Faculdades Integradas Espírito-Santenses
Unidade de Computação e Sistemas
Cursos: Ciência da Computação e Sistemas de Informação
Disciplina: **Linguagem de Programação II** - 2010/1
Prof. Henrique Monteiro Cristovão
Prof. Tiago Wirtti

Última atualização: 03-02-2010

Notas de Aula

Aqui você encontra os conteúdos da disciplina Linguagem de Programação II na ordem em que serão trabalhados. Eles foram escritos de forma resumida, por isso é importante a complementação do estudo nas bibliografias indicadas. Mas, lembre-se que para efetiva construção do seu conhecimento é imprescindível o desenvolvimento dos exercícios indicados pelo professor para cada assunto trabalhado.

Não esqueça de:

- trocar idéias com o professor e com os colegas, tanto no laboratório quanto no meio virtual (lista de discussão – endereço para cadastro disponível no site);
- fazer imediatamente os exercícios indicados para aquele assunto trabalhado afim de compreender melhor as abstrações da programação orientada a objetos .
- procurar o monitor da disciplina em caso de dúvida.

Obs.: este material é dinâmico e pode ser alterado ao longo do período letivo – observe a data da última atualização no início desta página.

Bibliografia recomendada:

CAELUM. **Java e orientação a objetos**. Disponível em:

<http://www.caelum.com.br/downloads/apostila/caelum-java-objetos-fj11.pdf>.

POO nos capítulos 4,6,7,9,10,11,12.

Java básico no capítulo 3.

SANTOS, Rafael. **Introdução à programação orientada a objetos usando JAVA**. Rio de Janeiro: Campus, 2003.

DEITEL, H. M.; DEITEL, P. J. **Java**: Como Programar. **6**. ed. Porto Alegre: Bookman, 2005.

Wikipédia, a enciclopédia livre. Disponível em <http://pt.wikipedia.org/>

4.1TEORIA: Tratamento de erros

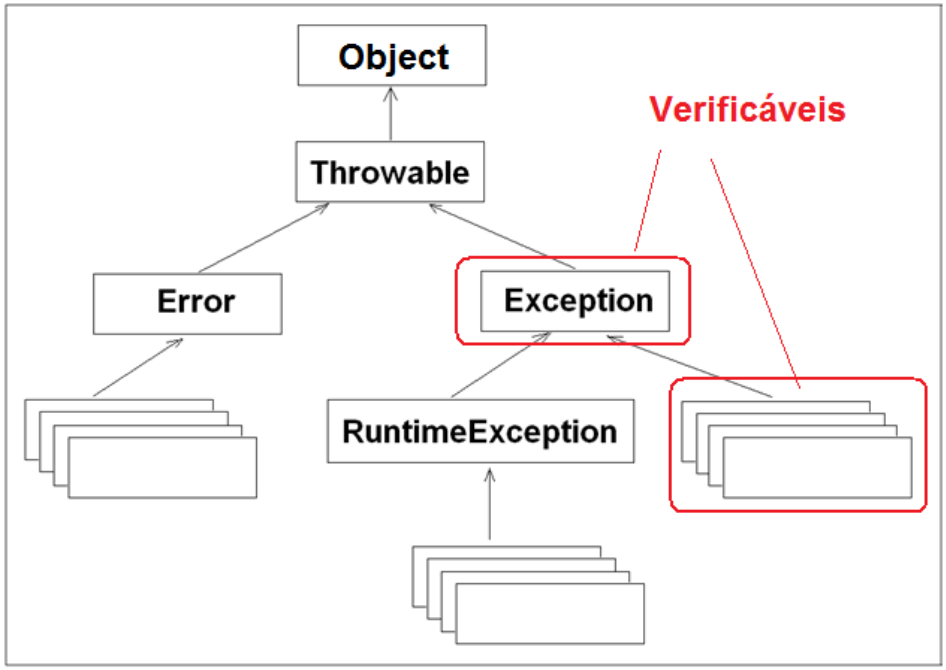
Em programação, erro (ou exceção) é qualquer evento que causa uma interrupção do fluxo normal de execução de uma seqüência de código.

O programador de classes deve implementar a sua classe de tal forma que o seu funcionamento não fique prejudicado caso a mesma seja utilizada incorretamente.

Algumas soluções inadequadas para o problema do tratamento de erros:

1. retorno de valor boolean para indicar o erro
2. retorno de valor fora da faixa de escopo de trabalho do método
3. exibição de mensagem de erro
4. parada do processamento
5. substituir o valor inválido por outro válido
6. indicar o erro num campo criado somente para este propósito

A LP Java trata erros de forma mais direta e elegante. Em Java todas as exceções são classes que implementam a interface **Throwable**.



As exceções são classificadas nos seguintes tipos:

- Verificáveis (*checked exceptions*):
  - Devem ser explicitamente tratadas no código;
  - ***Exception*** e suas subclasses (exceto ***RuntimeException*** e subclasses);
- Não verificáveis (*unchecked execptions*):
  - Podem ser ignoradas durante o processo de codificação;
  - ***Error***, ***RuntimeException*** e suas subclasses;

Como veremos a seguir, os comandos try-catch-throw-throws oferecem uma solução mais adequada para o tratamento de erros:

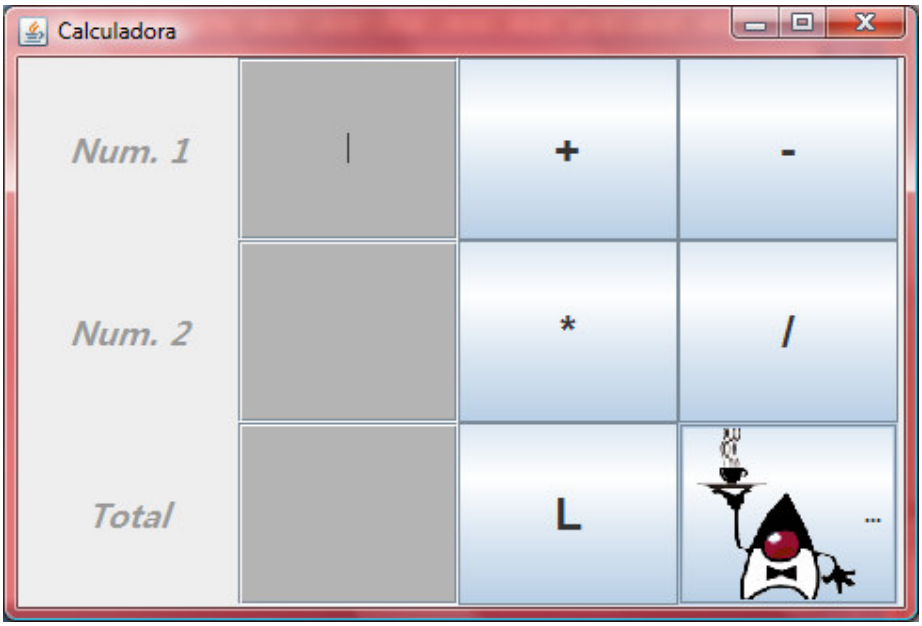
Exemplo: Tratamento de erros com try-catch-throw-throws.

```
package exemplo1;

public class TratamentoErros{
	public static void main(String args[]){
		try{
			int num1 = Integer.parseInt(args[0]);
			int num2 = Integer.parseInt(args[1]);
			if (num2 == 0){
			// não verificável

				throw new ArithmeticException("ERRO DE DIVISÃO POR ZERO");
			}
			else{
				System.out.println("num1/num2 = " + div(num1, num2));
			}
		}
		catch (ArithmeticException e){
			System.out.println(e.getMessage());
		}
		catch (ArrayIndexOutOfBoundsException e) {
			System.out.println("Número de parâmetros insuficiente");
		}
		catch (NumberFormatException e){
			System.out.println("Digite apenas numeros inteiros!!!");
		}
		finally {
			System.out.println("O finally sempre é executado!!!");
		}
	}
	public static int div(int n1, int n2) throws ArithmeticException {
			return n1/n2;
	}


}
```

4.2PRATICA: Tratamento de erros

a) Para a calculadora a seguir, faça o tratamento dos erros que podem ocorrer no método “actionPerformed”

b) Ttrate os erros com janelas de popup e mensagens informando ao usuário os procedimentos corretos:



Código do método actionPerformed (disponível também no projeto “Conceitos-Modulo-04”, pacote exemplo1:

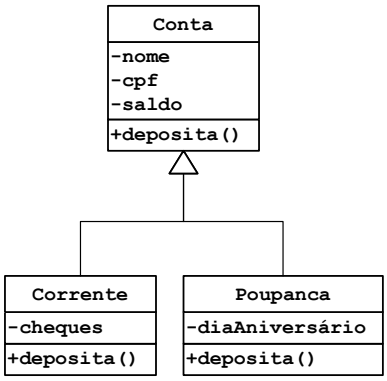
```
public void actionPerformed(ActionEvent e) {
	// TODO Auto-generated method stub
	// limpa campos
	if (e.getSource()==jButton_Limpa){
		jTextField_Num1.setText("");
		jTextField_Num2.setText("");
		jTextField_Resultado.setText("");
		return;
	}
	float N1=0,N2=0,result=0;
	// trata entrada de dados (divisor nulo)
	N1 = Float.parseFloat(jTextField_Num1.getText());
	N2 = Float.parseFloat(jTextField_Num2.getText());
	if (N2==0){
		jTextField_Resultado.setText("Erro!");
		JOptionPane.showMessageDialog(null, "Divisão por ZERO", "ERRO",
		JOptionPane.ERROR_MESSAGE);
		// lance uma exceção aqui
	}
	// tratar erro de entrada não numérica

	// processa operações
	if (e.getSource()==jButton_Soma) {
		result = N1 + N2;
		System.out.println("Botao +");
	}
	if (e.getSource()==this.jButton_Subtrai) {
		result = N1 - N2;
		System.out.println("Botao -");
	}
	if (e.getSource()==this.jButton_Multiplica) {
		result = N1 * N2;
		System.out.println("Botao x");
	}
	if (e.getSource()==this.jButton_Divide) {
		result = N1 / N2;
		System.out.println("Botao /");
	}
	this.jTextField_Resultado.setText(""+result);
}
```

c) Melhore a GUI desta aplicação.

4.3TEORIA: Classes abstratas

Como problema motivador, suponha a necessidade de representar informações de contas correntes e contas poupança. Vamos também supor que os algoritmos do depósito sejam diferentes para contas corrente e contas poupança.

Já visando o reaproveitamento de código, através de **herança**, e também o uso **polimórfico** do método que realiza o depósito, poderíamos organizar estas classes da seguinte forma:



Ainda sim existem dois problemas nesta solução:

1º) um usuário desavisado poderia criar um objeto da classe Conta, e gerar transtornos uma vez que não faz sentido um objeto desta classe.

2º) numa futura extensão deste sistema, por exemplo, na criação da classe Investimento, que também possui a relação "é uma Conta", um outro usuário descuidado poderia esquecer de implementar o método deposita() ou então implementá-lo com outro nome como fazDeposito(). Neste caso, numa chamada polimórfica do método deposita, o algoritmo da classe pai é que seria executado.

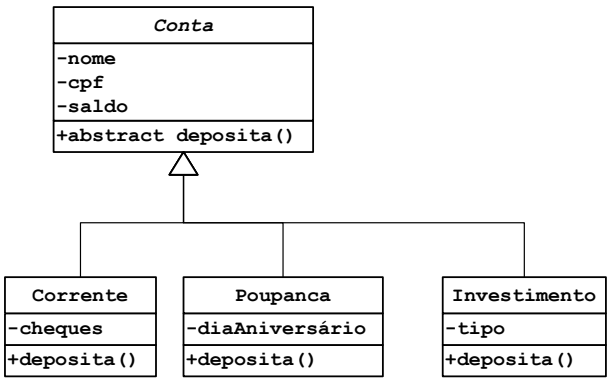
Soluções dos problemas:

- para se prevenir do primeiro problema podemos declarar a classe Conta como abstrata.
- para o segundo problema temos que declarar o método deposita() como abstrato. Métodos abstratos não possuem código, mas somente a declaração do seu cabeçalho.

Em ambos os casos o próprio compilador detecta os problemas e acusa erro.

Uma classe que possui um método abstrato também deve ser abstrata, pois senão um objeto seu poderia invocar a execução deste método sem algoritmo. Campos nunca são abstratos.

Exemplo:



No diagrama de classes UML uma classe abstrata é representada escrevendo seu nome em itálico. Ou então incluindo <abstract> abaixo do nome.

Exemplo de implementação de uma classe abstrata com método(s) abstrato(s):

```
abstract public class Conta {
   private String nome;
   private String cpf;
   private double saldo;

   abstract public void deposita(double _valor);

}
```

Como conseqüência destas declarações temos: a classe `Conta` não pode ser instanciada e todas as subclasses de `Conta` tem por obrigação a implementação do método `deposita`, ao menos que ele seja declarado como `abstract` também.

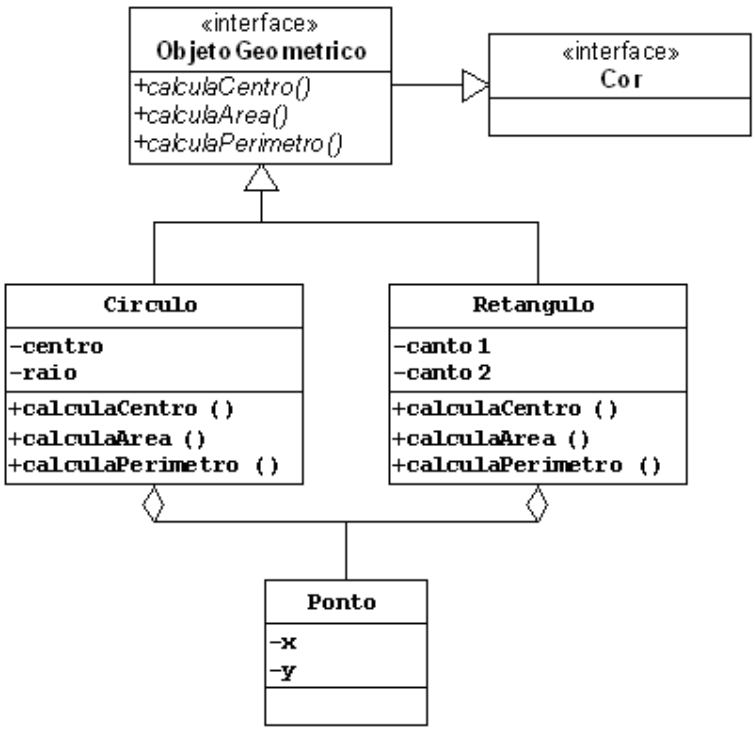
4.4TEORIA: Interfaces

Uma classe do tipo interface pode ser entendida como uma classe totalmente abstrata É como se fosse um contrato que diz o que as subclasses deverão fazer mas nada sobre como fazer.

Algumas características:

- Não é declarada com a palavra `class`, mas com `interface`.
- Os métodos são implicitamente declarados como `abstract` e `public`.
- Os campos são `static` e `final`, ou seja, só são permitidas constantes.
- A herança é feita com a palavra `implements` ao invés de `extends` e é permitida para várias classes simultaneamente.

Exemplo 1:



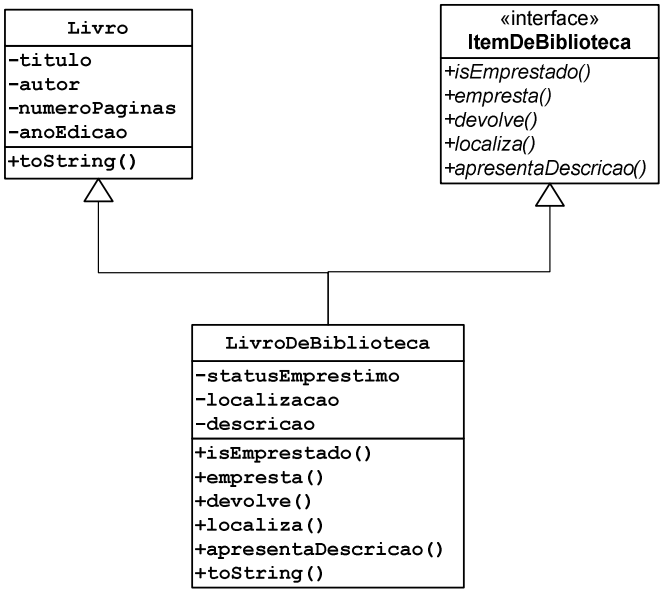
Obs.:

- a interface `ObjetoGeometrico` herda da interface `Cor`, que é uma classe formada por constantes com os códigos das cores. Esta também é uma aplicação de interface muito usada. As constantes ficam disponíveis em todas as subclasses de `ObjetoGeometrico`.
- Novos objetos geométricos, tais como triângulos e trapézios, representados por suas classes, herdarão da interface `ObjetoGeometrico` e serão obrigados a implementar os três métodos (`calculaCentro`, `calculaArea` e `calculaPerimetro`). Ales disto terão disponíveis os constantes dos códigos de cores.
- Herança de interface com interface deve ser escrita com `extends`.

Código:

```
public interface Cor {
   int VERMELHO = 234;
   int AZUL = 178;
   int AMARELO = 112;
}

public interface ObjetoGeometrico extends Cor {
      double calculaCentro();
      double calculaArea();
      double calculaperimetro();
}

public class Circulo implements ObjetoGeometrico {
      private Ponto centro;
      private double raio;

      public Ponto calculaCentro() {
            return this.centro;
      }
      public double calculaArea() {
            return Math.PI*this.raio*this.raio;
      }
      public double calculaPerimetro() {
            return 2*Math.PI*this.raio;
      }
}
public class Retangulo implements ObjetoGeometrico {
      // ...
}
```

Exemplo 2:



Obs.:

- Este é um caso de herança múltipla, pois `LivroDeBiblioteca` herda da classe Livro e da interface `ItemDeBiblioteca`.
- A interface `ItemDeBiblioteca` serve para moldar todos os novos itens (subclasses) que venham a aparecer neste sistema, e garantir uma padronização além diminuir a quantidade de problemas advindos de usos polimórficos.
- A declaração da classe `LivroDeBiblioteca` fica desta forma:

```
class LivroDeBiblioteca extends Livro implements ItemDeBiblioteca {
      // ...
}
```

A palavra `extends` deve ser escrita primeiro que `implements`.

4.5TEORIA: Pacotes e modificadores de acesso

O pacote é um agrupamento de código com a mesma finalidade. Um pacote é praticamente uma pasta onde você põe as classes que tem algo em comum. No Eclipse ele pode ser criado a partir do menu: File – New – Package. Uma pasta será criada com o nome do pacote.

Agora, com o conhecimento da abstração de pacote é que podemos apresentar de forma completa os quatro modificadores de acesso:

- Públicos (modificador ***public***):Visíveis dentro e fora do pacote
- Protegidos (modificador ***protected***):Visíveis dentro do pacote e fora apenas por herança
- Padrão (sem modificador):Visíveis apenas dentro do mesmo pacote
- Privados (modificador ***private***):Visíveis apenas dentro da própria classe

A visibilidade dos membros de uma capacote em relação a outros membros dentro e fora da classe e do próprio pacote é ditada pelas regras de visibilidade estabelecidas pelo uso dos modificadores de acesso.

Visibilidade	Público	Protegido	Padrão	Privado
Da mesma classe	**Sim**	**Sim**	**Sim**	**Sim**
De qualquer classe do mesmo pacote	**Sim**	**Sim**	**Sim**	**Não**
De uma subclasse do mesmo pacote	**Sim**	**Sim**	**Sim**	**Não**
De uma subclasse externa ao pacote	**Sim**	**Sim**	**Não**	**Não**
De uma classe C externa ao pacote que utiliza uma classe B (tb externa ao pacote), sendo B subclasse de uma classe A do pacote	**Sim**	**Não**	**Não**	**Não**

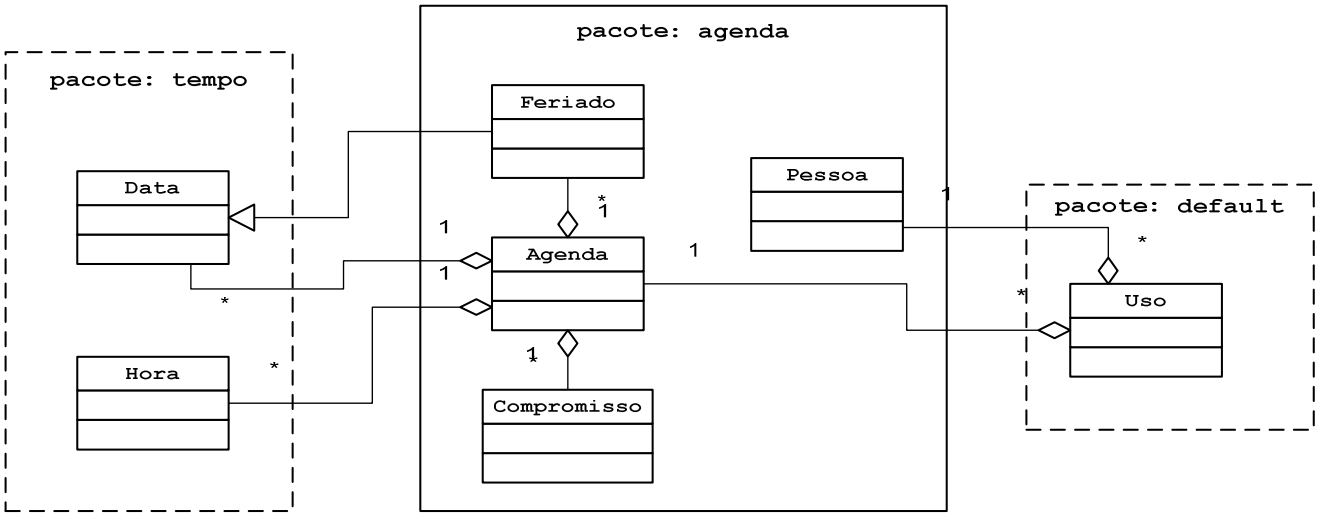
A declaração do pacote é feita no início do arquivo com a palavra reservada **package**:

```
package tempo;  // por convenção usa-se letra minúsculas
public class Data {
    ...
}
```

A utilização de uma classe em outro pacote é feita com a palavra reservada **import:**

```
import tempo.Data;  // permite a utilização da classe Data neste arquivo
    ...
```

Exemplo:



4.6PRÁTICA: Faça o que se pede usando as classes do pacote exemplo3, projeto "Conceitos-Modulo-04"

a) Mude as classes **Funcionario** e **Vendedor** para o pacote exemplo3.pacote1.
b) Mude a classe **ContratoVendedor** para o pacote exemplo3.pacote2.
c) Altere os modificadores de acesso dessas classes de forma que eles sejam visíveis na classe **ContratoVendedor**.
d) Altere os modificadores de acesso dos atributos da classe **Funcionario** para **protegido** e de **Vendedor** para **privado**.
e) Faça as alterações necessárias na classe **ContratoVendedor** para que o programa funcione corretamente.

4.7TEORIA: Enumerações

Uma enumeração é um conjunto de contantes que ajuda ao desenvolvedor definir valores fixos que serão usados no sistema. A enumeração dá controle e performance, ajudando a reduzir o número de erros na aplicação, pois o usuário terá um conjunto bem definido de valores possíveis para trabalhar.

As enums são derivadas da classe `java.lang.Enum` fornecidas na API do Java.

Entenda a aplicabilidade e a sintaxe através dos exemplos que se seguem.

Exemplo 1:

```
public enum DiaSemana {
    SEGUNDA, TERCA, QUARTA, QUINTA, SEXTA, SABADO, DOMINGO;
}

public class Uso {
    public static void main(String[] args) {
        System.out.println(DiaSemana.QUINTA);
    }
}
```

Saída na console:

```
QUINTA.
```

O **método `values()`** retorna um vetor de `DiaSemana` preenchido com os valores, por exemplo, o código:

```
DiaSemana[] vet = DiaSemana.values();
System.out.println(vet[0]);
```

provoca a saída:

```
SEGUNDA
```

Observe outro exemplo de uso desta enumeração, agora com o método `values()` conjugado com o `for`:

```
public class Uso {
    public static void main(String[] args) {
        for(DiaSemana dia : DiaSemana.values()) {
            System.out.println(dia);
        }
    }

}
```

Saída na console:

```
SEGUNDA
TERCA
QUARTA
QUINTA
SEXTA
SABADO
DOMINGO
```

Exemplo 2:

```
public enum Cor { BRANCO, PRETO, VERMELHO, AMARELO, AZUL; }

public class Figura {
      private Cor cor;
      private int tamanho;

      public Figura(Cor cor, int tamanho) {
         this.setCor(cor);
         this.setTamanho(tamanho);
      }
      public Cor getCor() {
            return cor;
      }
      public int getTamanho() {
            return tamanho;
      }
      public void setCor(Cor cor) {
            this.cor = cor;
      }
      public void setTamanho(int tamanho) {
            this.tamanho = tamanho;
      }
      // ...
      public String toString() {
            StringBuffer resultado = new StringBuffer();
            resultado.append(this.getTamanho());
            resultado.append(" - ");
            resultado.append(this.getCor());
            return resultado.toString();
      }
}

public class Uso {
      public static void main(String[] args) {
         Figura a = new Figura(Cor.AMARELO, 500);
         // ...
         a.setCor(Cor.VERMELHO);
         System.out.println(a);
      }
}
```

Saída na console:

```
500 - VERMELHO
```

A enumeração também pode ter campos. Veja uma extensão para a enum Cor onde tem-se a necessidade de armazenar o código das cores:

```
public enum Cor {
      BRANCO(45), PRETO(37), VERMELHO(12), AMARELO(98), AZUL(29);
      private int codigo;
      private Cor(int c) {
            this.codigo = c;
      }
      public int getCodigo() {
            return this.codigo;
      }
}
```

```
class Figura {
      private Cor cor;
      private int tamanho;

      public Figura(Cor cor, int tamanho) {
         this.setCor(cor);
         this.setTamanho(tamanho);
      }
      public Cor getCor() {
            return cor;
      }
      public int getTamanho() {
            return tamanho;
      }
      public void setCor(Cor cor) {
            this.cor = cor;
      }
      public void setTamanho(int tamanho) {
            this.tamanho = tamanho;
      }
      public String toString() {
            StringBuffer resultado = new StringBuffer();
            resultado.append(this.getTamanho());
            resultado.append(" - ");
            resultado.append(this.getCor());
            return resultado.toString();
      }
      // ...
}

public class Uso {
      public static void main(String[] args) {
            Figura a = new Figura(Cor.AMARELO, 500);
          // ...
            a.setCor(Cor.VERMELHO);
            // ...
            System.out.println("Objeto:         " + a.toString());
            System.out.println("Cor:            " + a.getCor());
            System.out.println("Tamanho:        " + a.getTamanho());
            System.out.println("Código da cor: " + a.getCor().getCodigo());
      }
}
```

Saída na console:

```
Objeto:         500 - VERMELHO
Cor:            VERMELHO
Tamanho:        500
Código da cor: 12
```

4.8PRÁTICA: Adaptando o relatório do projeto Paint para utilizar enumeração

a) Crie um tipo enumerado “TipoGeometrico”, como mostrado abaixo:

```
package lpii.paint.util;

public enum TipoGeometrico {
	LINHA, QUADRADO, RETANGULO, CIRCUNFERENCIA, ELIPSE;
}
```

b) Utilize este tipo enumerados nas chamadas do método toString de cada classe de geometria assim:

```
@Override
public String toString() {
	return TipoGeometrico.LINHA+"[p0=" + p0 + ", p1=" + p1 + "]";
}
```

Ao invés de

```
@Override
public String toString() {
	return "LINHA [p0=" + p0 + ", p1=" + p1 + "]";
}
```

c) Execute novamente os relatórios e observe os resultados.

d) Crie um tipo enumerado (TipoRelatorio) para substituir na classe Relatorio os textos `"ordemDesenho"`, `"ordemAlfabetica"`, `"ordemArea"` e `"ordemPerimetro"`. Utilize “TipoRelatorio” na classe Relatorio.

4.9TEORIA: Serialização de objetos

Objetos Java podem ser colocados à disposição num fluxo, por exemplo, para serem gravados num arquivo.

Para isto, a classe deve implementar a interface `Serializable` (`pacote Java.io`).

Normalmente classes da biblioteca Java já são implementações de `Serializable`, como por exemplo, `String`, `Date`.

Exemplo:

```
public class Data implements Serializable {
  ...
}
```

Gravação de objetos num arquivo.
Todas estas chamadas de métodos requerem tratamento de erros.
Use `IOException` ou outra classe derivada de `Exception`.

```
// cria um arquivo
ObjectOutputStream arq = new ObjectOutputStream(new FileOutputStream("arquivo.obj"));

// grava um objeto
arq.writeObject(objeto);

// descarrega os dados pendentes no buffer para o arquivo
```

```
arq.flush();

// fecha o arquivo
arq.close();
```

Leitura de objetos num arquivo.
Se a classe de recebimento dos dados não for uma implementação da classe `Serializable` deverá ocorrer o erro:
`java.lang.ClassCastException` em tempo de execução.

```
// Abertura do arquivo para leitura dos objetos
ObjectInputStream arq = new ObjectInputStream(new FileInputStream("arquivo.obj"));

// Lê um objeto
// Obs.:
//       - É necessário tratar a exceção classNotFoundException
//       - O objeto deve pertencer a uma classe que tenha implementado Serializable
//       - É necessário cast de conversão para dar forma ao objeto lido da classe Object
//       - A leitura é realizada na mesma ordem em que os objetos foram gravados (fila)
//       - Todos os objetos dependentes/agregados são também carregados
objeto = (tipo_do_objeto)arq.readObject();

// Fechamento do arquivo:
arq.close();
```

4.10PRATICA: Serialização de objetos na aplicação Paint

a) Usando a classe “SuporteArquivo” disponível no pacote **exemplo4** no projeto “Conceitos-Modulo-04”, faça o programa a seguir (execute e teste):

```
package exemplo4;

import java.io.File;

public class Teste {
      public static void main(String[] args) {
            // Serializando uma String
            String texto = new String("Texto do arquivo");
            // mude o caminho do arquivo conforme suas
            // necessidades usando \\ no lugar de \
            String nomeArquivo = "C:\\Tiago\\Faesa\\Disciplinas\\”+
            ”CC-LP-OFICIAL\\2010-1\\PROJETO-FINAL\\Conceitos-Modulo-04\\texto.ser";

            File arquivo = new File("C:\\Tiago\\Faesa\\Disciplinas\\”+
           ”CC-LP-OFICIAL\\2010-1\\PROJETO-FINAL\\Conceitos-Modulo-04\\texto.ser");
            SuporteArquivo.gravar(arquivo, texto);

            System.out.println("Verifique a pasta: “+
            "C:\\Tiago\\Faesa\\Disciplinas\\CC-LP-OFICIAL\\2010-1\\”+
            ”PROJETO-FINAL\\Conceitos-Modulo-04\\");
      }
}
```

b) Seu programa executou corretamente? Corrija-o (até que execute sem erros de caminho de arquivo) e execute novamente. Qual foi o resultado?

c) Agora verifique o conteúdo do diretório onde o seu arquivo foi serializado adicionando ao código anterior (dentro do main) o seguinte:

```
// verificando o conteúdo do diretório:
String nomeDir = "C:\\Tiago\\Faesa\\Disciplinas\\CC-LP-OFICIAL\\"+
"2010-1\\PROJETO-FINAL\\Conceitos-Modulo-04\\";
File diretorio = new File(nomeDir);
System.out.println(SuporteArquivo.listaArquivosPastaTrabalho(diretorio));
```

d) Lendo o arquivo serializado:

```
// Lendo o arquivo serializado
String conteudo = (String) SuporteArquivo.abrir(arquivo);
System.out.println(conteudo);
```