Introdução à Programação
Orientada a Objetos com Java

Expressões e
Estruturas de Controle,
parte I

Paulo Borba
Centro de Informática
Universidade Federal de Pernambuco

Depois de aprender a criar programas simples...

Precisamos aprender a criar
programas mais complexos,
que usam mais
tipos, operadores e instruções
da linguagem de programação

Este é o objetivo desta aula

O tipo `boolean`

- Valores: **`true false`**
- Operadores e expressões:
  - **`x & y`** (operador lógico de conjunção, AND)
  - **`x | y`** (operador lógico de disjunção, OR)
  - **`!x`** (operador lógico de negação, NOT)

`x` e **`y`** são expressões do tipo **`boolean`**

As expressões resultantes são do tipo **`boolean`**, gerando **`true`** ou **`false`** como resultado da avaliação

Expressões do tipo `boolean`

```
{  ...
   boolean b, c;
   b = true;
   c = !b;
   c = !(true | b) & c;
   b = c | !(!b);
   ...
}
```

Parênteses são usados para evitar ambigüidades

Qual o valor de **`b`** neste ponto?

Mais operadores de `boolean`

- **`x ^ y`** (operador lógico de disjunção exclusiva, XOR)
- **`x && y`** (operador condicional de conjunção)
- **`x || y`** (operador condicional de disjunção)

Equivalentes aos operadores lógicos correspondentes, mas o operando do lado direito do operador só é avaliado se necessário

Avaliando os operadores condicionais

```
{  ...
   boolean b = false;
   b = true | !(false & b);

   b = true || !(false & b);

   b = (false & b) || true;

   b = (false && b) || true;
   ...
}
```

Apenas as expressões sublinhadas são avaliadas

O tipo `int`

- Valores (32 bits):
  `-2147483648,...,0,...,2147483647`
- Operadores e expressões:
  - `+x` `x + y` (adição unária e binária)
  - `-x` `x - y` (subtração unária e binária)

`x` e `y` são expressões do tipo `int`

As expressões resultantes são do tipo `int`, gerando um inteiro como resultado da avaliação

Mais operadores aritméticos

- `x * y` (multiplicação)
- `x / y` (divisão inteira)
  - `17/3` dá como resultado `5`
- `x % y` (módulo, resto da divisão inteira)
  - `17%3` dá como resultado `2`

Operadores relacionais

- `x < y`
- `x <= y`
- `x > y`
- `x >= y`

`x` e `y` são expressões do tipo `int`

As expressões resultantes são do tipo `boolean`, gerando `true` ou `false` como resultado da avaliação

O comando `if-else`

```
if (expressaoBooleana) {
  comandos
} else {
  outros comandos
}
```

Se a avaliação de **expressaoBooleana** retornar **true**, **comandos** são executados , caso contrário, executa-se **outros comandos**

Débitos condicionais

```
class Conta {
  ...
  void debitar(double valor) {
    if (valor <= saldo) {
      saldo = saldo - valor;
    } else {
      Console c = new Console();
      c.println("Saldo Insuficiente!");
    }
  }...
}
```

Operadores de igualdade

- `x == y` (operador de igualdade)
- `x != y` (operador de diferença, desigualdade)

`x` e `y` são expressões do mesmo tipo, para qualquer tipo!

As expressões resultantes são do tipo `boolean`, gerando `true` ou `false` como resultado da avaliação

Semântica do operador de igualdade

```
x == y
```

Para avaliar esta expressão

- Avalia-se **x** obtendo um valor
- Avalia-se **y** obtendo outro valor
- Compara-se os dois valores obtidos e retorna-se **true** caso os dois sejam a mesma referência para um objeto ou o mesmo elemento de um tipo primitivo

Avaliando o operador de igualdade de tipos primitivos

```
boolean b, c;

b = true || false;

c = true && b;

b = b == c;
```

Qual o valor de b aqui?

Avaliando o operador de igualdade de tipos referência

```
boolean b;
Conta c, d;
c = new Conta();
d = null;
b = c == d;
d = new Conta();
b = c == d;
c = d;
b = c == d;
```

Qual o valor de b após cada uma das atribuições?

O método `equals` de Java...

```
x.equals(y)
```

- Para variáveis **x** e **y** do mesmo tipo (classe ou interface), é pré-definido em toda classe Java
- É equivalente a **x == y**, mas pode ser redefinido pelo programador para comparar não as referências mas os valores dos atributos dos objetos

O `equals` de `Conta` (quase!)

```
class Conta {...
  boolean equals(Conta c) {
    boolean r;
    r =    (numero == c.getNumero())
        && (saldo == c.getSaldo());
    return r;
  }
}
```

☺

Compara os valores dos atributos do objeto no qual o método está sendo executado com os valores correspondentes do objeto cuja referência é passada como parâmetro

A variável `this`

- Só pode ser lida, não pode-se atribuir um valor a ela
- Só pode aparecer dentro de métodos não estáticos
- Contém a referência para o objeto no qual um dado método está sendo executado (o método no qual ela aparece)

O valor da variável `this`

```
class Conta {...
  boolean equals(Conta c) {
    boolean r;
    r =   (this.getNumero() == c.getNumero())
       && (this.getSaldo() == c.getSaldo());
    return r;
  }
}         // Equivalente à definição anterior
```

☺

Na chamada `x.equals(y)` a variável `this` conterá a referência armazenada em `x` e `c` conterá a referência armazenada em `y`

Acessando atributos com `this`

```
class Conta {...
  boolean equals(Conta c) {
    boolean r;
    r =   (this.numero == c.getNumero())
       && (this.saldo == c.getSaldo());
    return r;
  }
}     // Equivalente às definições anteriores
```

☺

`this.x` representa o valor do atributo `x` do objeto no qual o método que contém `this.x` está sendo executado; é equivalente a `x`

Acessando atributos com referências

```
class Conta {...
  boolean equals(Conta c) {
    boolean r;
    r =   (this.numero == c.numero)
       && (this.saldo == c.saldo);
    return r;
  }
}     // Equivalente às definições anteriores
```

☺

`c.x` representa o valor do atributo `x` do objeto cuja referência está em `c`; mas se `x` for `private` isso só é valido na classe de `x`

Executando chamadas de métodos

`expressão.nomeDoMétodo(argumentos)`

Para avaliar esta chamada

- Avalia-se **`expressão`** obtendo-se uma referência para um objeto
- Avalia-se **`argumentos`**, da esquerda para a direita
- Executa-se **`nomeDoMétodo`** no objeto associado à referência obtida, mas...

`NullPointerException`

- A referência obtida pode ser **`null`**
  - Não está associada a nenhum objeto
  - O método não pode ser executado
  - É levantada a exceção **`NullPointerException`** de Java

Muito cuidado!
Isso não deve acontecer!

Verificando existência do objeto

```
class Conta {...
  boolean equals(Conta c) {
    boolean r;
    if (c != null)
      r =   (this.numero == c.getNumero())
         && (this.saldo == c.getSaldo());
    else r = false;
    return r;
  }
}   // Melhor do que às definições anteriores
```

☺

Não compara o conteúdo!

Comparando com o `equals`

```
class Conta {...
  boolean equals(Conta c) {
    boolean r;
    if (c != null)
☹     r =   (this.numero.equals(c.getNumero()))
        && (this.saldo == c.getSaldo());
    else r = false;
    return r;
  }
}  // Melhor do que às definições anteriores
```

numero pode ser null!

Evitando exceções

```
class Conta {...
  boolean equals(Conta c) {
    boolean r, s;
    if (c != null) {
      r = (numero==null) && (c.numero==null);
      s = (numero!=null) && (c.numero!=null)
          && (numero.equals(c.numero));
      r = (r || s) && (saldo==c.saldo);
    } else r = false;
    return r;
  }
}  // Melhor do que às definições anteriores
```

☹

Pouca legibilidade!

Legibilidade e métodos (auxiliares)

Um método longo, difícil de ler ou entender deve chamar métodos (auxiliares) que realizam partes da operação do método

Não qualquer parte, mas partes que correspondam a sub-operações lógicas

Reuso e métodos

Se uma operação é necessária em **mais de uma parte** de um programa, deve-se definir um método que realiza esta operação

Não reinvente a roda! Não perca produtividade, tempo, dinheiro

O método auxiliar `equals`

Aceita um objeto de qualquer classe

```
public class OperacoesObject {
  public static boolean equals(Object a,
                               Object b){
    return    ((a == null) && (b == null))
           || (   (a != null) && (b != null)
               && (a.equals(b)));
  }
}
```

Esta classe agrupa operações associadas a `Object`, uma classe pré-definida de Java

As classes pré-definidas...

- Têm um conjunto de métodos pré-definidos
- Não permitem a inclusão de novos métodos, mesmo que estes manipulem objetos destas classes

Como novos métodos são necessários, a saída é definir uma classe com métodos estáticos

O `equals` de `Conta` (finalmente!)

```
class Conta {...
  boolean equals(Conta c) {
    boolean r;
    if (c != null)
      r =   (OperacoesObject.equals(
                this.numero,c.getNumero()))
          && (this.saldo == c.getSaldo());
    else r = false;
    return r;
  }
}   // Melhor do que às definições anteriores
```

Resumindo...

- O tipo `boolean`
- O tipo `int`
- O comando `if-else`
- Os operadores `==` e `!=`
- O método `equals`
- A variável `this`
- `NullPointerException`
- Legibilidade, reuso e métodos