Introdução à Programação
Orientada a Objetos com Java

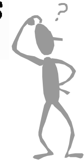
Expressões e
Estruturas de Controle,
parte II

Paulo Borba
Centro de Informática
Universidade Federal de Pernambuco

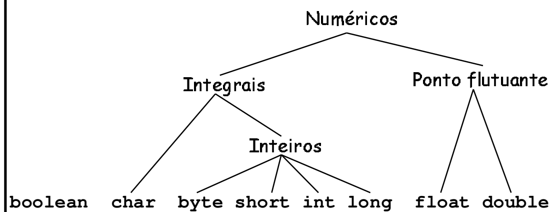
Depois de aprender a criar programas simples...

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

Este é o objetivo desta aula



Os tipos primitivos de Java



Os tipos inteiros

- `byte` (8 bits, -128 a 127)
- `short` (16 bits, -32768 a 32767)
- `int` (32 bits)
- `long` (64 bits)

```
int h, i, j;
h = 0x0000000c;
i = 12;
j = 014;
long l = 12L;
```

Literais inteiros em hexadecimal, decimal e octal

Literal `long`

O tipo `char`

Valores (16 bits):

- `'A'  '1'  'ç'  '@'  ' '  '\t'  '\\'  ...`
- Todos os 65536 caracteres do padrão Unicode (`www.unicode.org/charts`)
- Podem ser representados usando o código Unicode: `' '` é o mesmo que `'\u0020'`
- O caráter espaço (`' '`) não é o mesmo que a string que contém um espaço (`" "`)

Os tipos de ponto flutuante

- `float` (32 bits)
  - `(-)1.4012...e-45, (-)3.4028...e+38`
- `double` (64 bits)
  - `(-)4.940...e-324, (-)1.797...e+308`

Literais

```
double d = 5.6;
float f = 3.33F;
f = 4.342f;
```

Potência de 10

Que tipo usar?

- Use o menor tipo que contenha os valores que você está interessado em manipular
  - Para representar dias do mês: `byte`
  - Para representar números de alunos matriculados na universidade: `int`
  - Para representar o saldo de uma conta: `double` ou `float` mas cuidado com erros de precisão (melhor usar `BigDecimal`)

As operações nos numéricos...

São as mesmas de **`int`** mas os operandos são

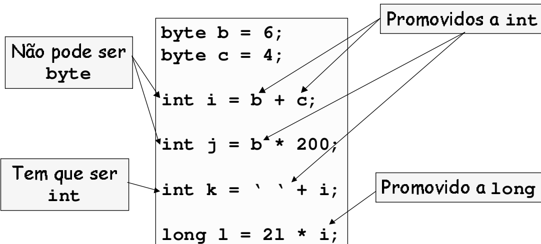
promovidos

para que tenham o tipo do operando de maior tipo

Os operadores unários só promovem **`byte`**, **`char`**, e **`short`** para **`int`**

No mínimo são promovidos a **`int`**

Promoção numérica



Conversão entre tipos

- A promoção numérica converte, implicitamente, uma expressão de um tipo "menor" para um tipo "maior"
- Pode-se também converter uma expressão de um tipo "maior" para um tipo "menor"
  - Implicitamente, ou
  - Usando casts

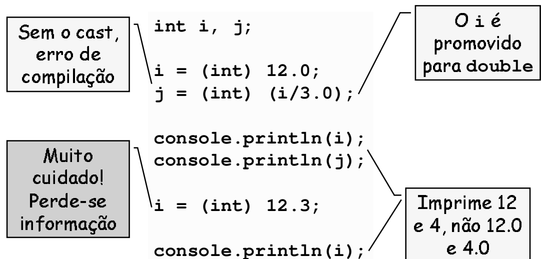
Conversão com casts

O cast indica o tipo para o qual a expressão deve ser convertida:

```
(tipo) expressão
```

- **`tipo`** é o nome do tipo "menor"
- **`expressão`** é uma expressão do tipo "maior"
- o resultado da avaliação é um valor de **`tipo`** que melhor representa o valor de **`expressão`** com seu tipo original

Usando casts



Casts, tipos primitivos e tipos referência

- Casts podem ser realizados entre quaisquer dois tipos primitivos, exceto `boolean`
- Casts podem ser realizados entre tipos referência, desde que certas condições sejam observadas (veremos depois...)

Conversões implícitas

Além das promoções numéricas temos:

- De "menor" para "maior" na atribuição:

```
double d = 2001;
d = 1111111111111123456L;
    // Perde precisão
```

- De `int` para `byte`, `char`, ou `short` na atribuição:

```
byte b = 127; // 128 daria erro
char c = i; // OK se 0<=i<=65535
b = 'a'; // Erro de compilação
```

Conversão de tipos e passagem de parâmetros

- Nenhuma conversão implícita é feita nas passagens de parâmetros
- O uso do cast torna-se necessário em todos os casos onde o tipo do parâmetro é diferente do tipo do respectivo argumento

O operador de atribuição

- **`x = exp`**
- **`x += exp`**
- **`x -= exp`**
- **`x *= exp`**
- **`x /= exp`**

`a = b = -1;`	⟺	`b = -1;` `a = b;`
`x += 1;`	⟺	`x = x + 1;`
`y -= k;`	⟺	`y = y - k;`

O operador condicional

```
cond?exp1:exp2
```

exp1 e exp2 são expressões do mesmo tipo

A avaliação desta expressão resulta no

- resultado da avaliação de **exp1** caso a expressão booleana **`cond`** gere **`true`**
- resultado da avaliação de **exp2** caso a expressão booleana **`cond`** gere **`false`**

O operador condicional e o comando `if-else`

```
int maior;
maior = (x > y) ? x : y;
```

Expressões

Equivalentes

```
int maior;
if (x > y) maior = x;
else maior = y;
```

Comandos

Operadores de incremento e decremento

- `++x`
- `--x`
- `x++`
- `x--`

Incrementa (decremetra) a variável `x` e retorna como resultado o novo valor dela

Incrementa (decremetra) a variável `x` mas retorna como resultado o valor antigo dela

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''
}
```

O mesmo pode ser feito sem o `else`, sem blocos, etc.

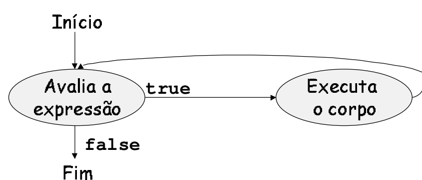
O comando `while`

```
while (expressãoBooleana)
  corpo
```

- Executa `corpo` várias vezes até que a avaliação da expressão retorne `false`
- A expressão é avaliada de novo após cada execução de `corpo`
- Não executa `corpo` nenhuma vez caso, de início, a avaliação da expressão retorne `false`

Diagrama de estados do comando `while`

Início



Fim

Usando o comando `while`

```
int soma = 0;
int valor = console.readInt();
while (valor >= 0) {
  soma = soma + valor;
  valor = console.readInt();
}
console.println(soma);
```

Lê e soma números até que um número negativo seja digitado

O `while` deve ser usado sempre que não sabe-se o número de vezes que certos comandos devem ser repetidos!

Entendendo o comando `while`

```
int soma = 0;
int valor = console.readInt();
while (valor >= 0) {
  soma = soma + valor;
  valor = console.readInt();
}
console.println(soma);
```

Se o primeiro número digitado for negativo, não executa-se o corpo do `while`

É executado quando o `while` termina, a expressão for `false`

Se um número negativo nunca for digitado, o programa não pára e a soma não será impressa

O comando `for`

```
for (int i = 0; i < valor; i = i+1)
  corpo
```

- Executa **corpo** um número específico de vezes: **valor** vezes
- Na primeira execução de **corpo**, o valor de **i** é 0
- O valor de **i** é incrementado após cada execução de **corpo**
- **i** pode ser acessada dentro de **corpo**, e deixa de existir após a execução do **for**

Usando o comando `for`

```
int soma = 0;
int valor = console.readInt();
for (int i = 0; i < valor; i = i+1) {
  soma = soma + console.readInt();
}
console.println(soma);
```

Lê a quantidade de números a serem digitados

Lê e soma uma certa quantidade de números

O **for** deve ser usado sempre que sabe-se o número de vezes que certos comandos devem ser repetidos!

Entendendo o comando `for`

```
int soma = 0;
int valor = console.readInt();
for (int i = 0; i < valor; i = i+1) {
  soma = soma + console.readInt();
}
console.println(soma);
```

Se **valor** for menor ou igual a 0, não executa-se o corpo do **for**

É executado quando o **for** termina

Os números negativos são somados também

Usando a variável do comando `for`

```
int soma = 0;
int somatorio = 0;
int valor = console.readInt();
for (int i = 0; i < valor; i = i+1) {
  soma = soma + console.readInt();
  somatorio = somatorio + i;
}
console.println(soma);
console.println(somatorio);
```

A variável pode ser usada dentro do corpo do **for**

Imprime o somatório de `0` até `valor-1`

A forma geral do comando `for`

```
for(inicialização;condição;incremento)
  corpo
```

- **inicialização** e **incremento** podem ser praticamente quaisquer comandos
- **condição** pode ser qualquer condição booleana
- **inicialização** deve inicializar a variável do **for**
- **incremento** deve incrementar a variável **for**

O comando `for` e o comando `while`

```
for(inicialização;condição;incremento)
  corpo
```

equivale a ...

```
inicialização;
while(condição) {
  corpo;
  incremento;
}
```

O comando `for` sem condição, etc.

```
for(;;)
  corpo
```

equivale a ...

```
while(true)
  corpo
```

Repetição infinita: cuidado!

O comando `do-while`

```
do {
  corpo
} while(expressaoBooleana)
```

- Executa **corpo**, pelo menos uma vez, até que a avaliação da expressão retorne **false**
- A expressão é avaliada de novo após cada execução de **corpo**

Os comandos `do-while` e `while`

```
do {
  corpo
} while(expressaoBooleana)
```

Equivalente a ...

```
corpo;
while(expressaoBooleana)
  corpo;
```

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** tem que ser integral
- 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`

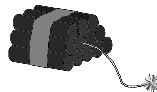
```
switch(expressao) {
  case rotulo1:
    Comandos1
    break;
  case r2: case r3:
    Comandos2
    break;
  ...
  default:
    Comandos
}
```

- 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!");
}
```

O comando `break`

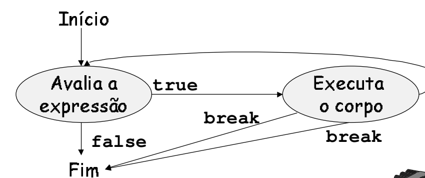


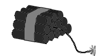
```
break
```

- Termina a execução de um comando **switch**
- Termina a execução de um loop (comandos **for**, **while**, **do-while**)
- Termina a execução de um bloco com rótulo

```
rotulo:{ comandos }
```

Diagrama de estados do comando `while` com comandos `break`





O comando `continue`

```
continue
```

- Termina a execução da iteração atual de um loop (**for**, **while**, **do-while**) e volta ao começo deste loop
- Todos os comandos que seriam executados após o **continue** são descartados

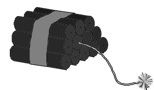
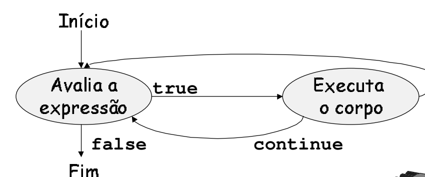
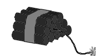


Diagrama de estados do comando `while` com comandos `continue`





Resumindo...

- Tipos primitivos de Java
- Conversões entre tipos e casts
- Operadores de atribuição, incremento, decremento, condicional
- Comandos de iteração (loops)
- Os comandos `continue`, `break`, e `switch`