


Tipos de Dados e Estruturas de Controle

Profa. Karen Selbach Borges



Tipos de Dados

- **Números Inteiros:** a faixa numérica de cada tipo pode ser calculado através da seguinte fórmula : $-2^{(\text{bits}-1)}$ até $2^{(\text{bits}-1)} - 1$
 - byte (1 byte)
 - short (2 bytes)
 - int (4 bytes)  Padrão
 - long (8 bytes)



Tipos de Dados

- Números Reais :
 - float (4 bytes)
 - double (8 bytes) ← Padrão

Erros Comuns	Explicação	Forma correta
<code>double d = 3,14;</code>	Os valores decimais são separados do inteiro por um ponto e não vírgula.	<code>double d = 3.14;</code>
<code>float f = 2.32;</code>	Como o padrão é double, então está sendo atribuído um valor com precisão numérica maior do que pode ser suportado por um float.	<code>float f = 2.32f;</code>



Tipos de Dados

- Outros :
 - Tipo Character : `char`
 - Representado entre apóstrofes ('...')
 - Ex1: `char letra = 'a';`
 - Ex2: `char quebraDeLinha = '\n';`
 - Tipo Lógico : `boolean`
 - Assume os valores `True` ou `False`
 - Ex1: `boolean teste = TRUE;`
 - Ex2: `boolean outroTeste = 1; // Erro muito comum de programadores C++;`



Tipos de Dados

- Cada tipo de dado tem um valor padrão (default), os quais são:

Tipo	Valor Default	Exemplo
Valores numérico	0	<code>int i = 0;</code> <code>byte b = 0;</code> <code>double d = 0.0;</code> <code>float f = 0f;</code>
Valores lógicos	False	<code>boolean teste = False;</code>
Caracter	'\u0000' ou '\0' (o caractere nulo)	<code>char c = '\u0000';</code>



Conversão Entre Tipos Numéricos



- **Conversões automáticas:** o valor resultante assume o tipo de dado do operador com maior precisão numérica.
- Exemplo:
`double x = 10.5 * 3 - 1.4f;`



Conversão Entre Tipos Numéricos



- **Conversões explícitas** (cast) : são possíveis de serem realizadas, mas com a possível perda de informações.
- Exemplo:
`float x = (float)10.5 * 3 - 1.4f;`
- Observação: não é uma boa prática !!!



Operadores Aritméticos

- Adição : $op1 + op2$
- Subtração : $op1 - op2$
- Multiplicação : $op1 * op2$
- Divisão : $op1 / op2$
- Resto da divisão : $op1 \% op2$



Operadores Aritméticos



- Lembre que, assim como na matemática, as operações de multiplicação, divisão e resto da divisão tem precedência sobre adição e subtração.
- Utilize parênteses para modificar a ordem de processamento dos operadores.



Operadores Aritméticos



- **Cuidado com as divisões por zero !**
 - Dividir um inteiro por zero o resultado será uma exceção do tipo `java.lang.ArithmeticException`
 - Dividir um número com ponto flutuante por zero o resultado será *infinity*.



Operadores de Atribuição

- Atribuição simples : =
- Atribuição Composta:

Operador	Exemplo	Equivale à
<code>+=</code>	<code>x +=4</code>	<code>x = x + 4</code>
<code>- =</code>	<code>x -=4</code>	<code>x = x - 4</code>
<code>* =</code>	<code>x *=4</code>	<code>x = x * 4</code>
<code>/ =</code>	<code>x /=4</code>	<code>x = x / 4</code>
<code>% =</code>	<code>x %=4</code>	<code>x = x % 4</code>



Ao usar atribuições compostas as conversões de tipo são feitas automaticamente

Operadores de Incremento/Decremento



- $op++ \neq ++op$
- $op-- \neq --op$
- Ex : `int m = 7;`
`int a = 2 * ++m; // a=16 e m=8`
`int b = 2 * m++; // b=16 e m=9`



Operadores Relacionais

- Maior : $op1 > op2$
- Menor : $op1 < op2$
- Igual : $op1 == op2$
- Maior igual : $op1 \geq op2$
- Menor igual : $op1 \leq op2$
- Diferente : $op1 \neq op2$



Operadores Lógicos

- `op1 && op2` : `op1` e `op2`. Avalia `op1` e condicionalmente `op2`.
- `op1 || op2` : `op1` ou `op2`. Avalia `op1` e condicionalmente `op2`.
- `op1 & op2` : `op1` e `op2`. Sempre avalia `op1` e `op2`.
- `op1 | op2` : `op1` ou `op2`. Sempre avalia `op1` e `op2`.
- `! op` : não `op`.



Estruturas de Decisão

if-else:

```
if (expressãoBooleana) {  
    // código executado caso expressãoBooleana seja verdadeira  
}  
else {  
    // código executado caso expressãoBooleana seja falsa  
}
```



Estruturas de Decisão

if-else-if:

```
if (expressãoBooleana1) {  
    // código executado caso expressãoBooleana seja verdadeira  
}  
else if (expressãoBooleana2) {  
    // código executado caso expressãoBooleana1 seja falsa e  
    expressãoBooleana2 seja verdadeira  
}  
else {  
    // código executado caso expressãoBooleana1 e  
    expressãoBooleana2 sejam falsas  
}
```



Estruturas de Decisão

- Observações sobre a estrutura if-else-if:
 - cuidado com atribuições dentro das expressões booleanas
 - é possível utilizar um operador ternário (?) com a seguinte sintaxe : **expressão ? valor1 : valor2**. Onde o valor da expressão será valor1 caso a expressão seja verdadeira ou valor2 caso contrário.




Estruturas de Decisão

switch-case:

```
switch (expressão)
{
    case valor 1:
        // código executado caso expressão seja igual valor1
        break;
    case valor 2:
        // código executado caso expressão seja igual valor2
        break;
    ...
    default:
        // código executado caso expressão seja diferente de
        todos os valores anteriores
}
```



Estruturas de Decisão

- Observações sobre a estrutura switch:
 - A expressão do switch deve retornar um dos seguintes tipo: byte, short, char e int.
 - São aceitos também enums e instâncias de Character, Byte, Short e Integer.
 - O argumento do case deve ser uma constante em tempo de execução. Ou seja, deve ser um valor numérico ou uma constante numérica.
 - A instrução default não é obrigatória e, quando presente, não precisa necessariamente ser a última instrução.
- O uso da instrução break não é obrigatório, mas se for omitido o fluxo seguirá normalmente para o outro case

Estruturas de Repetição



for:

```
for (inicialização; expressãoBooleana; incremento)
{
    // código executado enquanto
    // expressãoBooleana for verdadeira
}
```



Estruturas de Repetição



while:

```
while (expressãoBooleana)
{
    // código executado enquanto
    // expressãoBooleana for verdadeira
    // pode executar uma primeira vez ou não
}
```



Estruturas de Repetição



do-while:

```
do
{
    // código executado enquanto
    // expressãoBooleana for verdadeira
    // sempre vai executar pelo menos uma vez
}
while (expressãoBooleana)
```



Comandos de Desvio de Fluxo



- Break
 - Interrompe estruturas while, for, do/while ou switch
 - Execução continua com a primeira instrução depois da estrutura
- Continue
 - Quando executada em uma estrutura em uma estrutura while, for ou do/while , pula as instruções restantes no corpo dessa estrutura e prossegue com a próxima interação do laço.

