Fundamentos de Programação Java e Orientação a Objetos (OO)

Este documento destina-se a organizar a apresentação do conteúdo referente a Java e OO da disciplina de Programação II do Centro de Educação Superior do Alto Vale do Itajaí (CEAVI/UDESC). O aluno não deve utilizar este documento como única fonte de aprendizado, devendo buscar material e conteúdo nos livros citados ao final deste documento e no plano de ensino da disciplina.
Última atualização deste documento: 16-07-2013 17:00

Capítulo 1 – Fundamentos de Programação Java

Estrutura básica de um programa Java:

```
public class MeuPrograma {

    public static void main(String[] args) {
        // aqui é o ponto de partida do programa;
    }
}
```

Os tipos primitivos em Java são:

- **byte**: valores inteiros entre -128 e +127
- **short**: valores inteiros entre -32.768 e +32.767
- **int**: valores inteiros entre -2.147.483.648 e 2.147.483.647
- **long**: valores inteiros entre -9.223.372.036.854.775.808 e 9.223.372.036.854.775.807
- **float**: valores reais
- **double**: valores reais de maior precisão/capacidade
- **boolean**: valores booleanos (true/false)
- **char**: caracteres

Para armazenar cadeias de caracteres, o Java oferece a classe **String**;

Operadores aritméticos:

+, -, *, /, %

Operadores relacionais:

- == igual a
- != diferença
- > maior que
- < menor que
- >= maior ou igual a
- <= menor ou igual a
- .equals igualdade entre strings

Operadores lógicos:

- && E (conjunção)
- || OU (disjunção)
- ! NÃO (negação)

Entrada e saída de dados:

```
public class MeuPrograma {

    public static void main(String[] args) {
        String palavra = JOptionPane.showInputDialog("Informe Palavra");
        System.out.println("A palavra digitada foi: "+palavra);
    }
}
```

Conversão de String para...

- int Integer.parseInt(String s)
- float Float.parseFloat(String s)
- double Double.parseDouble(String s)
- char variavelString.charAt(0)
- boolean Boolean.parseBoolean(String s)

Estruturas de Seleção

Simples

```
if ( condição ) {
      instruções que são executadas se condição for verdadeira
}
```

Composta

```
if ( condição ) {
      instruções que são executadas se condição for verdadeira
}else{
      instruções que são executadas se a condição for falsa
}
```

Aninhada

```
if ( condição1 ) {
      ...
      if ( condição2 ) {
            ...
            if ( condição3 ) {
                  ...
             }
             ...
      }else{
            ...
      }
      ...
}else{
      ...
      if ( condição4 ) {
            ...
      }
      ...
}
```

Estrutura de Seleção Múltipla

```
switch (<variável ou expressão>) {
   case <valor 1> :
          bloco de instruções 1
        break;
   case <valor 2> :
        bloco de instruções 2
        break;
   .
   .
   .
   default :
```

```
      bloco de instruções default
}
```

Estruturas de Repetição

Repetição com teste no início:

```
while ( condição ) {
      instruções que são executadas
      enquanto condição for verdadeira
}
```

Repetição com teste no fim:

```
do {
      instruções que são executadas
      enquanto condição for verdadeira
} while ( condição )
```

Repetição com variável de controle:

```
for( inicialização ; comparação ; incremento/decremento ){
      bloco de instruções que serão repetidas.
}
```

Exemplo:

```
for( int numero = 0; numero <= 100; numero++ ){
      System.out.println(Valor número: " + numero);
}
```

Boas práticas de programação em Java.

As boas práticas de programação apresentam modos adequados de codificar um programa. Estas práticas são adotadas por desenvolveres Java em todo o mundo, para facilitar o entendimento de códigos fontes.

As boas práticas serão adotadas ao longo da disciplina para resolver exercícios e trabalhos, sendo também consideradas como **critérios de avaliação para correção das provas e trabalhos**.

Boa prática 1: Nomes de arquivo e de classes.
O nome do arquivo e da classe devem iniciar sempre com letra maiúscula.
Se for um nome composto, então cada parte do nome também deve iniciar com maiúsculas.

Exemplos:
ExercicioProgramacao.java ⇒ public class ExercicioProgramacao (correto)
ExercicioComNomeGrande.java ⇒ public class ExercicioComNomeGrande (correto)
exercício.java ⇒ public class exercício (errado, deve iniciar com letra maiúscula)
Exercicio_Clientes ⇒ public class Exercicio_Clientes (errado, não se deve usar underline "_")

Boa prática 2: Nomes de variáveis, métodos e parâmetros.
O nome de variáveis, métodos ou parâmetros, deve começar com uma letra minúscula, e na sequência pode conter outras letras, números, ou o caracter sublinhado "_".

Se for um nome composto, a inicial do primeiro nome deve ser minúscula, e dos demais nomes deve ser maiúscula.

Exemplos:
posicao1
posicao2
linhaInicial
PontosTotais ⇒ (errado, deve iniciar com letra minúscula)
nomerobo ⇒ (errado, só a inicial da primeira parte [nome] é que deve ser minúscula)

valor_salario ⇒ (errado, não se deve usar underline "_")
Calcular_Fatorial() ⇒ (errado, métodos devem deve iniciar com letra minúscula, sem "_")

Exercícios

1) Faça um programa Java que será a urna eletrônica na próxima eleição para reitor da UDESC. Considere que haverá quatro candidatos. Primeiramente, o programa deve solicitar o nome de cada candidato. Em seguida, o programa lê os votos dos eleitores. Cada eleitor vota informando um número, de acordo com a seguinte codificação:

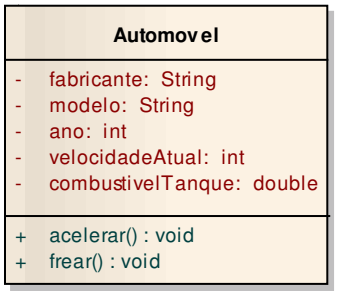
- 1, 2, 3, 4: voto para os respectivos candidatos;
- 5: voto em branco;
- 6: voto nulo;

Ao ser informado o número 0, a eleição é finalizada.
Ao fim da eleição, o programa deve informar:

a) total de votos de cada candidato;
b) total de votos nulos;
c) total de votos em branco;
d) percentual de nulos + brancos em relação ao total de votos
e) o nome do(s) candidato(s) vencedor(es) [em caso de empate, informe todos os nomes empatados – desconsidere votos nulos/brancos].

Capítulo 2 – Orientação a Objetos em Java

Objetos



No mundo real, um objeto é algo palpável, que possui características e pode ser manipulado.

A construção de programas torna-se mais natural e intuitiva quando representamos os objetos dentro dos nossos programas, do mesmo modo como eles são em nosso mundo real.

A representação dos objetos é feita por suas características e comportamentos, refletindo o modo como são no mundo real. Isto significa que podemos manipular os objetos no programa assim como manipulamos objetos no mundo real:

- podemos alterar características de objetos
- podemos ativar comportamentos do objeto

Um programa orientado a objetos é construido a partir da criação e manipulação de objetos.

Na orientação a objetos, as características dos objetos são chamadas de atributos, e seus comportamentos são chamados de métodos.

Classes

Objetos do mundo real não são construídos a partir "do nada"...

- Uma **casa** é construída a partir de uma **planta**
- Um **carro** é construído a partir de um **desenho de engenharia**
- Um **bolo** é construído a partir de uma **receita**

Nos programas, os objetos também não podem ser construídos "do nada". É necessário elaborar um **modelo (tipo) de objeto.** Este modelo de objeto é chamado de **CLASSE**.

Em uma classe, definimos quais serão os atributos e métodos dos objetos.

Somente após definir a classe é que podemos **construir** os objetos. O ato de construir um objeto é chamado de **instanciar** um objeto.

A partir de uma classe podemos instanciar vários objetos. Os objetos instanciados a partir de uma classe possuem os mesmos atributos e métodos, porém, os atributos podem ter **valores diferentes**.

Representação de classes de acordo com a UML (Unified Modeling Language): uma caixa dividida em três partes: a primeira parte contém conter o **nome** da classe, a segunda contém os **atributos** da classe e a terceira parte contém os **métodos** da classe.

Visibilidade de atributos e métodos e a representação no diagrama de classes UML:

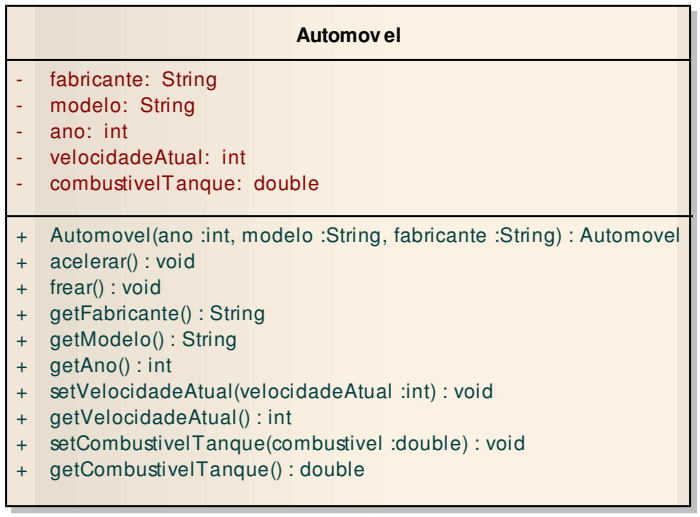
Visibilidade	Símbolo	Descrição
private	-	é acessado **apenas dentro da classe** onde foi declarado
protected	#	é acessado **dentro da classe** onde foi declarado e nas **classes descendentes**.
public	+	é acessado **dentro da classe** onde foi declarado, nas **classes descendentes e** em **qualquer classe externa**
package	~	é similar a public, porém é acessada apenas por classes que estejam no **mesmo pacote**

Classes: encapsulamento

Ensapsular significa proteger. Encapsular os atributos é boa prática da programação orientada a objetos. Ao encapsular os atributos, apenas o próprio objeto será capaz de alterar o valor de seus atributos. O objetivo é manter o objeto em ordem e consistente. Já imaginou se qualquer pessoa pudesse alterar uma característica sua, por exemplo, seu cabelo? O encapsulamento evita isso.

Se um objeto deseja permitir que seus atributos sejam alterados, a sua classe deve disponibilizar **métodos de acesso**. Convencionou-se que estes métodos iniciam com os prefixos get (quando o método recupera uma informação do objeto) e set (quando o método altera uma informação do objeto).

Na classe Automóvel os atributos estão encapsulados, pois todos foram declarados como private. A classe disponibiliza métodos de acesso get/set para alterar os valores dos atributos.



Classes: construtores

Construtores são métodos especiais, que constróem (instanciam) objetos de uma classe a partir do operador `new`. Podemos criar nossos próprios construtores nas classes. Criamos um construtor quando precisamos inicializar atributos ou ativar métodos durante a criação do objeto.
Regras para criar construtores:

- Devem ter exatamente o mesmo nome da classe
- A assinatura não pode declarar nenhum retorno, nem mesmo void

A classe Automóvel possui um construtor, que recebe como parâmetro o ano, modelo e fabricante.

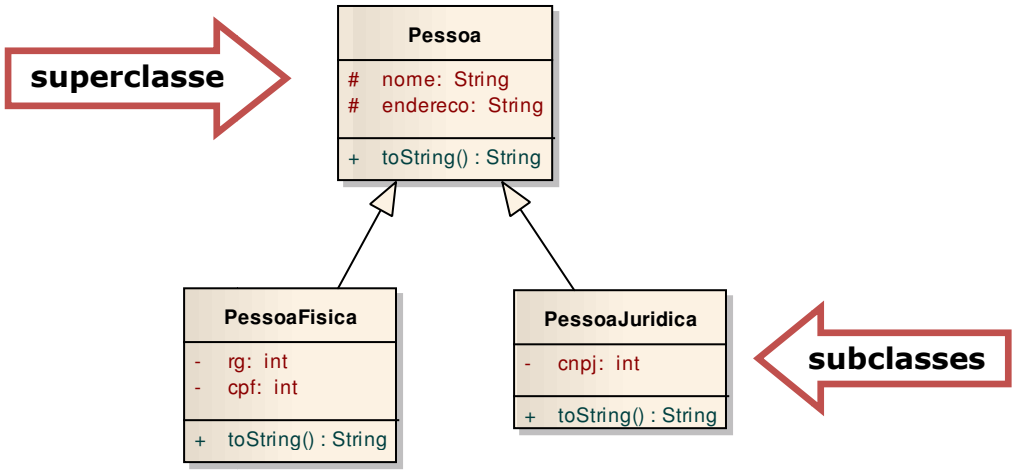
Exercícios

1) Crie um projeto, chamado **RevisaoOO**. Crie a classe Automovel apresentada no diagrama acima, com seus atributos e métodos. O método acelerar deve verificar se há combustivel, aumentar a velocidade do automóvel e diminuir a quantidade de combustível. O método frear deve diminuir a velocidade do automóvel. Em seguida crie uma classe TesteAutomovel, com o método main(), que deverá fazer o seguinte: (i) criar um Automovel gol; (ii) criar um automóvel celta; (iii) colocar 30 litros de gasolina no tanque do gol; (iv) colocar 10 litros de gasolina no tanque do celta; (v) acelerar o gol; (vi) imprimir os detalhes do gol; (vii) acelerar o celta; (viii) imprimir os detalhes do celta; (ix) frear o gol; (x) frear o celta; (xi) imprimir os detalhes dos dois automóveis.

Herança

A herança é um relacionamento de **generalização**, que estabelece uma hierarquia de **superclasses** e **subclasses**, onde as subclasses especializam superclasses mais gerais. A idéia de generalização vem do fato de que a superclasse é uma generalização de subclasses.

Subclasses herdam atributos e métodos da superclasse.



Em Java, subclasses são criadas utilizando a palavra extends, seguida do nome da superclasse.

```
public class PessoaFisica extends Pessoa{....}
```

Polimorfismo

Polimorfismo significa que um objeto pode assumir diversas formas (tipos) durante a execução do programa. O polimorfismo permite tratar objetos de modo mais geral ou mais específico, conforme a necessidade. Consequentemente, programas que utilizam polimorfismo são menores e mais simples.

A chave do polimorfismo está na herança. Objetos podem ser tratados como sendo do tipo de sua classe ou de suas superclasses.

Classes Abstratas

São classes que representam conceitos bastante gerais (abstratos), que não existem de fato no mundo real. Uma classe abstrata não pode ser instanciada. Seu propósito é fornecer uma superclasse apropriada, que contém atributos e métodos comuns, a partir da qual outras classes podem **herdar** estes atributos e métodos, **complementando-os**.

Classes abstratas podem conter **métodos abstratos** Um método abstrato não possui corpo, apenas assinatura. É o modo como o design orientado a objetos assegura que toda subclasse possuirá o método sem que seja necessário implementar um corpo vazio ou "sem sentido" para o método na superclasse.

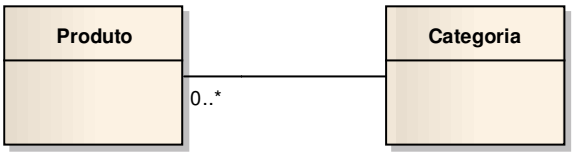
Relacionamentos entre classes

Além da herança, classes podem se relacionar de outras formas.

Associação

Relacionamento simples entre duas classes. Define que uma classe "**está associada**" a outra. Logo, em tempo de execução, os objetos destas classes também estarão associados. Em UML a associação é representada por uma linha sólida.

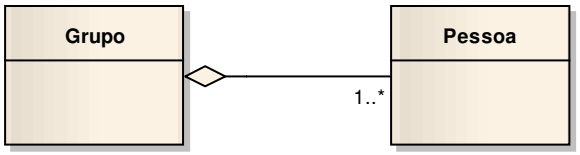
Na figura ao lado, a classe Categoria está associada com Produto. É possível especificar a **cardinalidade** da associação. A cardinalidade indica quantos objetos de uma classe estarão associados à outra classe. Por exemplo, em tempo de execução um objeto Categoria estará associado à 0..* (zero ou mais) Produtos.



Agregação

Relacionamento entre duas classes com o conceito de **todo-parte**: a classe todo possui partes, e as partes fazem parte de um todo. É representado na UML por uma linha sólida com um losango ao lado da classe agregadora. Em um relacionamento de agregação, um objeto parte pode fazer parte de vários objetos todo em tempo de execução.

Na figura ao lado, há um relacionamento de agregação entre Grupo e Pessoa: um grupo agrega várias pessoas. Ao mesmo tempo, o relacionamento de agregação permite que uma pessoa faça parte de vários grupos ao mesmo tempo. Em uma agregação, também é possível especificar a cardinalidade: um Grupo agrega 1..* (uma ou mais) Pessoas.

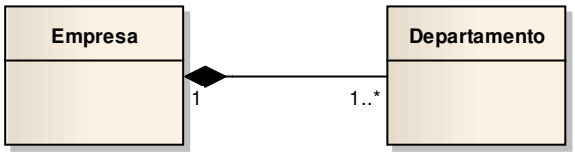


Composição

Relacionamento entre duas classes com o conceito **forte** de **todo-parte**. A classe todo possui partes, e as partes fazem parte de um **único** todo. Logo, a existência do objeto parte está condicionada a existência do objeto todo: se o todo for excluido, suas partes também são.

O relacionamento de composição traz a idéia de É representado na UML por uma linha sólida com um losango preenchido ao lado da classe compositora.

Na figura ao lado, há um relacionamento de composição entre Empresa e Departamento: uma empresa é composta por 1..* (um ou mais) Departamentos. Ao mesmo tempo, um departamento compõe apenas uma empresa. Note que o relacionamento de composição não permite que um objeto Departamento componha outros objetos Empresa, e a existência dos objetos de Departamento está condicionada a existência da Empresa.



Implementação dos relacionamentos de associação, agregação e composição.

Do ponto de vista de implementação, estes relacionamentos são todos implementados da mesma forma: através de atributos nas classes. Caso a cardinalidade exiga, estes atributos deverão ser listas ou mapas.

Tratamento de Exceções

Uma exceção é uma situação não usual, estranha ou inesperada, que ocorreu durante a execução e que pode causar falhas no programa. Para tornar o programa tolerante a falhas (que não pára ao ocorrerem falhas), as exceções devem ser **tratadas**. No tratamento de exceção, deve-se especificar quais ações serão tomadas para evitar a falha do programa.

Para tratamento de exceção, o Java disponibiliza as instruções **try/catch**. Uma instrução *try* pode conter um ou mais blocos *catch*, para capturar as exceções que eventualmente ocorrerem.

Exemplo de tratamento de exceção:

```
int numero;
try{
  numero = Integer.parseInt(JOptionPane.showInputDialog("Digite um NUMERO"));
  System.out.println("O número digitado: "+numero);
}catch(NumberFormatException excecao){
  System.out.println("O valor digitado NÃO É NÚMERO");
}
```

Ao ocorrer uma exceção, o bloco *try* termina imediatamente e o controle do programa é transferido para o primeiro dos blocos *catch* em que o tipo do parâmetro de exceção corresponde ao tipo da exceção ocorrida. Depois do tratamento, a execução do programa continua após o último bloco *catch*.

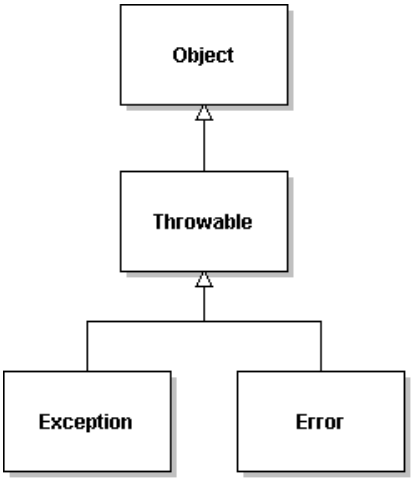
Hierarquia de exceções em Java

Throwable: é a superclasse de todo e qualquer erro ou exceção. Somente Throwable e suas subclasses podem ser utilizadas no catch(...).
Exception (e subclasses): tipo de exceção que representa uma situação de exceção "tratável", ou seja, em que o programa tem como se recuperar e prosseguir a execução. Exemplos:

- ArithmeticException (divisão por zero)
- NullPointerException (acesso a referência nula)

Error (e subclasses): tipo de exceção que representa uma situação anormal/séria, em que na maioria dos casos não há como ser tratada pelo programa. Exemplos:

- OutOfMemoryError (acabou a memória do Java)
- StackOverflowError (infinitas chamadas recursivas)



As figuras a seguir apresentam algumas subclasses de Exception e de Error:

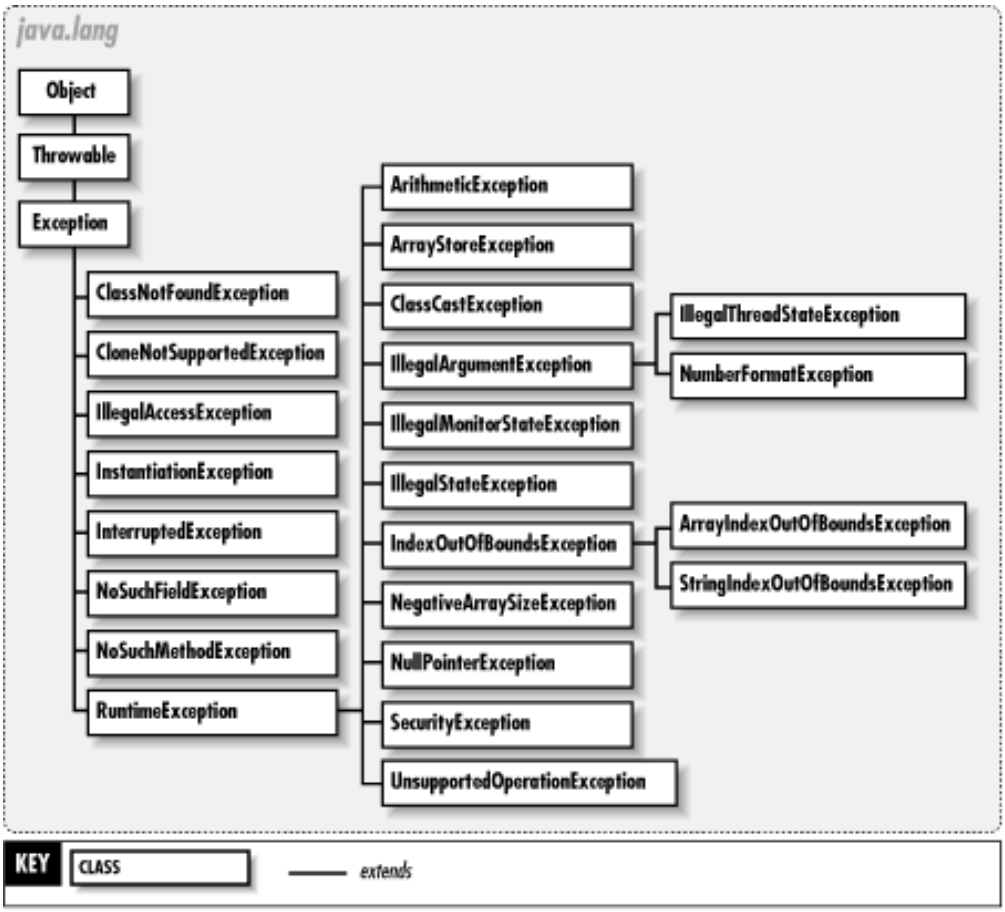


Figura 1 - Algumas subclasses de Exception (fonte: http://docstore.mik.ua/orelly/java-ent/jnut/ch12_01.htm)

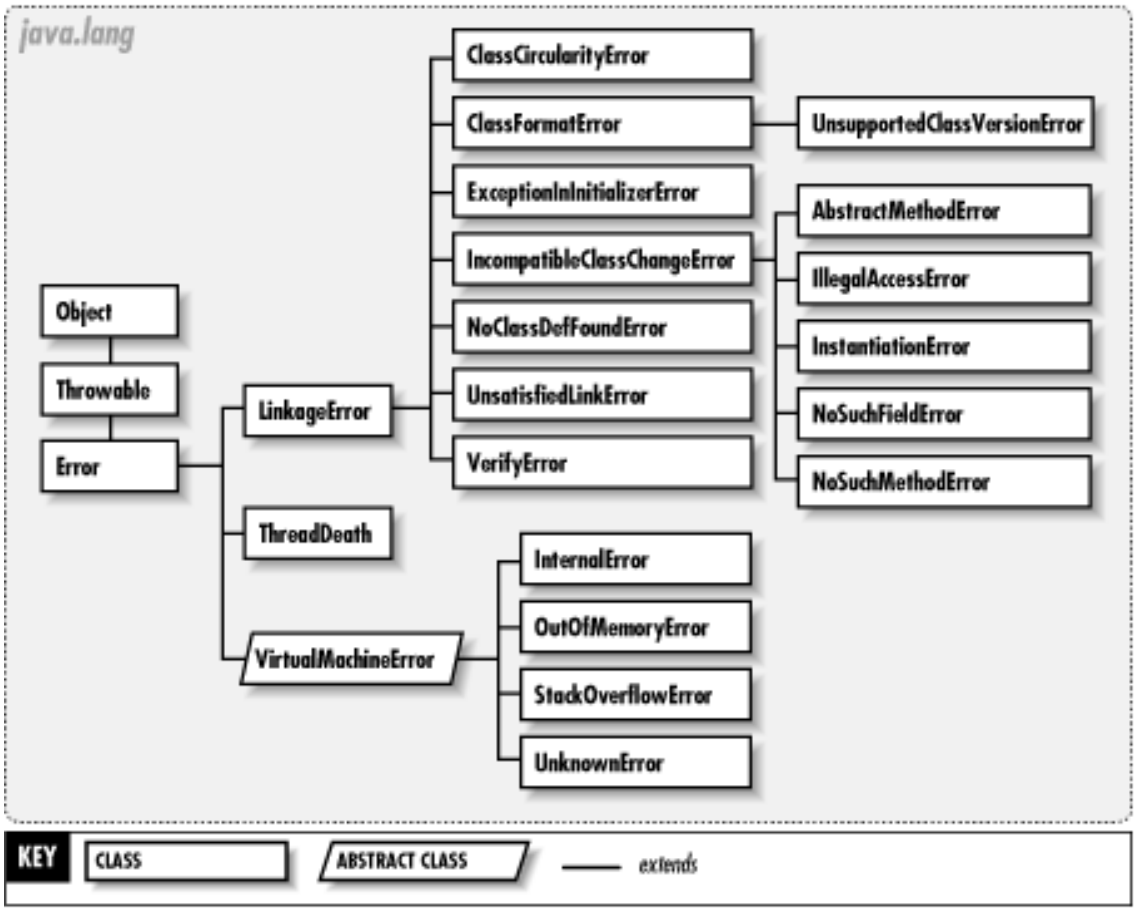


Figura 2 - Algumas subclasses de Error (fonte: http://docstore.mik.ua/orelly/java-ent/jnut/ch12_01.htm)

O Java agrupa as suas exceções em dois grandes conjuntos: exceções checadas e exceções não checadas.

Exceções checadas
São aquelas cujo tratamento pelo desenvolvedor é **obrigatório**.
Todas as exceções tratadas são subclasses de **Exception**. A ferramenta de desenvolvimento (ex: NetBeans), quando detecta possível ocorrência de exceção checada, acusa erro de compliação, obrigando o uso de um bloco try/catch ou da cláusula throws (ignorar a exceção e repassá-la adiante).
Exemplos de exceções tratadas: IOException, ClassNotFoundException, FileNotFoundException, etc.

Exceções não checadas.
O tratamento pelo desenvolvedor é **opcional**.
Todas as exceções não tratadas são subclasses de **RuntimeException**.
Exemplos de exceções não tratadas: NumberFormatException, ArithmeticException, IndexOutOfBoundException, NullPointerException, etc.

Tratamento de Exceções – bloco finally

Todo bloco de tratamento de exceção *try* pode possuir, ao final, um bloco *finally*. As instruções codificadas dentro do *finally* sempre serão executadas, ocorrendo ou não exceção. Em geral, um bloco *finally* é utilizado para liberar recursos utilizados dentro do bloco *try* (ex: fechar arquivos abertos, fechar conexões com o banco...) evitando vazamento de recursos e estouro de memória.

```
try {
    // comandos que podem gerar exceção
} catch (TipoDeExceção1 excecao) {
    // tratamento do TipoDeExceção1
```

```
} catch (TipoDeExceção2 excecao) {
    // tratamento do TipoDeExceção2
} finally {
    // instruções a serem executadas, ocorrendo ou não alguma exceção.
}
```

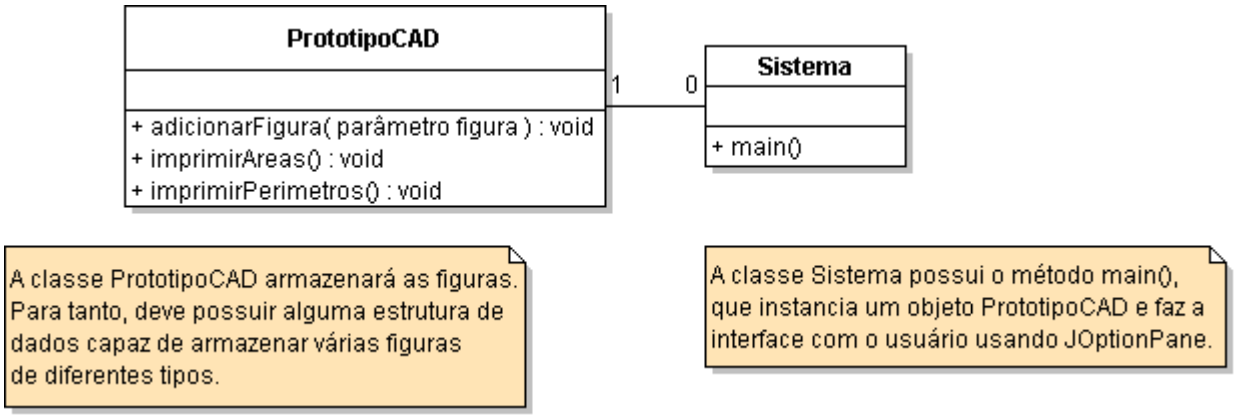
Capítulo 3 – Exercícios Adicionais

PrototipoCAD

PrototipoCAD é um programa que deve ser criado para manipular figuras geométricas. No momento ele será apenas um protótipo, mas se funcionar corretamente a empresa tem intenção de comercializar o produto (o nome sugerido pela gerência foi AutoCAD).

A versão solicitada do PrototipoCAD deve permitir cadastrar figuras geométricas. As figuras suportadas são: quadrado, retângulo, círculo e triângulo. Cada figura possui uma cor (cores possíveis: azul, verde, amarelo e branco), além dos dados necessários para calcular sua área e perímetro.

O PrototipoCAD deve possuir opções para incluir figuras, para imprimir áreas, e para imprimir perímetros. A interface com o usuário pode ser em modo texto ou gráfico, mas deve ficar em uma classe chamada Sistema, que conterá o método main(). **Importante**: a estrutura de dados que armazena as figuras deve ficar na classe PrototipoCAD. A classe Sistema deve instanciar um objeto PrototipoCAD e utilizar seus métodos para incluir as figuras e imprimir as informações quando solicitadas. O diagrama de classes abaixo apresenta o relacionamento entre as classes PrototipoCAD (com seus métodos) e a Sistema.



PEDE-SE:

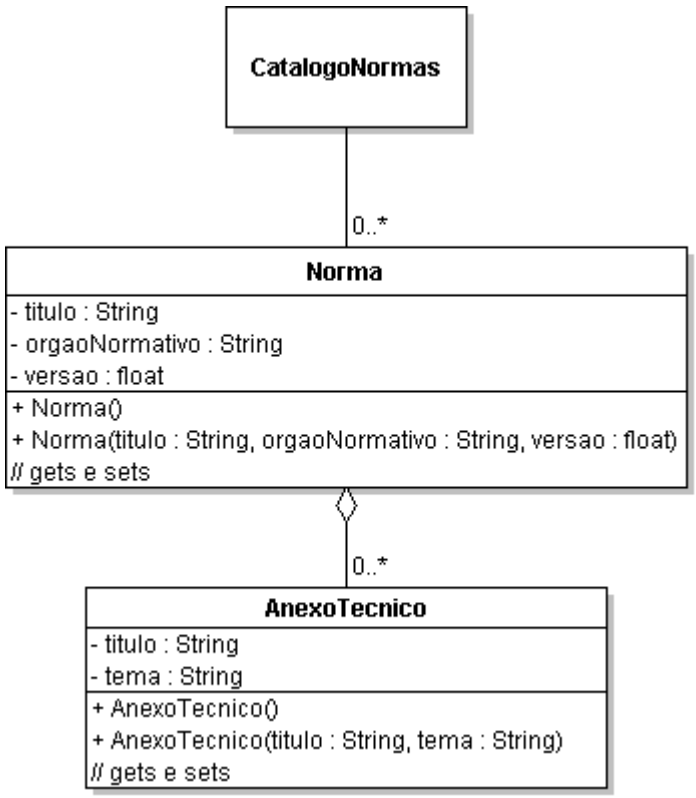
a) Identifique quais classes serão necessárias para resolver o exercício;
b) Faça um diagrama de classes, com as classes identificadas e seus atributos, bem como com os relacionamentos entre estas classes;
c) Codifique em JAVA o PrototipoCAD, seguindo o diagrama de classes elaborado. Implemente o método main() na classe Sistema para fazer a interface com o usuário, mostrando um menu com opções para incluir figura, imprimir areas e imprimir perimetros.

Fórmulas para calcular área e perímetro:

Quadrado área = $lado^2$ perímetro = 4 * lado	**Retângulo** área = base * altura perímetro = base + base + altura + altura
Círculo área = $\pi * raio^2$ perímetro = $2 * \pi * raio$	**Triângulo** (área é calculada pelo teorema de heron) s = (lado1 + lado2 + lado3) / 2; área = $\sqrt{s * (s - lado1) * (s - lado2) * (s - lado3)}$ perímetro = lado1 + lado2 + lado3

Parecer Técnico

Uma empresa está melhorando seus processos de software através da implantação de normas internacionais de qualidade de software. Uma norma pode conter diversos documentos complementares, conhecidos como anexos técnicos, que apresentam determinados temas. Para auxiliar a implantação, está sendo desenvolvido um sistema que organize os documentos das normas adquiridas pela empresa com a seguinte estrutura de classes:



PEDE-SE:

a) Implemente as classes e métodos solicitados.
b) Crie uma classe chamada Sistema, que deve conter o método main(). O sistema deve permitir cadastrar normas e imprimir as normas cadastradas. Para tanto, deve exibir um menu contendo as opções:
    1. Cadastra Norma: deve solicitar os dados da norma, e também os dados dos seus anexos técnicos, para cadastrar a norma.
    2. Imprimir normas: imprime os dados das normas cadastradas (apenas das normas)
    3. Imprimir normas com anexos: imprime os dados das normas cadastradas juntamente com seus anexos.

Observação:

- A entrada e saída de dados deve ser realizada apenas na classe Sistema.

Receita Federal (versão 1.0)

Construir um sistema para cadastrar declarações de imposto de renda. Cada contribuinte possui cpf, nome e valor da renda mensal, e um método para calcular o imposto a pagar. O imposto a pagar é calculado com base na tabela do IR apresentada ao lado.

Renda Mensal	% de Imposto
até 1.434,59	**isento**
de 1.434,60 até 2.150,00	**7,5%**
de 2.150,01 até 2.866,70	**15,0%**
de 2.866,71 até 3.582,00	**22,5%**
acima de 3.582,00	**27,5%**

O seu sistema deve possuir uma classse ReceitaFederal, com uma estrutura de dados para armazenar os contribuintes. Esta classe deve oferecer métodos para realizar as seguintes ações:

- Incluir contribuinte
  - solicita cpf, nome e valor da renda do contribuinte;
  - armazena o contribuinte;
- Retificar declaração:
  - solicita cpf do contribuinte;
  - se o contribuinte existir, então solicita a nova renda e altera no contribuinte;
  - se o contribuinte não existir, verifica se deseja incluir um novo, solicitando os demais dados;
- Imprimir declaração de contribuinte
  - solicita cpf do contribuinte;
  - imprime a declaração, cpf, nome e valor a pagar;
- Imprimir todas as declarações
  - imprime todas as declarações armazenadas.

O seu programa deve possuir uma classe Sistema, onde será implementado o método main(). Este método deve oferecer um menu para o usuário escolher qual das ações acima ele deseja executar.

PEDE-SE:

a) Identifique quais classes serão necessárias para resolver o exercício;
b) Faça um diagrama de classes, com as classes identificadas e seus atributos, bem como com os relacionamentos entre estas classes;
c) Codifique em Java o sistema seguindo o diagrama de classes elaborado.

Observação:

- A entrada e saída de dados pode ser realizada dentro de cada método da classe ReceitaFederal (isto é: incluirContribuinte(), retificarDeclaracao(), etc...)

Receita Federal (versão 2.0)

Modifique o sistema da receita federal desenvolvido no exercício anterior, para que em cada contribuinte seja possível cadastrar seus **dependentes**.

De cada dependente, basta cadastrar seu cpf e nome. Ao calcular o imposto a pagar, cada dependente cadastrado para o contribuinte resulta em um abatimento de R$ 150.00.

A classse ReceitaFederal deve continuar com uma estrutura de dados para armazenar os contribuintes. Nesta classe, deve ser adicionado um método para incluir dependente em contribuinte, que solicita cpf e nome do dependente, e adiciona no seu contribuinte. As impressões devem ser alteradas para que, ao imprimir um contribuinte, seja impresso também os dados de seus dependentes.

Bibliografia

BARNES, David J. **Programação Orientada a Objetos com Java:** Uma Introdução Prática Utilizando o Blue J. David J. Barnes, Michel Kölling. São Paulo: Pearson Prentice Hall, 2004. Número de chamada: 005.11 B261p.

DEITEL, Paul J; DEITEL, Harvey M. Java: como programar.8. ed. São Paulo: Pearson, 2010. xxix, 1144 p, il.

SANTOS, Rafael. **Introdução à Programação Orientada a Objetos Usando JAVA**. Rio de Janeiro: Campus, 2003. Número de chamada: 005.11 S237i.

SIERRA, Kathy; BATES, Bert. **Use a cabeça!**: Java.2. ed. Rio de Janeiro : Alta Books, c2007. xxvi, 470 p, il.

ARNOLD, Ken; GOSLING, James; HOLMES, David. **A linguagem de programação Java**.4. ed. Porto Alegre : Bookman, 2007. 799 p.

HORSTMANN, Cay S. **Big Java**. Porto Alegre : Bookman, 2004. xi, 1125 p, il. +, 1 CD-ROM.