Introdução à Programação

Expressões e Comandos Condicionais

Expressões e Comandos Condicionais em Java

```
public class MinhaJanela extends Window {
        Button botaoMudaCor, botaoFechar;
        public MinhaJanela() {
                int coordenadaY ;
                botaoMudaCor = new Button("Muda");
                botaoFechar = new Button("Fechar");
                this.include(botaoMudaCor,250,65);
                coordenadaY =  65 + botaoMudaCor.getHeight();
                this.include(botaoFechar,250,coordenadaY);
                this.setSize(500,400);
        }
        public void clickEvent(Component component) {
                if (component == botaoMudaCor)
                        this.setBackground(new Color(195,0,0));
                if (component == botaoFechar)

                        ...
        }
}
```

Expressão aritmética

Comandos Condicionais

Tópicos da Aula

- Hoje vamos acrescentar comportamentos mais complexos a métodos
  - Expressões Aritméticas
  - Mudança de fluxo de controle de métodos
    - Expressões booleanas
    - Comandos Condicionais (if e else)
- Depois vamos empregar estes conceitos para construir exemplos mais complexos em miniJava
  - Pegando eventos de mais de um componente
  - Classe MiniJavaSystem
- Finalmente, iremos reestruturar o código usando mais algumas técnicas de refactoring
  - *Extract method* e *inline method*

Expressões Aritméticas

- Uma **expressão aritmética** computa resultados numéricos e utiliza operadores aritméticos
- Operadores aritméticos
  - \+ Adição → Pode ser usado de forma unária **+** operando
    - Pode ser usado para concatenar Strings
  - \- Subtração → Pode ser usado de forma unária **-** operando
  - * Multiplicação
  - / Divisão
  - % Resto da divisão (operador módulo)
- Multiplicação, divisão e resto da divisão têm precedência maior sobre adição e subtração

```
resultado = 3 + 4 % 3 * 5
```
→ 8

Divisão e Resto da Divisão

- Se ambos operandos da expressão aritmética forem valores inteiros, o resultado será um inteiro (a parte decimal será descartada)
- Portanto

```
14/3   ➡  4
8/12   ➡  0

14%3   ➡  2
8%12   ➡  8
```

Atribuição de Expressões aritméticas

- Numa atribuição, a expressão aritmética é avaliada primeiro, para depois se atribuir o resultado da expressão à variável

Primeiro, a expressão do lado direito do operador = é avaliado

```
answer  =  sum / 4 + MAX * lowest;
```

4 1 3 2

Depois, o resultado é armazenado na variável

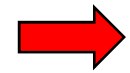
Atribuição de Expressões aritméticas

- O lado direito e esquerdo de um comando de atribuição podem conter a mesma variável

```
int contador  = 3;
```

Primeiro, 1 é adicionado ao valor original de `contador`

```
contador  =  contador + 1;
```



4

Depois o resultado é armazenado em contador (sobrescrevendo o seu valor original)

Operadores de Incremento e Decremento

- Operadores de *incremento* e *decremento* são operadores unários (usam um só operando)
- O operador de *incremento* (++) soma 1 ao seu operando
- O operador de *decremento* (--) subtrai 1 de seu operando
- A instrução

```
contador++;
```

é funcionalmente equivalente a

```
contador = contador + 1;
```

Operadores de Incremento e Decremento

- Estes operadores podem ser empregados de forma *pós-fixada* ou *pré-fixada*

```
contador++;  ou  ++ contador;
```

Incrementa (decrementa) a variável `contador` mas retorna como resultado o valor antigo dela

Incrementa (decrementa) a variável `contador` e retorna como resultado o novo valor dela

- Quando isolados têm comportamentos equivalentes
- Quando fazem parte de expressões maiores, eles podem ter comportamentos diferentes

Operadores de Incremento e Decremento

- Devem ser utilizadas com cuidado em expressões maiores

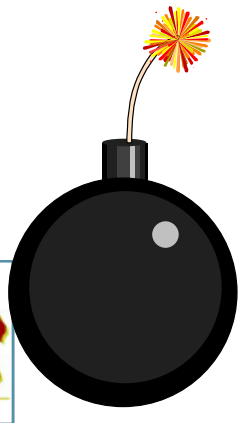
```
int contador  = 3;
contador++;
++contador;
int valor  = contador++;
valor  = ++ contador;
```

`contador` agora armazena 4

`contador` agora armazena 5

`valor` agora armazena 5 e depois `contador` é incrementado para 6

`contador` é incrementado para 7 e agora `valor` armazena 7



Operador de Atribuição

- É comum fazermos algum tipo de operação com uma variável e depois armazenar o valor da operação na própria variável
- Java oferece uma forma mais compacta de fazer isto, utilizando o operador de atribuição

`x += exp`	`x += 1;`	⟺	`x = x + 1;`
`x -= exp`	`x -= 2;`	⟺	`x = x - 2;`
`x *= exp`	`x *= k;`	⟺	`x = x * k;`
`x /= exp`	`x /= 3;`	⟺	`x = x / 3;`
`x %= exp`	`x %= 3;`	⟺	`x = x % 3;`

Modificando Fluxo de Controle

- A ordem de execução de um método é denominado **fluxo de controle**
- Exceto quando especificado de outra forma, a ordem de execução de um método é linear, isto é uma instrução após a outra em seqüencia
- Alguns comandos em programação nos permitem:
  - Decidir se a execução de uma instrução deve ou não ser feita
- Esta decisão é baseada em **expressões booleanas**

O Tipo `boolean`

- Valores: **`true false`**
- Operadores lógicos e expressões:
  - **`x && y`** (operador lógico de conjunção, AND)
  - **`x || y`** (operador lógico de disjunção, OR)
  - **`!x`** (operador lógico de negação, NOT)
  - **`x ^ y`** (operador lógico de disjunção exclusiva, XOR)
- Operandos da direita só são avaliados, se necessário

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

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

Tabela Verdade

- Uma tabela verdade contém todas as combinações true-false de uma expressão booleana

a	b	a && b	a || b	a ^ b
true	true	true	true	false
true	false	false	true	true
false	true	false	true	true
false	false	false	false	false

Expressões Booleanas

```
{   ...
    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?

true

Operadores relacionais

- Operadores relacionais, geralmente, fazem parte de expressões booleanas
  - `x < y`
  - `x <= y`
  - `x > y`
  - `x >= y`

`x` e `y` são expressões de algum tipo numérico

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

Operadores de igualdade

- Operadores de igualdade, também, pode fazer parte de expressões booleanas
  - `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?

true

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?

false

false

true

Modificando Fluxo de Controle

- Um **comando condicional** nos permite escolher qual deve ser a próxima instrução executada em um método
- A execução de uma determinada instrução depende de uma condição (expressão booleana)
- Um dos comandos condicionais presentes em Java é o ***if-else***

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**

Variações do comando if-else

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

```
if (expressaoBooleana)
   comando;
```

Se a avaliação da expressão retornar **false**, não executa-se nada

```
if (expressaoBooleana)
   comando;
else outroComando;
```

O uso do bloco só é necessário caso queira-se executar mais de um comando

Encadeando comandos
if-else

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

Exemplo de if-else

```
class Conta {
   String   numero, digito;
   double   saldo;

   void debito(double valor) {
      if (valor <= saldo)
           saldo = saldo - valor;
    }
   ...
```

O Comando `switch`

```
switch(expressao) {
   case rotulo1:
      Comandos1
      break;
   case rotulo2:
      Comandos2
      break;
   ...
   default:
      Comandos
}
```

Para executar um `switch`

- Avalia-se **`expressao`**
- Executa-se os comandos do **`case`** cujo rótulo é igual ao valor resultante da expressão
- Executa-se os comandos de **`default`** caso o valor resultante não seja igual a nenhum rótulo

Restrições do Comando `switch`

```
switch(expressao) {
   case rotulo1:
      Comandos1
      break;
   case rotulo2:
      Comandos2
      break;
   ...
   default:
      Comandos
}
```

- O tipo de **`expressao`** só pode ser :
  - char, byte, short, int
- Os rótulos são constantes diferentes
- Existe no máximo uma cláusula **`default`** (é opcional)
- Os tipos dos rótulos têm que ser o mesmo de **`expressao`**

Variações do Comando `switch`

```
switch(expressao) {
   case rotulo1:
      Comandos1
      break;
   case r2: case r3:
      Comandos2
      break;
   ...
   default:
      Comandos
}
```

- Vários rótulos podem estar associados ao mesmo comando
- Os comandos **`break`** são opcionais:
  - Sem o **`break`** a execução dos comandos de um rótulo continua nos comandos do próximo, até chegar ao final ou a um **`break`**

Usando o Comando `switch`

```
switch(resposta) {
   case 's': case 'S':
      retorno = true;
      break;
   case 'n': case 'N':
      retorno = false;
      break;
   default:
      retorno = false;
      console.println("Erro!");
}
```

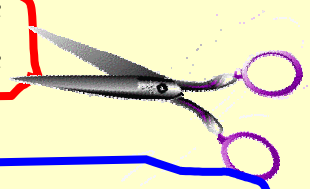
Mais Técnicas de Refactoring

- Extract Method
  - Consiste em extrair um trecho de código para um método auxiliar
  - Melhora a compreensão do código original, tornando o código mais modular

Antes do extract method

```
MinhaJanelaComComportamento() {
botao = new Button("Calc");
this.include(botao,250,65);

rotuloEntrada = new Label("Entrada");
campoEntrada = new TextField();
this.include(rotuloEntrada,40,60);
this.include(campoEntrada,100,60);
rotuloResultado = new Label("Resultado");
campoResultado = new TextField();
this.include(campoResultado,100,90);
this.include(rotuloResultado,40,90);
}
```



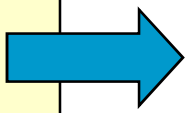
Depois do extract method

```
MinhaJanelaComComportamento() {
  configurarBotao();
  configurarEntrada();
  configurarResultado();
}
void configurarBotao(){
  botao = new Button("Calc");  this.include(botao,250,65);
}
void configurarEntrada(){
  rotuloEntrada = new Label("Entrada");
  campoEntrada = new TextField();
  this.include(rotuloEntrada,40,60);
  this.include(campoEntrada,100,60);
}
...
```

Inline method

- Inverso do extract method
  - Troca uma chamada de método pelo código correspondente
  - Deve ser usado quando o corpo do método faz algo que é tão claro quanto o seu nome

```
MinhaJanelaComComportamento() {
  criarEIncluirBotao();
}
void criarEIncluirBotao(){
  botao = new Button("Calc");
  this.include(botao,250,65);
}
```



```
MinhaJanelaComComportamento() {
  botao = new Button("Calc");
  this.include(botao,250,65);
}
```

Resumindo ...

- Expressões aritméticas
  - Operadores aritméticos
  - Operadores de incremento/decremento
- Mudança de fluxo de controle de um método
  - Expressões Booleanas
  - Comandos condicionais (*if* e *else*)
- Adicionando comportamento mais complexo aos componentes gráficos
  - Pegando eventos de mais um componente
  - Utilização da classe MiniJavaSystem
- Mais técnicas de Refactoring