Capítulo 7

Programas com Vários Objetos

Quais novidades veremos neste capítulo?

- Programas com vários objetos.

Até agora, todos os programas que vimos lidavam com apenas um objeto. No entanto, podemos ter programas que lidam com vários objetos. Estes objetos podem pertencer todos à mesma classe ou a classes diferentes. Exemplo:

1. Vários objetos do mesmo tipo

```
Flor rosa = new Flor();
Flor margarida = new Flor();
Flor florDeLaranjeira = new Flor();

rosa.cor("vermelha");
rosa.aroma("muito agradável");
margarida.aroma("sutil");
florDeLaranjeira.aroma("delicioso");
```

2. Vários objetos de tipos (classes) diferentes:

```
Cachorro floquinho = new Cachorro();
Gato mingau = new Gato();
Rato topoGiggio = new Rato();
Vaca mimosa = new Vaca();

floquinho.lata();
mingau.mie();
topoGiggio.comaQueijo();
mingau.persiga(topoGiggio)
floquinho.persiga(mingau);
mimosa.passePorCima(floquinho, mingau, topoGiggio);
```

Vejamos agora um exemplo de utilização de objetos de 3 tipos diferentes em conjunto. Neste exemplo, teremos 3 classes representando prismas, quadrados e triângulos retângulos e criaremos uma instância de quadrado, uma de triângulo retângulo e duas de prismas.

Note, no exemplo abaixo, que utilizamos um padrão diferente para nomear os métodos que servem para atribuir valores aos atributos. Segundo este padrão, muito utilizado por programadores avançados em linguagens como C++ e Smalltalk, o nome do método é exatamente o nome do atributo correspondente. Por exemplo, o método `void altura(double a)` é utilizado para definir o valor do atributo `altura` e assim por diante. Ao escrever seus programas, você é livre para escolher entre qualquer um dos padrões existentes, mas é importante que você seja coerente, ou seja, após escolher o padrão de nomeação, aplique-o consistentemente em todo o seu código.

```
class Prisma
{
  double altura;
  double areaDaBase;

  void altura(double a)
  {
    altura = a;
  }

  void areaDaBase(double a)
  {
    areaDaBase = a;
  }

  double volume()
  {
    return areaDaBase * altura;
  }
}

class Quadrado
{
  double lado;

  void lado(double l)
  {
    lado = l;
  }

  double area()
  {
    return lado * lado;
  }
}

class TrianguloRetângulo
{
  double cateto1;
  double cateto2;
```

```
  // note a indentação supercompacta!
  void cateto1(double c) { cateto1 = c; }
  void cateto2(double c) { cateto2 = c; }

  double area()
  {
    return cateto1 * cateto2 / 2.0;
  }
}
```

Agora, utilizando o interpretador, podemos criar objetos destes vários tipos e utilizá-los em conjunto:

```
Quadrado q = new Quadrado();
TrianguloRetângulo tr = new TrianguloRetângulo();
Prisma prismaBaseQuadrada = new Prisma();
Prisma prismaBaseTriangular = new Prisma();

q.lado(10.0);
tr.cateto1(20.0);
tr.cateto2(30.0);

prismaBaseQuadrada.altura(3.0);
prismaBaseTriangular.altura(1.0);

prismaBaseQuadrada.areaDaBase(q.area());
prismaBaseTriangular.areaDaBase(tr.area());

if(prismaBaseQuadrada.volume() > prismaBaseTriangular.volume())
    System.out.println("O prisma de base quadrada tem maior volume");
else if(prismaBaseTriangular.volume() > prismaBaseQuadrada.volume())
    System.out.println("O prisma de base triangular tem maior volume");
else
    System.out.println("Ambos os prismas têm o mesmo volume");
```

Nota sobre o interpretador do DrJava: para conseguir digitar todos os `if`s encaixados no interpretador do DrJava sem que ele tente interpretar cada linha em separado, é preciso utilizar Shift+Enter em vez de apenas Enter no final de cada linha dos `if`s encaixados. Apenas no final da última linha (a que contém o `println` final) é que se deve digitar apenas Enter para que o DrJava então interprete todas as linhas de uma vez.

Exercícios

1. Utilizando a classe `Conversor5` definida no exercício 1 do Capítulo 3, escreva uma classe contendo três métodos, onde cada método recebe uma temperatura `x` utilizando uma escala de temperaturas e imprime os valores de `x` nas demais escalas de temperatura.

2. Escreva uma classe `Rendimentos` que contenha os seguintes métodos a fim de contabilizar o total de rendimentos de uma certa pessoa em um certo ano:

- `rendimentosDePessoaFísica(double);`
- `rendimentosDePessoaJurídica(double);`
- `rendimentosDeAplicaçõesFinanceiras(double);`
- `rendimentosNãoTributáveis(double);`
- `double totalDeRendimentosTributáveis();`

Em seguida, escreva uma classe `TabelaDeAlíquotas` que possui:

- um método `alíquota()` que recebe como parâmetro o total de rendimentos tributáveis de uma pessoa e devolve um número entre 0 e 1.0 correspondente à aliquota de imposto que a pessoa deverá pagar e
- um método `valorADeduzir()` que recebe como parâmetro o total de rendimentos tributáveis de uma pessoa e devolve o valor a deduzir no cálculo do imposto.

Para escrever esta classe, utilize a tabela do IR 2006 abaixo:

Rendimentos Tributáveis	Alíquota	Parcela a deduzir
Até R$ 13.968,00		
De R$ 13.968,01 a R$ 27.912,00	0.15	R$ 1.904,40
acima de R$ 27.912,00	0.275	R$ 5.076,90

Agora escreva uma classe `CalculadoraDeImposto` que possui um único método que recebe como parâmetro o valor dos rendimentos tributáveis de uma pessoa e devolve o valor do imposto a ser pago.

Finalmente, escreva um trecho de código (para ser digitado no interpretador) que utiliza `Rendimentos` para definir os vários rendimentos de uma pessoa e `CalculadoraDeImposto` para calcular o imposto a pagar.

3. Suponha que você tenha as seguintes classes:

```
class A
{
  double a (int meses, double taxa)
  {
    return Math.pow((taxa + 100) / 100, meses) - 1;
  }
}

class B
{
  final double TAXA = 1.2;

  void b (double valorEmprestado, int meses)
  {
    A a = new A();
    double valorDaDívida = valorEmprestado + (a.a(meses, TAXA)*valorEmprestado);
```

```
      System.out.println("Dívida de " + valorDaDívida + " real(is), " +
                         "calculada com taxa de " + TAXA + "% ao mês.");
    }
  }
```

(a) O que fazem os métodos `a` (da classe `A`) e `b` (da classe `B`)? Não precisa entrar em detalhes. Dica: para saber o que `Math.pow` faz consulte a página `http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Math.html`;

(b) Os nomes `a` e `b` (dos métodos) e `A` e `B` (das classes) são péssimos. Por quê? Que nomes você daria? Sugira, também, outro nome para a variável objeto (criada no interpretador);

(c) Acrescente alguns comentários no código do método `b`;

(d) Seria mais fácil digitar o valor 1.2 quando necessário, em vez de criar uma constante `TAXA` e utilizá-la. Então, por que isso foi feito? Cite, pelo menos, dois motivos. A palavra chave `final` faz com que o valor da variável `TAXA`, uma vez atribuído, não possa ser alterado.

Capítulo 8

Laços e Repetições

Quais novidades veremos neste capítulo?

- A idéia de laços em linguagens de programação;
- O laço `while`;
- O operador que calcula o resto da divisão inteira: `%`.

8.1 Laços em linguagens de programação

Vamos apresentar para vocês um novo conceito fundamental de programação: o *laço*. Mas o que pode ser isso? Um nome meio estranho, não? Nada melhor do que um exemplo para explicar.

Vamos voltar ao nosso velho conversor de temperatura. Imagine que você ganhou uma passagem para Nova Iorque e que os EUA não estão em guerra com ninguém. Você arruma a mala e se prepara para a viagem. Antes de viajar você resolve conversar com um amigo que já morou nos EUA. Ele acaba lhe dando uma dica: guarde uma tabelinha de conversão de temperaturas de Fahrenheit para Celsius. Ela será muito útil, por exemplo, para entender o noticiário e saber o que vestir no dia seguinte. Você então se lembra que já tem um conversor pronto. Basta então usá-lo para montar a tabela. Você chama então o DrJava e começa uma nova sessão interativa.

```
Welcome to DrJava.
> Conversor4 c = new Conversor4()
> c.fahrenheitParaCelsius(0)
-17.77777777777778
> c.fahrenheitParaCelsius(10)
-12.222222222222221
> c.fahrenheitParaCelsius(20)
-6.666666666666667
> c.fahrenheitParaCelsius(30)
-1.1111111111111112
```

```
> c.fahrenheitParaCelsius(40)
4.444444444444445
> c.fahrenheitParaCelsius(50)
10.0
> c.fahrenheitParaCelsius(60)
15.555555555555555
> c.fahrenheitParaCelsius(70)
21.11111111111111
> c.fahrenheitParaCelsius(80)
26.666666666666668
> c.fahrenheitParaCelsius(90)
32.22222222222222
> c.fahrenheitParaCelsius(100)
37.77777777777778
> c.fahrenheitParaCelsius(110)
43.333333333333336
>
```

Pronto, agora é só copiar as linhas acima para um editor de textos, retirar as chamadas ao método `fahrenheitParaCelsius` (pois elas confundem) e imprimir a tabela.

Será que existe algo de especial nas diversas chamadas do método `fahrenheitParaCelsius` acima? Todas elas são muito parecidas e é fácil adivinhar a próxima se sabemos qual a passada. Ou seja, a lei de formação das diversas chamadas do método é simples e bem conhecida. Não seria interessante se fosse possível escrever um trecho de código compacto que representasse essa idéia? Para isso servem os laços: eles permitem a descrição de uma seqüência de operações repetitivas.

8.2 O laço `while`

O nosso primeiro laço será o `while`, a palavra inglesa para *enquanto*. Ele permite repetir uma seqüência de operações enquanto uma *condição* se mantiver verdadeira. Mais uma vez, um exemplo é a melhor explicação. Experimente digitar as seguintes linhas de código no painel de interações do DrJava (lembre-se que para digitarmos as 5 linhas do comando `while` abaixo, é necessário usarmos Shift+Enter em vez de apenas Enter no final das 4 linhas iniciais do `while`):

```
Welcome to DrJava.
> int a = 1;
> while (a <= 10)
{
  System.out.println("O valor atual de a é: " + a);
  a = a + 1;
}
```

o resultado será o seguinte:

```
O valor atual de a é: 1
O valor atual de a é: 2
O valor atual de a é: 3
O valor atual de a é: 4
O valor atual de a é: 5
O valor atual de a é: 6
O valor atual de a é: 7
O valor atual de a é: 8
O valor atual de a é: 9
O valor atual de a é: 10
>
```

Vamos olhar com calma o código acima. Primeiro criamos uma variável inteira chamada `a`. O seu valor inicial foi definido como 1. A seguir vem a novidade: o laço `while`. Como dissemos antes, ele faz com que o código que o segue (e está agrupado usando chaves) seja executado enquanto a condição `a <= 10` for verdadeira. Inicialmente `a` vale 1, por isso este é o primeiro valor impresso. Logo depois de imprimir o valor de `a`, o seu valor é acrescido de 1, passando a valer 2. Neste momento o grupo de instruções que segue o `while` terminou. O que o computador faz é voltar à linha do `while` e verificar a condição novamente. Como `a` agora vale 2, ele ainda é menor que 10. Logo as instruções são executadas novamente. Elas serão executadas *enquanto* a condição for verdadeira, lembra? Mais uma vez, o valor atual de `a` é impresso e incrementado de 1, passando a valer 3. De novo o computador volta à linha do `while`, verifica a condição (que ainda é verdadeira) e executa as instruções dentro das chaves. Esse processo continua até que o `a` passe a valer 11, depois do décimo incremento. Neste instante, a condição torna-se falsa e na próxima vez que a condição do `while` é verificada, o computador pula as instruções dentro das chaves do `while`. Ufa, é isso! Ainda bem que é o computador que tem todo o trabalho! Uma das principais qualidades do computador é a sua capacidade de efetuar repetições. Ele faz isso de forma automatizada e sem se cansar. O laço é uma das formas mais naturais de aproveitarmos essa característica da máquina.

Agora vamos ver como esse novo conhecimento pode nos ajudar a montar a nossa tabela de conversão de forma mais simples e flexível. Se pensarmos bem, veremos que as operações realizadas para calcular as temperaturas para tabela são semelhantes ao laço apresentado. Só que no lugar de simplesmente imprimir os diferentes valores de uma variável, para gerar a tabela chamamos o método `fahrenheitParaCelsius` várias vezes. Vamos agora adicionar um método novo à classe `Conversor4`, que terá a função de imprimir tabelas de conversão para diferentes faixas de temperatura. O código final seria:

```
class Conversor5
{
  /**
   * Converte temperatura de Celsius para Fahrenheit.
   */
  double celsiusParaFahrenheit(double celsius)
  {
    return celsius * 9.0 / 5.0 + 32;
  }

  /**
   * Converte temperatura de Fahrenheit para Celsius.
   */
  double fahrenheitParaCelsius(double fahr)
```

```
  {
    return (fahr - 32.0) * 5.0 / 9.0;
  }

  /**
   * Imprime uma tabela de conversão Faranheit => Celsius.
   */
  void imprimeTabelaFahrenheitParaCelsius(double inicio, double fim)
  {
    double fahr = inicio;
    double celsius;

    while (fahr <= fim)
    {
      celsius = fahrenheitParaCelsius(fahr);
      System.out.println(fahr + "F = " + celsius + "C");
      fahr = fahr + 10.0;
    }
  }
}
```

Muito melhor, não?

8.3 Números primos

Vejamos agora um novo exemplo. Todos devem se lembrar o que é um número primo: um número natural que possui exatamente dois divisores naturais distintos, o 1 e o próprio número. Vamos tentar escrever uma classe capaz de reconhecer e, futuramente, gerar números primos.

Como podemos reconhecer números primos? A própria definição nos dá um algoritmo. Dado um candidato a primo `x`, basta verificar se algum inteiro entre 2 e `x` - 1 divide `x`. Então para ver se um número é primo podemos usar um laço que verifica se a divisão exata ocorreu.

Porém, ainda falta um detalhe. Como podemos verificar se uma divisão entre números inteiros é exata. Já sabemos que se dividirmos dois números inteiros em Java a resposta é inteira. E o resto da divisão? Felizmente, há um operador especial que devolve o resto da divisão, é o operador `%`. Vejamos alguns exemplos:

```
Welcome to DrJava.
> 3 / 2
1
> 3 % 2
1
> 5 / 3
1
> 5 % 3
2
> int div = 7 / 5
> int resto = 7 % 5
> div
1
```

```
> resto
2
> div*5 + resto
7
>
```

Deu para pegar a idéia, não?

Agora vamos escrever uma classe contendo um método que verifica se um inteiro é primo ou não, imprimindo a resposta na tela. O nome que daremos à nossa classe é `GeradorDePrimos`. A razão para esse nome ficará clara no próximo capítulo.

```
class GeradorDePrimos
{
  /**
   * Imprime na tela se um número inteiro positivo é primo ou não.
   */
  void verificaPrimalidade(int x)
  {
    // Todos os números inteiros positivos são divisíveis por 1.
    int númeroDeDivisores = 1;
    // O primeiro candidato a divisor não trivial é o 2.
    int candidatoADivisor = 2;

    // Testa a divisão por todos os números menores ou iguais a x.
    while (candidatoADivisor <= x)
    {
      if (x % candidatoADivisor == 0)
        númeroDeDivisores = númeroDeDivisores + 1;
      candidatoADivisor = candidatoADivisor + 1;
    }

    // Imprime a resposta.
    if (númeroDeDivisores == 2)
      System.out.println(x + " é primo.");
    else
      System.out.println(x + " não é primo.");
  }
}
```

Será que esta é a forma mais eficiente de implementar este método? Será que podemos alterar o código para que ele forneça a resposta mais rapidamente? O que aconteceria se quiséssemos verificar a primalidade de 387563973. Pense um pouco sobre isso e depois dê uma olhada no exercício 6 deste capítulo.

Exercícios

1. Crie uma classe `Fatorial` com um método `calculaFatorial(int x)` que calcula o fatorial de `x` se este for um número inteiro positivo e devolve `-1` se `x` for negativo.

   Adicione o método `testaCalculaFatorial()` que testa o método `calculaFatorial(int x)` para diferentes valores de `x`.

2. Crie uma classe `Média` contendo um método `calculaMédia(int n)` que devolve a média dos valores 1, 2, 3, ..., n, onde n é o valor absoluto de um número fornecido ao método.

   Adicione o método `testaCalculaMédia()` que testa o método `calculaMédia(int n)` para diferentes valores de `n`.

3. Adicione as seguintes funcionalidades à classe `Conversor5` vista neste capítulo:

   (a) Crie o método `imprimeTabelaCelsiusParaFahrenheit`, que converte no sentido oposto do método `imprimeTabelaFahrenheitParaCelsius`.

   (b) Adicione um parâmetro aos métodos acima que permita a impressão de uma tabela com passos diferentes de 10.0. Ou seja, o passo entre a temperatura atual e a próxima será dado por esse novo parâmetro.

4. Escreva uma classe `Fibonacci`, com um método `imprimeNúmerosDeFibonacci (int quantidade)`, que imprime os primeiros `quantidade` números da seqüência de Fibonacci. A seqüência de Fibonacci é definida da seguinte forma.

   - $F_1 = 1$;
   - $F_2 = 1$;
   - $F_n = F_{n-1} + F_{n-2}$, para todo inteiro positivo $n > 2$.

   O método deve então imprimir $F_1, F_2, F_3, \ldots, F_{quantidade}$.

5. Abaixo, apresentamos uma pequena variação do método `verificaPrimalidade`. Ela não funciona corretamente em alguns casos. Você deve procurar um exemplo no qual esta versão não funciona e explicar o defeito usando suas próprias palavras. Note que a falha é sutil, o que serve como alerta: programar é uma tarefa difícil, na qual pequenos erros podem gerar resultados desastrosos. Toda atenção é pouca!

```
/**
 * Imprime na tela se um número inteiro positivo é primo ou não.
 */
void verificaPrimalidade(int x)
{
  // Todos os números inteiros positivos são divisíveis por 1.
  int númeroDeDivisores = 1;
  // O primeiro candidato a divisor não trivial é o 2.
  int candidatoADivisor = 2;

  // Testa a divisão por todos os números menores ou iguais a x.
  while (candidatoADivisor <= x)
  {
    candidatoADivisor = candidatoADivisor + 1;
    if (x % candidatoADivisor == 0)
      númeroDeDivisores = númeroDeDivisores + 1;
  }

  // Imprime a resposta.
  if (númeroDeDivisores == 2)
```

```
      System.out.println(x + " é primo.");
    else
      System.out.println(x + " não é primo.");
  }
```

6. O laço no nosso `verificaPrimalidade` é executado mais vezes do que o necessário. Na verdade poderíamos parar assim que `candidatoADivisor` chegar a `x/2` ou mesmo ao chegar à raiz quadrada de `x`. Pense como mudar o programa levando em consideração estes novos limitantes.

7. Escreva uma classe Euclides, com um método `mdc` que recebe dois números inteiros `a1` e `a2`, estritamente positivos, com `a1 >= a2`, e devolve o máximo divisor comum entre eles, utilizando o algoritmo de Euclides.

   Breve descrição do algoritmo de Euclides (para maiores detalhes, consulte seu professor de Álgebra):

   - Dados `a1` e `a2`, com `a1` >= `a2`, quero o m.d.c.(`a1`, `a2`).
   - Calcule `a3` = `a1` % `a2`.
   - Se `a3` = 0, fim. A solução é `a2`.
   - Calcule `a4` = `a2` % `a3`.
   - Se `a4` = 0, fim. A solução é `a3`.
   - Calcule `a5` = `a3` % `a4`.
   - ...

   Nota importante: o operador binário % calcula o resto da divisão de n por m, quando utilizado da seguinte maneira: n % m. Curiosidade: ele também funciona com números negativos! Consulte seu professor de Álgebra ;-)

Capítulo 9

Expressões e Variáveis Lógicas

Quais novidades veremos neste capítulo?

- Condições como expressões lógicas;
- Variáveis booleanas;
- Condições compostas e operadores lógicos: `&&`, `||` e `!`;
- Precedência de operadores.

9.1 Condições como expressões

Já vimos que em Java e outras linguagens de programação, as condições exercem um papel fundamental. São elas que permitem que diferentes ações sejam tomadas de acordo com o contexto. Isso é feito através dos comandos `if` e `while`.

Mas, o que são condições realmente? Vimos apenas que elas consistem geralmente em comparações, usando os operadores `==`, `>=`, `<=`, `>`, `<` e `!=`, entre variáveis e/ou constantes. Uma característica interessante em linguagens de programação é que as condições são na verdades expressões que resultam em verdadeiro ou falso. Vamos ver isso no DrJava:

```
Welcome to DrJava.
> 2 > 3
false
> 3 > 2
true
> int a = 2
> a == 2
true
> a >= 2
true
```

```
> a < a + 1
true
>
```

Vejam que cada vez que digitamos uma condição o DrJava responde `true` (para verdadeiro) ou `false` (para falso).

Para entender bem o que ocorre, é melhor imaginar que em Java as condições são expressões que resultam em um dos dois valores lógicos: "verdadeiro" ou "falso". Neste sentido, Java também permite o uso de variáveis para guardar os resultados destas contas, como vemos abaixo.

```
> boolean comp1 = 2 > 3
> comp1
false
> int a = 3
> boolean comp2 = a < a + 1
> comp2
true
>
```

Com isso, acabamos de introduzir mais um tipo de variável, somando-se aos tipos `int` e `double` já conhecidos: o tipo `boolean`, que é usado em variáveis que visam conter apenas os valores booleanos (verdadeiro ou falso). O nome é uma homenagem ao matemático inglês George Boole (1815-1864). Em português este tipo de variável é chamada de variável *booleana*

Agora que começamos a ver as comparações como expressões que calculam valores booleanos, torna-se mais natural a introdução dos operadores lógicos. Nós todos já estamos bem acostumados a condições compostas. Algo como "eu só vou à praia se tiver sol *e* as ondas estiverem boas". Nesta sentença a conjunção *e* une as duas condições em uma nova condição composta que é verdadeira somente se as duas condições que a formam forem verdadeiras.

Em Java o "*e*" lógico é representado pelo estranho símbolo `&&`. Ou seja, uma condição do tipo `1 <= a <= 10` seria escrita em Java como `a >= 1 && a <= 10`. Da mesma forma temos um símbolo para o *ou* lógico. Ele é o símbolo `||`. Isso mesmo, duas barras verticais. Por fim, o símbolo `!` antes de uma expressão lógica nega o seu valor. Por exemplo, a condição "a não é igual a 0"poderia ser escrita em Java como `!(a == 0)`[1].

Tabelas da verdade contém todos os resultados que podem ser obtidos ao se aplicar uma operação lógica sobre variáveis booleanas. Abaixo apresentamos as tabelas da verdade para os operadores `&&`, `||` e `!`.

`&&` (*e*)	`true`	`false`
`true`	`true`	`false`
`false`	`false`	`false`

`||` (*ou*)	`true`	`false`
`true`	`true`	`true`
`false`	`true`	`false`

`!` (*não*)	`true`	`false`
	`false`	`true`

[1] Daí vem a explicação para o fato do sinal de diferente conter o ponto de exclamação.

Por fim, podemos montar expressões compostas unindo, através dos operadores descritos acima, condições simples ou respostas de expressões lógicas anteriores que foram armazenadas em variáveis boolenas. Mais uma vez um exemplo vale mais que mil palavras.

```
Welcome to DrJava.
> (2 > 3) || (2 > 1)
true
> boolean comp1 = 2 > 3
> comp1
false
> comp1 && (5 > 0)
false
> !(comp1 && (5 > 0))
true
> int a = 10
> (a > 5) && (!comp1)
true
> boolean comp2 = (a > 5) && comp1
> comp2
false
>
```

Também podemos "misturar" operadores aritméticos e comparadores, sempre que isso faça sentido. Por exemplo,

```
> (a - 10) > 5
false
> a - (10 > 5)
koala.dynamicjava.interpreter.error.ExecutionError: Bad type in subtraction
>
```

Note que a última expressão resultou em um erro. Afinal de contas, ela pede para somar, a uma variável inteira, o resultado de uma expressão cujo valor é booleano, misturando tipos. Isto não faria sentido em Java.

Outra coisa que pode ser feita é a criação de métodos que devolvem um valor booleano. Assim a resposta dada por esses métodos pode ser usada em qualquer lugar onde uma condição faça sentido, como um `if` ou um `while`. Por exemplo, se alterarmos o método `verificaPrimalidade`, dado no capítulo anterior, para devolver a resposta (se o número é primo ou não), em vez de imprimir na tela, teríamos o seguinte método `éPrimo`:

```
/**
 * Verifica se um número inteiro positivo é primo ou não.
 */
boolean éPrimo(int x)
{
  // Todos os números inteiros positivos sao divisíveis por 1.
  int númeroDeDivisores = 1;
  // O primeiro candidato a divisor não trivial é o 2.
  int candidatoADivisor = 2;
```

```
  // Testa a divisão por todos os números menores ou iguais a x.
  while (candidatoADivisor <= x)
  {
    if (x % candidatoADivisor == 0)
      númeroDeDivisores = númeroDeDivisores + 1;
    candidatoADivisor = candidatoADivisor + 1;
  }

  if (númeroDeDivisores == 2)
    return true;
  else
    return false;
}
```

Note que, desta forma, podemos escrever algo do tipo:

```
if (!éPrimo (x))
   // faça alguma coisa
```

A construção acima deixa muito claro o significado da expressão, pois a lemos como "se não é primo x", o que é bem próximo do que seria uma frase falada em português: "se x não é primo". Quanto mais próximo for o seu código da linguagem falada, mais fácil será para outras pessoas o compreenderem; e a clareza do código é um dos principais objetivos do bom programador.

9.2 Precedência de operadores

Como acabamos de apresentar vários operadores novos, devemos estabelecer a precedência entre eles. Lembre-se que já conhecemos as regras de precedência dos operadores aritméticos há muito tempo. Já a precedência dos operadores lógicos é coisa nova. A tabela abaixo apresenta os operadores já vistos, listados da precedência mais alta (aquilo que deve ser executado antes) à mais baixa:

operadores unários	- !
operadores multiplicativos	* / %
operadores aditivos	+ -
operadores de comparação	== != > < >= <=
"*e*" lógico	&&
"*ou*" lógico	||
atribuição	=

Entre operadores com mesma precedência, as operações são computadas da esquerda para a direita.

Note, porém, que, nos exemplos acima, abusamos dos parênteses mesmo quando, de acordo com a tabela de precedência, eles são desnecessários. Sempre é bom usar parênteses no caso de expressões lógicas (ou mistas), pois a maioria das pessoas não consegue decorar a tabela acima. Assim, mesmo que você tenha uma ótima memória, o seu código torna-se mais legível para a maioria dos mortais.

9.3 Exemplos

Primeiro, vamos retomar o método `verificaLados` da classe `TianguloRetângulo3` vista no Capítulo 6. Nele, testamos se não há lado de comprimento nulo. Entretanto, parece mais natural e correto forçar todos os lados a terem comprimento estritamente positivo:

```
if ((a > 0) && (b > 0) && (c > 0))
{
  // Aqui vão os comandos para verificar a condição pitagórica.
}
```

Podemos também usar condições compostas para escrever uma versão mais rápida do método `éPrimo` do capítulo anterior.

```
/**
 * Verifica se um número inteiro positivo é primo ou não.
 */
boolean éPrimo(int x)
{
  // Todos os números inteiros positivos são divisíveis por 1.
  int númeroDeDivisores = 1;
  // O primeiro candidato a divisor não trivial é o 2.
  int candidatoADivisor = 2;

  // Testa a divisão por todos os números menores ou iguais a x/2 ou
  // até encontrar o primeiro divisor.
  while ((candidatoADivisor <= x/2) && (númeroDeDivisores == 1))
  {
    if (x % candidatoADivisor == 0)
      númeroDeDivisores = númeroDeDivisores + 1;
    candidatoADivisor = candidatoADivisor + 1;
  }

  if ((númeroDeDivisores == 1) && (x != 1) && (x != 0) && (x != -1))
    return true;
  else
    return false;
}
```

Melhor ainda podemos finalmente escrever a classe `GeradorDePrimos` de forma completa. O método mais interessante é o `próximoPrimo` que devolve o primeiro número primo maior do que o último gerado. Este exemplo já é bem sofisticado, vocês terão que estudá-lo com calma. Uma sugestão: tentem entender o que o programa faz, um método por vez. O único método mais complicado é o `éPrimo`, mas este nós já vimos.

```
class GeradorDePrimos
{
  // Limite inferior para busca de um novo primo.
  int limiteInferior = 1;

  /**
   * Permite mudar o limite para cômputo do próximo primo.
   */
  void carregaLimiteInferior(int limite)
  {
```

```
    limiteInferior = limite;
  }

  /**
   * Verifica se um número inteiro positivo é primo ou não.
   */
  boolean éPrimo(int x)
  {
    // Todos os números inteiros positivos sao divisíveis por 1.
    int númeroDeDivisores = 1;
    // O primeiro candidato a divisor não trivial é o 2.
    int candidatoADivisor = 2;

    // Testa a divisão por todos os números menores ou iguais a x/2 ou
    // até encontrar o primeiro divisor.
    while ((candidatoADivisor <= x/2) && (númeroDeDivisores == 1))
    {
      if (x % candidatoADivisor == 0)
        númeroDeDivisores = númeroDeDivisores + 1;
      candidatoADivisor = candidatoADivisor + 1;
    }

    if ((númeroDeDivisores == 1) && (x != 1) && (x != 0) && (x != -1))
      return true;
    else
      return false;
  }

  /**
   * A cada chamada, encontra um novo primo maior que limiteInferior.
   */
  int próximoPrimo()
  {
    // Busca o primeiro primo depois do limite.
    limiteInferior = limiteInferior + 1;
    while (!éPrimo(limiteInferior))
      limiteInferior = limiteInferior + 1;

    return limiteInferior;
  }
}
```

Note o uso do atributo `limiteInferior` no exemplo acima. Ele guarda uma informação importante: o valor do último número primo encontrado. É importante que ele seja um atributo dos objetos do tipo `GeradorDePrimos` e não uma variável local do método `próximoPrimo()` para que o seu valor sobreviva às sucessivas chamadas ao método `próximoPrimo()`. Se ele fosse uma variável local, a cada nova execução do método, seu valor seria zerado.

Não deixem de brincar um pouco com objetos da classe `GeradorDePrimos` para enteder melhor como ela funciona! Agora um desafio para vocês: usando o método `próximoPrimo`, escrevam um novo método `void imprimePrimos (int quantidade)` que imprime uma dada quantidade de números primos a partir do `limiteInferior`. Experimentem executar o método no DrJava passando 50 como parâmetro.

Exercícios

1. Escreva uma classe `TrianguloRetângulo` com um método denominado `defineLados(double x1, double x2, double x3)` que recebe três valores e verifica se eles correspondem aos lados de um triângulo retângulo. Em caso afirmativo, o método devolve `true`, caso contrário ele devolve `false`. Note que o programa deve verificar quais dos três valores corresponde à hipotenusa. Construa duas versões do método, uma contendo três `if`s e outra contendo apenas um `if`! Em seguida, crie um novo método que verifica se ambos os métodos retornam o mesmo resultado para diferentes combinações de `x1`, `x2` e `x3`.

2. Escreva uma classe `Brincadeiras` que possua 3 atributos inteiros. Escreva um método para carregar valores nestes atributos e, em seguida, escreva os seguintes métodos:

   (a) `troca2Primeiros()` que troca os valores dos dois primeiros atributos. Por exemplo, se antes da chamada do método o valor dos atributos é <1, 2, 3>, depois da chamada, eles deverão valer <2, 1, 3>.

   (b) `imprime()` que imprime o valor dos 3 atributos.

   (c) `imprimeEmOrdemCrescente()` que imprime o valor dos 3 atributos em ordem crescente.

3. A linguagem Java oferece operadores que, se usados corretamente, ajudam na apresentação e digitação do código, tornando-o mais enxuto. Veremos neste exercício dois deles: os operadores de incremento e de decremento. Verifique o funcionamento desses operadores usando os métodos da classe abaixo.

```
class Experiência
{
  void verIncremento(int n)
  {
    int x = n;
    System.out.println("Número inicial x -> " + x);
    System.out.println("x++              -> " + x++);
    System.out.println("Novo valor de x  -> " + x);

    x = n;
    System.out.println("Número inicial x -> " + x);
    System.out.println("++x              -> " + ++x);
    System.out.println("Novo valor de x  -> " + x);
  }
  void verDecremento(int n)
  {
    int x = n;
    System.out.println("Número inicial x -> " + x);
    System.out.println("x--              -> " + x--);
    System.out.println("Novo valor de x  -> " + x);

    x = n;
    System.out.println("Número inicial x -> " + x);
    System.out.println("--x              -> " + --x);
    System.out.println("Novo valor de x  -> " + x);
  }
}
```

Entenda bem o código e observe os resultados. Em seguida, tire suas conclusões e compare-as com as conclusões de seus colegas.

Além dos operadores de incremento e decremento, também existem os seguintes operadores resumidos: `+=`, `-=`, `*=` e `/=`. Eles são úteis quando queremos efetuar uma operação em uma variável e guardar o resultado na mesma. Isto é, `a = a * 2;` é completamente equivalente a `a *= 2;`.