UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

DISCIPLINA INF01121 - MODELOS DE LINGUAGENS DE PROGRAMAÇÃO

Professor Leandro Krug Wives

Laboratório de linguagens imperativas 02 – Introdução à linguagem Java

1) **Relembrar uso do Netbeans (ver roteiro do primeiro laboratório de Java): criar projetos, pacotes e arquivos-fonte.**

2) **Relembrar como criar primeiro programa em Java: ola mundo.**
   a) Crie um projeto denominado OlaMundo.
   b) Crie uma classe no projeto, denominada OlaMundo, dentro do pacote olamundo.
   c) Coloque o seguinte código nela e execute-o:

```
package olamundo;

public class OlaMundo {

      public static void main(String[] args) {

         System.out.println("Olá Mundo!");

      }

}
```

3) **Métodos construtores: revisão e aprofundamento**

Lembra-se da classe Pessoa? Segue uma versão simplificada:

```
public class Pessoa {
      private String nome;
      private Integer idade;

      public void setNome(String nome){
            this.nome = new String(nome);
      }

      public String getNome(){
            return this.nome;
      }

      public void setIdade(int idade){
            this.idade = new Integer(idade);
      }

      public Integer getIdade(){
            return idade;
      }

}
```

Quando criamos objetos, o seu método construtor é chamado:

```
public class ProgramaExemplo1 {

   public static void main(String argumentos[]){
      Pessoa p1 = new Pessoa();

      System.out.println(p1.getNome() + " tem " + p1.getIdade() + " anos");
   }
}
```

No entanto, este método não foi definido na classe Pessoa que criamos. Quando isso acontece, a linguagem cria um método padrão para nós.

Vamos criar nosso próprio, para provar que ele é chamado:

```
public class Pessoa {
	private String nome;
	private Integer idade;

	public Pessoa(){
		System.out.println("> Nova instancia de Pessoa criada");
	}

	public void setNome(String nome){
		this.nome = new String(nome);
	}

	public String getNome(){
		return this.nome;
	}

	public void setIdade(int idade){
		this.idade = new Integer(idade);
	}

	public Integer getIdade(){
		return idade;
	}

}
```

Ao executar o ProgramaExemplo, você perceberá que o método construtor foi chamado, onde uma nova instância de Pessoa foi criada. No entanto, este método não inicializa os atributos da classe. Vamos alterá-lo:

```
public class Pessoa {
	private String nome;
	private Integer idade;

	public Pessoa(){
		System.out.println("> Nova instancia de Pessoa criada");
		this.setNome(“Indefinido”);
		this.setIdade(0);
	}

	public void setNome(String nome){
		this.nome = new String(nome);
	}

	public String getNome(){
		return this.nome;
	}

	public void setIdade(int idade){
		this.idade = new Integer(idade);
	}

	public Integer getIdade(){
		return idade;
	}

}
```

Agora, quando você executar o ProgramaExemplo, você perceberá que os atributos receberam valores-padrão.

O método construtor pode ter diversas formas (isso se chama polimorfismo ou sobrecarga). Isso significa que podemos criar diferentes métodos construtores, que recebem parâmetros diferentes. Veja o exemplo:

```
public class Pessoa {
	private String nome;
	private Integer idade;

	public Pessoa(){
		System.out.println("> Nova instancia de Pessoa criada");
		this.setNome(“Indefinido”);
		this.setIdade(0);
	}

	public Pessoa(String nome, int idade){
		System.out.println("> Nova instancia de Pessoa criada");
		this.setNome(nome);
		this.setIdade(idade);
	}


	public void setNome(String nome){
		this.nome = new String(nome);
	}

	public String getNome(){
		return this.nome;
	}

	public void setIdade(int idade){
		this.idade = new Integer(idade);
	}

	public Integer getIdade(){
		return idade;
	}

}
```

Agora podemos criar pessoas e já atribuir valores iniciais a elas:

```
public class ProgramaExemplo1 {

   public static void main(String argumentos[]){
      Pessoa p1 = new Pessoa(“Leonardo”, 21);

      System.out.println(p1.getNome() + “ tem ” + p1.getIdade() + “ anos”);
   }
}
```

Polimorfismo é um conceito importante, pois permite que você crie um método com diversos parâmetros diferentes, sobrecarregando-o. Isso pode ser feito não só no construtor, mas em qualquer método (menos nos operadores tradicionais, como +, -, / etc.).

OBS: em C++ você também pode sobrecarregar os operadores tradicionais da linguagem!

Leia a página de Marshall Cline sobre sobrecarga de operadores em c++: http://www.parashift.com/c++-faq-lite/operator-overloading.html

4) Herança: revisão e aprofundamento

Levando em conta a última versão da classe Pessoa acima, vamos definir uma nova classe chamada de aluno: um aluno é uma pessoa, mas que tem outros atributos. No nosso caso, ele terá um atributo extra: seu número de matrícula

```
public class Aluno extends Pessoa{

    private int matricula;

    public Aluno() {
```

```
        System.out.println("> Nova instancia de Aluno criada");
    }

    public void setMatricula(int matricula) {
        this.matricula = matricula;
    }

    public int getMatricula() {
        return matricula;
    }

}
```

Vamos também definir um outro programa-exemplo para testá-lo:

```
public class ProgramaExemplo2 {

   public static void main(String argumentos[]){

      Aluno p1 = new Aluno();

      System.out.println(p1.getNome() + " tem " + p1.getIdade() + " anos");
      System.out.println("seu código de matrícula é:" + p1.getMatricula());

   }
}
```

Perceba o uso de extends, que é uma palavra-reservada que indica que você está criando uma classe com base em outra já existente, herdando tudo que ela tinha. Analise os demais métodos e veja que você está acrescentando novas funcionalidades específicas para a classe filha: matrícula e seus métodos de acesso. Note também que os dois construtores foram chamados: o de Pessoa e o de Aluno!

Altere a classe Aluno da seguinte forma:

```
public class Aluno extends Pessoa{

    private int matricula;

    public Aluno() {
        System.out.println("> Nova instancia de Aluno criada");
    }

    public Aluno(String nome, int idade, int matricula)
    {
        this.setMatricula(matricula);
    }

    public void setMatricula(int matricula) {
        this.matricula = matricula;
    }

    public int getMatricula() {
        return matricula;
    }

}
```

Como só ajustamos o código da matrícula, os demais valores assumiram os valores definidos pelo construtor padrão de Pessoa! Podemos resolver isso chamando o construtor adequado, da seguinte forma:

```
public class Aluno extends Pessoa{

    private int matricula;

    public Aluno() {
        System.out.println("> Nova instancia de Aluno criada");
    }
```

```
    public Aluno(String nome, int idade, int matricula)
    {
        super(nome, idade);
        this.setMatricula(matricula);
    }

    public void setMatricula(int matricula) {
        this.matricula = matricula;
    }

    public int getMatricula() {
        return matricula;
    }

}
```

O método super() chama o construtor da classe-pai. Super também serve para chamarmos outros métodos específicos da classe pai. Veja o exemplo:

```
public class Aluno extends Pessoa{

    private int matricula;

    public Aluno(String nome, int idade, int matricula) {
        super.setNome(nome);
        super.setIdade(idade);
        setMatricula(matricula);
        System.out.println("> Nova instancia de Aluno criada");
    }

    public void setMatricula(int matricula) {
        this.matricula = matricula;
    }

    public int getMatricula() {
        return matricula;
    }

}
```

Herança é um conceito importante, pois permite o reuso de código. Você pode acrescentar funcionalidades a um componente já existente, estendendo-o. Pode também modificar um componente com problemas e todos os dependentes deles serão corrigidos. Veja:

```
public class Pessoa {
      private String nome;
      private Integer idade;

      public Pessoa(){
            this.setNome(“Indefinido”);
            this.setIdade(0);
      }

      public Pessoa(String nome, int idade){
            this.setNome(nome);
            this.setIdade(idade);
      }

      public void setNome(String nome){
            if(!nome.equals(“”)) this.nome = new String(nome);
      }

      public String getNome(){
            return this.nome;
      }

      public void setIdade(int idade){
            if(idade > 0) this.idade = new Integer(idade);
      }

      public Integer getIdade(){
            return idade;
```

```
        }
}
```

Exercício: elabore uma classe chamada funcionário que estenda a classe Pessoa. Esta classe deve ter um atributo denominado salário, protegido. Crie os construtores e os métodos de acesso de forma adequada. Elabore um programa que teste a classe definida por você, testando os construtores e os métodos, ajustando os atributos e mostrando o resultado na tela.

5) Composição

A composição de objetos difere da herança. Lá, você cria uma nova classe com base em outra, herdando seus métodos e atributos (características e comportamentos). A composição também permite o reuso de código, mas age de forma diferente, onde você utiliza uma classe dentro de outra. O relacionamento entre elas é do tipo: tem-um. Exemplo: um carro tem-um motor, tem-um conjunto de rodas, tem-um chassi, etc. O motor não é-um carro, mas sim, faz parte dele.

A composição é extremamente simples, bastando você criar objetos dentro de outros. Uma pessoa, por exemplo, possui um nome, que é um objeto String. Os objetos que participam da composição são declarados como atributos do objeto maior que os possui. Todos eles são criados no construtor do objeto maior.

Exemplo do Carro:

```
class Carro{
   Motor  motor;
   Chassi chassi;
   Vector rodas;

   public Carro() {
       motor = new Motor();
       rodas = new Vector();
       rodas.add(new Roda());
       rodas.add(new Roda());
       rodas.add(new Roda());
       rodas.add(new Roda());
       chassi = new Chassi();
       chassi.add(motor);
       chassi.add(rodas);
   }

   public String toString(){
      return new String("Motor: " + motor.getTipo());
   }
}

class Motor{
   String tipo;

   public Motor() {
     tipo = new String("1.0");
   }

   public String getTipo(){
      return tipo;
   }
}




class Roda{
   String tipo;

    public Roda() {
```

```
        tipo = new String(“Aro 14”);
    }
}

class Chassi{
   Motor motor;
   Vector rodas;

   public add(Motor motor){
      this.motor = motor;
   }

   public add(Vector rodas){
      this.rodas = rodas;
   }
}
```

class Exemplo{

```
    public static void main(String args[])
    {
      Carro c = new Carro();
      System.out.println( c );
    }

}
```

6) Polimorfismo

Polimorfismo vem de muitas formas. É a capacidade que a linguagem tem de permitir com que um método seja descrito de diversas formas, mas com diferentes parâmetros.

Veja o exemplo:

```
import java.util.*;

public class Oficina {

    Random r = new Random();

    public Veiculo proximo() {
        Veiculo v;
        int code = r.nextInt();
        if (code%2 == 0)
            v = new Automovel();
        else
            v = new Bicicleta();

        return v;
    }

    public void manter(Veiculo v) {
        v.vistoria();
        v.conserto();
        v.limpeza();
    }


    public static void main(String[] args) {
        Oficina o = new Oficina();
        Veiculo v;

        for (int i=0; i<5; ++i) {
            v = o.proximo();
            o.manter(v);
        }
    }

}
```

```
class Veiculo {

    public Veiculo() {
        System.out.print("Veiculo ");
    }

    public void vistoria() {
        System.out.println("Veiculo.vistoria");
    }
    public void conserto() {
        System.out.println("Veiculo.conserto");
    }
    public void limpeza() {
        System.out.println("Veiculo.limpeza");
    }
}

class Automovel extends Veiculo{

    public Automovel() {
        System.out.println("Automovel");
    }
    public void vistoria() {
        System.out.println(">> Automovel.vistoria");
    }
    public void conserto() {
        System.out.println(">> Automovel.conserto");
    }
    public void limpeza() {
        System.out.println(">> Automovel.limpeza");
    }

}

class Bicicleta extends Veiculo {

    public Bicicleta() {
        System.out.println("Bicicleta");
    }
    public void vistoria() {
        System.out.println(">> Bicicleta.vistoria");
    }
    public void conserto() {
        System.out.println(">> Bicicleta.conserto");
    }
    public void limpeza() {
        System.out.println(">> Bicicleta.limpeza");
    }

}
```

Neste caso, a oficina atende 5 veículos de diferentes tipos (automóveis ou bicicletas). Para simular isso, o componente Random foi utilizado. Se o número sorteado for par, considera-se um automóvel, caso contrário, uma bicicleta.

De forma independente do tipo do veículo, todos eles são vistoriados, consertados e limpos. No entanto, a forma como estes métodos funcionam são diferentes para cada veículo. Usando herança, podemos estabelecer um método básico para uma destas atividades e especializá-las nas classes filhas. A própria linguagem, de forma dinâmica, seleciona o método adequado de acordo com o tipo de objeto criado. Isso é chamado de polimorfismo.

Outro tipo de polimorfismo (mais comum ou simples) é o utilizado nos métodos construtores. Assim, temos um método com nome semelhante mas assinatura diferente (com parâmetros diferentes). O exemplo do carro na atividade 5 também usa polimorfismo: o método add (adicionar) do chassi tem duas formas: uma que adiciona motores ao chassi e outra que adiciona um vetor de rodas.

Exercício: Com as classes Pessoa, Funcionário e Aluno, crie um programa semelhante ao da oficina. Antes, no entanto, crie um método denominado mostrar em cada classe, que mostre o seu conteúdo (atributos). O programa deve criar 5 funcionários ou alunos de forma aleatória e imprimir o tipo da classe (veja a aula seguinte que aborda Reflexão). Além disso, deve chamar o método mostrar para que os objetos mostrem o seu conteúdo na tela.