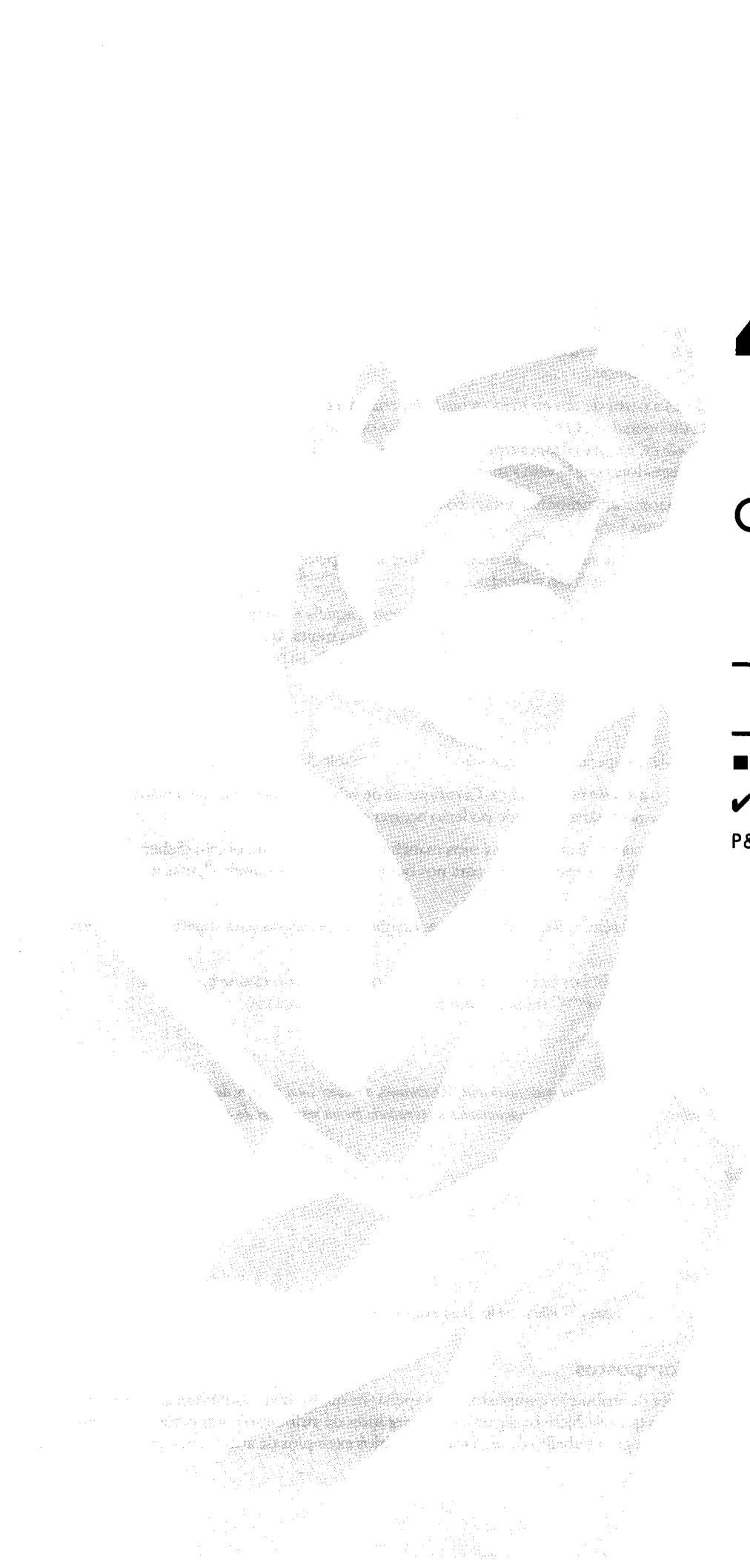

4

Operadores

Objetivos para a certificação

- ■ Usando Operadores
- ✔ Exercícios rápidos
- **P&R** Teste individual

Se você tiver variáveis, irá alterá-las. Aumentará seu valor, as somará, deslocará seus bits, as inverterá e comparará umas com as outras. Neste capítulo, você aprenderá como fazer tudo isso em Java. Adicionalmente, você aprenderá como fazer coisas que provavelmente nunca usará no mundo real, mas que é quase certo de estarem no exame.

Objetivo para a certificação

Operadores Java (Objetivo 7.6 do exame)

*7.6 Escrever código que aplique corretamente os operadores apropriados, incluindo os de atribuição (limitados a =, +=, -=), os operadores aritméticos (limitados a +, -, *, /, %, ++, - -), os operadores de comparação (limitados a <, <=, >, >=, ==, !=), o operador instanceof, os operadores lógicos (limitados a &, |, ^, !, &&, | |) e o operador condicional (?:) para produzir um resultado desejado. Escrever código que determine a igualdade de dois objetos ou dois tipos primitivos.*

Os *operadores* Java produzem novos valores a partir de um ou mais *operandos* (apenas para que fique tudo bem claro, os *operandos* são itens à esquerda ou direita do operador). O resultado da maioria das operações é um valor `booleano` ou numérico. E como você já sabe que *Java não é C++*, não ficará surpreso com o fato de seus operadores não poderem ser sobrecarregados. No entanto, há alguns operadores que já vêm sobrecarregados:

- O operador + pode ser usado para adicionar um primitivo numérico a outro, ou para realizar uma operação de concatenação se um dos operandos for uma String.
- Os operadores &, | e ^ podem todos eles ser usados de duas maneiras diferentes, apesar de que, nesta versão do exame, as suas funcionalidades para operação com bits não são cobradas.

Fique alerta. A parte do exame referente aos operadores e atribuições normalmente é aquela em que os candidatos conseguem menos pontos. Além disso, operadores e atribuições acabam entrando em muitas questões de outros tópicos... Seria uma pena você conseguir destrinchar uma questão realmente complexa sobre threads, só para errar a resposta por causa de uma instrução de pré-incremento.

Operadores de Atribuição

Abordamos a maior parte da funcionalidade do operador de atribuição, "=", no Capítulo 3. Para resumir:

- Ao se atribuir um valor a um primitivo, o *tamanho* faz diferença. Certifique-se de saber quando uma conversão implícita irá ocorrer, quando a conversão explícita é necessária, e quando poderão ocorrer cortes.
- Lembre-se de que uma variável de referência não é um objeto; é uma maneira de se chegar a um objeto (Sabemos que todos os leitores que são programadores de C++ estão loucos para nos ouvir dizer "é um ponteiro", mas não iremos fazer isso).
- Ao se atribuir um valor a uma variável de referência, o *tipo* faz diferença. Lembre-se das regras para supertipos, subtipos e arrays.

Em seguida, abordaremos mais alguns detalhes sobre os operadores de atribuição que caem no exame e, quando chegarmos no Capítulo 7, veremos como o operador "=" trabalha com Strings (que são imutáveis).

OBSERVAÇÕES PARA O EXAME

Não desperdice tempo preparando-se para tópicos que não estejam mais no exame! Resumindo, o exame Java 5 difere do exame 1.4 por se afastar dos bits e se aproximar da API. Muitos tópicos da versão 1.4 relacionados a operadores foram removidos do exame, de modo que neste capítulo você NÃO verá

- Operadores de mudança de bit
- Operadores bitwise
- Complementos binários
- Assuntos referentes à divisão por zero

Não é que esses tópicos não sejam importantes, é apenas que eles não caem mais no exame, e nosso objetivo é nos concentrarmos no exame.

Operadores de Atribuição Compostos

Na verdade, existem cerca de 11 operadores de atribuição compostos, mas apenas os quatro mais comumente usados (+=, -=, *= e /=), caem no exame (não importa o que os objetivos digam). Os operadores de atribuição compostos permitem que os digitadores preguiçosos economizem algum trabalho de digitação. Eis vários exemplos de atribuições, primeiro sem usar um operador composto,

```
y = y - 6;
x = x +2 * 5;
```

E agora com operadores compostos:

```
y -= 6;
x += 2 * 5;
```

As duas últimas atribuições dão o mesmo resultado que as duas primeiras.

OBSERVAÇÕES PARA O EXAME

Versões antigas do exame colocavam uma grande ênfase na precedência de operadores (como: Qual é o resultado de: `x = y++ + ++x/z;`*). Tirando um conhecimento bastante básico sobre a precedência (que * e / têm precedência sobre + e -, por exemplo), você não precisará estudar a precedência de operadores, exceto o fato de que, ao usar um operador composto, a expressão no lado direito do = sempre será avaliada primeiro. Por exemplo, você poderia esperar que*

```
x *= 2 + 5;
```

fosse avaliada desta forma:

```
x = (x * 2) + 5; // precedência incorreta
```

uma vez que a multiplicação têm precedência sobre a adição. Porém, em vez disso, a expressão à direita sempre é colocada entre parênteses. É avaliada desta forma:

```
x = x * (2 + 5);
```

Operadores de comparação

O exame aborda seis operadores de comparação (<, <=, >, >=, == e !=) Os operadores de comparação sempre resultam em um valor booleano (`true` ou `false`). Esse valor `booleano` é usado com mais freqüência em testes `if`, como vemos a seguir:

```
int x = 8;
if (x < 9) {
      // faz algo
}
```

mas o valor resultante também pode ser atribuído diretamente a um tipo `booleano` primitivo:

```
class CompareTest {
   public static void main(String [] args) {
      boolean b = 100 > 99;
      System.out.println("The value of b is " + b);
   }
}
```

Java tem quatro operadores de comparação que podem ser usados para comparar qualquer combinação de inteiros, números de ponto flutuante ou caracteres:

- `>` maior que
- `>=` maior ou igual a
- `<` menor que
- `<=` menor ou igual a

Examinemos algumas comparações válidas:

```
class GuessAnimal {
   public static void main(String [] args) {
      String animal = "unknown";
      int weight = 700;
      char sex = 'm';
      double colorWaveLength = 1.630;
```

```
        if (weight >= 500) animal = "elephant";
        if (colorWaveLength > 1.621) animal = "gray " + animal;
        if (sex <= 'f') animal = "female " + animal;
        System.out.println("The animal is a " + animal);
    }
}
```

No código anterior, usamos um operador de comparação entre caracteres. Também é válido comparar um tipo primitivo de caractere com qualquer número (embora não seja um estilo de programação adequado). A execução da classe anterior exibirá o seguinte:

```
The animal is a gray elephant
```

Mencionamos que os caracteres podem ser usados com operadores de comparação. Quando compara um caractere com outro ou um caractere com um número, a linguagem Java usa o valor Unicode do caractere como o valor numérico e compara os números.

Operadores de igualdade

Java tem também dois operadores de comparação (chamados de "operadores de igualdade") que comparam dois *itens* semelhantes e retornam um valor `boolean` que representa o que é verdadeiro sobre os dois itens serem iguais. Esses operadores são

- `==` igual (também conhecido como "igual a")
- `!=` diferente (também conhecido como "diferente de")

Cada comparação pode envolver dois números (incluindo o tipo `char`), dois valores `booleanos` ou duas variáveis de referência de objeto. Você não pode comparar tipos incompatíveis, no entanto. O que significaria perguntar se um tipo `boolean` é igual a um tipo `char`? Ou se um objeto `Button` é igual a um array `String`? (Exatamente, não tem sentido, e é por isso que não podemos fazê-lo). Há quatro tipos diferentes de *itens* que podem ser testados:

- Números
- Caracteres
- Tipos booleanos primitivos
- Variáveis de referência de objeto

Porém, o que o operador == examina realmente? O valor da variável – em outras palavras, o padrão de bits.

Igualdade de tipos primitivos

A maioria dos programadores está familiarizada com a comparação de valores primitivos. O código a seguir mostra alguns testes de igualdade em variáveis primitivas:

```
class ComparePrimitives {
   public static void main(String [] args) {
      System.out.println("character 'a' == 'a'? " + ('a' == 'a'));
      System.out.println("character 'a' == 'b'? " + ('a' == 'b'));
      System.out.println("5 != 6? " + (5 != 6));
      System.out.println("5.0 == 5L? " + (5.0 == 5L));
      System.out.println("true == false? " + (true == false));
   }
}
```

Esse programa produzirá a saída abaixo:

```
character 'a' == 'a'? true
character 'a' == 'b'? false
5 != 6? true
5.0 == 5L? true
true == false? false
```

Como podemos ver, se um número de ponto flutuante for comparado com um inteiro e os valores forem os mesmos, o operador == retornará `true` como esperado.

OBSERVAÇÕES PARA O EXAME

Não confunda o sinal = com o operador = = em uma expressão booleana. O código a seguir é válido:

```
11. boolean b = false;
12. if (b = true) { System.out.println("b is true");
13.} else { System.out.println("b is false"); }
```

Olhe cuidadosamente! Você pode ficar tentado a pensar que a saída será "`b is false`*", mas examine o teste booleano da linha 2. A variável booleana* `b` *não está sendo comparada com* `true`*, está sendo configurada com* `true`*, portanto,* `println` *será executado e obteremos "*`b is true`*". O resultado de qualquer expressão de atribuição é igual ao valor da variável após a atribuição. Essa substituição do sinal = pelo operador = = só funciona com variáveis* `booleanas`*, já que o teste* `if` *pode ser feito apenas em expressões booleanas. Portanto, o código abaixo não será compilado:*

```
7. int x = 1;
8. if (x = 0) { }
```

Já que `x` *é um inteiro (e não um* `booleano`*), o resultado de (*`x = 0`*) será 0 (o resultado da atribuição). Os inteiros não podem ser usados onde um valor booleano é esperado, portanto, o código da linha 8 não funcionará a menos que seja alterado de uma atribuição (=) para um teste de igualdade (= =) como vemos a seguir:*

```
8. if (x == 0) { }
```

Igualdade de variáveis de referência

Como vimos anteriormente, duas variáveis de referência podem apontar para o mesmo objeto, como o trecho de código abaixo demonstra:

```
JButton a = new Button("Exit");
JButton b = a;
```

Depois que esse código for executado tanto a variável `a` quanto `b` referenciarão o mesmo objeto (um objeto JButton com o rótulo `Exit`). As variáveis de referência podem ser testadas com o operador == para sabermos se estão referenciando um objeto em comum. Lembre-se de que o operador == estará examinando os bits da variável, portanto, com relação às variáveis de referência, se os bits das duas variáveis forem idênticos, elas estarão referenciando o mesmo objeto. Examine o código abaixo:

```
import java.awt.Button;
class CompareReference {
   public static void main(String [] args) {
      Button a = new Button("Exit");
      Button b = new Button("Exit");
      Button c = a;
      System.out.println("Is reference a == b? " + (a == b));
      System.out.println("Is reference a == c? " + (a == c));
   }
}
```

Esse código cria três variáveis de referência. As duas primeiras, `a` e `b`, são objetos JButton diferentes, que por acaso têm o mesmo nome. A terceira variável de referência, `c`, é inicializada para referenciar o mesmo objeto que `a` está referenciando. Quando esse programa for executado, a saída a seguir será produzida:

```
Is reference a == b? false
Is reference a == c? true
```

Ela nos mostra que `a` e `c` estão referenciando a mesma instância de um objeto `JButton`. O operador == não testará se dois objetos são "significativamente equivalentes", um conceito que veremos com muito mais detalhes no Capítulo 7, onde abordaremos o método `equals( )` (não confundir com o *operador* de igualdade que examinamos aqui).

Igualdade para Enums

Depois que você declarou um enum, ele não pode ser expandido. Em tempo de execução, não é possível criar novas constantes enum. É claro que você pode ter tantas variáveis quanto quiser apontando para uma dada constante enum, de forma que é importante que se possa comparar duas variáveis de referência enum para ver se são "iguais", ou seja, se elas se referem à mesma constante enum. Você pode usar ou o operador == ou o método `equals()` para determinar se duas variáveis estão se referindo à mesma constante enum:

```
class EnumEqual {
   enum Color {RED, BLUE} // ; é opcional
   public static void main(String[] args) {
      Color c1 = Color.RED; Color c2 = Color.RED;
      if(c1 == c2) { System.out.println("==");  }
      if(c1.equals(c2)) { System.out.println("dot equals");  }
} }
```

(Sabemos o quanto } } é feio, mas estamos preparando você para o exame). Esse exemplo produz a saída:

```
= =
dot equals
```

Operador de comparação instanceof

O operador `instanceof` é usado somente com variáveis de referência de objeto, e você pode empregá-lo para verificar se um objeto é de um tipo específico. Por tipo, queremos dizer tipo de interface ou classe – em outras palavras, se o objeto referenciado pela variável à esquerda do operador passaria no teste É-MEMBRO do tipo de interface ou classe do lado direto do operador (o Capítulo 2 abordou os relacionamentos É-MEMBRO com detalhes). O exemplo abaixo:

```
public static void main (String [] args) {
   String s = new String("foo");
   if (s instanceof String) {
      System.out.print("s is a String");
   }
}
```

exibe isto: `s is a String`

Mesmo se o objeto testado não for realmente uma instanciação do tipo de classe que estiver do lado direito do operador, `instanceof` ainda retornará verdadeiro se o objeto puder ser atribuído a esse tipo.

O exemplo a seguir demonstra o teste em um objeto com o uso de `instanceof`, para saber se ele é a instância de um dos seus subtipos, antes de se tentar uma conversão "redutora":

```
class A { }
class B extends A {
   public static void main (String [] args) {
      A myA = new B();
      m2(myA);
   }
   public static void m2(A a) {
      if (a instanceof B)
         ((B)a).doBstuff(); // convertendo uma referência A
                            // em uma referência B
   }
   public static void doBstuff() {
      System.out.println("'a' refers to a B");
   }
}
```

O código acima compila e produz esta saída:

```
'a' refers to a B
```

Em exemplos como esse, o uso do operador `instanceof` protege o programa contra a tentativa de tentar uma conversão inválida.

Você pode testar a referência a um objeto confrontando-a com seu próprio tipo de classe ou com qualquer uma de suas superclasses. Isso significa que *qualquer* referência a um objeto será avaliada como verdadeira se você usar o operador `instanceof` junto ao tipo `Object`, da seguinte forma,

```
B b = new B();
if (b instanceof Object) {
   System.out.print("b is definitely an Object");
}
```

que exibirá:

```
b is definitely an Object
```

OBSERVAÇÕES PARA O EXAME

Procure perguntas com instanceof que testem se um objeto é a instância de uma interface, quando a classe do objeto for implementada indiretamente. Uma implementação indireta ocorre quando uma das superclasses do objeto implementa uma interface, mas a própria classe da instância não o faz – por exemplo,

```
interface Foo { }
class A implements Foo { }
class B extends A { }
. . .
A a = new A();
B b = new B();
```

os resultados a seguir serão verdadeiros:

```
a instanceof Foo
b instanceof A
b instanceof Foo // implementado indiretamente
```

Um objeto será considerado de um tipo de interface específico (significando que passará no teste `instanceof`*) se alguma de suas superclasses implementar a interface.*

Além disso, é válido testar se a referência de um objeto `null` é a instância de uma classe. Isso sempre resultará em `false`, é claro. Por exemplo:

```
class InstanceTest {
   public static void main(String [] args) {
      String a = null;
      boolean b = null instanceof String;
      boolean c = a instanceof String;
      System.out.println(b + " " + c);
   }
}
```

exibe isto: `false false`

Erro de Compilação de instanceof

Não é possível usar o operador `instanceof` para testar em duas hierarquias de classes diferentes. Por exemplo, o seguinte código não compilará:

```
class Cat { }
class Dog {
   public static void main(String [] args) {
      Dog d = new Dog();
      System.out.println(d instanceof Cat);
   }
}
```

A compilação falha - `d` jamais poderá se referir a um `Cat` ou a um subtipo de `Cat`.

OBSERVAÇÕES PARA O EXAME

Lembre-se que os arrays são objetos, mesmo quando armazenam tipos primitivos. Procure códigos que possam ter essa aparência na pergunta:

```
int [] nums = new int[3];
if (nums instanceof Object) { } // o resultado é true
```

Um array será sempre uma instância de Object. Qualquer array.

A Tabela 4-1 resume o uso de `instanceof` presumindo-se o seguinte:

```
interface Face { }
class Bar implements Face{ }
class Foo extends Bar { }
```

Tabela 4-1 Operandos e resultados do uso do operador `instanceof`

Primeiro operando (referência sendo testada)	**Operando de instanceof (tipo que compararemos com a referência)**	**Resultado**
null	Qualquer tipo de interface ou classe	false
instância de Foo	Foo, Bar, Face, Object	true
instância de Bar	Bar, Face, Object	true
instância de Bar	Foo	false
Foo[ ]	Foo, Bar, Face	false
Foo[ ]	Object	true
Foo[1]	Foo, Bar, Face, Object	true

Operadores aritméticos

Temos certeza de que você está familiarizado com os operadores aritméticos básicos.

- `+` adição
- `-` subtração
- `x` multiplicação
- `/` divisão

Eles podem ser usados da maneira padrão:

```
int x = 5 * 3;
int y = x - 4;
System.out.println("x - 4 is " + y); // Exibirá 11
```

O Operador Resto (%)

Um operador com o qual você pode não estar familiarizado é o operador de resto %. O operador de resto divide o operando esquerdo pelo direito com o resultado sendo o resto, como o código abaixo demonstra:

```
class MathTest {
   public static void main (String [] args) {
      int x = 15;
      int y = x % 4;
     System.out.println("The result of 15 % 4 is the remainder of
15 divided by 4. The remainder is " + y);
   }
}
```

A execução da classe MathTest exibirá a linha a seguir:

```
The result of 15 % 4 is the remainder of 15 divided by 4. The remainder is 3
```

(Lembre-se: As expressões são avaliadas da esquerda para a direita, por padrão. Você pode modificar essa seqüência, ou *procedência*, adicionando parênteses. Lembre-se também de que os operadores *, / e % têm maior precedência do que os operadores + e -.)

Operador de concatenação de strings

O sinal de adição também pode ser usado para concatenar duas strings, como vimos anteriormente (e com certeza veremos de novo):

```
String animal = "Grey " + "elephant";
```

A concatenação de Strings se tornará interessante quando você combinar números com objetos String. Verifique o código a seguir:

```
String a = "String";
int b = 3;
int c = 7;
System.out.println(a + b + c);
```

O operador + atuará como um sinal de adição quando somar as variáveis int de `b + c`? Ou tratará `3` e `7` como caracteres e os concatenará individualmente? O resultado será `String10` ou `String37`? Certo, você teve tempo suficiente para pensar sobre isso. Os valores `int` foram simplesmente tratados como caracteres e acrescentados à direita da string. Portanto, o resultado é:

```
String37
```

De forma que poderíamos ler o código anterior assim:

> "Começa com a `String a`, "`String`", e adiciona o caractere `3` (o valor de `b`) a ela, para produzir uma nova string, "`String3`", adicionando, então, o caractere `7` (o valor de `c`) ao resultado para produzir outra string, "`String37`", e exibi-la em seguida".

No entanto, se você inserir parênteses envolvendo as duas variáveis `int`, como vemos abaixo:

```
System.out.println(a + (b + c));
```

obterá: `String10`

O uso de parênteses fará com que (`b + c`) seja tratado primeiro, portanto, o operador + funcionará como o operador de adição, dado que os dois operandos são valores `int`. O ponto chave aqui é que, dentro dos parênteses, o operando da esquerda não é uma `String`. Se fosse, o operador + executaria a concatenação de `Strings`. O código anterior pode ser lido da seguinte forma:

> "Soma os valores de `b + c`, em seguida, converte a soma em uma `string` e a concatena com a `string` da variável `a`".

A regra a lembrar é:

> *Se um dos operandos for uma* `String`*, o operador + atuará como um operador de concatenação de* `Strings`*. Se os dois operandos forem números, o operador + funcionará como o operador de adição.*

Em algumas situações você poderá ter problemas para descobrir se, digamos, o operador da esquerda é ou não uma `String`. No exame, não espere que isso seja óbvio. (Na verdade, já que mencionamos nisso, *nunca* espere que seja óbvio). Examine o código a seguir:

```
System.out.println(x.foo() + 7);
```

Você não terá meios de saber como o operador + está sendo usado até que descubra o que o método `foo( )` retornará! Se retornar uma `String`, então, `7` será concatenado a string retornada. Mas se `foo( )` retornar um número, o operador + será usado para somar o algarismo `7` com o valor retornado por `foo( )`.

Finalmente, você precisa saber que é válido misturar o operador aditivo composto (+=) com Strings, desta forma:

```
String s = "123";
s += "45";
s += 67;
System.out.println(s);
```

Uma vez que nos dois casos o operador += foi usado, e o operando da esquerda era uma String, ambas as operações foram concatenações, resultando em

```
1234567
```

OBSERVAÇÕES PARA EXAME

Se você não compreender como a concatenação de Strings funciona, principalmente dentro de uma instrução de saída, poderá realmente ser mal sucedido no exame, mesmo se souber o resto da reposta à pergunta! Já que tantas perguntas indagam, "Qual será o resultado?", você precisa saber não só qual a saída do código que estiver sendo executado, mas também como esse resultado será exibido. Mesmo havendo pelo menos meia dúzia de perguntas que testarão diretamente seu conhecimento em strings, a concatenação de strings aparecerá em outras perguntas de cada objetivo. Faça um teste! Por exemplo, você pode ver uma linha como

```
int b = 2;
System.out.println("" + b + 3);
```

que exibirá

```
23
```

mas se a instrução de saída for alterada para

```
System.out.println(b + 3);
```

então, o resultado será 5.

Acréscimo e decréscimo

A linguagem Java possui dois operadores que aumentarão ou diminuirão uma variável em exatamente uma unidade. Esses operadores são compostos por dois sinais de adição (++) ou dois sinais de subtração (- -):

- ++ acréscimo (prefixo e sufixo)
- - - decréscimo (prefixo e sufixo)

O operador é inserido antes (prefixo) ou depois (sufixo) de uma variável para alterar o valor. O fato de o operador vir antes ou depois do operando pode alterar o resultado de uma expressão. Examine o código abaixo:

```
1. class MathTest {
2.    static int players = 0;
3.    public static void main (String [] args) {
4.       System.out.println("players online: " + players++);
5.       System.out.println("The value of players is " + players);
6.       System.out.println("The value of players is now " + ++players);
7.    }
8. }
```

Observe que na quarta linha do programa, o operador de incremento está *depois* da variável `players`. Isso significa que estamos usando o operador pós-fixado, o que fará com que a variável `players` seja aumentada em uma unidade, *mas somente depois que seu valor for usado na expressão.* Quando executarmos esse programa, ele exibirá o seguinte:

```
%java MathTest
players online: 0
The value of players is 1
The value of players is now 2
```

Observe que quando a variável é exibida na tela, primeiro informa que o valor é igual a 0. Já que usamos o operador de incremento *pós-fixado*, o acréscimo não ocorrerá até o momento *após* a variável `players` ser usada na instrução de saída. Entendeu? A partícula *pós* de pós-fixado significa *depois*. A linha seguinte, a de número 5, não aumenta o valor de `players` ; apenas o exibe na tela, portanto, o valor recém-aumentado a ser exibido será 1. A linha 6 aplica o operador *pré-fixado* a `players`, o que significa que o acréscimo ocorrerá *antes* que o valor da variável seja usado (*pré* significa *antes*). Portanto, a saída será igual a 2.

Espere por perguntas que combinem os operadores de incremento e diminuição com outros operadores, como no exemplo abaixo:

```
int x = 2;
int y = 3;
if ((y == x++) | (x < ++y)) {
   System.out.println(«x = « + x + « y = « + y);
 }
```

O código anterior exibirá `x = 3 y = 4`

Você pode ler o código assim: "Se 3 for igual a 2 OU menor do que 4..."

A primeira expressão compara `x` e `y`, e o resultado é `false`, porque o acréscimo de `x` não ocorrerá antes que o teste do operador == seja executado. A seguir, aumentamos `x` e agora ele é igual a 3. Depois verificamos se `x` é menor do que `y`, mas *aumentamos o valor de* `y` *antes de compará-lo com* `x`! Portanto, o segundo teste lógico é (3<4). O resultado é `true`, de modo que a instrução de saída será executada.

OBSERVAÇÕES PARA O EXAME

Procure perguntas que usem os operadores de incremento e diminuição em uma variável `final`. *Já que as variáveis* `final` *não podem ser alteradas, os operadores de incremento e diminuição não poderão ser usados com elas e qualquer tentativa de fazer isso resultará em um erro do compilador. O código a seguir não será compilado,*

```
final int x = 5;
int y = x++;
```

e produzirá o erro

```
Test.java:4: cannot assign a value to final variable x
int y = x++;
```

Você pode esperar uma violação como essa camuflada em um trecho complexo de código. Se identificá-la, saberá que o código não será compilado e poderá prosseguir sem percorrer o resto do código.

Pode parecer que essa questão está testando o seu conhecimento sobre um complexo operador aritmético, quando, na verdade, está testando o seu conhecimento sobre o modificador `final`.

Operador condicional

O *operador condicional* é um operador ternário (tem três operandos) usado para avaliar expressões `booleanas` de modo semelhante à instrução `if`, exceto por, em vez de executar um bloco de código se o resultado do teste for `true`, atribuir um valor à variável. Em outras palavras, o objetivo do operador condicional é decidir qual dos dois valores atribuir a uma variável. Ele é construído com os símbolos ? (ponto de interrogação) e : (dois pontos). Os parênteses são opcionais. Sua estrutura é a seguinte:

x = (expressão `booleana`) ? valor a atribuir se `true`: valor a atribuir se `false`

Examinemos um operador condicional no código:

```
class Salary {
   public static void main(String [] args) {
      int numOfPets = 3;
      String status = (numOfPets<4)?"Pet limit not exceeded":"too many pets";
      System.out.println("This pet status is " + status);
   }
}
```

Você poderia ler o código anterior assim:

"Configura `numOfPets` igual a 3. A seguir, atribui uma `String` a variável status. Se `numOfPets` for menor do que 4, atribuirá "`Pet limit not exceeded`"; do contrário, atribuirá "`too many pets`".

O operador condicional começa com uma operação `booleana`, seguida de dois valores possíveis para a variável à esquerda do operador (=). O primeiro valor (que fica à esquerda dos dois pontos) será atribuído se o teste condicional (`booleano`) tiver um resultado `true` e o segundo valor será atribuído se o resultado do teste for `falso`. Você também pode aninhar operadores condicionais em uma instrução:

```
class AssignmentOps {
   public static void main(String [] args) {
      int sizeOfYard = 10;
      int numOfPets = 3;
      String status = (numOfPets<4)?"Pet count OK"
            :(sizeOfYard > 8)? "Pet limit on the edge"
            :"too many pets";
```

```
        System.out.println("Pet status is " + status);
    }
}
```

Não espere muitas perguntas que usem operadores condicionais, mas lembre-se de que, às vezes, os operadores condicionais são confundidos com as instruções assertivas, portanto, certifique-se de poder diferenciá-los. O Capítulo 5 abordará as assertivas com detalhes.

Operadores Lógicos

Os objetivos para o exame especificam seis operadores "lógicos" (&, |, ^, !, && e | |). Alguns documentos oficiais da Sun usam outra terminologia para esses operadores, mas, para os nossos propósitos, os "operadores lógicos" são os seis listados acima e nos objetivos para o exame.

Operadores Bitwise (Não Caem no Exame!)

Certo, isto será um pouco confuso. Dos seis operadores lógicos listados acima, três deles (&, | e ^) podem também ser usados como operadores "bitwise". Os operadores bitwise foram incluídos em versões anteriores do exame, mas não caem no exame Java 5. Eis algumas instruções válidas que usam operadores bitwise:

```
byte b1 = 6 & 8;
byte b2 = 7 | 9;
byte b3 = 5 ^ 4;
System.out.println(b1 + " " + b2 + " " + b3);
```

Os operadores bitwise comparam duas variáveis bit por bit, e retornam uma variável cujos bits foram definidos com base em se as duas variáveis sendo comparadas tinham bits correspondentes que estavam ou ambos "ligados" (&), ou um ou outro "ligado" (|), ou exatamente um "ligado" (^). A propósito, se rodarmos o código anterior, teremos

```
0 15 1
```

Tendo dito tudo isso sobre operadores bitwise, o principal a se lembrar é isto:

OS OPERADORES BITWISE NÃO CAEM NO EXAME!

Então por que falamos deles? Se cair nas suas mãos um livro antigo de preparação para o exame, ou se encontrar alguns exames simulados que ainda não foram devidamente atualizados, você poderá se deparar com questões envolvendo operações bitwise. A não ser que se sinta tão culpado a ponto de achar que merece esse tipo de punição, pode ignorar esse tipo de questão simulada.

Operadores Lógicos de Abreviação

Há cinco operadores lógicos no exame que são usados para avaliar instruções que contenham mais de uma expressão `booleana`. Desses cinco, os mais comumente usados são os dois operadores lógicos *de abreviação*. São eles:

- && E de abreviação
- | | OU de abreviação

Eles são usados para juntar pequenas expressões `booleanas` a fim de formar expressões `booleanas` maiores. Os operadores && e | | avaliam somente valores `booleanos`. Para uma expressão E (&&) ser verdadeira, ambos os operandos precisam ser `true` – por exemplo,

```
if ((2 < 3) && (3 < 4)) { }
```

A expressão anterior só é `true`, porque tanto o operando um (`2 < 3`) quanto o operando dois (`3 < 4`) são `true`.

O recurso de abreviação do operador && consiste no fato *de ele não perder seu tempo em avaliações inúteis*. O operador de abreviação && avaliará o lado esquerdo da operação primeiro (operando um) e se esse operando tiver um resultado `false`, ele não examinará o lado direito da equação (operando dois), já que o operador saberá que a expressão completa não será `true`.

```
class Logical {
   public static void main(String [] args) {
      boolean b = true && false;
      System.out.println("boolean b = " + b);
   }
}
```

Quando executarmos o código anterior, obteremos:

```
boolean b = false
```

O operador || é semelhante ao operador &&, exceto por ele avaliar como `true` se *qualquer um dos operadores* for `true`. Se o primeiro operando de uma operação OU for `true`, o resultado também será, portanto, o operador de abreviação || não perderá tempo examinando o lado direito da equação. No entanto, se o primeiro operando for `false`, o operador terá que avaliar o segundo operando para saber se o resultado da operação OU será `true` ou `false`. Preste bastante atenção no exemplo a seguir; você verá algumas perguntas como esta no exame:

```
1. class TestOR {
2.    public static void main (String [] args) {
3.     if ((isItSmall(3)) || (isItSmall(7))) {
4.        System.out.println("Result is true");
5.     }
6.     if ((isItSmall(6)) || (isItSmall(9))) {
7.        System.out.println("Result is true");
8.     }
9.    }
10.
11.   public static boolean isItSmall(int i) {
12.      if (i < 5) {
13.         System.out.println("i < 5");
14.         return true;
15.      } else {
16.       System.out.println("i >= 5");
17.          return false;
18.       }
19.   }
20. }
```

Qual será o resultado?

```
% java TestOR
i < 5
Result is true
i >= 5
i >= 5
```

Aqui está o que aconteceu quando o método `main( )` foi executado:

1. Quando chegamos à linha 3, o primeiro operando da expressão || (em outras palavras, o lado esquerdo da operação ||) foi avaliado.
2. O método `isItSmall(3)` foi chamado, exibiu "`i < 5`" e retornou *verdadeiro*.
3. Já que o primeiro operando da expressão ||, da linha 3, é igual a `true`, o operador || não avaliará o segundo operando. Portanto, nunca veremos a frase "`i >= 5`", que teria sido exibida caso o segundo operando fosse avaliado (chamando `isItSmall(7)`).
4. A linha 6 será avaliada agora, começando pelo primeiro operando da expressão ||.
5. O método `isItSmall(6)` é chamado, exibe "`i >= 5`" e retorna `false`.
6. Já que o primeiro operando da expressão ||, da linha 6, é `false`, o operador || não poderá ignorar o segundo operando; ainda há uma chance da expressão ser `true`, se o segundo operando for avaliado como `true`.
7. O método `isItSmall(9)` é chamado e exibe "`i >= 5`".
8. O método `isItSmall(9)` retorna `false`, portanto, a expressão da linha 6 é `false` e a linha 7 nunca será executada.

OBSERVAÇÕES PARA O EXAME

Os operadores && e || só funcionam com operandos `booleanos`. *O exame pode tentar enganá-lo, usando inteiros com esses operadores;*

```
if (5 && 6) { }
```

onde parece que estamos tentando usar um operador bit a bit E nos bits que representam os inteiros 5 e 6, mas o código nem mesmo será compilado.

Operadores lógicos (que não são de abreviação)

Existem dois operadores lógicos *que não são de abreviação.*

- & E não-abreviado
- | OU não-abreviado

Esses operadores são usados em expressões lógicas, assim como os operadores && e | |; porém, já que não são os operadores de abreviação, *avaliarão os dois lados da expressão, sempre!* Eles serão ineficientes. Por exemplo, mesmo se o primeiro operando (lado esquerdo) de uma expressão & for `false`, o segundo também será avaliado – *ainda que agora tenha se tornado impossível o resultado ser* `true`*!* E o operador | será ineficiente na mesma medida; se o primeiro operando for `true`, ele ainda dará prosseguimento e avaliará o segundo operando, *mesmo quando* souber *que a expressão será* `true`.

No exame, você verá muitas perguntas usando tanto operadores lógicos de abreviação quanto os que não usam a abreviação. Você terá que saber exatamente quais operandos serão avaliados e quais não serão, já que o resultado variará se o segundo operando da expressão for ou não avaliado:

```
int z = 5;
if(++z > 5 || ++z > 6) z++;   // z = 7 depois deste código
```

contra:

```
int z = 5;
if(++z > 5 | ++z > 6) z++;   // z = 8 depois deste código
```

Operadores Lógicos ^ e !

Os dois últimos operadores lógicos no exame são

- ^ OU exclusivo (XOR)
- ! inversão booleana

O operador ^ (OU exclusivo) avalia somente valores `booleanos`. O operador ^ relaciona-se com os operadores de não-abreviação que acabamos de revisar, no sentido de que sempre avalia *ambos* os operandos, o do lado direito e o do lado esquerdo, em uma expressão. Para uma expressão OU exclusivo (^) ser verdadeira, EXATAMENTE um operando precisa ser `true` – por exemplo,

```
System.out.println("xor " + ( (2<3) ^ (4>3) ) );
```

produz a saída: `xor false`

A expressão acima avalia como `false` porque TANTO o primeiro operando (2 < 3) quanto o segundo (4 > 3) avaliam como `true`.

O operador ! (inversão booleana) retorna o oposto do valor atual de um `booleano`:

```
if(!(7 == 5) ) { System.out.println("not equal"); }
```

pode ser lido como "se não for verdade que 7 == 5", e a instrução produz esta saída:

```
not equal
```

Eis um outro exemplo usando `booleanos`:

```
boolean t = true;
boolean f = false;
System.out.println("! " + (t & !f) + " " + f);
```

produz a saída:

```
! true false
```

No exemplo anterior, repare que o teste & teve sucesso (exibindo `true`), e que o valor da variável `booleana f` não se modificou, de modo que ela exibiu `false`.

Resumo para a certificação

Se você estudou este capítulo atentamente, deve ter obtido uma compreensão sólida sobre os operadores Java, e deverá saber o que significa igualdade, quando seu conhecimento sobre o operador == for avaliado. Recapitulemos os pontos

importantes do que você aprendeu neste capítulo.

Os operadores lógicos (&& e | |) só podem ser usados para avaliar duas expressões `booleanas`. A diferença entre && e & é que o operador && não se preocupará em testar o operando direito, caso o esquerdo for avaliado como `false`, pois o resultado da expressão && nunca pode ser `true`. A diferença entre | | e | é que o operador | | não testará o operando direito se o esquerdo for avaliado como `true`, porque o resultado já será reconhecido como `true` nesse momento.

O operador == pode ser usado para comparar valores de tipos primitivos, mas também pode ser empregado para determinar se duas variáveis de referência referenciam o mesmo objeto.

O operador `instanceof` é usado para determinar se o objeto sendo referido por uma variável de referência passa no teste É-UM para um tipo especificado.

O operador + é sobrecarregado para realizar tarefas de concatenação de `Strings`, e pode concatenar também `Strings` com tipos primitivos, mas cuidado – essa operação pode ser complicada.

O operador condicional (também conhecido como "operador ternário") tem uma sintaxe incomum, com três operandos – não a confunda com uma instrução assert complexa.

Os operadores ++ e - - serão usados em todo o exame, e você deve prestar a atenção a se eles são prefixados ou posfixados à variável que está sendo atualizada.

Prepare-se para muitas perguntas no exame envolvendo os tópicos deste capítulo. Mesmo em perguntas que testarem seu conhecimento referente a outro objetivo, o código freqüentemente estará usando operadores, atribuições, passagem de objetos e tipos primitivos, e assim por diante.

✓ Exercícios rápidos

Aqui estão alguns pontos-chave de cada seção deste capítulo.

Operadores de comparação (Objetivo 7.6)

- ❑ Os operadores de comparação sempre resultam em um valor `booleano` (`true` ou `false`).
- ❑ Há seis operadores de comparação: >, >=, <, <=, == e !=. Os dois últimos (== e !=) são, às vezes, chamados de operadores de igualdade.
- ❑ Quando compara caracteres, a linguagem Java usa o valor Unicode do caracter como o valor numérico.
- ❑ Operadores de igualdade
  - ❑ Quatro tipos de itens podem ser testados: números, caracteres, booleanos, variáveis de referência.
  - ❑ Há dois operadores de igualdade: == e !=.
- ❑ Ao comparar variáveis de referência, == retorna `true` somente se ambas as referências apontarem para o mesmo objeto.

Operador instanceof (Objetivo 7.6)

- ❑ `instanceof` só é usado com variáveis de referência e verifica se o objeto é de um tipo específico.
- ❑ O operador `instanceof` só pode ser usado para testar objetos (ou valores `null`) confrontando-os com tipos de classe que estejam na mesma hierarquia da classe.
- ❑ Para interfaces, um objeto passa no teste `instanceof` se alguma de suas superclasses implementar a interface do lado direito do operador `instanceof`.

Operadores aritméticos (Objetivo 7.6)

- ❑ Há quatro operadores principais: adição, subtração, multiplicação e divisão.
- ❑ O operador de resto (%) retorna o resto de uma divisão.
- ❑ As expressões são avaliadas da esquerda para a direita, a não ser que você adicione parênteses, ou que alguns operadores na expressão tenham maior precedência do que outros.
- ❑ Os operadores *, / e % têm maior precedência do que + e -.

Operador de concatenação de strings (Objetivo 7.6)

- ❑ Se um dos operandos for uma `String`, o operador + concatenará os operandos.
- ❑ Se os dois operandos forem numéricos, o operador + somará os operandos.

Operadores de acréscimo/decréscimo (Objetivo 7.6)

- O operador pré-fixado (++ e --) será executado antes do valor ser usado na expressão.
- O operador pós-fixado (++ e --) será executado depois que o valor for usado na expressão.
- Em qualquer expressão, os dois operandos são avaliados integralmente antes que o operador seja aplicado.
- O valor das variáveis `final` não pode ser aumentado ou diminuído.

Ternário (operador condicional) (Objetivo 7.6)

- Retorna um entre dois valores baseando-se nos casos de uma expressão `booleana` ser `true` ou `false`.
- Retorna o valor depois de ? se a expressão for `true` .
- Retorna o valor depois de : se a expressão for `false`.

Operadores lógicos (Objetivo7.6)

- O exame aborda seis operadores lógicos: &, |, ^, !, && e ||.
- Os operadores lógicos trabalham com duas expressões (exceto !) que devem resultar em valores `booleanos`.
- Os operadores && e & retornarão `true` somente se os dois operandos forem *verdadeiros*.
- Os operadores || e | retornarão `true` se um ou os dois operandos forem *verdadeiros*.
- Os operadores && e || são conhecidos como operadores de abreviação.
- O operador && não avaliará o operando direito se o esquerdo for `false`.
- O operador || não avaliará o operando direito se o esquerdo for `true`.
- Os operadores & e | sempre avaliam os dois operandos.
- O operador ^ (chamado de "XOR lógico") retorna `true` se exatamente um operando for verdadeiro.
- O operador ! (chamado de operador de "inversão"), retorna o valor contrário do operando `booleano` que o precede.

Teste individual

1. Dado:

```
class Hexy {
 public static void main(String[] args) {
   Integer i = 42;
   String s = (i<40)?"life":(i>50)?"universe":"everything";
   System.out.println(s);
 }
}
```

Qual é o resultado?

A. `null`

B. `life`

C. `universe`

D. `everything`

E. A compilação falha.

F. É lançada uma exceção no tempo de execução.

2. Dado:

```
1. class Example {
2.   public static void main(String[] args) {
3.     Short s = 15;
```

```
4.     Boolean b;
5.     // insira o código aqui
6.   }
7. }
```

Qual opção, inserida indepentemente na linha 5, irá compilar? (Marque todas as corretas)

A. `b = (Number instanceof s);`

B. `b = (s instanceof Short);`

C. `b = s.instanceof(Short);`

D. `b = (s instanceof Number);`

E. `b = s.instanceof(Object);`

F. `b = (s instanceof String);`

3. Dado:

```
1. class Comp2 {
2.   public static void main(String[] args) {
3.     float f1 = 2.3f;
4.     float[][] f2 = {{42.0f}, {1.7f, 2.3f}, {2.6f, 2.7f}};
5.     float[] f3 = {2.7f};
6.     Long x = 42L;
7.     // insira o código aqui
8.       System.out.println("true");
9.   }
10. }
```

E os cinco seguintes fragmentos de código:

```
F1.  if(f1 == f2)
F2.  if(f1 == f2[2][1])
F3.  if(x == f2[0][0])
F4.  if(f1 == f2[1,1])
F5.  if(f3 == f2[2])
```

O que é verdadeiro?

A. Um deles compilará, apenas um será `true`.

B. Dois deles compilarão, apenas um será `true`.

C. Dois deles compilarão, dois serão `true`.

D. Três deles compilarão, apenas um será `true`.

E. Três deles compilarão, exatamente dois serão `true`.

F. Três deles compilarão, exatamente três serão `true`.

4. Dado:

```
class Fork {
 public static void main(String[] args) {
   if(args.length == 1 | args[1].equals("test")) {
     System.out.println("test case");
    } else {
     System.out.println("production " + args[0]);
    }
 }
}
```

E a chamada de linha de comando:

```
java Fork live2
```

Qual é o resultado?

A. `test case`

B. `production`

C. `test case live2`

D. A compilação falha.

E. É lançada uma exceção no tempo de execução.

5. Dado:

```
class Foozit {
 public static void main(String[] args) {
   Integer x = 0;
   Integer y = 0;
   for(Short z = 0; z < 5; z++)
     if((++x > 2) || (++y > 2))
       x++;
   System.out.println(x + " " + y);
 }
}
```

Qual é o resultado?

A. `5 1`

B. `5 2`

C. `5 3`

D. `8 1`

E. `8 2`

F. `8 3`

G. `10 2`

H. `10 3`

6. Dado:

```
class Titanic {
 public static void main(String[] args) {
   Boolean b1 = true;
   boolean b2 = false;
   boolean b3 = true;
   if((b1 & b2) | (b2 & b3) & b3)
     System.out.print("alpha ");
   if((b1 = false) | (b1 & b3) | (b1 | b2))
     System.out.print("beta ");
 }
}
```

Qual é o resultado?

A. `beta`

B. `alpha`

C. `alpha beta`

D. A compilação falha.

E. Não é produzida nenhuma saída.

F. É lançada uma exceção em tempo de execução.

7. Dado:

```
class Feline {
 public static void main(String[] args) {
   Long x = 42L;
   Long y = 44L;
   System.out.print(" " + 7 + 2 + " ");
   System.out.print(foo() + x + 5 + " ");
   System.out.println(x + y + foo());
 }
 static String foo() { return "foo"; }
}
```

Qual é o resultado?

A. `9 foo47 86foo`

B. `9 foo47 4244foo`

C. `9 foo425 86foo`

D. `9 foo425 4244foo`

E. `72 foo47 86foo`

F. `72 foo47 4244foo`

G. `72 foo425 86foo`

H. `72 foo425 4244foo`

I. A compilação falha.

8. Coloque os fragmentos no código para produzir a saída 33. Observação: você deve usar cada fragmento exatamente uma vez.

```
class Incr {
   public static void main(String[] args) {
      Integer x = 7;
      int y = 2;
      x ___ ___;
      ___ ___ ___;
      ___ ___ ___;
      ___ ___ ___;
      System.out.println(x);
   }
}
```

FRAGMENTOS:

y	y	y	y
y	x	x	
-=	*=	*=	*=

9. Dado:

```
1. class Maybe {
2.   public static void main(String[] args) {
3.     boolean b1 = true;
4.     boolean b2 = false;
5.     System.out.print(!false ^ false);
6.     System.out.print(" " + (!b1 & (b2 = true)));
7.     System.out.println(" " + (b2 ^ b1));
8.   }
9. }
```

O que é verdadeiro?

A. A linha 5 produz `true`.

B. A linha 5 produz `false`.

C. A linha 6 produz `true`.

D. A linha 6 produz `false`.

E. A linha 7 produz `true`.

F. A linha 7 produz `false`.

10. Dado:

```
class Sixties {
public static void main(String[] args) {
  int x = 5;  int y = 7;
  System.out.print(((y * 2) % x));
  System.out.print(" " + (y % x));
 }
}
```

Qual é o resultado?

A. `1 1`

B. `1 2`

C. `2 1`

D. `2 2`

E. `4 1`

F. `4 2`

G. A compilação falha.

H. É lançada uma exceção em tempo de execução.

Respostas

1. **D** está correta. Este é um ternário aninhado dentro de um ternário com um pouco de unboxing junto. Ambas as expressões ternárias são falsas.

   A, **B**, **C**, **E** e **F** estão incorretas com base no exposto acima. (Objetivo 7.6)

2. **B** e **D** usam corretamente boxing e `instanceof` juntos.

   A está incorreta porque os operandos estão invertidos. **C** e **E** usam sintaxe `instanceof` incorreta. **F** está errada porque `Short` não está na mesma árvore de herança que `String`. (Objetivo 7.6)

3. **D** está correta. Os fragmentos **F2**, **F3** e **F5** compilarão, e apenas **F3** é `true`.

A, **B**, **C**, **E** e **F** estão incorretas. **F1** está incorreta porque não é possível comparar um tipo primitivo com um array. **F4** usa sintaxe incorreta para acessar um elemento de um array bidimensional. (Objetivo 7.6)

4. **E** está correta. Pelo fato de a abreviação (| |) não ter sido usada, ambos os operandos são avaliados. Uma vez que `args[1]` está além dos limites do array args, é lançada uma `ArrayIndexOutOfBoundsException`.

   A, **B**, **C** e **D** estão incorretas com base no exposto acima. (Objetivo 7.6)

5. **E** está correta. Nas duas primeiras vezes em que o teste `if` é executado, tanto `x` como `y` são incrementadas uma vez (só se alcança `x++` na terceira iteração). A partir da terceira iteração do loop, `y` nunca mais é usado, por causa do operador de abreviação.

   A, **B**, **C**, **D**, **F**, **G** e **H** estão incorretas com base no exposto acima . (Objetivo 7.6)

6. **E** está correta. No segundo teste `if`, a expressão mais à esquerda é uma atribuição, e não uma comparação. Depois que `b1` foi definido como `false`, os demais testes são todos `false`.

   A, **B**, **C**, **D** e **F** estão incorretas com base no exposto acima . (Objetivo 7.6)

7. **G** está correta. A concatenação roda da esquerda para a direita e, se algum dos operandos for uma `String`, os operandos são concatenados. Se ambos forem números, eles são adicionados um ao outro. O unboxing funciona em conjunto com a concatenação.

   A, **B**, **C**, **D**, **E**, **F**, **H** e **I** estão incorretas com base no exposto acima . (Objetivo 7.6)

8.
```
class Incr {
 public static void main(String[] args) {
   Integer x = 7;
   int y = 2;
    x *= x;
   y *= y;
   y *= y;
   x -= y;
    System.out.println(x);
 }
}
```

   Sim, sabemos que isso é complicado, mas você poderá encontrar algo parecido no exame real. (Objetivo 7.6)

9. **A**, **D** e **F** estão corretas. O ^ (xor) retorna `true` se exatamente um operando for `true`. O ! inverte o valor `booleano` do operando. Na linha 6, `b2 = true` é uma atribuição, e não uma comparação, e não é avaliada porque & não serve de abreviação para ela.

   B, **C** e **E** estão incorretas com base no exposto acima . (Objetivo 7.6)

10. **F** está correta. O operador % (resto, também conhecido como módulo) retorna o resto de uma operação de divisão.

    A, **B**, **C**, **D**, **E**, **G** e **H** estão incorretas com base no exposto acima . (Objetivo 7.6)