine()` não pode lê-lo, `FileReader` lança uma `IOException`.

Exceções de runtime (não checadas) são instâncias da classe `RuntimeException` ou de uma de suas subclasses. Você não precisa declarar exceções não checadas na cláusula `throws` do método que lança esta exceção. E também o método chamador não precisa ter que tratá-la -- embora ele possa. Exceções não checadas geralmente são lançadas apenas por problemas relacionados ao ambiente da Java Virtual Machine (JVM). Como tal, os programadores não devem lançá-las.

`java.lang.ArithmeticException` é um exemplo de uma exceção não checada lançada quando uma condição aritmética excepcional ocorre. Por exemplo, um inteiro "dividido por zero" lança uma instância desta classe. O código a seguir ilustra como usar uma exceção não checada:

```
public static int divide(int dividend, int divisor) throws
    DivideException
{
    float q;
    try
    {
        q = dividend/divisor;
    }
    catch(ArithmeticException ae)
    {
        throw new FussyDivideException("Can't divide by zero.");
    }
}

public class DivideException extends Exception
{
    public DivideException(String s)
    {
        super(s);
    }
}
```

`divide()` força que o método chamador garanta que ele não tentará dividir por zero. Ele faz isto tratando `ArithmeticException` -- uma exceção não checada (e então lança `DivideException`) uma exceção checada.

Para ajudá-lo a decidir se deve usar uma exceção checada ou não, siga a seguinte regra geral: se a exceção significa uma situação que o método chamador deve saber lidar, então a exceção deve ser checada, do contrário ela pode ser não checada.

Quando devo usar exceções?

A especificação da linguagem Java estabelece que "uma exceção será lançada quando restrições semânticas forem violadas", o que basicamente implica que uma exceção é lançada em situações que são, à princípio, não possíveis ou em violações sérias do comportamento aceitável do sistema.

A fim de esclarecermos bem os tipos de comportamentos que podem ser classificados como "normais" ou excepcionais, dê uma olhada em alguns exemplos de códigos:

Exemplo 1

```

Employee getEmployee()
{
    try
    {
        Employee person = object.searchEmployee("João Ribeiro");
    }
    catch(NoEmployeeFoundException nefe)
    {
        // trata a exceção
    }
}

```

Exemplo 2

```

Employee getEmployee()
{
    Employee person = object.searchEmployee("João Ribeiro");
    if(person == null)
        // trata a situação
}

```

No exemplo 1, se a busca pelo empregado não é bem sucedida, então a exceção `NoEmployeeFoundException` é lançada; enquanto que no exemplo 2, uma simples verificação por `null` faz o trabalho. Desenvolvedores encontram situações similares à anterior no seu trabalho do dia-a-dia; o truque é construir uma estratégia saudável e eficiente.

Assim, seguindo a filosofia geral por trás das exceções, você deveria descartar a possibilidade de que a busca não retornará nada? Quando uma busca não encontra nada, não é mais um caso de processamento normal? Então, a fim de usar exceções sensatamente, escolha a abordagem do exemplo 2 no lugar do exemplo 1.

Um bom exemplo de uma situação excepcional: se de alguma forma a instância do objeto que invoca o método `searchEmployee` for nulo, isto se transforma em uma violação fundamental da semântica do método `getEmployee`.

Qual a melhor forma de usar exceções?

Todos os desenvolvedores Java devem procurar capturar diferentes tipos de exceções e saber o que fazer com elas. Isto fica cada vez mais complicado a medida que o código deve se transformar de mensagens de erro de exceções críticas de nível de sistema para mensagens de nível de aplicação amigáveis para o usuário. Isto é verdadeiro particularmente para código do tipo API, onde você pluga seu código em uma outra aplicação e você não é responsável pela GUI.

Geralmente, existem três abordagens para tratamento de exceções:

- Capturar e tratar todas as exceções.
- Declarar exceções em cláusulas `throws` do método e deixá-las passar adiante.
- Capturar exceções e mapeá-las em uma classe de exceção customizada e relançá-las.

Vamos dar uma olhada em algumas situações para cada uma dessas opções e tentar desenvolver uma possível solução. Exemplo 1

```

Employee getEmployee()
{

```

```

try
{
    Employee person = object.searchEmployee("João Ribeiro");
}
catch(MalformedURLException ume)
{
    // faz algo
}
catch(SQLException sqle)
{
    // faz algo
}
}

```

Em um extremo, você poderia capturar todas as exceções e então encontrar alguma forma de avisar o método chamador que algo está errado. Esta abordagem, como ilustrada no exemplo anterior precisa retornar valores nulos ou outro valor especial para método chamador para sinalizar o erro.

Como uma estratégia de projeto, esta abordagem apresenta significantes desvantagens. Você perde todo o suporte de tempo de compilação, e o método chamador deve se preocupar em testar todos os valores de retorno possíveis. E também, o código normal e o código de tratamento de erro se misturam, o que leva a confusão.

Employee getEmployee() throws MalformedURLException, SQLException

```

{
    Employee person = object.searchEmployee("João Ribeiro");
}

```

Este segundo código mostra outro extremo. O método getEmployee() declara todas as exceções lançadas pelo método que ele chama em sua cláusula throws. Assim, getEmployee(), apesar de ciente das situações de exceção, escolhe não tratá-las e as repassa para o método que o chamou. Resumindo, ele age como um propagador das exceções lançadas pelo método que ele chama. Contudo isto não oferece uma solução viável, já que como toda a responsabilidade pelo processamento dos erros é enviada para a hierarquia acima dele, isso pode apresentar alguns problemas significantes, particularmente em casos onde existem muitos limites do sistema. Suponha, por exemplo, que você possui um sistema de RH que chama o método getEmployee() dentro deste seu sistema. Este seu sistema teria que tratar todas as exceções que o método getEmployee() venha a lançar. Não importando se a exceção fosse uma MalformedURLException ou uma SQLException, mesmo que para o sistema só importasse algo como "Falha na busca". Vamos ver agora um terceiro exemplo: Exemplo 3

Employee getEmployee() throws EmployeeException

```

{
    try
    {
        Employee person = object.searchEmployee("João Ribeiro");
    }
    catch(MalformedURLException ume)
    {
        // faz algo
        throw new EmployeeException("Falha na busca", ume);
    }
}

```

```

catch(SQLException sqle)
{
    // faz algo
    throw new EmployeeException("Falha na busca", sqle);
}
}

```

Este terceiro caso representa o meio do caminho entre os dois extremos apresentados no primeiro e no segundo exemplos. Ele utiliza uma classe de exceção personalizada chamada `EmployeeException`. Esta classe apresenta uma característica especial que entende a exceção real lançada como um argumento e transforma uma mensagem de nível de sistema em outra mais relevante para o nível da aplicação. E ainda você mantém a flexibilidade de saber exatamente o que causou a exceção tendo a exceção original como parte da nova instância do objeto de exceção, que é útil para propósitos de depuração. Esta abordagem fornece uma solução elegante para projetar APIs e aplicações que ultrapassam os limites de um sistema.

O código a seguir mostra a classe `EmployeeException`, que usamos como nossa exceção personalizada no terceiro exemplo. Ela toma dois argumentos. Uma é uma mensagem, que pode ser exibida em um fluxo de erro; a outra é a exceção real, que causou a exceção a ser lançada. Este código mostra como empacotar outra informação dentro de uma exceção personalizada. A vantagem deste empacotamento é que, se a chamada de método realmente quer saber a verdadeira causa da `EmployeeException`, tudo o que ela deve fazer é chamar `getHiddenException()`. Isto permite ao método chamador decidir se quer lidar com exceções específicas ou parar em `EmployeeException` mesmo.

```

public class EmployeeException extends Exception
{
    private Exception hiddenException_;

    public EmployeeException(String error, Exception excp)
    {
        super(error);
        hiddenException_ = excp;
    }

    public Exception getHiddenException()
    {
        return(hiddenException_);
    }
}

```

Conclusão

É importante ter uma estratégia muito bem pensada e consistente para tratamento de exceções para garantir a eficiência e a boa prática de programação. O tratamento de exceções deve ser considerado não como um melhoramento mas como parte integrante do processo de desenvolvimento. O poder das exceções fornecem uma base sobre a qual podemos desenvolver aplicações que são robustas e confiáveis.

Resumo sobre exceções

1) O que é uma exceção?

Um dos maiores desafios para os programadores é a maneira de manipular erros durante a execução de forma eficiente e elegante. Java trata isso por um rigoroso esquema de tratamento de exceção. Um tratador de exceções é chamado com frequência de bloco try-catch, devida a sua estrutura.

No bloco try o código passa se não houver nenhum problema, porém se ocorrer um erro, o Java gera um objeto capaz de indicar o problema. A tarefa do bloco catch é interceptar qualquer objeto de exceção emitido do interior do bloco try.

Exemplo:

```
int z = 0;
try{
    z=x/y;
    System.out.println( "Z = " + Z);
} catch ( Exception e) {
    System.out.println( "Erro");
}
finally {
    System.out.println( "Fará sempre");
}
```

2) Emissão de exceções de forma explícita - throws

Os métodos também podem emitir exceções de forma explícita. Um método pode declarar que é capaz de emitir uma exceção, usando para isso a cláusula throws.

Exemplo das próprias classes do Java (JDK):

Na classe Integer existe um método chamado parseInt (String s)
public static int parseInt (String s) throws NumberFormatException
throw significa que o método pode emitir um certo tipo de exceção.

Exemplo:

```
String s="100", s2="40";
try {
    System.out.println("O primeiro numero e : " + Integer.parseInt(s);
    System.out.println("O segundo numero e : " + Integer.parseInt(s2);
}
catch ( NumberFormatException e ) {
    System.out.println(" Não e possivel converter String em numero!!");
}
```

3) Exceções mais comuns em java

- ArithmeticException, NullPointerException
- ClassCastException, NegativeArraySizeException
- ArrayIndexOutOfBoundsException, EOFException

Uma exceção em Java é um objeto que é instância de uma subclasse de java.lang.Throwable

- java.lang.Error e suas subclasses são erros considerados fatais e não devem ser capturados pelo programa
- java.lang.Exception e suas subclasses são erros considerados recuperáveis e podem ser tratados pelo programa

4) Levantando Exceções

Levanta uma exceção do tipo especificado que será tratada em algum catch. Se a exceção não for tratada, o programa será interrompido com uma mensagem de erro.

Comando:

```
throw new <tipo_da_exceção>
```

Ex:

```
if (indiceVetor > tamanho-1){  
    throw new ArrayIndexOutOfBoundsException();  
}
```

5) Criando suas próprias Exceções

Define uma subclasse de Exception e lança a exceção quando preciso.

Ex:

```
class MinhaExcecao extends Exception{  
    // código da excecao  
}
```

no programa

```
if ( situacaoExcecao) {  
    throw new MinhaExcecao( );  
}
```

O pacote java.util

Aqui veremos brevemente as classes principais deste pacote. É importante para o programador conhecê-las para poder fazer uma escolha apropriada da classe/interface que apresente um comportamento que se adeque às suas necessidades. A partir das novas certificações do Java 2 o conhecimento de classes de coleções é constantemente avaliado. As questões são bem básicas, e requerem o conhecimento de onde e como você pode usá-las, em vez de um conhecimento detalhado dos campos e métodos.

As coleções antigas

A API do Java 2 inclui novas interfaces e classes para aumentar a quantidade de coleções disponíveis. Versões anteriores de Java incluíam:

- vector
- hashtable
- array
- BitSet

Dessas, apenas array era cobrada na certificação 1.1. Uma das notáveis ausências do Java 1.1 era o suporte para ordenação, uma necessidade muito comum em qualquer situação de programação.

As novas coleções

Na raiz da API Collection está a interface Collection. Ela nos fornece uma série de métodos comuns que todas as classes de coleção devem ter. Você provavelmente nunca terá que criar sua própria classe que implementa Collection já que o Java fornece uma série de sub-interfaces e classes que usam a interface Collection.

A API do Java 2 inclui as seguintes novas interfaces de coleções:

- Sets
- Maps

Classes que implementam a interface Collection armazenam objetos como elementos que não são tipos primitivos. Esta abordagem tem uma desvantagem, pois criar objetos gera um overhead de performance e os elementos os elementos devem ser reconvertidos de Object para o tipo apropriado antes de serem usados. Isto também significa que as coleções não checam se os elementos são todos do mesmo tipo, já que um objeto pode ser quase qualquer coisa.

Set

Um Set (Conjunto) é uma interface de coleção que não pode conter elementos duplicados. Dessa forma, se assemelha com o conceito de um conjunto de registros retornado de uma tabela de um banco de dados relacional. A parte mais interessante da interface Set é o método de adição:

```
add(Object o);
```

Qualquer objeto passado para o método add deve implementar o método equals de forma que o valor possa ser comparado com objetos existentes na classe. Se o conjunto já contém este objeto a chamada ao add deixa o conjunto inalterado e retorna falso. A idéia de retornar falso quando se tenta adicionar um elemento parece mais com a abordagem usada em C/C++ do que Java. Muitos métodos similares de Java lançariam uma Exception neste tipo de situação.

List

Um List (lista) é uma interface de coleção ordenada que pode conter duplicatas.

Alguns métodos importantes são:

- add
- remove
- clear

A documentação do JDK trás um exemplo do uso de List para gerenciar um controle de lista de uma GUI real contendo uma lista de nomes de Planetas.

Map

Map é uma interface, classes que a implementam não podem conter chaves duplicadas, e é similar a uma hashtable.

Por que usar Collections em vez de arrays?

A grande vantagem de coleções sobre arrays é que as coleções podem crescer em tamanho, você não tem que atribuir o tamanho no momento da criação. A desvantagem de coleções é que elas armazenam apenas objetos e não tipos primitivos e isto trás um inevitável overhead de performance. Arrays não suportam ordenação diretamente, mas isto pode ser superado usando-se métodos estáticos de Collections. Aqui está um exemplo:

```
import java.util.*;
public class Sort{
    public static void main(String argv[]){
        Sort s = new Sort();
    }
    Sort(){
        String s[] = new String[4];
        s[0]="z";
        s[1]="b";
        s[2]="c";
        s[3]="a";
        Arrays.sort(s);
        for(int i=0;i< s.length;i++)
            System.out.println(s[i]);
    }
}
```

As coleções Set e Map garantem a unicidade, List não garante a unicidade de seus itens mas os ordena.

Utilizando Vectors

O exemplo a seguir ilustra como você pode adicionar objetos de diferentes classes a um Vector. Isto contrasta com arrays onde cada elemento deve ser do mesmo tipo. O código então navega através de cada objeto imprimindo na saída padrão. Ele acessa implicitamente o método toString de cada objeto.

```
import java.awt.*;
import java.util.*;
public class Vec{
    public static void main(String argv[]){
        Vec v = new Vec();
        v.amethod();
    }//Fim do main

    public void amethod(){
        Vector mv = new Vector();
        //Note como um vetor pode armazenar objetos
        //de diferentes tipos
        mv.addElement("Hello");
        mv.addElement(Color.red);
        mv.addElement(new Integer(99));
        //Isto causaria um erro
        //Já que um vetor não armazenará primitivos
        //mv.addElement(99)
        //Caminha por todos os elementos do vetor
        for(int i=0; i< mv.size(); i++){
```

```

        System.out.println(mv.elementAt(i));
    }
} //Fim do método
}

```

Antes do Java 2 a classe Vector era a principal forma de criar uma estrutura de dados redimensionável. Elementos podem ser removidos da classe Vector com o método remove.

Usando Hashtables

Hashtables são um pouco parecidos com o conceito do Visual Basic de uma Collection usada com uma chave. Ele age como um Vector, exceto que em vez de referenciar os elementos pelo número, você os referencia por uma chave. A parte hash do nome refere-se ao termo matemático que trata da criação dos índices. Um hashtable pode oferecer buscas mais rápidas do que um Vector.

BitSet

Um BitSet como o nome já diz, armazena uma sequência de Bits. Não se confunda com a parte "set" do nome. Ele não é um conjunto no sentido matemático ou de banco de dados, nem está relacionado com os Sets disponíveis no Java 2. É mais apropriado pensar nele como um vetor de bits. Um BitSet pode ser útil para uma armazenagem eficiente de bits onde os bits são utilizados para representar valores verdadeiro/falso. A alternativa de se usar algum tipo de coleção contendo valores Boolean pode ser menos eficiente.

De acordo com Bruce Eckel em "Thinking in Java"

"É eficiente apenas do ponto de vista de tamanho; se você está procurando por eficiência de acesso, é levemente mais lento do que usar um array de algum tipo nativo."

O BitSet é uma classe que deve ser mais útil para propósitos de criptografia ou processamento de imagens.

Manipulação de arquivos e "Serilização" de Objetos

Arquivos

- Outro dispositivo de entrada e saída de vital importância é disco manipulado através do conceito de arquivo.
- Um arquivo é uma abstração utilizada para uniformizar a interação entre o ambiente de execução e os dispositivos externos.
- A interação de um programa com um dispositivo através de arquivos passa por três etapas:
 - abertura ou criação de um arquivo
 - transferência de dados
 - fechamento do arquivo
- Em java a classe File permite representar arquivos nesse nível de abstração.
Class File

- Usada para representar o sistema de arquivos. É apenas uma abstração: a existência de um objeto File não significa a existência de um arquivo ou diretório.

- Contém métodos para testar a existência de arquivos, para definir permissões (nos S.Os onde for aplicável), para apagar arquivos, criar diretórios, listar o conteúdo de diretórios, etc.

Classe File

Alguns métodos

```
public String getParent();   retorna o diretório (objeto File) pai
public list();              retorna lista de arquivos contidos no diretório
public boolean isFile();    retorna se é um arquivo
public boolean isDirectory(); retorna se é um diretório
public boolean delete();    tenta pagar o diretório ou arquivo
public long length();       retorna o tamanho do arquivo em bytes
public boolean mkdir();     cria um diretório com o nome do arquivo
public String getAbsolutePath(); retorna o path
```

Exemplo

```
File diretorio = new File("c:\\novo");
diretorio.mkdir(); // cria, se possível
File subdir1 = new File( diretorio, "subdir1");
subdir1. mkdir();
File subdir2 = new File( diretorio, "subdir2");
subdir2. mkdir();
File arquivo = new File( diretorio, "arquivoVazio.txt");
FileWriter f = new FileWriter(arquivo);
f.close();
String[] arquivos = diretorio. list();
for (int i =0;i<arquivos.length; i++) {
    File filho = new File( diretorio, arquivos[ i]);
    System. out. println(filho.getAbsolutePath());
}
}
```

- Há várias formas diferentes de ler e escrever dados: Seqüencialmente, aleatoriamente, como bytes, como caracteres, linha por linha e palavra por palavra.
- APIs Java para I/O oferecem objetos que abstraem fontes e destinos (chamados de nós), fluxos de bytes e caracteres
- Existem genericamente dois grupos:
 - Entrada e Saída de bytes: InputStream e OutputStream;
 - Entrada e Saída de chars: Reader e Writer;

Os arquivos são abertos criando-se objetos destas classes de fluxo que herdaram de InputStream, OutputStream, Reader, Writer.

Classes FileInputStream, FileOutputStream, FileReader, FileWriter, InputStreamReader

As funcionalidades de transferência seqüencial de dados a partir "de", ou "para", um arquivo não é suportada pela classe File.

As classes acima oferecem pelo menos um construtor que recebe como argumento um objeto da classe File e implementam os métodos básicos de transferência de dados. InputStreamReader é um filtro que converte bytes em chars.

Exemplo: Escrita seqüencial de Arquivo - FileOutputStream

```
File diretorio = new File("c:\\tmp");
diretorio.mkdir();
File arquivo = new File( diretorio, "lixo.txt");
FileOutputStream out = new FileOutputStream(arquivo);
out.write( new byte[]{ 'l', 'i', 'x', 'o' } );
File subdir = new File( diretorio, "subdir");
subdir.mkdir();
String[] arquivos = diretorio.list();
for (int i =0;i<arquivos.length; i++) {
    File filho = new File( diretorio, arquivos[ i]);
    System.out.println(filho.getAbsolutePath());
}
if (arquivo.exists()) {
    arquivo.delete(); //O bloco de código acima
}
out.close();
```

Exemplo: Escrita sequencial de Arquivo - FileOutputStream

```
System.out.print("Digite o texto");
FileOutputStream f0 = new FileOutputStream("c:\\Saida0.txt");
byte a = (byte)System.in.read();
while(a!='\n'){
    f0.write(a);
    a=(byte)System.in.read();
}
```

Exemplo : Leitura sequencial de Arquivo - FileInputStream

```
File arquivo = new File("c:\\listaAlunos.txt");
FileInputStream in = new FileInputStream(arquivo);
InputStreamReader conversor = new InputStreamReader(in);
BufferedReader bf = new BufferedReader(conversor);
boolean continua=true; String linha;
while(continua){
    linha = bf.readLine();
    if (linha==null){
        continua=false;
    }else{
        System.out.println(linha);
    }
}
```

```

}
bf.close();
in.close();

```

Exemplo: Escrita sequencial de Arquivo - FileReader e
Leitura sequencial de Arquivo - FileWriter

A maneira mais eficiente de ler um arquivo de texto é usar FileReader com um BufferedReader. Para gravar, use um FileWriter com um PrintWriter.

Exemplo: Leitura sequencial de Arquivo - FileReader

```

FileReader f = new FileReader("c:\\arq.txt");
BufferedReader in =new BufferedReader(f);
String linha =in.readLine();
while(linha !=null ){
    System.out.println(linha);
    linha =in.readLine();
}
in.close();

```

Exemplo: Gravação sequencial de Arquivo - FileWriter

```

InputStreamReader conversor = new InputStreamReader(System.in);
BufferedReader bf = new BufferedReader(conversor);
boolean continua=true; String linha;
FileWriter f = new FileWriter("c:\\arq.txt");
PrintWriter out =new PrintWriter(f);
System.out.println("Digite");
while(continua){
    linha = bf.readLine();
    if (linha.equals("fim")){
        continua=false;
    }else{
        out.println(linha);
    }
}
bf.close();
out.close();

```

outro exemplo

```

FileWriter f = new FileWriter("c:\\arq.txt");
PrintWriter out =new PrintWriter(f);
out.println("Ana");
out.println("Alice");
out.println("Lucio");
out.close();

```

Class RandomAccessFile

- Construindo uma instância do RandomAccessFile, você pode procurar por qualquer posição desejada dentro de um arquivo , e então ler ou escrever um montante de dados desejados.
- Esta classe oferece acesso aleatório através do uso de um ponteiro.
- Construindo uma instância do RandomAccessFile no modo 'r' , se o arquivo não existir dispara uma exceção "FileNotFoundException".
- Construindo uma instância do RandomAccessFile no modo 'rw', se o arquivo não existir um arquivo de tamanho zero é criado.
- Construindo uma instância do RandomAccessFile , você pode procurar por qualquer posição desejada dentro de um arquivo, e então ler ou escrever um montante desejado de dados.

Class RandomAccessFile - leitura

```
File fileName = new File("c:\\Alunos.txt");
RandomAccessFile obj = new RandomAccessFile(fileName , "rw");
int i=0; String result;
while(i<obj.length()){
    result=obj.readLine();
    if(result==null){
        break ;
    }
    System.out.println(result);

    i++;
}
```

Class RandomAccessFile - gravação

```
InputStreamReader conversor = new InputStreamReader(System.in);
BufferedReader bf = new BufferedReader(conversor);
File fileName = new File("c:\\Alunos2Chamada.txt");
RandomAccessFile obj = new RandomAccessFile(fileName , "rw");
int i=0; String result;
boolean continua=true; String linha;
while(continua){
    linha = bf.readLine();
    if (linha.equals("fim")){
        continua=false;
    }else{
        obj.writeBytes(linha+"\n");
    }
}
obj.close();
```

Serialização de objetos - interface Serializable **Class ObjectOutputStream e ObjectInputStream**

- Java permite a gravação direta de objetos em disco ou seu envio através da rede. Neste caso o objeto deve declarar implementar java.io.Serializable.

- Um objeto que implementa a interface `Serializable` poderá, ser gravado em qualquer stream usando o método `writeObject()` de `ObjectOutputStream` e poderá ser recuperado de qualquer stream usando o método `readObject()` de `ObjectInputStream`.

Exemplo gravação e leitura de objetos serializados

```
class Aluno implements java.io.Serializable{
    private String nome;
    public Aluno(String nome){
        this.nome=nome;
    }
    public String getNome(){
        return nome;
    }
}
```

// trecho de programa

```
Aluno a = new Aluno("Mario");
Aluno b = new Aluno("Alice");
File arquivo = new File("c:\\\\GuardaObjetos.txt");
FileOutputStream fOut = new FileOutputStream(arquivo);
ObjectOutputStream objOut = new ObjectOutputStream(fOut);
objOut.writeObject(a);
objOut.writeObject(b);
```

```
FileInputStream fIn = new FileInputStream(arquivo);
ObjectInputStream objIn = new ObjectInputStream(fIn);
Aluno primeiro=(Aluno)objIn.readObject();
System.out.println(primeiro.getNome());
Aluno segundo=(Aluno)objIn.readObject();
System.out.println(segundo.getNome());
```

Classes do Pacote swing

JComboBox

- `int getItemCount()` - Retorna a quantidade de itens no `JComboBox`.
- `void addItem(String)` - Adiciona uma `String` ao final do `JComboBox`
- `object getItemAt(int)` - Retorna o Objeto na posição indicada por `int`. Para transformar esse objeto em uma `String` use casting. Ex: `(String)Obj.getItemAt(3)`
- `void insertItemAt(String, int)` - Insere a `String` na posição `int`.
- `object getSelectedItem()` - Retorna o objeto selecionado. Para transformar esse objeto em uma `String` use casting. Ex: `(String)Obj.getSelectedItem()`

JLabel

- `void setText(String)` - Define a string de informação do `Label`
- `void setLabeFor(Jcomponent)` - Define o componente associado ao rótulo

JTextField

- void setText(String) - Define a string do campo de texto
- String getText() - Lê a string presente no campo de texto
- void requestFocus() - Coloca o "Foco" no Objeto
- void addFocusListener(Objeto da Classe que Implementa a Interface Correspondente)
Associa ao Evento

JCheckBox

- boolean isSelected() - Determina se o Check Box está Selecionado ou Não
- void setSelected(boolean) - Marca ou Desmarca o Check Box.
- void addActionListener(Objeto da Classe que Implementa a Interface Correspondente)
Associa ao Evento

JRadioButton

- boolean isSelected() - Determina se o JRadioButton está Selecionado ou Não
- void setSelected(boolean) - Marca o JRadioButton.
- void addActionListener(Objeto da Classe que Implementa a Interface Correspondente)
Associa ao Evento

ButtonGroup

- void add(JComponent) - adiciona o componente ao grupo de botões

JButton

- void addActionListener(Objeto da Classe que Implementa a Interface Correspondente)
Associa ao Evento

JTextArea

- void setText(String) - Define a string da área de texto
- String getText() - Retorna todas as linhas de um TextArea como uma String só. Junta as Linhas, o começo de uma nova linha está sempre no fim da linha anterior.
- void append(String s) - Adiciona a String s ao final do TextArea

JOptionPane

- static JOptionPane.showMessageDialog(null, String, String, JOptionPane.ERROR_MESSAGE)

TREINAMENTO DE MODELAGEM

1. Faça o diagrama de classe seguindo a especificação dos enunciados abaixo:

a.) Modele um sistema de registro de alunos em disciplinas de uma Faculdade. As disciplinas devem ser criadas e os alunos devem ser matriculados, se houver vaga na disciplina (cada disciplina pode ter até 40 alunos matriculados). Entre outras coisas, o sistema deve ser capaz de informar:

- i. quantos alunos existem em cada disciplina
- ii. a disciplina que tem um maior número de alunos
- iii. o nome e a matrícula dos alunos de cada disciplina
- iv. o semestre de entrada de cada aluno da disciplina
- v. se um aluno está matriculado em uma disciplina
- vi. o nome e código de todas as disciplinas da Faculdade

b.) Modele um sistema para ser usado nas disputas do Circuito de Tênis da Empresa LTII. O circuito é composto por partidas de Tênis e cada jogador deve ser inscrito para participar no Torneio. O resultado de cada partida deve ser registrado no sistema. O ganhador de partidas da 1º fase recebe 10 pontos e o ganhador de partidas da 2º fase recebe 15 pontos. Cada jogador pode participar em até 6 partidas. Entre outras coisas, o sistema deve ser capaz de :

- i. Informar o ganhador do circuito (entende-se como aquele que teve um maior número de pontos)
- ii. Informar o total de pontos de cada jogador
- iii. Informar o ganhador de uma partida
- iv. Informar o jogador que esta na frente, considerando somente as partidas da 1º fase do torneio.

c.) Modele um sistema de atendimento ao cliente em uma Loja. A loja vende os seguintes tipos de produtos: Chocolate, Refrigerante, Biscoito. A loja deve ser capaz de:

- i. cadastrar produtos para venda,
- ii. mostrar os produtos disponíveis,
- iii. retornar o preço de um dos produtos cadastrados
- iv. vender os produtos. (Todos os produtos do tipo refrigerante têm um desconto de 10%)

d.) Modele um sistema de locadora. A locadora deve ser capaz de cadastrar clientes, filmes e locações. Entre outras coisas, com o sistema de locadora deve ser possível:

- i. Consultar se um filme está disponível
- ii. Listar filmes de um dos gêneros (Comédia, Drama, Suspense, Aventura)
- iii. Efetuar devoluções de filmes

e.) Modele um sistema de livraria. (Semelhante ao anterior)

f.) Modele um sistema de pagamento de empregados da Empresa LTII. A empresa tem 10 empregados e deve ser capaz de.

- i. Cadastrar os seus empregados
- ii. Informar o resumo do cartão de ponto do empregado (quantas faltas, quantas horas extras no mês)

iii. Informar o valor líquido do salário do empregado no mês, considerando o seu resumo do cartão de ponto, a quantidade de dependentes (a empresa paga 5% a mais por dependente - até o limite de 3 dependentes) e faixa de imposto de renda e INSS. Considere que o valor descontado por dia de trabalho é calculado sobre o total de 30 dias de trabalho no mês e que o valor da hora extra é 1.5 da hora normal do funcionário (tomando em consideração 30x8 horas mensais).

iv. Informar o valor total da folha do mês

g.) Refaça a modelagem anterior para atender a uma empresa que tenha dois tipos de funcionário: o funcionário administrativo e o funcionário professor. Este último tem um total de 12 dias trabalhados por mês e o valor da hora extra é de 2.5 da hora normal (levando em consideração 12 x 8 horas mensais).

h.) Modele um sistema para ser usado em um Banco. Este Banco tem contas poupança e conta corrente. Entre outras atividades, o Banco deve ser capaz de :

i. Cadastrar clientes e cadastrar as contas

ii. Executar uma operação de retirada ou depósito sobre uma conta cadastrada

iii. Mostrar todas operações sobre uma conta

iv. Informar saldo de uma conta

Exercícios

Resolva os exercícios abaixo:

LabI: Um programa “Simples”

Nesse exercício você deve rodar um programa em Java que imprima na tela a frase “oi Mamãe?”. O nome do arquivo fonte deve ser SimpleProgram.java.

Etapas

1. Defina uma classe com o nome de SimpleProgram e nela o método main. No método main imprima a frase.
2. Compile o arquivo fonte SimpleProgram.java.
3. Execute SimpleProgram.

Demonstração

O que deve aparecer na tela após a execução deve ser similar a isso:
Oi Mamãe?.

LabII: Usando “For Loops” em Java

Para esse exercício você vai usar um loop(For) para imprimir algumas strings e outro para simular a multiplicação de (6*3). O nome do arquivo fonte deve ser ForLoop.java.

Etapas

1. Faça um “For” que imprima "oi" 5 vezes.
2. Defina uma variável “soma” com valor inicial 0.
3. Use dois Loops para incrementar o valor da soma e atingir o resultado = 18 (que é 6*3).
4. Imprima no final o valor de “soma”.

Demonstração

O que deve aparecer na tela após a execução deve ser similar a isso:

```
oi  
oi  
oi  
oi  
oi  
oi  
18
```

LabIII – A : Usando Arrays em Java

Para esse exercício você vai criar e inicializar um vetor de inteiros e um de objetos String.

Etapas

1. Declare um vetor de inteiros de tamanho 5.
2. Use um loop while para preencher o vetor com valores acima de 10.
3. Use um For para imprimir os valores do vetor.
4. Declare um vetor de strings inicializado com Zé, João, e Tonho.

5. Use um For para imprimir os valores do vetor.
6. Mude o valor do primeiro elemento do vetor para Maria.
7. Use um For para imprimir os valores do vetor.

Demonstração

O que deve aparecer na tela após a execução deve ser similar a isso:

```
10
11
12
13
14
Zé
João
Tonho
Maria
João
Tonho
```

LabIII - B: Usando Arrays em Java

Construa um tipo abstrato de dados que possibilite a manipulação de vetores. O tipo deve ter comportamentos que permitam encher e ordenar um vetor de inteiros imprimindo o vetor ordenado ao final. O tamanho do vetor é 10. O vetor de inteiros deve ser ordenado usando operador ternário.

Demonstração

Simule esse programa ordenando o vetor[5|8|32|67|99|15|12|9|1|99];

LabIII - C: Usando Arrays em Java

Modifique a questão anterior de que maneira que o tipo permita também empilhar e desempilhar objetos Strings imprimindo a pilha a cada operação. O tamanho da pilha é 10. Se a pilha estiver cheia ou vazia deve ser lançado um aviso.

Demonstração

Simule esse programa empilhando as Strings "Ola", "Tudo bem", "Como vai" e chamando o método desempilha 4 vezes

LabIV: Usando If em Java

Para esse exercício você vai definir variáveis para armazenar temperaturas e imprimir, quando ou não, o clima está quente, frio, normal, ou extrema. A temperatura atual deve ser passada pela linha de comando.

Etapas

1. Defina as variáveis quente=40, frio=10, e atual que recebe o valor passado. Imprima o valor de atual.
2. Use a estrutura "IF" para testar se a temperatura atual é igual a frio. Se verdadeiro, imprima "FRIO." Senão, teste para ver se a temperatura atual é igual a quente. Se for, imprima "Quente".
3. Use a estrutura "IF" para testar se a temperatura atual está entre frio e quente. Se verdadeiro, imprima "Normal". Senão, imprima "Temperatura Extrema".
4. Repita o LabIV usando o "Case" no lugar do if.(opcional)

Demonstração

O que deve aparecer na tela após a execução deve ser similar a isso:
Atual tem valor 30
Normal

LabV: Definindo e Usando Métodos em Java

Para esse exercício, você vai definir e usar alguns métodos em Java.

Etapas

1. Defina um método chamado "Welcome" em uma classe MetodoClass, que não tem argumentos(parâmetros) nem valor de retorno. O método deve apenas imprimir "Seja Bem Vindo!!!."
2. Faça o método main da MetodoClass chamar o Welcome ().
3. Defina outro método chamado addTwo que pega um valor inteiro e soma 2 a ele, retornando o resultado.
4. No método main da MetodoClass, defina uma variável local inteira com valor 3 e em seguida chame addTwo(i) passando ela como parâmetro. Imprima o valor

retornado pelo método. Repita o passo anterior mudando o valor da variável para 19. Imprima o resultado

Demonstração

O que deve aparecer na tela após a execução deve ser similar a isso:

```
Seja bem-vindo
addTwo(3) é 5
addTwo(19) é 21
```

LabVI: A Classe MusicStore

Para esse exercício você vai implementar a primeira versão de um tipo abstrato de dados definido pelo programador de nome MusicStore. Essa Classe será usada exercícios subsequentes.

Etapas

1. Monte e implemente o MusicStore como uma classe pública com um método público de nome displayHoursOfOperation. Esse método imprime na tela o período diário de funcionamento de uma loja de música(discos).
2. Monte e implemente o TestMusicStore como uma classe pública com o método main que execute as seguintes tarefas:
 - Criar uma instância do MusicStore
 - Invocar o método displayHoursOfOperation para período diário de funcionamento
3. Compile os fontes de MusicStore e TestMusicStore.
4. Execute TestMusicStore.

Demonstração

O que deve aparecer na tela após a execução deve ser similar a isso:

```
Período:
Diariamente das 9:00 - 21:00
```

LabVII: MusicStore com um Dono

Para esse exercício você adiciona uma variável de instância chamada "owner" e o método setOwner a MusicStore.

Etapas

1. Adicione a variável "owner" a MusicStore. O tipo deve ser String inicializado com "sem dono."
2. Adicione o método setOwner to MusicStore. Esse método deve modificar o valor da variável "owner".
3. Modifique TestMusicStore para mudar o nome do dono da loja.
4. Compile os fontes de MusicStore e TestMusicStore.
5. Execute TestMusicStore.

Demonstração

O que deve aparecer na tela após a execução deve ser similar a isso:

Período:

Diariamente das: 9:00 - 21:00

Roberto, Proprietário

LabVIII: MusicStore - Aberta ou Fechada?

Para esse exercício você adiciona algumas variáveis e métodos para manipular as condições de aberta ou fechada da MusicStore.

Etapas

1. Crie as variáveis openTime e closeTime para a MusicStore. O tipo de dados deve ser inteiro com valores 9 e 21 respectivamente.
2. Crie 4 métodos de acesso para as variáveis definidas: setOpen, getOpen, setClose, e getClose.
3. Crie um método de nome isOpen que retorna o valor "booleano" indicando se a loja está aberta ou fechada no momento. O método deve comparar as variáveis openTime e closeTime com o valor da hora do sistema. Você usará o método getHourInt descrito abaixo para obter o valor da hora do sistema.
4.

```
int getHourInt() {  
    Calendar c = Calendar.getInstance();  
    return c.get(Calendar.HOUR_OF_DAY);  
}
```


5. Crie por conveniência o método `getOpenClosedMessage`. Ele deve retornar uma mensagem avisando quando a loja está fechada ou aberta, baseada no valor do método `isOpen`.
6. Então, modifique o método `displayHoursOfOperation`, que antes mostrava valores arbitrários para os horários de abertura e fechamento da loja, para mostrar os valores especificados nas variáveis `openTime` e `closeTime`.
7. Modifique a `TestMusicStore` para mostrar mensagens do tipo "Estamos Abertos!" ou "Estamos Fechados!".
8. Compile os fontes de `MusicStore` e `TestMusicStore`.
9. Execute `TestMusicStore`.

Demonstração

O que deve aparecer na tela após a execução deve ser similar a isso:

Período:

Estamos abertos!!

Diariamente das: 9:00 - 21:00

Roberto, Proprietário

LabIX: MusicStore – Concatenação de String

Para esse exercício você (1) vai modificar o `displayHoursOfOperation` para que ele leia do teclado a hora de abertura e de fechamento (2) criar um método `toString` para `MusicStore`.

Etapas

1. Modifique o método `displayHoursOfOperation` para que ele apresente os horários corretos de abertura e fechamento, isso é, horários consistentes com os valores armazenados nas respectivas variáveis lidos do teclado.
2. Crie um método `toString` para `MusicStore` que concatene junto a informação pertinente para a instância corrente e retorne a String resultante.
3. Modifique a `TestMusicStore` para Testar/Mostrar a funcionalidade do método `toString`.
4. Compile os fontes de `MusicStore` e `TestMusicStore`.
5. Execute `TestMusicStore`.

Demonstração

O que deve aparecer na tela após a execução deve ser similar a isso:

Período:

[Dono = Carlos , Abre = 9, Fecha = 21]

LabX: MusicStore - Adicionando Títulos

Para esse exercício adicione a capacidade de manipulação de múltiplos títulos de música(discos) , isso é, armazenar, recuperar, e Mostrar títulos.

Etapas

1. Crie uma classe chamada MusicTitle com 2 Strings, title e artist, inicializadas com "sem nome". Implemente get e set métodos de acesso para ambas as variáveis.
2. Crie a variável de instância titles do tipo MusicTitle[] para MusicStore, inicializando-a com null, e então implemente os métodos de acesso setTitles() e getTitles().
3. Crie um método de nome displayMusicTitles() para MusicStore que percorra o vetor de títulos e mostre o nome e o artista do título.
4. Modifique TestMusicStore para Testar/Mostrar a funcionalidade do método displayMusicTitles().
5. Compile os fontes.
6. Execute TestMusicStore.

Demonstração

O que deve aparecer na tela após a execução deve ser similar a isso:

Título 1:

Título: A Festa

Artista: Ivete Sangalo

Título 2:

Título: Luna Nueva

Artista: Diego Torres

LabXI: Banco - Herança

Para esse exercício suponha um pequeno Banco que possui 4 Clientes. Cada cliente possui uma única Conta associada com número e saldo. Essa Conta pode ser Corrente ou Poupança. A cada mês a Conta deve atualizar os saldos [se a conta for Corrente é debitado 1 real do saldo para manutenção e se for Poupança é acrescido 1% no saldo]. Modele a estrutura e implemente estas classes.

Regras

1. O Cliente (nome, Conta (número, saldo)) se cadastre no Banco fornecendo um saldo inicial.
2. Se possa depositar, retirar dinheiro e consultar saldo de uma conta pelo seu número ou pelo nome do cliente.
3. Se possa aplicar a atualização mensal quando necessário.
4. Se exiba todo o valor guardado no Banco.
5. Compile os fontes.

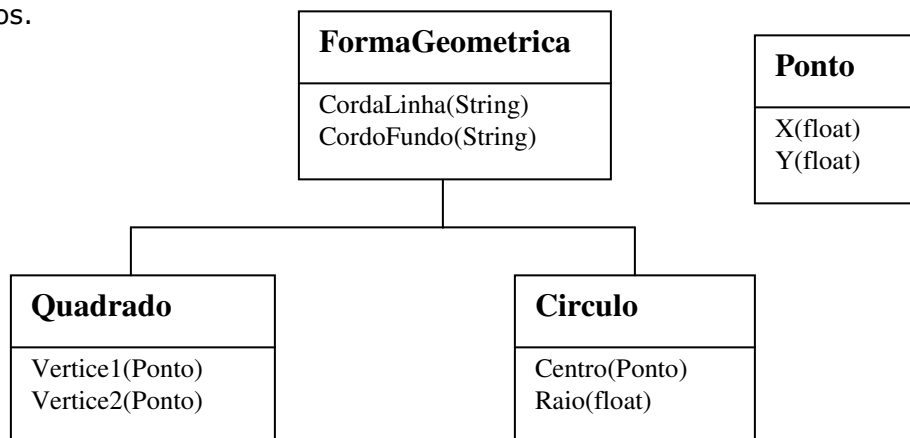
Demonstração

Crie um programa de exemplo que efetue as seguintes operações:

- Cadastre com saldo inicial de 100 reais cada, 3 clientes com conta poupança, e 1 clientes com conta corrente;
- Saque 30 pelo número da conta na 1ª conta corrente criada;
- Deposite 34 reais pelo nome do cliente na 1ª conta poupança criada;
- Atualize os valores devido a virada de mês
- Exiba pelo número da conta o saldo da 1ª conta-corrente
- Exiba o valor guardado no Banco.

LabXII: Formas Geométricas - Herança

A figura abaixo representa um diagrama simplificado de classes onde só aparecem os atributos.



Implemente as classes acima usando a linguagem Java, respeitando o fato de que os métodos deverão ser implementados de maneira a atender os requisitos abaixo relacionados:

Regras

1. Cada classe deve ter dois construtores. Um com a assinatura "default" e outro que recebe como argumento os atributos da classe.
2. Todos os atributos devem ter métodos de acesso (tanto para definição quanto para recuperação) aos seus valores.
3. As classes círculo e quadrado devem possuir um método para calcular a área da respectiva forma geométrica e outro para imprimir o tipo da forma geométrica(Nome da Classe), os valores de seus atributos e a área.

Demonstração

Crie um programa de exemplo que efetue as seguintes operações:

- Crie um Quadrado e um Círculo usando variáveis do tipo FormaGeométrica.
- Exiba o resultado do método imprime de cada Forma Geométrica Criada.

LabXIII: Forno - Agregação

Escreva um programa em Java que simule um forno, um termostato e um usuário. Primeiro o termostato é configurado pelo usuário que define a temperatura chamada de inicial. Depois o usuário passa a temperatura do forno para o termostato. Então esse compara as duas temperaturas e caso a temperatura do forno estiver abaixo da inicial o termostato avisa que o forno deve ser ligado. Quando o forno é ligado ou desligado um aviso é emitido. O programa termina aqui. Dica: Crie classes separadas para cada uma das entidades do programa(Usuário, Forno, Termostato).

Demonstração

Os valores das temperaturas necessárias para a realização do programa devem ser definidos por você no método main do programa. Crie uma classe chamada de Teste que tem o método main para testar o programa.

LabXIV: PARXY – Comparação de Objetos

Defina, desenvolva e teste uma classe PARXY, cujas instâncias são pares (x,y) de dois objetos de qualquer classe. Cada instância desta classe deverá ser capaz de responder às seguintes mensagens:

- getX() / putX(obj) – devolve / define o objeto em x;
- getY() / putY(obj) – devolve / define o objeto em y;
- iguaisXY() – indica se os objetos X e Y "são o mesmo" objeto.

Demonstração

Crie uma instância de PARXY e use todos os seus métodos para testar a sua implementação.

LabXV: PONTO3D

Desenvolva a classe PONTO3D, cujas instâncias são pontos de três coordenadas inteiras (x,y,z). Cada instância desta classe deverá ser capaz de responder às seguintes mensagens:

- putX(int), putY(int), putZ(int) – altera as coordenadas do ponto;
- getX(),getY(),getZ() – devolve as coordenadas do ponto.

LabXVI: PLANO3D

Desenvolva a classe PLANO3D, cujas instâncias são planos formados a partir de três PONTO3D não alinhados (i.e. não pertencem à mesma reta). Cada instância desta classe deverá ser capaz de responder às seguintes mensagens:

- horizontal() – verdadeiro se o plano for horizontal;
- perpendicular() – verdadeiro se o plano for perpendicular a qualquer dos eixos(X ou Y).

Demonstração

Crie uma instância de PLANO3D e use todos os seus métodos para testar a sua implementação.

LabXVII: RETÂNGULO

Defina, desenvolva e teste uma classe Retangulo, cujas instâncias apresentem um comportamento semelhante ao de um Retângulo cujas dimensões (comprimento e largura) são valores inteiros. Cada instância desta classe, deverá ser capaz de responder às seguintes mensagens:

- getComprimento() / setComprimento(comp) – devolve / define o comprimento;
- getLargura() / setLargura(l) – devolve / define a largura;
- getArea(c,l) – calcula a área do retângulo;
- perimetro() – calcula o perímetro do retângulo;
- divide() – o retângulo é dividido e é retornado o retângulo(uma nova instância) resultante da divisão;

Demonstração

Crie uma instância de Retangulo e use todos os seus métodos para testar a sua implementação.

LabXVIII: Lista de Objetos

Defina e desenvolva uma classe Lista, cujas instâncias são listas não limitadas de objetos de qualquer tipo. Cada instância desta classe deverá ser capaz de responder às seguintes mensagens:

- putObj(obj) – coloca obj numa posição livre da lista e retorna o índice dessa posição;
- remObj(indice) – remove o obj que está na posição índice;
- posObj(obj) – determina o índice na lista de obj (0 caso não exista);
- temObj(obj) – determina se contém numa posição da lista, o objeto obj;
- isEmpty(ind) – determina se a lista está vazia na posição ind (null);
- isEmpty() – determina se a lista está vazia (se apenas contiver objetos null);
- tamanho() – determina o número de objetos não null que a lista contém.

Demonstração

Crie uma instância de Lista e use todos os seus métodos para testar a sua implementação.

LabXIX: CLASSES ABSTRATAS

Defina, desenvolva e teste uma classe abstrata PARXY, que implemente as funcionalidades comuns a classes cujas instâncias são pares ordenados (x,y) de dois objetos da mesma classe. Defina e teste as subclasses PARXYInt (pares de inteiros), PARXYNomes (pares de Strings), PARXYVectores (pares de Vectores de Inteiros). As instâncias destas classes deverão ser capazes de responder às seguintes mensagens:

- getX() / putX(a) – devolve / define o objecto em x;
- getY() / putY(a) – devolve / define o objecto em y;
- iguaisXY() – indica se os objectos X e Y têm igual valor (equals());
- xMaior() – indica se o objecto X é maior que o objecto y;

Demonstração

Crie instâncias de PARXY para cada uma de suas subclasses e use todos os seus métodos para testar a sua implementação.

LabXX: Tratamento de Exceções

Construa a classe TestaStack que, usando uma instância de Stack (veja documentação anexada), deve efetuar a seguinte seqüência de ações de modo a que o programa não termine (tratar exceção): push(3), push(4), pop(), pop(), pop(), push(5).

LabXXI - A: Tratamento de Exceções

Construa a classe TestaStack que, usando uma instância de Stack (veja documentação do pacote java.util), deve efetuar a seguinte seqüência de ações de modo a que o programa não termine (tratar exceção): push(3), push(4), pop(), pop(), pop(), push(5).

LabXXI - B: Tratamento de Exceções

Construa e teste a classe Queue (fila de tamanho fixo que ao contrário da stack, tem um comportamento FIFO (FirstIn-FirstOut)).

LabXXI - C: Tratamento de Exceções

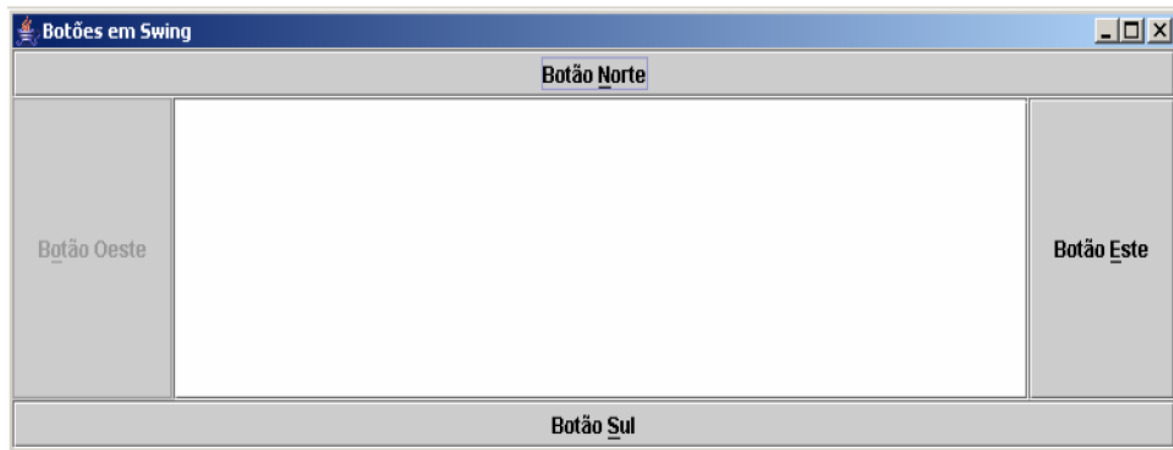
Construa a classe StackInt (subclasse de Stack) que apenas permite que os seus elementos sejam inteiros, não permitindo repetições.

LabXXI - D: Tratamento de Exceções

Pretende-se criar a classe Console que através de métodos estáticos permita fazer a leitura validada de tipos primitivos. Esta classe poderá vir a ser utilizada em problemas futuros.

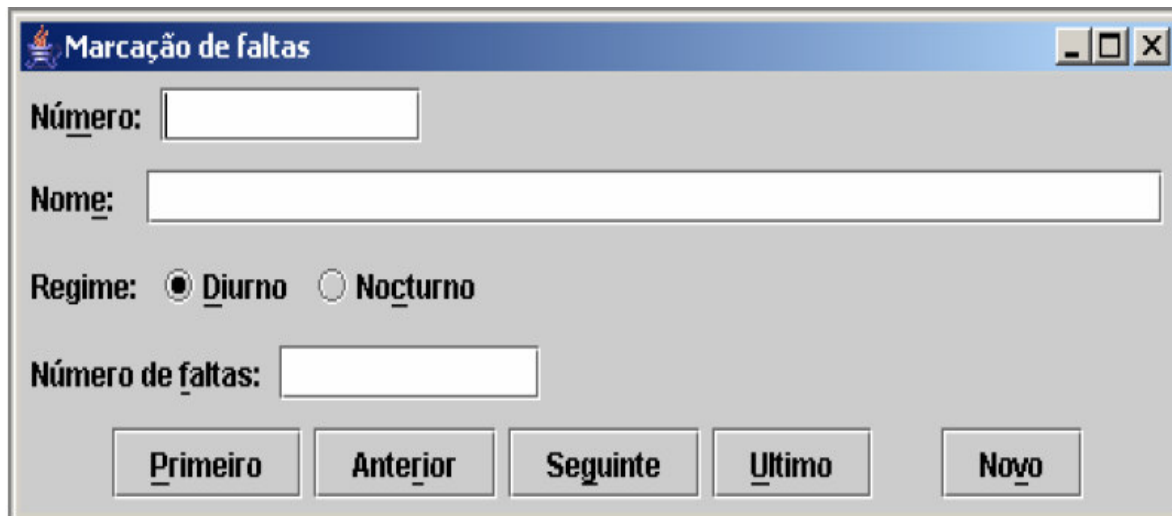
LabXXII: Interfaces Gráficas

Crie um programa em Java, utilizando a API Swing que construa uma interface gráfica com quatro botões. Um a Norte, outro a Sul, Leste e Oeste. Os botões não precisam ter qualquer funcionalidade. O programa deverá ter o aspecto da figura a seguir apresentada:



LabXXIII: Interfaces Gráficas

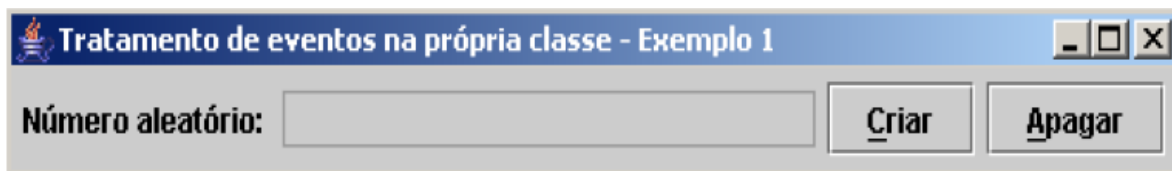
Crie um programa em Java, utilizando a API Swing, que graficamente tenha o aspecto da figura a seguir.



Deverá ser possível editar cada campo, no entanto não é necessário fazer qualquer tipo de funcionalidade nos botões.

LabXXIV: Interfaces Gráficas

Crie um programa em Java, utilizando a API Swing, que crie um número aleatório, bastando para isso clicar num botão. Deverá ser possível apagar o número escolhido. Use uma caixa de texto e dois botões para estas funcionalidades. O programa deverá ter o seguinte aspecto:

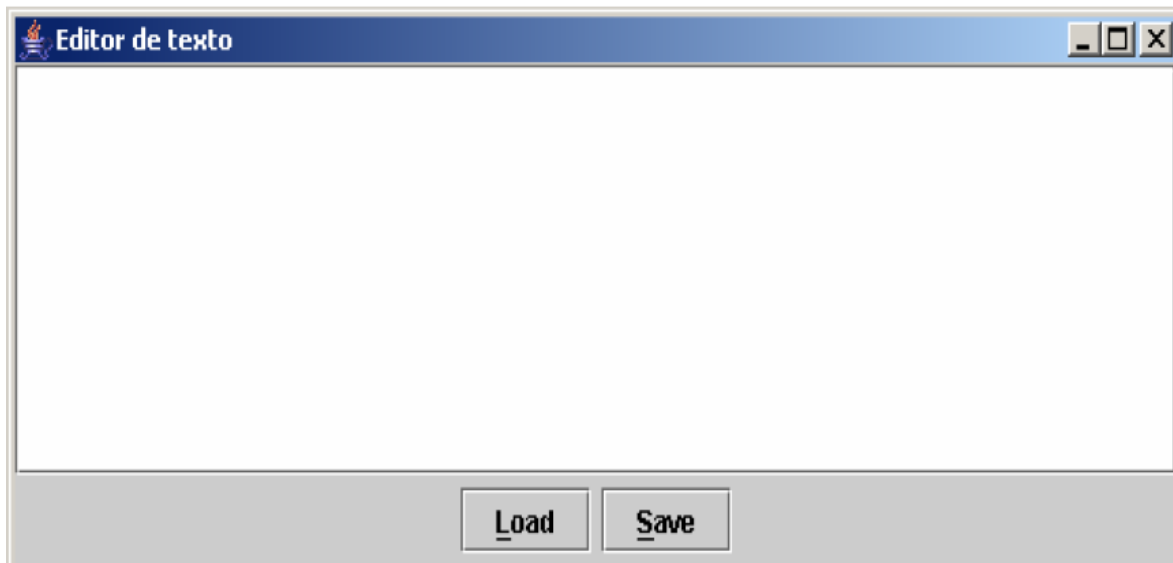


LabXXV: Interfaces Gráficas

Crie um programa em Java, utilizando a API Swing, que simule um Editor de Texto, para isso exigem-se as funcionalidades: Abrir Arquivo; Salvar Arquivo. Tanto para abrir quanto para salvar um arquivo será necessário usar um objeto do tipo JFileChooser e outro do tipo File (veja o trecho de exemplo de código). Para área de texto use o JEditorPane.

```
JFileChooser caixaSelecaoArquivo=new JFileChooser();
caixaSelecaoArquivo.showOpenDialog(this);
File arquivo= caixaSelecaoArquivo.getSelectedFile();
```

O programa deverá possibilitar a escrita de textos e deverá ter este aspecto:



LabXXVI: Interfaces Gráficas

Como é de conhecimento geral entrada de dados em Java não é tão simples como em outras linguagens de programação que apenas solicitam a leitura do tipo específico desejado. Em java todos os dados obtidos do teclado são do tipo String. Para torná-los inteiros, float ou double é necessário a realização de conversões. Para esse laboratório, crie uma subclasse de JTextField que deve ter métodos para: 1)ler um valor inteiro, 2)ler um valor float e 3)ler um valor String. Crie uma classe Pessoa que tem os atributos nome, peso e altura. Para preencher a informações de uma Pessoa, um formulário deve ser criado e o usuário deve digitar o seu nome, peso (em Kg) e altura (em centímetros). Em seguida deve ser criados um objeto do tipo Pessoa com estas informações e o índice de massa corpórea (valor do peso sobre a altura elevada ao quadrado) deve ser mostrado na tela.

LabXXVII: Interfaces Gráficas

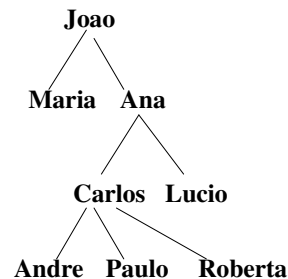
Seu João gostaria de fazer um sistema para cadastrar seus descendentes e informar a linhagem de um descendente (selecionado) até ele. Uma classe Pessoa deve ser criada para isso. Esta classe deve ter um método abstrato chamado cadastrarFilho que será implementado nas suas subclasses Filho e Filha. O método "cadastrarFilho" cria uma pessoa, cadastra-o como filho e depois retorna para a classe do Formulário a nova pessoa da família. Cada pessoa criada pode ser: uma filha, um filho ou um filho natimorto.

Por causa de uma doença de família, no caso dos descendentes de filhas existe 50% de chance da pessoa morrer antes de completar o 1º mês de vida se o filho for do sexo masculino. Isso deve ser determinado através o método Random do pacote java.util no ato do cadastramento do filho.

No caso dos descendentes de filhos o segundo filho sempre nasce morto, independente do sexo. Nos dois casos citados, o descendente deve ser do tipo FilhoNatimorto.

Perceba que a depender do tipo de pessoa a forma de retornar o método do nome é diferente. Aos homens é acrescentado o prefixo "Sr", às mulheres "Sra" e aos natimortos um "Natimorto".

No Formulário existe a pessoa João (que é o patriarca) e um Arraylist de todos os seus descendentes, independente do grau de parentesco. Os eventos tratados devem ser o de inserir um novo filho e mostrar a linhagem de um descendente cadastrado. Quando uma pessoa é cadastrada seu nome deve aparecer no JComboBox de Familiares. A interface gráfica segue o modelo dado a seguir.



Exemplo:

**Carlos é filho de Ana
neto de Joao**

**Assim a ordem para
mostrar a linhagem é :**

**Sr Carlos
Sra Ana
Sr Joao**

LabXXVIII: Interfaces Gráficas

Faça um programa orientado a objetos para cadastrar pessoas que se inscreveram em um Projeto de Pesquisa que tem como pesquisador maior a Sra. Mara Andrade. Os pesquisadores que trabalham neste Projeto de Pesquisa podem ser Coordenadores de Pesquisa que têm pesquisadores e outros coordenadores de pesquisa sob sua supervisão ou então simples professores.

Uma classe Pesquisador deve ser criada para auxiliar a sua solução. Todo pesquisador deve ser capaz de informar quantos pesquisadores existem sobre sua supervisão. Todo pesquisador possui um coordenador de pesquisa, que também é do tipo pesquisador.

A classe Projeto de Pesquisa deve ser construída para armazenar os pesquisadores cadastrados (método `inserirPesquisador`), para retornar a quantidade de pesquisadores associados a um pesquisador (método `getQtdPesquisadores`) e para retornar o valor a ser pago pelo pesquisador para se inscrever no Projeto (método `getValorPago`).

Para calcular o valor a ser pago pelo pesquisador é preciso levar em consideração que professores pagam R\$ 220,00 mais 30% do que paga seu coordenador de pesquisa,

enquanto que coordenadores de pesquisa pagam R\$ 300,00 menos 2,5% por cada pesquisador que esteja vinculado a ele.

A classe de Interacao com os usuários deve seguir o modelo apresentado na figura e os nomes dos componentes apresentados no Anexo. Para cada pesquisador cadastrado no Projeto deve ser possível executar as consultas mostradas na figura (quantidade de pesquisadores associados e a valor pago para entrar no projeto). Lembre-se que os pesquisadores associados de um coordenador de pesquisa envolvem tanto seus pesquisadores (e coordenadores de pesquisa) como os pesquisadores de seus coordenadores.

The figure consists of two screenshots of a software interface and a hierarchical diagram below them.

Left Screenshot: Shows a window with two tabs: "Cadastramento Funcionário" and "Consultas". The "Cadastramento Funcionário" tab is active. It contains three input fields: "NomePesquisador:" with the value "Maria", "Tipo:" with a dropdown menu showing "Professor", and "Coordenador:" with a dropdown menu showing "Jose". Below these fields is a "Cadastrar" button. A list of names (Mara Andrade, Davi, Jose) is visible below the "Coordenador:" dropdown.

Right Screenshot: Shows the same window with the "Consultas" tab active. It contains a "Nome Pesquisador:" dropdown menu showing "Davi", a "Consulta" button, a "Qtd de pesquisadores:" input field with the value "4", and a "Valor Pago:" input field with the value "270.0". A list of names (Mara Andrade, Manoel, Davi, Roberto, Jose, Maria, Marcus) is visible below the "Nome Pesquisador:" dropdown.

Hierarchical Diagram:

```
graph TD; A["Mara Andrade (##)"] --> B["Davi (##)"]; A --> C["Manoel (*)"]; B --> D["Roberto(*)"]; B --> E["Jose (##)"]; E --> F["Maria(*)"]; E --> G["Marcus(*)"];
```

(*) São os pesquisadores que são apenas professores, no exemplo.
(##) São os pesquisadores que são coordenadores de pesquisa.

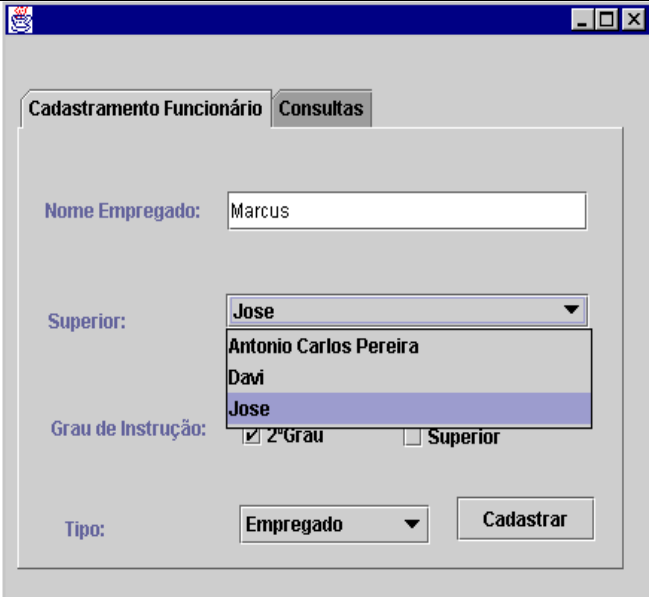
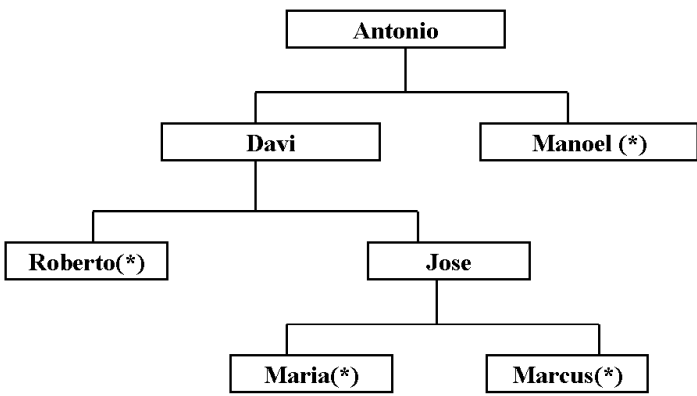
LabXXIX: Interfaces Gráficas

Em anexo temos um programa orientado a objetos para cadastrar pessoas que trabalham em uma empresa onde o chefe maior é seu Antonio Carlos Pereira. As pessoas que trabalham nesta empresa podem ser chefes que têm empregados e outros chefes sob sua supervisão ou então empregados. As classes Pessoa, Empregado, Chefe e Empresa devem ser implementadas.

O método `getListaEmpregadosAssociados()`, da classe Pessoa é abstrato e precisa ser implementado por você nas subclasses. Com este método é possível mostrar a lista dos empregados subordinados (diretos e indiretos) desta Pessoa, conforme pode ser visto no exemplo dado abaixo.

Toda pessoa deve ser capaz de informar a lista dos empregados sobre sua supervisão, bem como a lista de seus superiores. Os empregados subordinados são aqueles que são empregados diretos de subordinados da Pessoa. Toda Pessoa possui um superior, que também é do tipo pessoa.

A classe de Interacao com os usuários deve seguir o modelo apresentado na figura. Para cada nova pessoa cadastrada na Empresa já deve ser possível executar a consulta de quais os empregados que estão sob sua supervisão, bem como a lista dos chefes.

TELA DE CADASTRAMENTO	EXEMPLO
	 <p data-bbox="971 1602 1526 1654">(*) São as pessoas que são Empregados, no exemplo. Marcus foi o último a ser cadastrado na figura.</p>

Cadastramento Funcionário Consultas

Nome Funcionário: Davi

Tipo Consulta: Subordinados

Lista :

- Roberto
- Maria
- Marcus
- José

Consulta

Cadastramento Funcionário Consultas

Nome Funcionário: Davi

Tipo Consulta: Superiores

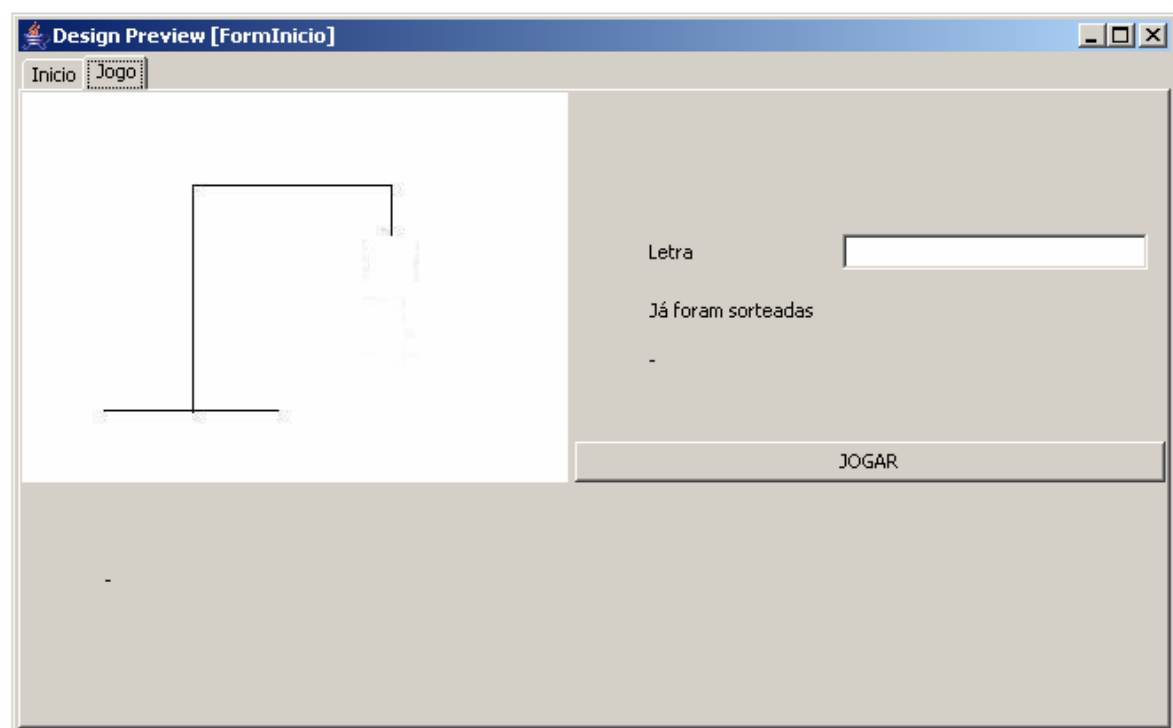
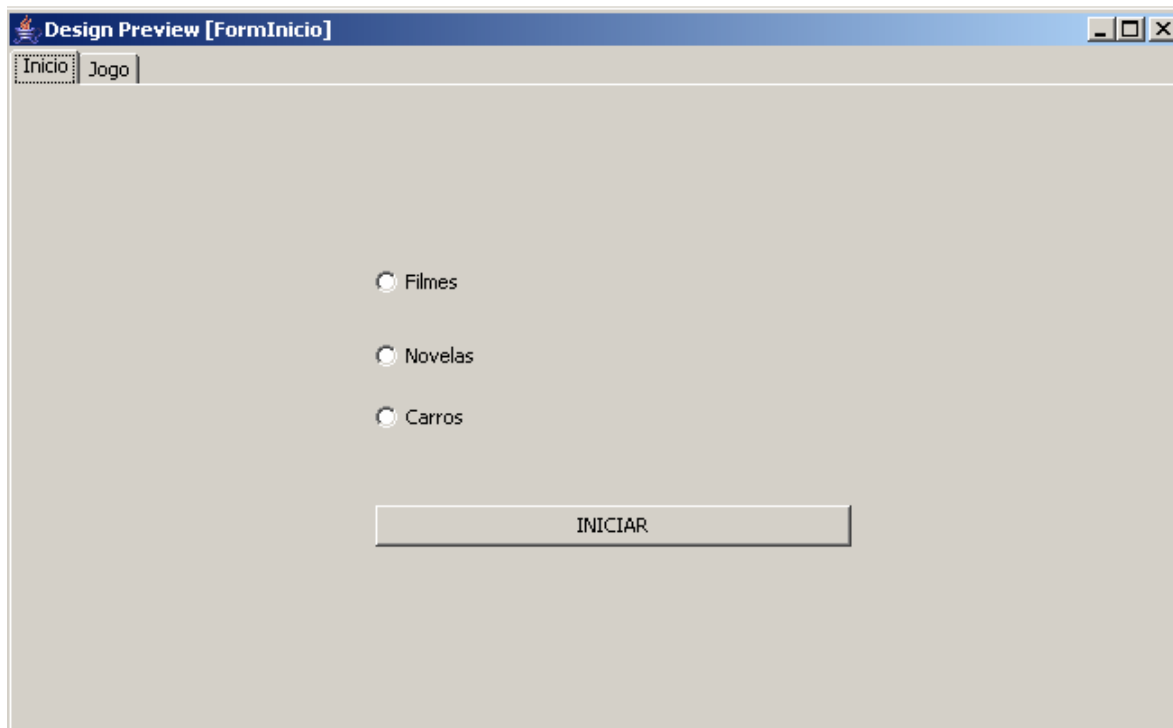
Lista :

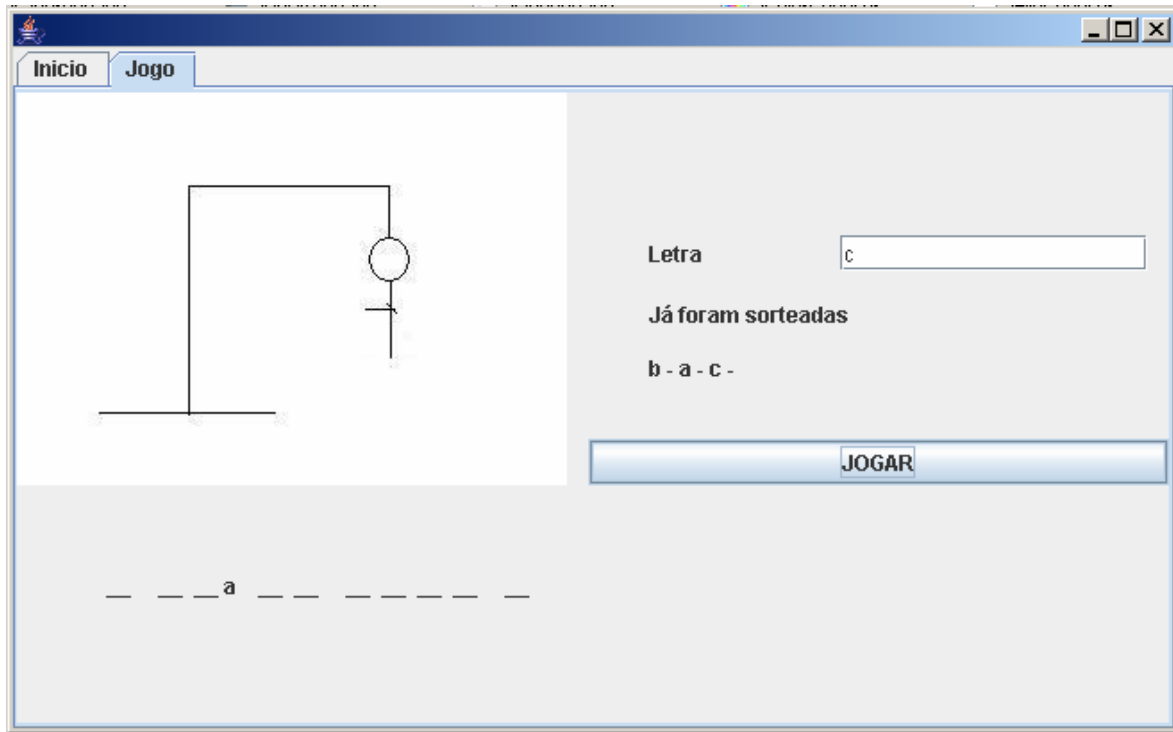
- Antonio Carlos Pereira

Consulta

LabXXX: Interfaces Gráficas

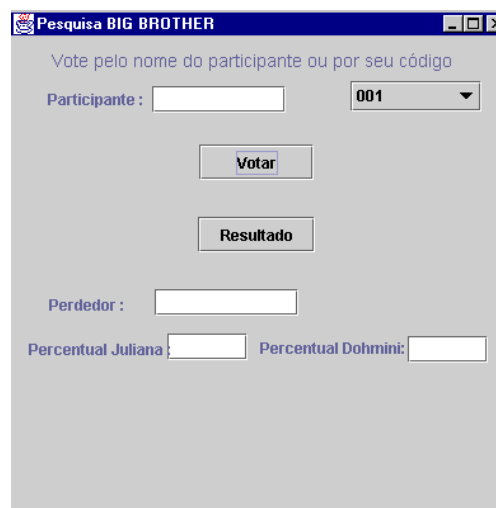
Escreva um programa que simule um Jogo de Forca. Inicialmente uma categoria deve ser escolhida (Filmes, Novelas ou Carros). Com base na categoria selecionada deve ser sorteada uma palavra deve ser sorteada e assim o jogo se inicia. Cada categoria deve conter quatro palavras. Um usuário pode cometer no máximo cinco erros. Analise a sequência de figuras a seguir para contruir a sua solução.





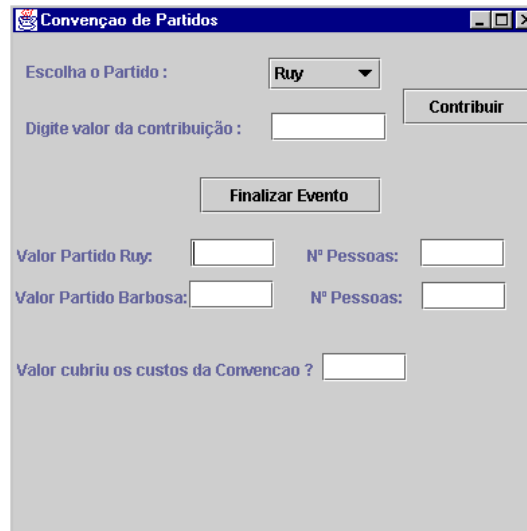
LabXXXI: Interfaces Gráficas

Faça um sistema para representar um pesquisa que foi feita com a finalidade de determinar qual dos dois participantes do BIG Brother irá deixar o programa nesta semana. O sistema deve permitir a votação pelo nome(Juliana ou Dhomini) ou pelo código do participante (001, 002), respectivamente. O sistema deve ser capaz de determinar qual o candidato que teve maior número de votos e qual a porcentagem dos dois candidatos. Se houver empate Dhomini deve ser o vencedor por já ter ido mais vezes ao paredão.



LabXXXII: Interfaces Gráficas

Faça um programa orientado a objeto para atualizar as contas de partidos políticos que estão juntos em uma convenção. Durante o evento pessoas individuais podem contribuir para os partidos no estande da convenção. No final do evento os partidos devem doar 15% do que arrecadaram para pagar os custos do evento, se o valor arrecadados for menor/igual do que R\$ 100.000 e 18% se for maior. O programa deve informar quanto foi arrecadado pelos partidos e se o valor repassado para a convenção cobriu os custos estimados (um total de R\$ 20.000) (Lembre-se de atualizar o saldo das contas dos partidos). Deve informar também quantas pessoas contribuíram para cada partido. Considere que, neste exemplo, teremos apenas dois partidos : Partido Ruy e o Partido Barbosa.



Convenção de Partidos

Escolha o Partido : Ruy

Contribuir

Digite valor da contribuição :

Finalizar Evento

Valor Partido Ruy: Nº Pessoas:

Valor Partido Barbosa: Nº Pessoas:

Valor cobriu os custos da Convencao ?

LabXXXIII: Interfaces Gráficas

Considere uma enquete que está sendo realizada com os ônibus que chegam a um estádio de futebol. Para cada ônibus são feitas duas perguntas: Que time a caravana torce (Bahia ou Vitória) e quantos torcedores há neste grupo. O patrocinador da enquete tem R\$ 10.000 para gastar e está disposto a colocar na conta dos Times o equivalente por torcedor (valor/total de entrevistados) até o limite de R\$ 10.000. Faça um programa orientado a objetos para resolver a questão. Este programa deve ter as classes Enquete, Time e Conta, além do Formulário(faça o formulário como achar melhor). Os Times devem ser capazes de responder no final da enquete quanto eles ganharam, quantos torcedores participaram da enquete e quanto de juros eles receberam depois de um mês aplicando o dinheiro. A conta do Bahia é do BBV e tem 2% de taxa de juros enquanto a conta do Vitória é do REAL que tem 1,98% de taxa de juros mensal.

LabXXXIV: Interfaces Gráficas

Faça um programa para auxiliar o departamento de pessoal de uma empresa X. O programa deve ter uma classe "Sistema de Pagamento", uma classe "Empregado" e uma classe "Desconto". Um sistema de pagamento tem um empregado e cada empregado tem descontos associados a ele. O desconto de imposto de renda (retido) segue a tabela 1 enquanto que o desconto de INSS segue a tabela 2. O salário líquido de um empregado deve ser calculado, o valor de imposto de renda e de INSS, bem como valor do FGTS no mês (8% do salário bruto). Para isso são informados pelo usuário os vencimentos que compõem o salário do empregado (exemplo: salário base, adicional de periculosidade, adicional de tempo de serviço, valor das horas extras etc). A soma dos vencimentos compõe o salário bruto do funcionário. Veja o Formulário e os resultados que seu sistema deve informar.

Valor mensal	Alíquota	Parcela a deduzir depois do cálculo do imposto
Até R\$ 1058,00	Não tem	Não tem
De R\$ 1058,01 até R\$ 2115,00	15% Deduzir R\$ 158,7	
Acima de R\$ 2115,00	27,5%	Deduzir R\$ 423,08

Tabela 1

Valor mensal	Percentual de desconto
Até R\$ 720,00	7,65%
De R\$ 720,01 até R\$ 1200,00	9,00%
De R\$ 1200,01 até 2 400,00	11,00%
Acima de R\$ 2400,01	R\$ 264,00 (valor fixo)

Tabela 2

Sistema de Pagamento

Nome Empregado:

CPF: Inserir

Vencimento: Inserir Vencimento

Resultado

Total dos Vencimentos:

Valor Líquido:

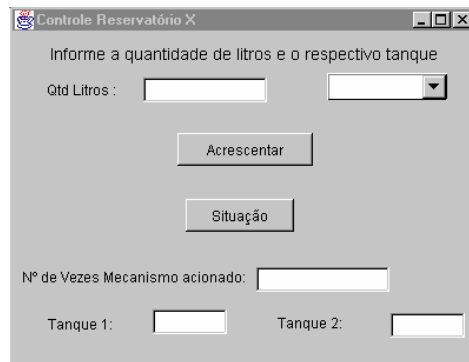
F.G.T.S:

Imposto Retido:

INSS:

LabXXXV: Interfaces Gráficas

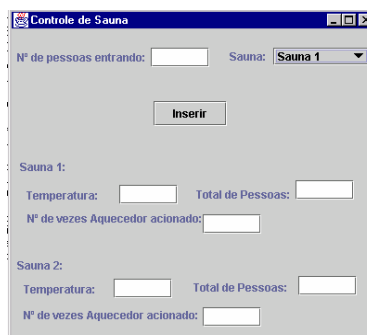
Faça um sistema orientado a objeto para controlar o enchimento de dois tanques de um reservatório. O sistema deve ser capaz de receber litros de água e direcionar sempre para o tanque que estiver mais vazio. Atingindo o limite de 100.000 litros (equivalente a capacidade máxima de cada tanque), cada vez que uma determinada quantidade de água for acrescentada, o tanque deve escoar 10% da sua capacidade total enquanto a quantidade for maior do que a capacidade de armazenamento. O sistema deve ser capaz de informar a quantidade de litros de água de cada tanque e a quantidade de vezes que o mecanismo de escoamento foi acionado.



The screenshot shows a window titled "Controle Reservatório X". Inside, there is a label "Informe a quantidade de litros e o respectivo tanque". Below it is a text input field for "Qtd Litros:" followed by a dropdown menu. There are two buttons: "Acrescentar" and "Situação". Below these buttons is a text input field for "Nº de Vezes Mecanismo acionado:". At the bottom, there are two text input fields labeled "Tanque 1:" and "Tanque 2:".

LabXXXVI: Interfaces Gráficas

Faca um programa orientado a objetos para controlar a entrada/saída de pessoas e a temperatura das saunas A e B de um clube. As saunas devem manter uma temperatura mínima de 43 e a máxima de 47 grau, com o limite máximo de 10 pessoas simultaneamente. Seu programa deve controlar a entrada e saída dos usuários da sauna não permitindo que o limite de pessoas seja excedido. A entrada de uma pessoa na sauna acarreta no aumento de 2 grau na temperatura da sauna. Quando uma pessoa sai da sauna a temperatura é reduzida em 1 grau. Uma vez que a porta foi aberta (tanto para saídas quanto para entradas), para as pessoas seguintes temos uma diminuição de 0,5 grau por pessoa. Cada vez que a temperatura sai do intervalo proposto para ela [43 a 47 graus] o aquecedor da sauna é acionado a fim de retornar a temperatura ao valor inicial de 45 graus. O sistema deve ser capaz de controlar as entradas e saídas das pessoas nas saunas A e B, como mostra o formulário, mostrando continuamente a temperatura ambiente, o número atual de pessoas e quantas vezes o aquecedor foi acionado em cada sauna (veja o formulário).

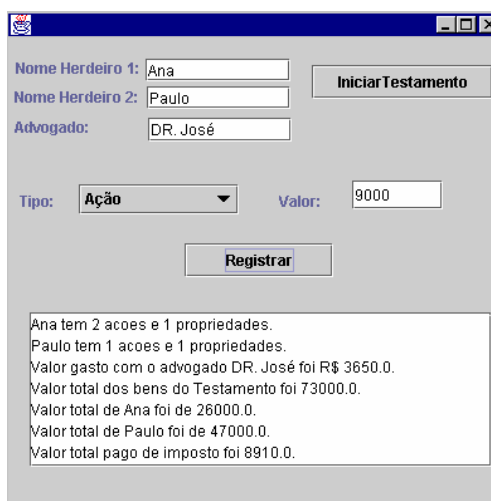


The screenshot shows a window titled "Controle de Sauna". It has two main sections for "Sauna 1" and "Sauna 2". At the top, there is a label "Nº de pessoas entrando:" followed by a text input field and a dropdown menu labeled "Sauna: Sauna 1". Below this is an "Inserir" button. For each sauna, there are three labels and input fields: "Temperatura:", "Total de Pessoas:", and "Nº de vezes Aquecedor acionado:". The "Sauna 2" section is identical to the "Sauna 1" section.

LabXXXVII: Interfaces Gráficas

Faça um programa orientado a objetos para auxiliar os cálculos necessários quando um Testamento é executado. Este programa deve ser composto de uma classe de interação com usuário, FormulárioTestamento, e as classes Testamento, Herdeiro, Herança e Advogado. O FormulárioTestamento pode ser visto abaixo. Este programa deve funcionar para muitos testamentos sendo necessário apenas iniciar novamente o testamento por acionar o botão IniciarTestamento que pode ser visto na figura. Os bens (ação ou propriedade) são inseridos alternadamente começando com o bem do herdeiro 1. O advogado do testamento receberá, de honorário, um percentual de 5% sobre o valor dos bens dos herdeiros. O testamento deve informar o resumo da herança como pode ser visualizado na figura. Cada herdeiro tem sua herança e deve poder informar a quantidades de propriedades e ações existentes nela, bem como o valor de imposto pago pelos bens de sua herança. Considere que as propriedades têm um imposto de 10% e as ações 17% sobre o seu valor. O herdeiro deve ser capaz de listar a quantidade dos tipos de bens (propriedade e ações) que ele possui.

OBS: O Resumo é composto pela quantidade de cada um dos bens de cada herdeiro, o valor total dos honorários do advogado, o valor total do testamento, o valor total de cada um dos herdeiros e o valor pago de imposto pela herança. Utilize a figura para se guiar na elaboração deste resumo.



Nome Herdeiro 1: Ana

Nome Herdeiro 2: Paulo

Advogado: DR. José

Iniciar Testamento

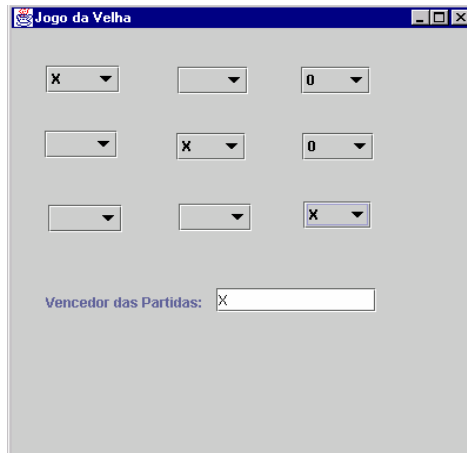
Tipo: Ação Valor: 9000

Registrar

Ana tem 2 acoes e 1 propriedades.
Paulo tem 1 acoes e 1 propriedades.
Valor gasto com o advogado DR. José foi R\$ 3650.0.
Valor total dos bens do Testamento foi 73000.0.
Valor total de Ana foi de 26000.0.
Valor total de Paulo foi de 47000.0.
Valor total pago de imposto foi 8910.0.

LabXXXVIII: Interfaces Gráficas

Faca um programa orientado a objetos para executar um Jogo da Velha. O programa deve possuir uma interface gráfica que retrate o jogo da velha e a lógica de uma partida. Cada partida tem um estado (finalizada ou ativa), um vencedor e uma matriz de posições que representa cada uma das posições do jogo da velha. Para cada jogada executada é necessário conferir se a partida está em andamento, se foi encerrada com um dos jogadores ganhando ou se foi encerrada sem vencedor. No momento que uma sequência for completada então o jogo é finalizado e na interface gráfica deve ser mostrada a referência do vencedor (neste caso X ou O), conforme pode ser visto no formulário a seguir. Não permita que depois de marcada uma posição o jogador tente modificá-la.



LabXXXIX: Interfaces Gráficas

Considere uma pequena Empresa que tem apenas dois empregados (Mário e Luís, com salários base R\$2000,00). Cada Empregado possui seu Cartão de Ponto que contém uma estrutura interna para armazenar as faltas de cada mês do ano. Faça um programa orientado a objetos para atualizar o Cartão de Ponto de cada funcionário. A empresa irá atualizar o programa com a quantidade de faltas e também os meses (e dias) onde elas ocorreram em cada um dos dois empregados. Assim cada empregado deve anotar no seu cartão de ponto as faltas existentes.

A Empresa possui um bonus de R\$1.000,00 para dar ao empregado que tenha uma menor quantidade de faltas no ano e está disposta a aumentar o salário do empregado neste valor. Informe qual o funcionário que receberá este aumento e atualize o seu salário. Imprima também, para cada empregado, quais as faltas que ocorreram (mês/dia) durante o ano, justificando assim o bônus para o funcionário mais assíduo.

Obs: Para simplificar, considere que existem 12 meses no ano e 31 dias em cada mês. Sugerimos o uso de uma matriz do tipo boolean como estrutura interna do cartão de ponto para controle das faltas. Faça a interface gráfica você mesmo!!.

LabXL: Interfaces Gráficas

Faça um programa Orientado a Objeto em Java que calcule o salário do mês de funcionários em uma Escola e quanto será a folha de pagamento do mês. Solicite que seja informado o valor do salário base do funcionário, quantos dias este faltou de trabalho, a quantidade de horas extras efetuadas durante o mês e o tipo de Funcionário (Funcionário Administrativo ou Professor). Para os funcionários administrativos considere que o valor descontado por dia de trabalho é calculado sobre o total de 30 dias de trabalho no mês e que o valor da hora extra é 1.5 da hora normal do funcionário (tomando em consideração 30x8 horas mensais). No caso dos professores não existem descontos por causa de faltas, já que estes sempre terão de repor as aulas não ministradas. Além disso, os professores trabalham apenas 12 dias por mês com uma carga horária de 8 horas por dia e o valor da hora extra deles é 2.5 da hora normal.

Obs: Faça a interface gráfica você mesmo!!

LabXLI: Interfaces Gráficas

Construa um sistema orientado a objetos, utilizando programação em três níveis, que atenda ao cadastramento e vendas de produtos de uma Farmácia. Nesta farmácia teremos no máximo o cadastramento de 100 medicamentos diferentes, divididos em controlados, não-controlados e comuns. Para cada medicamento deve ser informada a quantidade que está sendo disponibilizada na farmácia, seu preço e código (que deve ser único). Medicamentos controlados não podem ser totalmente vendidos para o público, pois esta farmácia está ligada a um hospital. Assim, para este tipo de medicamento a quantidade em estoque nunca pode ser de menos de 20% da quantidade inicialmente cadastrada. Como este mês a farmácia está em promoção, na compra de um produto comum, com preço abaixo de R\$ 2,50, o cliente recebe dois produtos. É possível devolver um produto desde que ele seja do tipo controlado. Tenha cuidado de vender apenas produtos existentes no estoque. Implemente os conceitos de herança, polimorfismo, classe abstrata e redefinição de métodos, mostrando o que significam e onde eles foram implementados no seu código.

Obs: Faça a interface gráfica você mesmo!!

LabXLII: Interfaces Gráficas

Construa um sistema orientado a objetos, utilizando programação em três níveis, que atenda a venda de bilhetes de cinema. Neste cinema existem 3 salas grandes, 4 médias e 5 pequenas, cada uma com respectivamente 500, 300 e 100 lugares. Cada sala tem uma identificação que corresponde as 10 sessões durante o dia, que começam uma hora da tarde e acabam dez horas da noite. Bilhetes vendidos para salas grandes independente do horário do filme custam R\$12,00, para salas médias eles custam R\$ 12,00 em horário noturno e R\$ 9,00 no vespertino e em salas pequenas nos dois turnos, vespertino e noturno, o valor do bilhete é de R\$ 9,00. Considere que não deve ser possível vender mais bilhetes do que cadeiras disponíveis e que nas salas grandes sempre deve ser deixado um percentual de 10% sem vender. Informe estas situações aos clientes. Informe quanto foi o lucro do cinema com a vendas dos bilhetes para a 12 salas e informe quantas pessoas estarão presentes em cada uma das salas (indique o nome da sala e a quantidade de bilhetes vendidos). Implemente os conceitos de herança, polimorfismo, classe abstrata e redefinição de métodos, mostrando o que significam e onde eles foram implementados no seu código.

Obs: Faça a interface gráfica você mesmo!!

LabXLIII: Interfaces Gráficas

Crie uma especialização (RandomProva) da class Random existente na biblioteca java.util implementando uma sobrecarga do método nextInt para que este retorne um valor inteiro dentro de intervalo passado por parâmetro[x, y]. Em seguida crie uma classe Loteria que deve possuir um objeto ArrayList de números inteiros que foram sorteados utilizando a classe RandomProva. Além de sortear 10 números entre 10 e 100 colocando-os no objeto ArrayList (evitando duplicação de valores), a classe Loteria deve ser capaz de informar a sequência de números sorteados, informar se um dado número foi sorteado e informar se

um jogo (representado por 10 números) foi premiado. Para testar se as classes elaboradas estão corretas solicite inicialmente que a Loteria sorteie os números do Jogo "Nosso Jogo Milionário". Permita em seguida que o usuário digite os 10 números que ele jogou para que a loteria possa informar se ele ganhou ou não o prêmio. No final imprima os números sorteados. Olhe os métodos da classe Random e da classe ArrayList para responder esta questão.

Obs: Faça a interface gráfica você mesmo!!

LabXLIV: Interfaces Gráficas

Crie uma classe ZeroOuUm que herde de uma classe Jogo refletindo a lógica de um jogo Zero ou Um. Inicialmente jogadores são cadastrados. A lógica do jogo exige que cada jogador informe valores 0 ou 1 (aleatórios utilizando um objetos da classe Random para isso) ganhando o jogador que tiver resultado diferente dos demais. Crie uma sobrecarga do método jogar que recebe a quantidade de jogadores que participarão do jogo e os seus nomes em um vetor de String. Crie um método para retornar em linhas distintas o nome do vencedor e a rodada que cada jogador saiu do jogo conforme o exemplo abaixo:

Para um jogo com 4 jogadores

Mario – vencedor

Luis e Andre – perdedor - 2º rodada

Pedro – perdedor - 1º rodada

Crie também um método que permita acompanhar cada jogada do jogo.


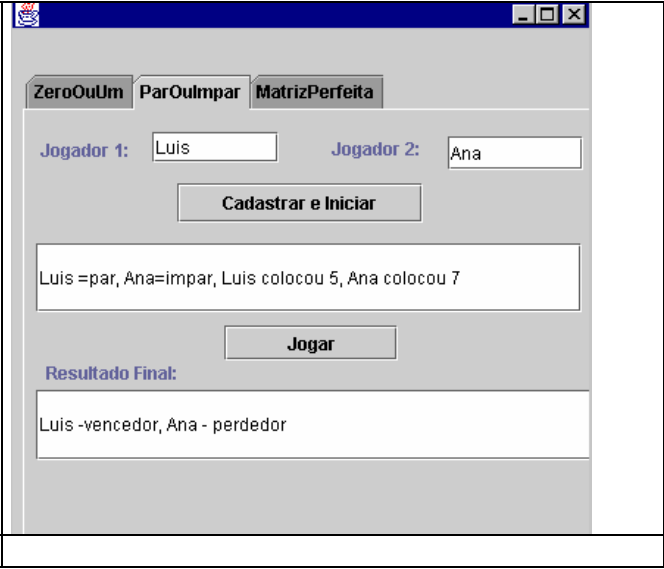
Crie uma classe ParOuImpar que também herde da classe Jogo e reflita a lógica de um jogo par ou impar entre dois jogadores. Defina em um método separado, de forma aleatória, quem será par e quem será impar. Informe o vencedor e o perdedor em linhas distintas conforme o exemplo abaixo:

Mario - vencedor

Luis – perdedor

Crie uma classe MatrizPerfeita que herde da classe Jogo e que implemente a lógica de uma matriz perfeita. O Jogador ganha o Jogo em uma matriz perfeita quando a soma de todos os valores existentes nas colunas e linhas de uma matriz [3 x 3] são iguais (veja exemplo abaixo). No início do jogo 3 valores aleatórios, de 0 a 9, em 3 posições aleatórias e distintas são sorteados para inicializar a matriz. Em seguida, o usuário preenche as posições restantes da matriz e solicita o resultado do jogo. Informe o nome do jogador e se ele perdeu ou ganhou.

<table><tr><td>0</td><td>4</td><td>2</td></tr><tr><td>4</td><td>1</td><td>1</td></tr><tr><td>2</td><td>1</td><td>3</td></tr></table> <div><div> </div><div>6</div><div> </div><div>6</div><div> </div><div>6</div></div> <div>= 6</div>	0	4	2	4	1	1	2	1	3	<p>Neste exemplo os valores aleatórios inicializados na matriz estão nas posições [0,0], [1,1] e [2,0]</p> <p>Resultado deste Exemplo: Luis – vencedor</p>
0	4	2								
4	1	1								
2	1	3								
<table><tr><td>0</td><td>6</td><td>2</td></tr><tr><td>8</td><td>3</td><td>1</td></tr><tr><td>2</td><td>1</td><td>7</td></tr></table> <div><div> </div><div>10</div><div> </div><div>10</div><div> </div><div>10</div></div> <div>= 8</div> <div>= 13</div> <div>= 10</div>	0	6	2	8	3	1	2	1	7	<p>Neste exemplo os valores aleatórios inicializados na matriz estão nas posições [0,0], [1,1] e [1,2]</p> <p>Resultado do Exemplo: Luis - perdedor</p>
0	6	2								
8	3	1								
2	1	7								

	
--	---

<div> <div>ZeroOuUm</div> <div>ParOuImpar</div> <div>MatrizPerfeita</div> </div>	
<div>Jogador: <input type="text" value="Jane"/></div>	
<div> <div>0</div> <div>4</div> <div>2</div> </div>	<div>Jogar</div>
<div> <div>4</div> <div>1</div> <div>1</div> </div>	
<div> <div>2</div> <div>1</div> <div>3</div> </div>	
<div>Resultado Final:</div> <div><input type="text" value="Jane - vencedor"/></div>	

LabXLV: Interfaces Gráficas

A Administradora de cartão de Crédito SEMAT emite mensalmente a fatura dos seus clientes. Essa fatura contém o valor a pagar pelas compras de um mês no cartão, o nome do titular do cartão e número do cartão. O valor a pagar representa a soma de todos os débitos (compras) abatidos dos créditos (bônus, pagamentos adiantados etc.) durante aquele período. O cliente pode optar por pagar o mínimo que é 10% do valor total mais encargos de 12% do saldo residual. Faça um programa que usando a interface gráfica abaixo, calcule o valor a pagar da fatura de um cliente e imprima esse valor por extenso. Ex.: Total=R\$ 12,37 -> Doze reais e trinta e sete centavos. Obs: Assuma que o valor total não pode ser superior a R\$ 1.000,00(mil reais). Sua solução deve ser escrita na linguagem Java, implementando os conceitos de orientação a objetos vistos em sala de aula. Preocupe-se com a semântica da sua solução.

Fatura do Mês -SEMAT

Nome

N. Cartão

Incluir Cliente

Débitos

Incluir Débito

Créditos

Incluir Crédito

Pagamento Total

Pagamento Mínimo

Valor:

LabXLVI: Interfaces Gráficas

Faca um programa orientado a objetos para comparar o desempenho dos dois pilotos da equipe Ferrari de formula 1 em uma corrida. O programa deve possuir uma interface gráfica que possibilite o cadastro dos pilotos e dos dados relativos a uma corrida. Ao final deve ser exibido na área de texto os resultados do piloto melhor colocado o seu nome e a sua velocidade média. Para cada piloto devem ser cadastrados: nome, número de voltas percorridas, volta mais rápida(tempo em segundos), o tempo total de corrida(em segundos). Da corrida deve-se obter o numero total de voltas, o tamanho em KM do circuito e o país onde ela ocorre.

Sistema de Resultados - Ferrari

Nome Tempo

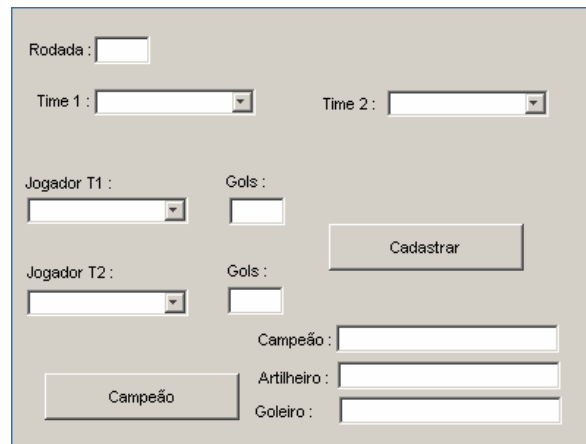
N Voltas Volta Mais Rápida

País Tamanho

N Voltas

LabXLVII: Interfaces Gráficas

O campeonato brasileiro é composto de 24 times quem se enfrentam ao longo do ano em 46 rodadas. Cada time pode ter até 23 jogadores(goleiros, atacantes, zagueiros...) inscritos no torneio. Cada partida distribui três pontos para o vencedor ou um ponto para cada time em caso de empate. Ao final das 46 rodadas o campeão será aquele que atender aos seguintes critérios de desempate: 1) Número de pontos; 2) Partidas vencidas; 3) Saldo de gols 4) Gols marcados 5) Gol sofridos. Escreva uma aplicação OO em Java que usando a interface abaixo que diga qual foi o Campeão brasileiro de 2003, o artilheiro e o goleiro mais "vasado". Assuma que o usuário vai digitar corretamente os dados de todas as rodadas.



Rodada :

Time 1 : Time 2 :

Jogador T1 : Gols :

Jogador T2 : Gols :

Campeão :

Artilheiro :

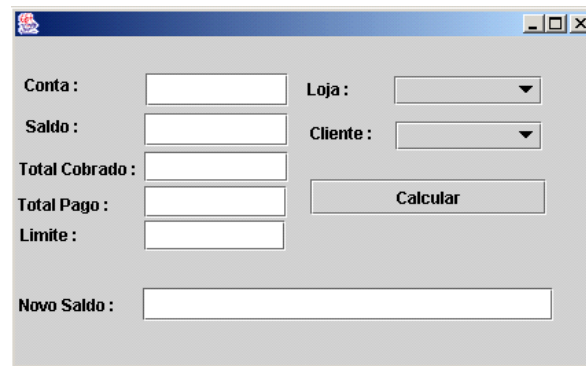
Goleiro :

LabXLVIII: Interfaces Gráficas

Desenvolva um aplicativo Java que gerencia as vendas de uma rede de lojas. O programa determina se um cliente de uma loja excedeu o limite de crédito em uma conta-corrente virtual da rede de lojas. Para cada cliente, os seguintes fatos estão disponíveis:

- a) Número da conta
- b) Saldo no início do mês
- c) Total dos itens cobrados desse cliente no mês
- d) Total de pagamentos feitos pelo cliente no mês
- e) Limite autorizado de crédito

O programa deve ler cada um desses fatos a partir da interface abaixo. Quando uma loja da rede for selecionada, os clientes dessa loja devem ser carregados no jcombobox de clientes. O programa Deve calcular o novo saldo (= saldo inicial + cobranças - pagamentos), exibir o novo saldo e determinar se o novo saldo excede o limite de crédito do cliente de cada uma das lojas especificadas na interface. Para aqueles clientes cujo limite de crédito for excedido, o programa deve exibir a mensagem, "Limite de crédito excedido". Defina para sua solução as classes Rede, Loja e Cliente.



Conta : Loja :

Saldo : Cliente :

Total Cobrado :

Total Pago :

Limite :

Novo Saldo :