Programação Orientada a Objetos

Prof. Dr. Josenalde Barbosa de Oliveira

josenalde.oliveira@ufrn.br

https://github.com/josenalde/apds

Tipos primitivos e classes Wrappers (empacotadoras)

- Classes que encapsulam tipos primitivos existentes e fornecem operações sobre estes
- Permite que tipos primitivos sejam tratados como objetos

Grupo numérico INTEIRO: byte (Byte), short (Short), int (Integer), long (Long)

Grupo numérico PONTO FLUTUANTE: float (Float), double (Double)

Grupo CARACTER: char (Char)

Grupo BOOLEANO: boolean (Boolean)

Não necessita ser instanciada (removido nas versões atuais Java):

```
Integer x = 1;
Float y = 4f;
```

classes Wrappers

- Checagens numéricas (`isNaN, isInfinite, compareTo, etc.`)

```
Integer a = 10;
Integer b = 20;
// 10<20, saída -1
System.out.println(a.compareTo(b));
```

- Conversões (parseInt, parseFloat, valueOf, intValue, toString, etc.)
- Tipo String não considerada classe Wrapper pois não existe tipo primitivo string em Java, no entanto possui vários construtores sobrecarregados, métodos de conversão de tipos e operações para lidar com caracteres e intervalos de dados (substrings, buscas etc.)

```
length() // comprimento da String
charAt(int index) // caractere na posição específica da String
concat(String str) // concatena duas strings (+)
equals (Object anObject) // compara se duas strings são iguais
valueOf(int i) // retorna String com base no valor de um inteiro
valueOf(float f)
valueOf(double d)
valueOf(boolean b)
lastIndexof(String str) // retorna o último índice da String
isEmpty() // retorna TRUE se possue tamanho 0
isBlank() // retorna verdadeiro se só possui espaços em branco
substring(int start[, int end]) // retorna String a partir do start até end
trim() // remove espaços em branco ao redor da string
split() // divide string com base em separador ou REGEX
indexOf(char c) // índice de determinado caracter
```

classe Scanner

- Facilita o uso de métodos sobre o InputStream
- next() (até encontrar caractere em branco)
- nextLine() (até encontrar \n)
- nextInt()
- nextFloat()
- nextDouble()
- nextBoolean()
- hasNextInt(), hasNextFloat(),hasNextDouble()... – verifica se próximo é...

```
Scanner cin = new Scanner(System.in);
String nome = cin.nextLine();
if (cin.hasNextInt()) Integer v1 = cin.nextInt();
if (cin.hasNextFloat()) Float v2 = cin.nextFloat();
if (cin.hasNextDouble()) Double v3 = cin.nextDouble();
if (cin.hasNextBoolean()) Boolean v4 = cin.nextBoolean();
```

O método toString (polimorfismo de sobrescrita)

- Seja o modelo UML de classes
  - Cliente (-nome:String): abstract
  - PessoaFisica(-cpf:String) extends Cliente
  - PessoaJuridica(-cnpj:String) extends Cliente

  - Pedido(-numero:int, -listaItens:List<Item>, -cliente:Cliente) associado unidirecional a Cliente (1 – 0..*) e unidirecional a Produto(-código:int, -nome:String, -preco:float)
  - Item(-produto:Produto,-quantidade:int) classe associativa a Produto e Pedido

```
public class Produto {
          //...
          @Override
          public String toString() {
                    String descProduto = "\nCodigo: " + codigo;
                    descProduto += "\nNome: " + nome;
                    descProduto += "\nPreço: " + preco;
                    return descProduto;
          }
```

O método toString (polimorfismo de sobrescrita)

- Classe abstrata Cliente e suas subclasses

```
public abstract class Cliente {
	//...
	@Override
	public String toString() {
		return "\Nome: " + nome;
	}
```

```
public class PessoaFisica extends Cliente {
	//...
	@Override
	public String toString() {
			String msg = super.toString();
			msg += "\nCPF: " + cpf;
			return msg;
	}
```

```
public class PessoaJuridica extends Cliente {
	//...
	@Override
	public String toString() {
			String msg = super.toString();
			msg += "\nCNPJ: " + cnpj;
			return msg;
	}
```

O método toString (polimorfismo de sobrescrita)

- Classe associativa Item

```
public class Item {
          //...
          @Override
          public String toString() {
                 String msg = "\n " + produto; // chama toString() de Produto
                 msg += "\nQuantidade: " + quantidade;
                 return msg;
          }
```

- A classe Pedido está associada à classe Cliente, logo toString de Pedido deve referenciar objeto Cliente tal que haja chamada implícita ao toString das classes concretas PessoaFisica e PessoaJuridica. O método toString deve também percorrer a lista de Itens.

```
public class Pedido {
          //...
          @Override
          public String toString() {
                    String msg = "\nNúmero: " + numero;
                    msg += cliente; // toString automático do cliente em questão
                    Iterator<Item> it = listaItens.iterator();
                    while (it.hasNext()) {
                        msg += it.next(); // chama toString de cada Item
                    }
                    return msg;
          }
```

O método toString (polimorfismo de sobrescrita)

- Como ficaria na classe Principal

```
public class Principal {
          public static void main(String[] args) {
                    Produto p1 = new Produto();
                    p1.setCodigo(1);
                    p1.setNome("Console XBOX");
                    p1.setPreco(3000);
                    Produto p2 = new Produto();
                    p2.setCodigo(2);
                    p2.setNome("tv led 32\"");
                    p2.setPreco(2000);
                    Item it1 = new Item();
                    it1.setProduto(p1);
                    it1.setQuantidade(2);
                    Item it2 = new Item();
                    it2.setProduto(p2);
                    it2.setQuantidade(3);
                    PessoaFisica c1 = new PessoaFisica();
                    c1.setNome("Pedro");
                    c1.setCPF("111.111.111-11");
                    Pedido ped1 = new Pedido();
                    pe1.setCliente(c1);
                    ped1.adicionarItem(it1);
                    ped1.adicionarItem(it2);
                    System.out.println(ped1);
          }
```

Tratamento de erros

- Uso do bloco try...catch...finally para capturar vários tipos de Erros

```
public class Principal {
          public static void main(String[] args) {
                    int x = 3;
                    int y = 0;
                    double res = Double.NaN;
                    try {
                         res = x / y; // bloco com lógica normal, funcionalidades
                    } catch (ArithmeticException e) { // se usar Exception é a superclasse
                         res = Double.NaN; // pode exibir o erro no objeto e
                         System.out.println(e);
                    } finally { // executado de qualquer forma, com ou sem erro
                         System.out.println("Resultado: " + res);
          }
          }
}
```

```
public class Principal {
          public static void main(String[] args) {
                    Pessoa pessoa = null;
                    try {
                         pessoa.setNome("John");
                    } catch (NullPointerException e) {
                         pessoa = new Pessoa();
                    } finally { // executado de qualquer forma, com ou sem erro
                         pessoa.setNome("John");
                    }
          }
}
```

Tratamento de erros

- Podemos criar quantos blocos catch achemos necessários para tratar exceções que podem ocorrer dentro do try, ao fim, ainda coloca-se um catch geral, com Exception e. [Não coloque Exception antes das exceções específicas, senão só capta a Exception geral, da superclasse]
- finally é importante para liberar recursos, fechar conexões com banco de dados, liberar tokens de segurança, ou simplesmente, executar instruções independente de ocorrer ou não exceção
- O finally só não é executado se dentro do bloco try há um método exit(). Neste caso os recursos são liberados diretamente pelo sistema operacional.
- Pode-se deslocar o tratamento para outro local, não necessariamente onde ocorre a exceção. Exemplo:

Tratamento de erros

```
public class Robo {
          public void calcularVelocidade(int tempo, int distancia) {
                    try {
                         this.velocidade = distancia / tempo;
                    } catch (ArithmeticException e) {
                         //notificar velocidade inválida
                    }
          }
}
```

```
public class Robo {
          public void calcularVelocidade(int tempo, int distancia) throws ArithmeticException {
                        this.velocidade = distancia / tempo;
          }
}
```

```
public class Principal {
          public static void main(String[] args) {
                    Robo r1 = new Robo();
                    try {
                      //ler tempo, ler dist
                      r1.calcularVelocidade(tempo, dist);
                    } catch (ArithmeticException e) {
                         // informar erro
                    } finally {
                        // informar que a velocidade atual do robô é r1.getVelocidade()
                    }
          }
}
```

Tratamento de erros

```
public class Jogador {
          public void setIdade(int idade) throws Exception {
                    Exception e = new Exception(“Jogador menor de 18 anos”);
                    if (idade < 18) throw e;
                    this.idade = idade;
          }
}
```

```
public class Principal {
          public static void main(String[] args) {
                    Jogador j = new Jogador();
                    try {
                      j.setIdade(7);
                    } catch (Exception ex) {
                          // informar erro
                    }
          }
}
```

Tratamento de erros – classe de exceções

```
public class IdadeJogadorException extends RuntimeException {
          public IdadeJogadorException(String message) {
                    super(message);
          }
}
```

```
public class Jogador {
          public void setIdade(int idade) throws IdadeJogadorException {
                    if (idade < 18) {
                       throw new IdadeJogadorException(“Idade menor que 18 anos”);
                    }
                    this.idade = idade;
          }
}
```

```
public class Principal {
          public static void main(String[] args) {
                    Jogador j = new Jogador();
                    try {
                      j.setIdade(7);
                    } catch (IdadeJogadorException ex) {
                          // informar erro ex
                    }
          }
}
```

Enumerações

```
public enum ForcaArmadaEnum {
          MB(“Marinha do Brasil”), EB(“Exército Brasileiro”), FAB(“Força Aérea Brasileira”);
          private final String descricao;
          private ForcaArmadaEnum(String descricao) {
                    this.descricao = descricao;
          }
          @Override
          public String toString() {
                    return descricao;
          }
}
```

```
public class Militar {
          private String nome;
          private ForcaArmadaEnum forca;
          // getters and setters
}
```

```
public class Principal {
          public static void main(String[] args) {
                    Militar m = new Militar();
                    // Entre com sua Força (MB, EB ou FAB)
                    // String forca = cin.nextLine();
                    militar.setForca(ForcaArmadaEnum.valueOf(forca));
          } // outra opção é usar MENU com opções e cada opção é um número começando em 0
}
```

Ver outros usos aqui: https://www.devmedia.com.br/enumeracoes-em-java/25839