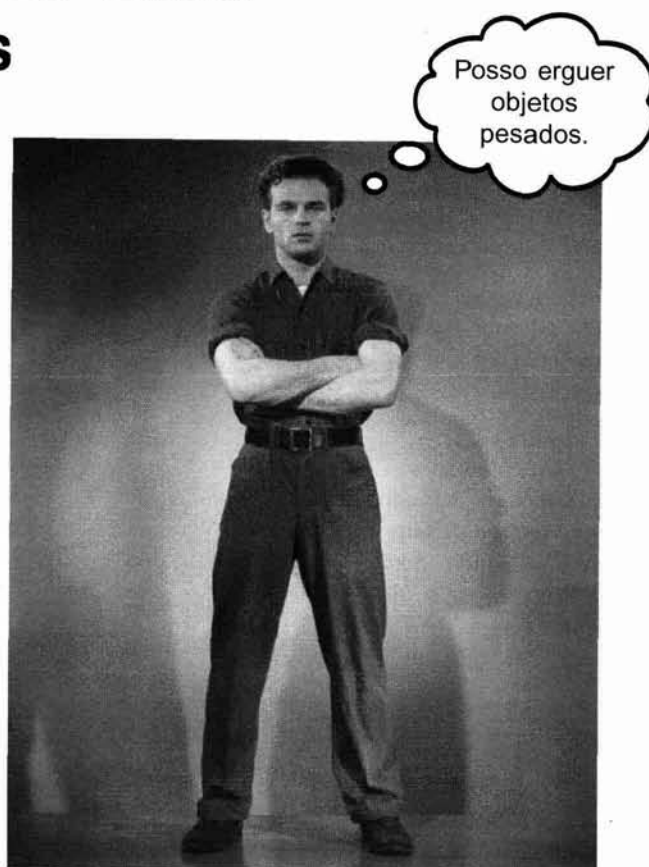


Métodos Extra Fortes



Fortaleceremos nossos métodos. Você esmiuçou as variáveis, brincou com alguns objetos e escreveu um pouco de código. Mas estávamos vulneráveis. Precisamos de mais ferramentas. Como os **operadores**. Precisamos de mais operadores, para que possamos fazer algo um pouco mais interessante do que, digamos, *latir*. E **loops**. Precisamos de loops, mas o que há de errado com os discretos loops *while*? Precisamos de loops **for** se quisermos fazer algo sério. Poderia ser útil **gerar números aleatórios**. E **converter uma string em um inteiro**, sim, isso seria avançado. É melhor aprendermos isso também. E por que não aprender tudo *criando* algo real, para sabermos como é escrever (e testar) um programa a partir do zero. **Talvez um jogo**, como a Batalha Naval. Essa é uma tarefa pesada, portanto, precisarei de *dois* capítulos para terminar. Construiremos uma versão simples neste capítulo e, em seguida, uma mais poderosa e sofisticada no Capítulo 6.

Construiremos um jogo no estilo Batalha Naval: "Sink a Dot Com"

Será você contra o computador, mas diferente do jogo de Batalha Naval real, aqui nenhum navio será nosso. Em vez disso, sua tarefa será afundar os navios do computador no menor número de tentativas.

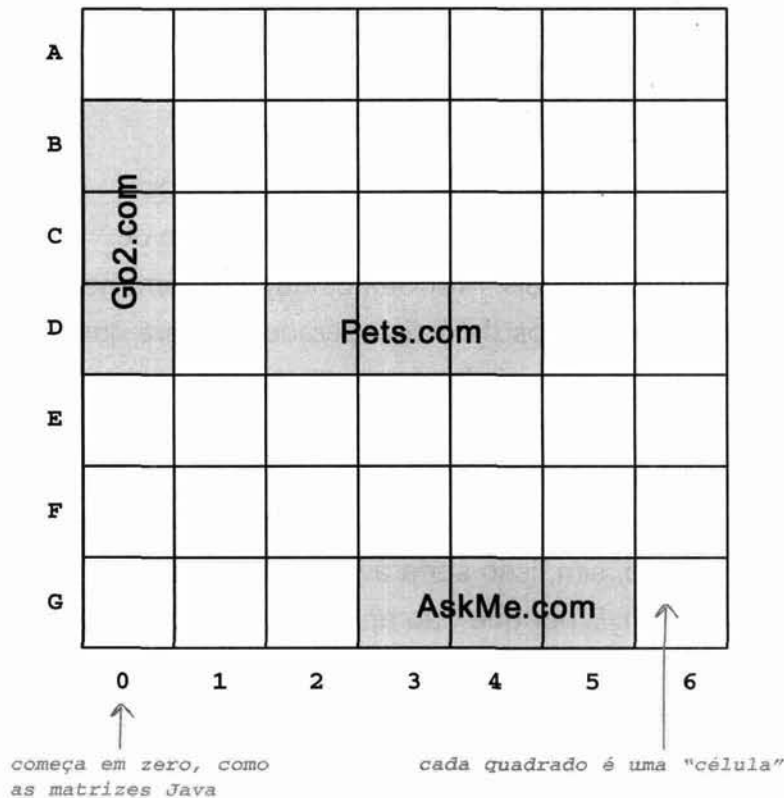
Ah, e não afundaremos navios. Eliminaremos Dot Coms (empresas na Internet). (Demonstrando assim a importância das empresas para que você possa avaliar o custo deste livro.)

Objetivo: afundar todas as Dot Coms do computador no menor número de tentativas. Você receberá uma classificação ou nível, baseado em como foi seu desempenho.

Preparação: quando o programa do jogo for iniciado, o computador inserirá três DotComs em uma **grade virtual 7 x 7**. Concluída essa etapa, o jogo solicitará seu primeiro palpite.

Como você jogará: ainda não aprendemos a construir uma GUI, portanto essa versão funcionará na linha de comando. O computador solicitará que você insira um palpite (uma célula), que deve ser digitado na linha de comando como "A3", "C5", etc. Em resposta a seu palpite, você verá um resultado na linha de comando, "Correto", "Errado" ou "Você afundou a Pets.Com" (ou qualquer que seja a Dot Com de sorte do dia). Quando você tiver eliminado todas as três Dot Coms, o jogo terminará exibindo sua classificação.

grade 7 x 7



Você vai construir o jogo Sink a Dot Com, com uma grade 7 x 7 e três Dot Coms. Cada Dot Com ocupa três células.

parte de interação do jogo

```
Arquivo Editar Janela Ajuda Vender

%java DotComBust

Insira um palpite A3
errado

Insira um palpite B2
errado

Insira um palpite C4
errado

Insira um palpite D2
correto

Insira um palpite D3
correto

Insira um palpite D4

Ora! Você afundou a Pets.com :(
eliminar

Insira um palpite B4
errado

Insira um palpite G3
correto

Insira um palpite G4
correto

Insira um palpite G5

Ora! Você afundou a AskMe.com :(
```

Primeiro, um projeto de alto nível

Sabemos que precisamos de classes e métodos, mas como eles devem ser? Para responder isso, precisamos de mais informações sobre o que o jogo deve fazer.

Primeiro, temos que descrever o fluxo geral do jogo. Aqui está a idéia básica:

1 O usuário inicia o jogo

- A O jogo cria três Dot Coms
- B O jogo insere as três Dot Coms em uma grade virtual

2 O jogo começa

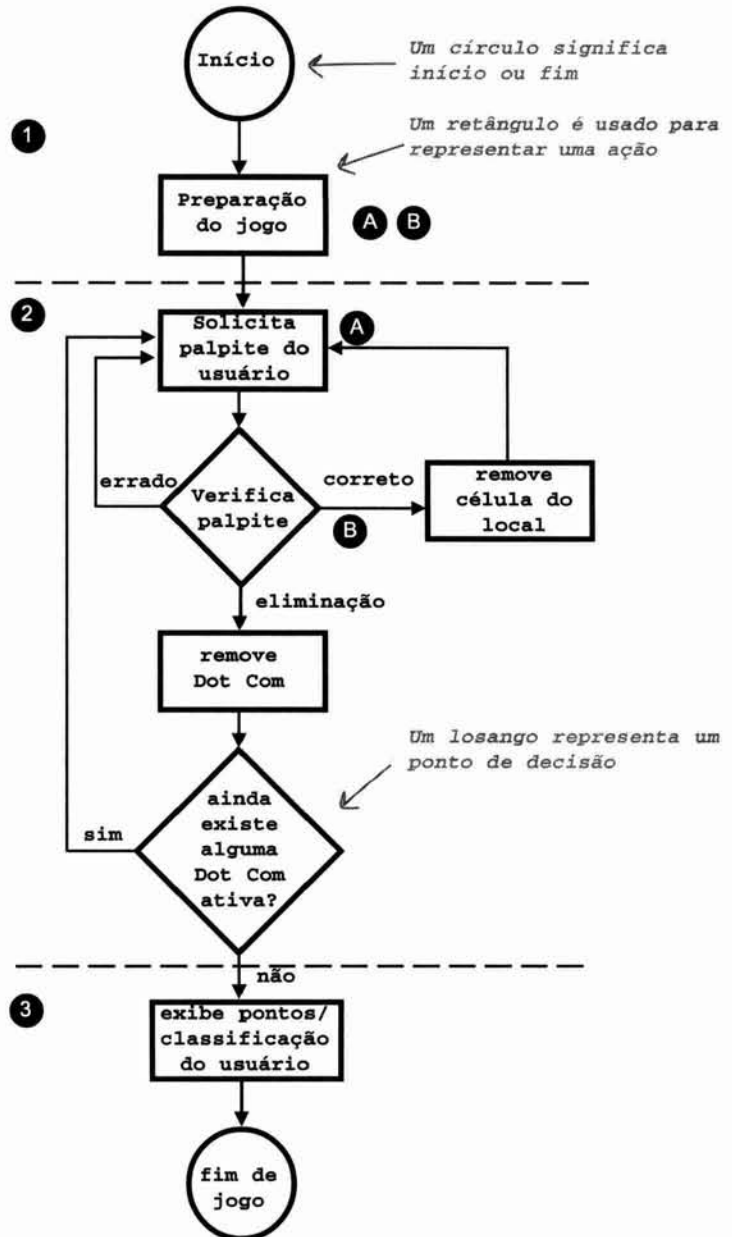
Repita as etapas a seguir até não haver mais Dot Coms:

- A Solicita ao usuário um palpite ("A2", "C0", etc.)
- B Confronta o palpite do usuário com as Dot Coms para procurar um acerto, um erro ou uma eliminação. Toma a medida apropriada: se for um acerto, excluir a célula (A2, D4, etc.). Se for uma eliminação, excluir a Dot Com.

3 O jogo termina

Fornece ao usuário uma classificação, baseando-se na quantidade de palpites.

Agora temos uma idéia do tipo de coisas que o programa precisa fazer. A próxima etapa é definir de que tipos de **objetos** precisaremos para fazer o trabalho. Lembre-se, pense como Brad em vez de



Uau! Um diagrama de fluxo real

O "Jogo Dot Com Simples"

Uma introdução mais amigável

Parece que precisaremos de pelo menos duas classes, uma classe Game e uma classe DotCom. Mas antes de construirmos o jogo *Sink a Dot Com* completo, começaremos com uma versão simplificada, o *Jogo Dot Com Simples*. Construiremos a versão simples neste capítulo, seguida pela versão sofisticada que construiremos no próximo capítulo.

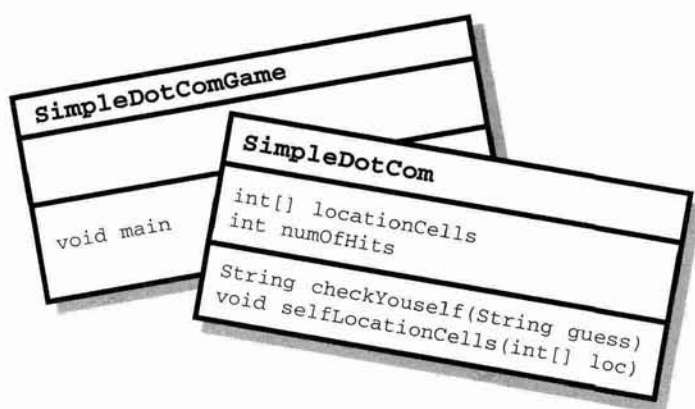
Tudo será mais simples nesse jogo. Em vez de uma grade 2-D, ocultaremos a Dot Com em uma única *linha*. E em vez de *três* Dot Coms, usaremos *uma*.

No entanto, o objetivo será o mesmo, logo, o jogo ainda precisará criar uma instância da Dot Com, atribuir a ela um local qualquer na linha, solicitar a entrada do usuário e, quando todas as células da Dot Com tiverem sido adivinhadas, o jogo terminará. Essa versão simplificada nos ajudará muito na construção do jogo completo. Se conseguirmos fazer essa versão menor funcionar, poderemos convertê-la na versão mais complexa posteriormente.

Nessa versão simples, a classe Game não terá variáveis de instância, e todo o código do jogo ficará no método main(). Em outras palavras, quando o programa for iniciado e main() começar a ser executado, ele criará uma

e somente uma instância da Dot Com, selecionará um local para ela (três células consecutivas na única linha virtual de sete células), solicitará ao usuário um palpite, verificará esse palpite e repetirá isso até todas as três células terem sido adivinhadas.

Lembre-se de que a linha virtual é... *Virtual*. Em outras palavras, ela não existe em nenhum local do programa. Contanto que o usuário e o jogo saibam que a Dot Com está oculta em três células consecutivas entre sete delas (começando em zero), a linha propriamente dita não terá que ser representada no código. Você pode ficar tentado a construir uma matriz de sete ints e, em seguida, atribuir a Dot Com a três dos sete elementos da matriz, mas não é preciso fazer isso. Tudo de que precisamos é uma matriz que contenha apenas as três células que a Dot Com ocupa.



Desenvolvendo uma classe

Como programador, provavelmente você tem uma metodologia/processo/abordagem para escrever código. Bem, nós também. Nossa sequência foi projetada para ajudá-lo a ver (e aprender) o que pensamos quando trabalhamos na codificação de uma classe. Não se trata necessariamente da maneira como nós (ou você) escrevemos códigos no dia-a-dia. É claro que, nesse contexto, você seguirá a abordagem que suas preferências pessoais, o projeto ou seu chefe impuserem. Nós, no entanto, podemos fazer o que quisermos. E quando criamos uma classe Java como uma “experiência de aprendizado”, geralmente o fazemos desta forma:

- Definimos o que a classe deve *fazer*.
- Listamos as **variáveis de instância e métodos**.
- Escrevemos um **código preparatório** para os métodos. (Você verá isso em breve.)
- Escrevemos um **código de teste** para os métodos.
- **Implementamos** a classe.
- **Testamos** os métodos.
- **Depuramos e reimplementamos** quando necessário.
- Agradecemos por não ser necessário testar nosso assim chamado aplicativo de *experiência de aprendizado* com usuários ativos reais.

- 1 O jogo é iniciado, cria UMA Dot Com e define um local para ela em três células da linha única de sete células. Em vez de “A2”, C4”, etc., os locais são apenas números inteiros. Por exemplo: 1, 2, e 3 são os locais das células nessa figura:



- 2 O jogo começa a ser disputado. Solicitará um palpite ao usuário e, em seguida, verificará se ele acertou alguma das três células da Dot Com. Se houver um acerto, ele incrementará a variável numOfHits.
- 3 O jogo terminará quando todas as três células tiverem sido adivinhadas (a variável numOfHits será igual a 3) e informará ao usuário quantos palpites ele usou para afundar a Dot Com.

Uma interação completa do jogo

```

Arquivo Editar Janela Ajuda Destruir

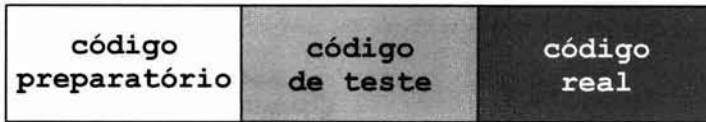
%java SimpleDotComGame
insira um número 2
correto
insira um número 3
correto
insira um número 4
errado
insira um número 1
correto
Você usou 4 palpites
    
```

O poder do cérebro

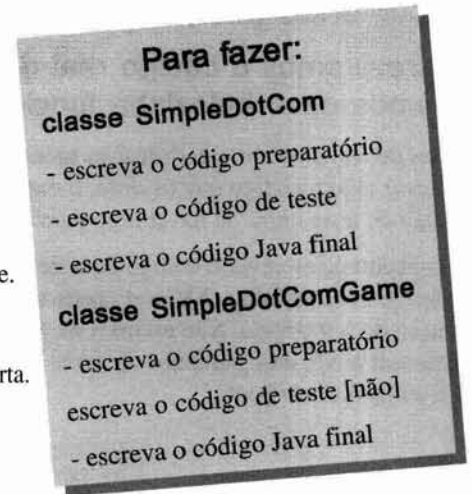
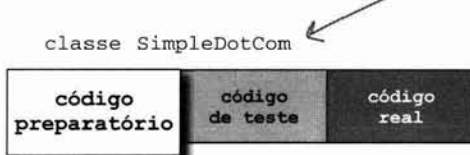
Flexione esses dendrites.

Como você definiria que classe ou classes construir *primeiro*, quando estiver escrevendo um programa? Supondo que todos os programas, exceto os menores, precisem de mais de uma classe (se você estiver seguindo os bons princípios da OO e não tiver *uma* classe que execute muitas tarefas diferentes), onde iniciaria?

As três coisas que escreveremos para cada classe:



Essa barra será exibida no primeiro conjunto de páginas para lhe mostrar em que parte você está trabalhando. Por exemplo, se você encontrar essa figura na parte superior de uma página, significa que estará trabalhando no código preparatório da classe SimpleDotCom.



código preparatório

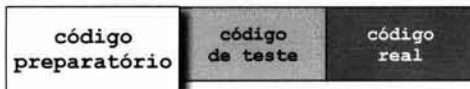
Um tipo de pseudocódigo, para ajudá-lo a focar a lógica sem se preocupar com a sintaxe.

código de teste

Uma classe ou os métodos que testarão o código real e avaliarão se ele está fazendo a coisa certa.

código real

A implementação real da classe. Trata-se do código Java real.



SimpleDotCom
<pre>int[] locationCells int numOfHits</pre>
<pre>String checkYourself(String guess) void setLocationCells(int[] loc)</pre>

Você terá uma idéia de como o código preparatório (nossa versão do pseudocódigo) funciona quando examinar esse exemplo. Ele é como um intermediário entre o código Java real e uma descrição simples da classe em português. A maioria dos códigos preparatórios inclui três partes: declarações de variáveis de instância, declarações de métodos, lógica dos métodos. A parte mais importante do código preparatório é a lógica dos métodos, porque ela define *o que* tem que acontecer, que posteriormente converteremos em *como*, quando escrevermos realmente o código do método.

Declare uma **matriz int** para armazenar os locais das células. Chame-a de **locationCells**.

Declare um **int** para armazenar o número de acertos. Chame-o de **numOfHits** e configure-o com 0.

Declare um método **checkYourself()** que use uma **String** para o palpite do usuário ("1", "3", etc.), verifique a string e retorne um resultado que represente um "acerto", "erro" ou "eliminação".

Declare um método de configuração **setLocationCells()** que use uma **matriz int** contendo os três locais das células na forma de **números inteiros** (2, 3, 4, etc.).

Método: **String checkYourself(String userGuess)**

Capture o palpite do usuário como um parâmetro de **String**

Converta o palpite do usuário em um **int**

Repita isso para cada local de célula da matriz **int**

// Confronte o palpite do usuário com o local da célula

Se o palpite do usuário estiver correto

Incremente o número de acertos

// Verifique se essa foi a última célula:

Se o número de acertos for igual a 3, **retorne** "eliminação" como resultado

Caso contrário não terá sido uma eliminação, portanto, **retorne**, "correto"

End If

Caso contrário o palpite do usuário não estará correto, portanto, **retorne** "errado"

End If

Fim da iteração

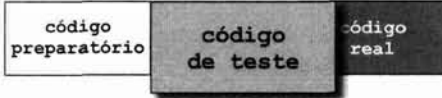
Fim do método

Método: void setLocationCells(int[] cellLocations)

Capture os locais das células como um parâmetro de *matriz int*

Atribua o parâmetro dos locais das células à variável de instância desses locais

Fim do método



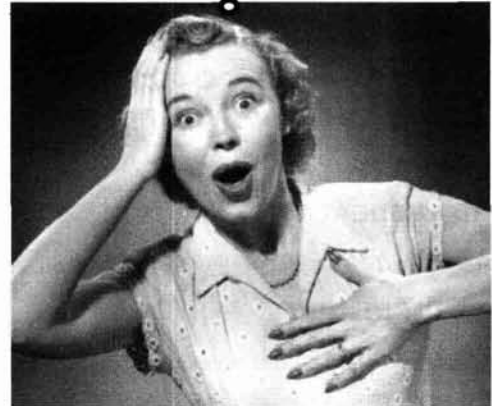
Meu Deus! Por um minuto achei que você não ia escrever seu código de teste primeiro. Nossa! Não me assuste assim.

Escrevendo a implementação dos métodos

Escreveremos o código real do método agora e faremos essa belezinha funcionar.

Antes de começarmos a codificar os métodos, faremos uma pausa para escrever algum código que os *teste*. É exatamente isso, escreveremos o código de teste *antes* de haver algo para testar!

O conceito de escrever o código de teste primeiro é uma das práticas da Extreme Programming (XP) e ela pode tornar mais fácil (e rápida) a criação de seu código. Não estamos dizendo necessariamente que você deva usar a XP, mas gostamos da parte sobre escrever os testes primeiro. E o termo XP *soa* bem.



Extreme Programming (XP)

A Extreme Programming (XP) é uma novidade no mundo da metodologia de desenvolvimento de softwares. Considerada por muitos “a maneira como os programadores querem realmente trabalhar”, a XP surgiu no fim dos anos 1990 e tem sido adotada por empresas que vão da loja de garagem com apenas duas pessoas à Ford Motor Company. O destaque da XP é que o cliente obtém o que deseja, quando deseja, mesmo quando as especificações são alteradas na última hora.

A XP se baseia em um conjunto de práticas testadas que foram projetadas para funcionar em conjunto, embora muitas pessoas selecionem algumas e adotem somente uma parte das regras. Essas práticas incluem coisas como:

Criar versões pequenas, mas frequentes.

Desenvolver em ciclos repetitivos.

Não inserir nada que não esteja na especificação (não importa o quanto você fique tentado a criar funcionalidades “para uso futuro”).

Escrever o código de teste *primeiro*.

Não seguir prazos apertados; cumprir as horas normais.

Redefinir (aperfeiçoar o código) quando e onde notar a oportunidade.

Não lançar nada que não tenha passado por todos os testes.

Definir prazos realistas, baseando-se em versões pequenas.

Manter a simplicidade.

Programar em pares e com rotatividade para que todos conheçam bem tudo sobre o código.

Escrevendo o código de teste da classe SimpleDotCom

Precisamos escrever um código de teste que consiga criar um objeto SimpleDotCom e executar seus métodos. Para a classe SimpleDotCom, só nos preocupamos realmente com o método *checkYourself()*, embora seja *preciso* implementar o método *setLocationCells()* para que o método *checkYourself()* seja executado corretamente.

Examine bem o código preparatório a seguir, do método *checkYourself()* (O método (*setLocationCells()*) é um método de configuração simples, portanto, não nos preocuparemos com ele, mas em um aplicativo ‘real’ poderíamos querer um método ‘de configuração’ mais robusto, que *pudéssemos* testar.)

Em seguida, pergunte para você mesmo: “Se o método *checkYourself()* fosse implementado, que código de teste eu poderia escrever que me provasse que ele está funcionando corretamente?”

Baseado nesse código preparatório:

```
Método: String checkYourself(String userGuess)
  Capture o palpite do usuário como um parâmetro de String
  Converta o palpite do usuário em um int
  Repita isso para cada local de célula da matriz int
    // Confronte o palpite do usuário com o local da célula
    Se o palpite do usuário estiver correto
      Incremente o número de acertos
      // Verifique se essa foi a última célula:
      Se o número de acertos for igual a 3, retorne "eliminação" como resultado
      Caso contrário não terá sido uma eliminação, portanto, retorne "correto"
    End If
  Caso contrário o palpite do usuário não estará correto, portanto, retorne "errado"
End If
Fim da iteração
Fim do método
```

Aqui está o que devemos testar:

1. Instanciar um objeto SimpleDotCom.
2. Atribuir um local para ele (uma matriz de 3 ints, como {2, 3, 4}).
3. Criar uma String que represente um palpite do usuário ("2", "0", etc.).
4. Chamar o método checkYourself(), passando para ele o palpite de usuário fictício.
5. Exibir o resultado para avaliar se está correto ("bem-sucedido" ou "com falhas").

Não existem Perguntas Idiotas

P: Talvez eu não esteja entendendo alguma coisa aqui, mas como exatamente executar um teste em algo que ainda não existe?

R: Isso não é possível. Nunca dissemos que você começaria *executando* o teste; começará *escrevendo* o teste. Quando estiver escrevendo o código de teste, você não terá nada em que usá-lo, portanto, provavelmente não poderá compilá-lo até escrever o código 'stub' que possa ser compilado, mas isso fará com que o teste falhe (por exemplo, retornando nulo).

P: Ainda não entendi. Por que não esperar até o código ser escrito e, então, projetar o código de teste?

R: O ato de planejar (e escrever) o código de teste ajudará a clarear seus pensamentos sobre o que o método propriamente dito precisa fazer.

Assim que seu código de implementação estiver concluído, você já terá um código de teste apenas esperando para validá-lo. Além disso, você *sabe* que, se não o fizer agora, *nunca* o fará. Há sempre algo mais interessante a fazer.

O ideal seria escrever um pequeno código de teste e, em seguida, criar *apenas* o código de implementação que você precisa que passe no teste. Depois escreva um pouco *mais* de código de teste e crie *apenas* o novo código de implementação que terá que passar *nesse* novo teste. A cada repetição do teste, você executará *todos* os testes já escritos, para que continue a avaliar se seus últimos acréscimos ao código não interromperão código já testado.

código preparatório	código de teste	código real
---------------------	------------------------	-------------

Código de teste da classe SimpleDotCom

```

public class SimpleDotComTestDrive {
    public static void main (String[] args) {
        SimpleDotCom dot = new SimpleDotCom();
        int[] locations = {2,3,4};
        dot.setLocationCells(locations);
        String userGuess = "2";
        String result = dot.checkYourself(userGuess);
        String testResult = "failed";
        if (result.equals("hit") ) {
            testResult = "passed";
        }
        System.out.println(testResult);
    }
}

```

← instancia um objeto SimpleDotCom

← cria uma matriz int para o local das dot com (3 ints consecutivos entre 7 possíveis)

← chama o método de configuração na variável dot com

← cria um palpite de usuário fictício

← chama o método checkYourself() no objeto dot com e passa para ele o palpite fictício

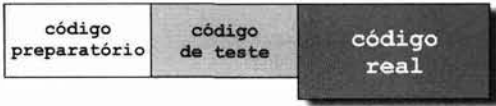
← se o palpite fictício (2) retornar um "acerto", o código estará funcionando

← exibe o resultado do teste (bem-sucedido ou com falhas)



Aponte seu lápis

Nas próximas páginas implementaremos a classe SimpleDotCom e posteriormente retornaremos à classe de teste. Se examinarmos o código de teste anterior, o que mais deve ser adicionado? O que *não* estamos testando nesse código, que *deveríamos* testar? Escreva suas idéias (ou linhas de código) a seguir:



O método checkYourself()

Não há uma conversão perfeita de código preparatório para código Java; você verá alguns ajustes. O código preparatório nos deu uma idéia muito melhor do *que* o código precisa fazer e agora temos que encontrar o código Java que consiga definir a *maneira* de fazer.

Em segundo plano, pense em que partes desse código você pode querer (ou ter que) aperfeiçoar. Os números dentro do círculo são para indicar as coisas (recursos de sintaxe e linguagem) que você ainda não viu. Elas são explicadas na outra página.



código preparatório	código de teste	código real
---------------------	-----------------	-------------

As novidades

O que ainda não vimos se encontra nesta página. Pare de se preocupar! O resto dos detalhes está no final do capítulo. Isso é o suficiente para que você possa continuar.

Um método da classe `Integer` que sabe como "converter" uma `String` no inteiro que ela representa.

Uma classe que vem com a Java.

Usa uma `String`.

1 Convertendo uma `String` em um `int`

`Integer.parseInt("3")`

Leia essa declaração do loop `for` desta forma: "repita para cada elemento da matriz '`locationCells`': extraia o próximo elemento da matriz e atribua-o à variável `int` '`cell`'."

O sinal de dois-pontos (`:`) significa "de", portanto, o conjunto todo significa "para cada valor `int` DE `locationCells`..."

2 O loop `for`

`for (int cell : locationCells) { }`

Declara uma variável que armazenará um elemento da matriz. A cada vez que o loop `for` percorrido, essa variável (nesse caso uma variável `int` chamada "`cell`"), armazenará um elemento diferente da matriz, até que não haja mais elementos (ou o código faça uma "interrupção"... Consulte o item 4 a seguir).

A matriz que o loop percorrerá. A cada vez que o loop `for` percorrido, o próximo elemento da matriz será atribuído à variável "`cell`". (Veremos mais sobre isso no final deste capítulo.)

3 O operador pós-incremento

`numOfHits++`

O sinal `++` significa somar 1 ao que quer que venha antes (em outras palavras, incrementar em 1).

`numOfHits++` é o mesmo (neste caso) que dizer `numOfHits = numOfHits + 1`, exceto por ser um pouco mais eficiente.

4 Instrução `break`

`break;`

Fará você sair de um loop. Imediatamente. Neste exato momento. Sem iteração, nenhum teste booleano, saia agora!

Não existem Perguntas Idiotas

P: O que aconteceria em `Integer.parseInt()` se você passasse algo que não fosse um número? A instrução reconhece números por extenso, como em "three"?

R: `Integer.parseInt()` só funciona com strings que representem os valores `ascii` dos dígitos (0, 1, 2, 3, 4, 5, 6, 7, 8, 9). Se você tentar converter algo como "two" ou "blurp", o código será interrompido no tempo de execução. (Por *interrompido*, queremos dizer na verdade que ele *lançará uma exceção*, mas não falaremos sobre exceções até chegarmos ao capítulo sobre elas. Portanto, por enquanto, *interrompido* é o que chega mais próximo.)

P: No começo do livro, havia um exemplo de loop *for* muito diferente desse — há dois tipos diferentes de loop *for*?

R: Sim! Na primeira versão do Java havia apenas um tipo de loop *for* (que será explicado posteriormente neste capítulo) com a seguinte aparência:

```
for (int i = 0; i < 10; i++) {
    // faz algo 10 vezes
}
```

Você pode usar esse formato para qualquer tipo de loop de que precisar. Mas... A partir do Java 5.0 (Tiger), também pode usar o loop *for* *aperfeiçoado* (essa é a descrição oficial) quando seu loop tiver que percorrer os elementos de uma matriz (ou *outro* tipo de conjunto, como você verá no *próximo* capítulo). Você sempre poderá usar o loop *for* antigo para percorrer uma matriz, mas o loop *for* *aprimorado* tornará isso mais fácil.

código
preparatóriocódigo
de testecódigo
real

Código final de SimpleDotCom e SimpleDotComTester

```
public class SimpleDotComTestDrive {

    public static void main (String[] args) {
        SimpleDotCom dot = new SimpleDotCom();
        int[] locations = {2,3,4};
        dot.setLocationCells(locations);
        String userGuess = "2";
        String result = dot.checkYourself(userGuess);
    }
}
```

```
public class SimpleDotCom {

    int[] locationCells;
    int numOfHits = 0;

    public void setLocationCells(int[] locs) {
        locationCells = locs;
    }

    public String checkYourself(String stringGuess) {
        int guess = Integer.parseInt(stringGuess);
        String result = "miss";
        for (int cell : locationCells) {
            if (guess == cell) {
                result = "hit";
                numOfHits++;
                break;
            }
        } // fora do loop

        if (numOfHits ==
            locationCells.length) {
            result = "kill";
        }
        System.out.println(result);
        return result;
    } // fecha o método
} // fecha a classe
```

O que devemos ver quando executarmos esse código?

O código de teste cria um objeto SimpleDotCom e fornece um local para ele nas posições 2, 3 e 4. Em seguida, envia um palpite de usuário fictício igual a "2" para o método checkYourself(). Se o código estiver funcionando corretamente, devemos ver a exibição do resultado:

```
Java SimpleDotComTestDrive
hit
```

Há um pequeno erro à espreita aqui. O código será compilado e executado, mas em alguns momentos... Não se preocupe por enquanto, mas *teremos* que enfrentar isso um pouco mais adiante.

código preparatório	código de teste	código real
---------------------	-----------------	-------------



Aponte seu lápis

Construímos a classe de teste e a classe SimpleDotCom. Mas ainda não temos o *jogo* real. Dado o código da página anterior, e as especificações do jogo real, escreva suas idéias para o código preparatório da classe do jogo. Forneceremos uma linha ou outra para ajudá-lo a começar. O código do jogo real está na próxima página, portanto, *não olhe a página até ter feito esse exercício!*

Você deve obter algo entre 12 e 18 linhas (incluindo as que escrevemos, porém *sem* incluir as linhas que apresentam apenas uma chave).

Método `public static void main(String[] args)`

Declare uma variável `int` para armazenar o número de palpites do usuário chamada `numOfGuesses`

Gere um número aleatório entre 0 e 4 que será a célula da posição inicial

Enquanto a dot com existir:

Capture entradas do usuário na linha de comando

A classe SimpleDotComGame precisa fazer isto:

1. Criar apenas um objeto SimpleDotCom.
2. Criar um local para ele (três células consecutivas na mesma linha de sete células virtuais).
3. Pedir ao usuário um palpite.
4. Verificar os palpites.
5. Repetir isso até a dot com ser eliminada.
6. Informar ao usuário quantos palpites ele usou.

Uma interação completa do jogo

```
Arquivo Editar Janela Ajuda Destruir
%java SimpleDotComGame
insira um número 2
correto
insira um número 3
correto
insira um número 4
errado
insira um número 1
correto
Você usou 4 palpites
```

Código preparatório da classe SimpleDotComGame

Tudo acontece em `main()`

Há algumas coisas em que você terá apenas que acreditar. Por exemplo, temos uma linha de código preparatório que diz, "CAPTURE entradas do usuário na linha de comando". Na verdade, isso vai um pouco além do que gostaríamos de implementar nesse momento. Mas, felizmente, estamos usando a OO. E isso significa que você solicitará a *outra* classe/objeto que faça o que é necessário, sem se preocupar com a *maneira* como será feito. Quando você escrever o código preparatório, deve presumir que *de alguma forma* será capaz de fazer o que for preciso, assim poderá dedicar todo o seu poder mental à construção da lógica.

código
preparatóriocódigo
de testecódigo
real

```
public static void main(String[] args)
```

Declare uma variável `int` para armazenar o número de palpites do usuário, chamada `numOfGuesses`, e configure-a com 0.

Crie uma nova instância de `SimpleDotCom`.

Gere um número aleatório entre 0 e 4 que será a célula da posição inicial.

Crie uma matriz `int` com 3 inteiros usando o número gerado aleatoriamente, esse número incrementado em 1 e esse número incrementado em 2 (exemplo: 3,4,5).

Chame o método `setLocationCells()` na instância de `SimpleDotCom`.

Declare uma variável booleana que representará o estado do jogo, chamada `isAlive`, e **configure-a** com verdadeiro.

Enquanto a dot com existir (`isAlive == true`):

Capture entradas do usuário na linha de comando.

// **Verifique** o palpite do usuário

Chame o método `checkYourself()` na instância de `SimpleDotCom`.

Incremente a variável `numOfGuesses`.

// **Verifique** se a dot com foi eliminada

Se o resultado for "kill"

Configure `isAlive` com falso (o que significa que não entraremos no loop novamente).

Exiba o número de palpites do usuário.

End If

End While

Fim do método.

dica metacognitiva



Não use apenas uma parte do cérebro por muito tempo. Usar apenas o lado esquerdo do cérebro por mais de 30 minutos é como usar apenas seu *braço* esquerdo durante esse período. Dê a cada lado de seu cérebro uma pausa, alternado-os em intervalos regulares. Quando você passar para um dos lados, o outro descansará e se recuperará. As atividades do lado esquerdo do cérebro incluem coisas como seqüências em etapas, resolução de problemas lógicos e análise, enquanto o lado direito se encarrega de metáforas, resolução de problemas que usam a criatividade, comparação de padrões e visualização.

DISCRIMINAÇÃO DOS PONTOS

- Seu programa Java deve começar com um projeto de alto nível.
- Normalmente escrevemos três coisas quando criamos uma nova classe:
código preparatório
código de teste
código real (Java)
- O código preparatório deve descrever *o que* fazer e não *como* fazê-lo. A implementação vem depois.
- Use o código preparatório como ajuda no projeto do código de teste.
- Escreva o código de teste *antes* de implementar os métodos.
- Use loops *for* em vez de *while* quando souber quantas vezes deseja repetir o código do loop.
- Use o operador *pré/pós-incremento* para adicionar uma unidade a uma variável (`x++`).
- Use o operador *pré/pós-decremento* para subtrair uma unidade de uma variável (`x--`).
- Use `Integer.parseInt()` para capturar o valor `int` de uma `String`.
- `Integer.parseInt()` só funcionará se a `String` representar um dígito ("0", "1", "2", etc.).
- Use `break` para sair antecipadamente de um loop (isto é, mesmo se a condição do teste booleano ainda for verdadeira).

código preparatório	código de teste	código real
---------------------	-----------------	-------------

O método main() do jogo

Exatamente como você fez com a classe SimpleDotCom, pense nas partes desse código que pode querer (ou ter que) aperfeiçoar. Os números dentro de um círculo são para as coisas que queremos destacar. Elas serão explicadas na outra página. Ah, se estiver querendo saber por que saltamos a fase do código de teste nessa classe, não precisamos de uma classe de teste para o jogo. Ele só tem um método, portanto, o que você faria em seu código de teste? Criaria uma classe *separada* que chamasse *main()* nessa classe? Não é necessário.



Declare uma variável para armazenar a contagem de palpites do usuário e configure-a com 0.

Crie um objeto SimpleDotCom. Gere um número aleatório entre 0 e 4.

Crie uma matriz int com o local das três células e chame setLocationCells no objeto dot com. Declare uma variável booleana isAlive

enquanto a dot com existir. Capture a entrada do usuário. // Verifique-a

Chame checkYourself() no objeto dot com.

Incremente numOfGuesses. Se o resultado for "kill", Configure gameAlive com falso.

Exiba o número de palpites do usuário.

```
public static void main(String[] args) {
```

```
    int numOfGuesses = 0;
```

```
    GameHelper helper = new GameHelper();
```

```
    SimpleDotCom theDotCom = new SimpleDotCom();
```

```
    int randomNum = (int) (Math.random() * 5);
```

```
    int[] locations = {randomNum, randomNum+1, randomNum+2};
```

```
    theDotCom.setLocationCells(locations);
```

```
    boolean isAlive = true;
```

```
    while(isAlive == true) {
```

```
        String guess = helper.getUserInput("insira um número");
```

```
        String result = theDotCom.checkYourself(guess);
```

```
        numOfGuesses++;
```

```
        if (result.equals("kill")) {
            isAlive = false;
```

```
        System.out.println("Você usou " + numOfGuesses + " palpites");
    } // encerra a instrução if
} // encerra while
} // encerra main
```

cria uma variável para controlar quantos palpites o usuário usou
essa é uma classe especial que criamos e que contém o método de captura de entradas do usuário. Por enquanto, considere-a como parte do Java

cria o objeto dot com

gera um número aleatório para a primeira célula e o usa para criar a matriz de locais de células

fornece à dot com sua localização (a matriz)

cria uma variável booleana que será usada no teste do loop while para registrar se o jogo continua ativo. Repete o loop enquanto o jogo estiver ativo.

captura a string da entrada do usuário

solicita à dot com para verificar o palpite; salva o resultado retornado em uma string

incrementa a contagem de palpites

foi uma "eliminação"? Se for, configura isAlive com falso (para não entrarmos novamente no loop) e exibe a contagem de palpites do usuário

código preparatório	código de teste	código real
---------------------	-----------------	-------------

random() e getUserInput()

Duas coisas que precisam de uma explicação um pouco melhor estão nessa página. Trata-se apenas de uma visão geral para que você possa continuar; mais detalhes sobre a classe GameHelper se encontram no fim deste capítulo.

Isso é uma 'conversão' e ela forçará o que estiver imediatamente após a ficar com o seu tipo (isto é, o tipo entre parênteses). `Math.random` retornará um tipo `double`, portanto teremos que convertê-lo em um `int` (queremos um número inteiro entre 0 e 4). Nesse caso, a conversão eliminará a parte fracionária do tipo `double`.

O método `Math.random` retornará um número no intervalo entre zero e menor que um. Portanto, essa fórmula (com a conversão), retornará um número de 0 a 4 (isto é, de 0 a 4,999..., convertido em um inteiro).

1 Gere um número aleatório

```
int randomNum = (int) (Math.random( ) * 5)
```

Declaramos uma variável `int` que armazenará o número aleatório fornecido.

Uma classe que vem com a Java.

Um método da classe `Math`.

Uma instância que criamos anteriormente, de uma classe que construímos para auxiliar o jogo. Ela se chama `GameHelper` e você ainda não a viu (mas verá).

2 Capturando a entrada do usuário usando a classe GameHelper

```
String guess = helper.getUserInput("insira um número");
```

Declaramos uma variável de `String` que armazenará a string da entrada fornecida pelo usuário ("3", "5", etc.).

Um método da classe `GameHelper` que solicita ao usuário entrada na linha de comando, a lê depois que o usuário pressiona Enter e retorna o resultado como uma `String`.


Uma última classe: GameHelper

Criamos a classe *dot com*.

Criamos a classe *do jogo*.

Só falta a classe auxiliar - a do método `getUserInput()`. O código para capturar entrada na linha de comando vai além do que queremos explicar agora. Ele abre caminho para muitos tópicos, o que é melhor deixarmos para depois. (Na verdade no Capítulo 14.)

Basta copiar* o código da próxima página e compilá-lo em uma classe chamada `GameHelper`. Insira todas as três classes (`SimpleDotCom`, `SimpleDotComGame`, `GameHelper`) no mesmo diretório e faça com ele seja seu diretório de trabalho.

Sempre que você encontrar o logotipo  **Código predefinido**, verá um código que terá que digitar do modo em que se encontra e acreditar em seu funcionamento. Pode confiar. Você aprenderá como esse código funciona *posteriormente*.



*Sabemos como você aprecia digitar, mas para os raros momentos em que preferir fazer outra coisa, disponibilizamos o Código Predefinido em wickedlysmart.com.

código preparatório	código de teste	código real
---------------------	-----------------	-------------



**Código
pré-definido**

```
import java.io.*;

public class GameHelper {

    public String getUserInput(String prompt) {
        String inputLine = null;
        System.out.print(prompt + " ");
        try {
            BufferedReader is = new BufferedReader(new InputStreamReader(System.in));
            inputLine = is.readLine();
            if (inputLine.length() == 0 ) return null;
        } catch (IOException e) {
            System.out.println("IOException: " + e);
        }
        return inputLine;
    }
}
```

Agora podemos jogar

Veja o que acontecerá quando executarmos o código e inserimos os números 1, 2, 3, 4, 5, 6. Parece funcionar bem.

Uma interação completa do jogo (sua pontuação pode variar)

```
Arquivo Editar Janela Ajuda Sorria

%java SimpleDotComGame
insira um número 1
errado
insira um número 2
errado
insira um número 3
errado
insira um número 4
correto
insira um número 5
correto
insira um número 6
eliminar
Você usou 6 palpites
```

O que é isso? Um erro?

Opa!

Veja o que acontece quando inserimos 1, 1, 1.

Uma interação diferente do jogo (epa)

```
Arquivo Editar Janela Ajuda Desmaiar

%java SimpleDotComGame
insira um número 1
correto
insira um número 1
correto
insira um número 1
eliminar
Você usou 3 palpites
```



Aponte seu lápis



É uma situação-limite!
Encontraremos o erro?
Corrigiremos o erro?

Não perca o próximo capítulo, onde responderemos essas perguntas e muitas outras...

Mas por enquanto, veja se consegue ter uma idéia do que deu errado e de como corrigir.

Mais informações sobre os loops for

Abordamos todo o código do jogo *neste* capítulo (mas voltaremos a ele para terminar a versão sofisticada no próximo capítulo). Não quisemos interromper seu trabalho com alguns dos detalhes e informações secundárias, portanto retornaremos a eles aqui. Começaremos com os detalhes dos loops `for` e, se você é programador de C++, poderá ler apenas superficialmente essas últimas páginas...

Loops `for` comuns (não-aperfeiçoados)

operador pós-incremento

o código a ser repetido entra aqui (o corpo)

```
for(int i = 0; i < 100; i++) { }
```

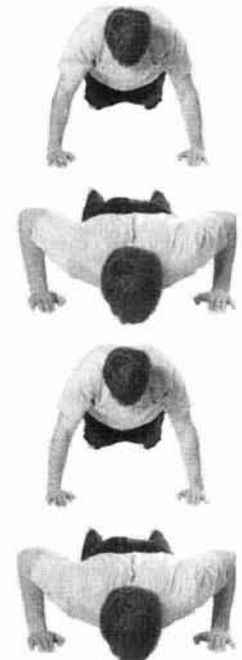
inicialização teste expressão de
 booleano iteração

O que significa em português simples: “repetir 100 vezes.”

repita 100 vezes:

Como o compilador interpreta:

- criar uma variável *i* e configurar com 0.
- repetir enquanto *i* for menor que 100.
- no fim de cada iteração do loop, acrescentar uma unidade a *i*.



Parte um: *inicialização*

Use essa parte para declarar e inicializar uma variável que será usada dentro do corpo do loop. Geralmente essa variável é usada como um contador. Na verdade você pode inicializar mais de uma variável aqui, mas veremos isso posteriormente no livro.

Parte dois: *teste booleano*

É aqui que o teste condicional entrará. Independentemente do que houver nele, *terá* que ser convertido em um valor booleano (você sabe, *verdadeiro* ou *falso*). Você pode ter um teste, como $(x \geq 4)$, ou até mesmo chamar um método que retorne um booleano.

Parte três: *expressão iterativa*

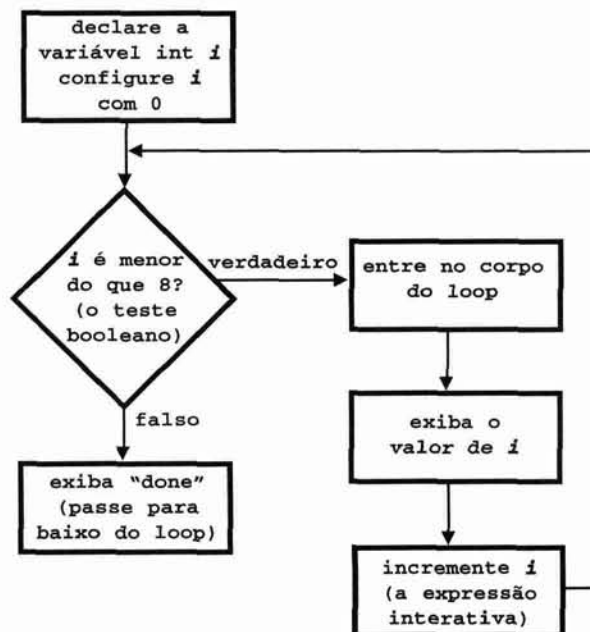
Nessa parte, insira uma ou mais coisas que você deseja que ocorram a cada passagem do loop. Lembre-se de que elas acontecerão no *final* de cada loop.

Percorrendo um loop

```
for (int i = 0; i < 8; i++) {
    System.out.println(i);
}
System.out.println("done");
```

saída:

```
File Edit Window Help Repeat
%java Test
0
1
2
3
4
5
6
7
done
```



Diferença entre `for` e `while`

Um loop `while` apresenta apenas o teste booleano; não tem uma expressão interna de inicialização ou iteração. Ele será útil quando você não souber quantas vezes o loop será executado e quiser continuar a execução apenas enquanto alguma condição for verdadeira. Mas se você *souber* quantas vezes o loop será executado (por exemplo, dependendo do tamanho de uma matriz, 7 vezes, etc.), um loop `for` será mais simples. Aqui está o loop anterior reescrito usando-se `while`:

```
int i = 0;  ← temos que declarar e inicializar o contador
while (i < 8) {
    System.out.println(i);
    i++;  ← temos que incrementar o contador
}
System.out.println("done");
```

++ --

Operador de pré e pós-incremento/decremento

O atalho para se adicionar ou subtrair 1 unidade de uma variável.

x++;

é o mesmo que:

x = x + 1;

As duas instruções significam a mesma coisa nesse contexto:

“adicione 1 unidade ao valor atual de x” ou “*incremente* x em 1 unidade”

E:

x- -;

é o mesmo que:

x = x - 1;

É claro que isso não é tudo. A inserção do operador (antes ou depois da variável) pode afetar o resultado.

Inserir o operador *antes* da variável (por exemplo, ++x), significa “*primeiro*, incremente x em 1 unidade e, *em seguida*, use esse novo valor de x”. Isso só será importante quando x++ fizer parte de alguma expressão maior e não de apenas uma instrução.

int x = 0; int z = ++x;

produzirá: x é igual a 1, z é igual a 1

Mas, se inserirmos o sinal ++ *depois* de x, teremos um resultado diferente:

int x = 0; int z = x++;

produzirá: x é igual a 1, mas **z é igual a 0!** Z receberá o valor de x e, *em seguida*, x será incrementado.

O loop `for` aperfeiçoado

A partir do Java 5.0 (Tiger), a linguagem passou a ter um segundo tipo de loop `for` chamado *for* aperfeiçoado, que torna mais fácil a iteração por todos os elementos de uma matriz ou outros tipos de conjunto (você aprenderá sobre *outros* conjuntos no próximo capítulo). Na verdade isso é tudo que o loop `for` aperfeiçoado fornece — uma maneira mais simples de percorrer todos os elementos do conjunto, mas já que essa é a finalidade mais comum de um loop `for`, valeu a pena adicioná-lo à linguagem. Revisitaremos o loop `for` aperfeiçoado no próximo capítulo, quando falarmos sobre conjuntos que *não são* matrizes.

Declara uma variável de iteração que armazenará apenas um elemento da matriz.

O sinal de dois-pontos (:) significa "DE".

O código a ser repetido entra aqui (o corpo).

```
for (String name: nameArray) { }
```

Os elementos da matriz DEVEM ser compatíveis com o tipo declarado para a variável.

A cada iteração um elemento diferente da matriz será atribuído à variável "name".

O conjunto de elementos que você deseja percorrer. Suponhamos que em algum ponto anterior do código tivéssemos:

```
String[] nameArray = {"Fred", "Mary", "Bob"};
```


 Na primeira iteração, a variável name teria o valor "Fred", na segunda o valor "Mary", etc.

O que isso significa em português claro: "a cada elemento de nameArray, atribua o elemento à variável 'name' e execute o corpo do loop."

Como o compilador interpretaria:

- Criar uma variável de string chamada *name* e configurá-la com nulo.
- Atribuir o primeiro valor de *nameArray* à variável *name*.
- Executar o corpo do loop (o bloco de código dentro das chaves).
- Atribuir o próximo valor de *nameArray* a *name*.
- Repetir enquanto ainda houver elementos na matriz.

Nota: dependendo da linguagem de programação que tiverem usado no passado, algumas pessoas podem chamar o loop for aperfeiçoado de "for each" ou "for in", porque é assim que se interpreta: "para (for) CADA (each) elemento DO (in) conjunto..."

Parte um: declaração da variável de iteração

Use essa parte para declarar e inicializar uma variável que será usada dentro do corpo do loop. A cada iteração do loop, essa variável armazenará um elemento diferente do conjunto. O tipo da variável deve ser compatível com os elementos da matriz! Por exemplo, você não pode declarar uma variável de iteração *int* para usar com uma matriz *String[]*.

Parte dois: o conjunto atual

Deve ser uma referência que aponte para uma matriz ou outro conjunto. Não se preocupe ainda com os outros tipos de conjunto que não são matrizes - você os verá no próximo capítulo.

Convertendo uma string em um inteiro

```
int guess = Integer.parseInt(stringGuess);
```

O usuário digitará seu palpite na linha de comando quando o jogo solicitar. Esse palpite chegará na forma de uma string ("2", "0", etc.), e o jogo passará essa string para o método `checkYourself()`.

Mas os locais das células são simplesmente os inteiros de uma matriz, e você não poderá comparar um inteiro com uma string.

Por exemplo, *isso não funcionará*:

```
String num = "2";
int x = 2;
if (x == num) //confronto incompatível!
```

Tentar compilar esse código fará o compilador rir e zombar de você:

```
operator == cannot be applied to int,java.lang.String
if (x == num) { }
```

Portanto, para contornar as diferenças, temos que converter a string "2" no inteiro 2. Embutida na biblioteca de classes Java temos uma classe chamada *Integer* (certo, trata-se da classe *Integer* e não de um tipo primitivo *int*) e uma de suas tarefas é pegar strings que representam números e convertê-las em números reais.

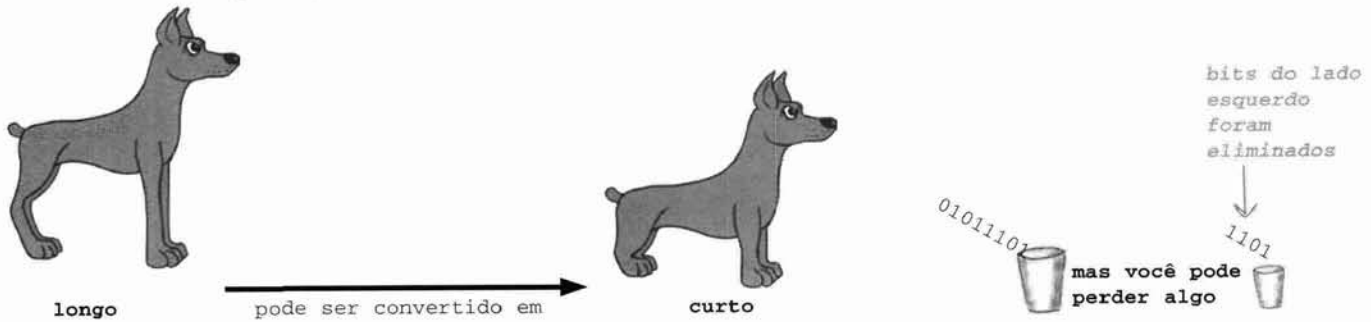
uma classe que vem com o Java

usa uma string

```
Integer.parseInt("3")
```

um método da classe *Integer* que sabe como "converter" uma string no inteiro que ela representa.

Convertendo tipos primitivos



No Capítulo 3 falamos sobre os tamanhos dos vários tipos primitivos e que você não pode inserir algo grande diretamente em um recipiente pequeno:

```
long y = 42;
int x = y;    // não será compilado
```

Um tipo **longo** é maior do que um **inteiro** e o compilador não terá certeza de onde esse **longo** saiu. Pode ter estado bebendo com outros longos e recebendo valores realmente altos. Para forçar o compilador a espremer o valor de uma variável primitiva maior em um tipo menor, você pode usar o operador **de conversão**. Ele tem esta aparência:

```
long y = 42;    // até aqui tudo bem
int x = (int) y; // x = 42 muito bom!
```

Se você inserir a conversão, estará solicitando ao compilador que pegue o valor de `y`, converta-o para o tamanho de um inteiro e configure `x` com o que sobrar. Se o valor de `y` for maior do que o valor máximo de `x`, acabaremos com um número estranho (mas calculável*):

```
long y = 40002;
// 40002 excede o limite de 16 bits de um tipo curto
short x = (short) y; // agora x é igual a -25534!
```

Mesmo assim, o importante é que o compilador lhe permita continuar. E digamos que você tivesse um número de ponto flutuante e quisesse apenas a parte inteira (**int**) dele:

```
float f = 3.14f;
int x = (int) f;    // x será igual a 3
```

Mas nem **pense** em converter algo em um booleano ou vice-versa — deixe como está.

**Isso envolve os bits de sinais, o sistema binário, 'complementos de dois' e outros detalhes, que serão discutidos no começo do Apêndice B.*



Exercício



Seja a JVM

O arquivo Java dessa página representa um arquivo-fonte completo. Sua tarefa é personificar a JVM e determinar qual será a saída quando o programa for executado?

```
class Output {
    public static void main(String [] args) {
        Output o = new Output();
        o.go();
    }
    void go() {
        int y = 7;
        for(int x = 1; x < 8; x++) {
            y++;
            if (x > 4) {
                System.out.print(++y + " ");
            }
            if (y > 14) {
                System.out.println(" x = " + x);
                break;
            }
        }
    }
}
```

```
Arquivo Editar Janela Ajuda OM
% java Output
12 14
```

OU

```
Arquivo Editar Janela Ajuda Incenso
% java Output
12 14 x = 6
```

OU

```
Arquivo Editar Janela Ajuda Fé
% java Output
13 15 x = 6
```

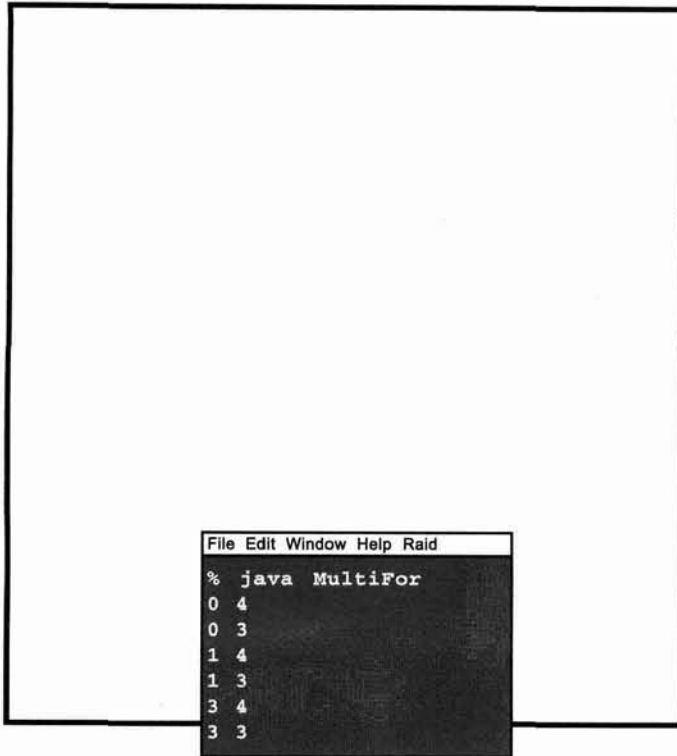



Exercício



Ímãs com código

Um programa Java funcional está todo misturado sobre a geladeira. Você conseguiria reorganizar os trechos de código para criar um programa Java funcional que produzisse a saída listada a seguir? Algumas das chaves caíram no chão e são muito pequenas para que as recuperemos, portanto, fique à vontade para adicionar quantas delas precisar!



```
File Edit Window Help Raid
% java MultiFor
0 4
0 3
1 4
1 3
3 4
3 3
```

x++;

if (x == 1) {

System.out.println(x + " " + y);

class MultiFor {

for(int y = 4; y > 2; y-) {

for(int x = 0; x < 4; x++)

public static void main(String [] args) {



Mensagens misturadas

Um programa Java curto é listado a seguir. Um bloco do programa está faltando. Seu desafio é **comparar o bloco de código candidato** (à esquerda) **com a saída** que você veria se ele fosse inserido. Nem todas as linhas de saída serão usadas e algumas delas podem ser usadas mais de uma vez. Desenhe linhas conectando os blocos de código candidatos à saída de linha de comando correspondente. (As respostas estão no final do capítulo.)

```
class MixFor5 {
    public static void main(String [] args) {
        int x = 0;
        int y = 30;
        for (int outer = 0; outer < 3; outer++)
        {
            for(int inner = 4; inner > 1; inner-)
            {
                
                y = y - 2;
                if (x == 6) {
                    break;
                }
                x = x + 3;
            }
            y = y - 2;
        }
        System.out.println(x + " " + y);
    }
}
```

O código candidato entra aqui

Candidatos:

x = x +3;

x = x +6;

x = x +2;

x++;

x--;

x = x +0;

Saídas possíveis:

45 6

36 6

54 6

60 10

18 6

6 14

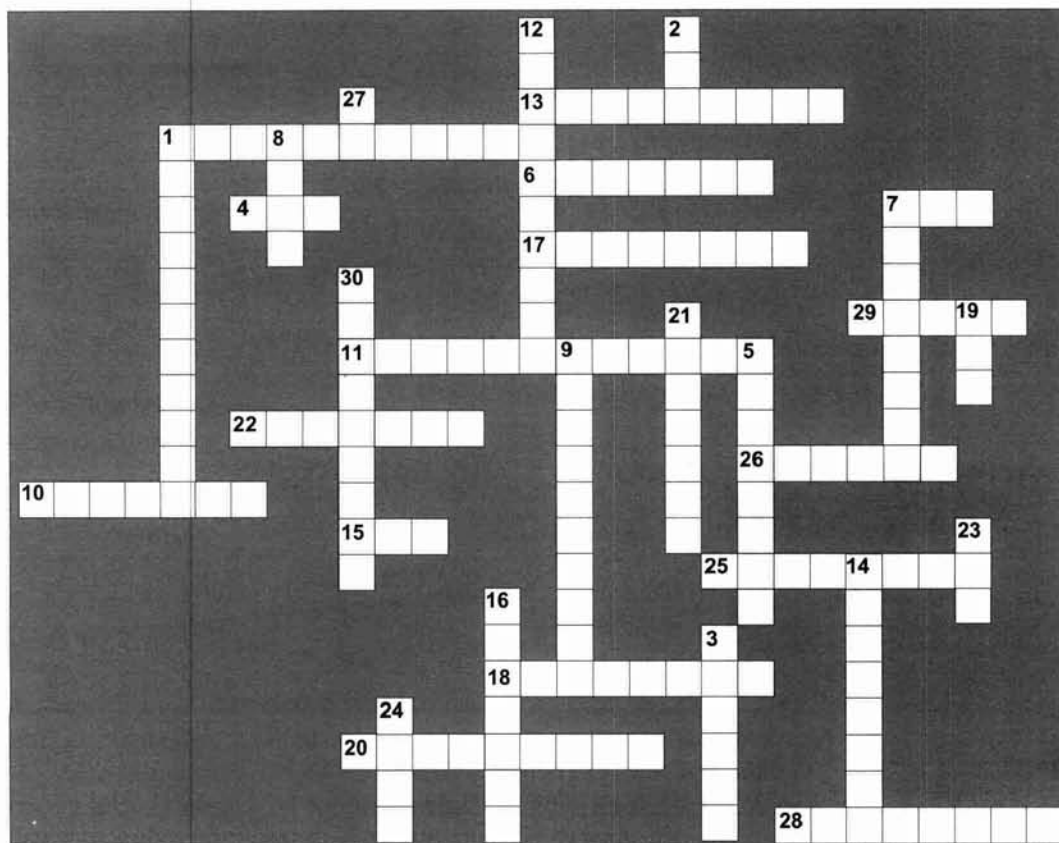
12 14

Compare cada candidato com uma das saídas possíveis



Cruzadas Java

Como um jogo de palavras cruzadas o ajudará a aprender Java? Bem, todas as palavras **têm** relação com o Java. Além disso, as pistas fornecem metáforas, trocadilhos e coisas do tipo. Esses atalhos mentais disponibilizarão rotas alternativas para o aprendizado Java, diretamente em seu cérebro!



Horizontais

1. Palavra engraçada em informática que significa construir
4. Loop de várias partes
6. Teste primeiro
7. 32 bits
10. Resposta do método
11. Código preparatório
13. Alteração
15. O grande kit de ferramentas
17. Uma unidade da matriz
18. De instância ou local
20. Kit de ferramentas automático
22. Parece um tipo primitivo, mas...
25. Não conversível
26. Método de Math
28. Método conversor
29. Sair antes

Verticais

1. Estabelecer o primeiro valor
2. Tipo de incremento
3. Cavalo-de-batalha da classe
5. O pré é um tipo de _____
7. Um ciclo
8. While ou For
9. Atualizar uma variável de instância
12. Contagem regressiva
14. Compilar e _____
16. Pacote de comunicação
19. Mensageiro dos métodos (abrev.)
21. Como se
23. Adicionar depois
24. A casa do pi
27. Valor do operador ++
30. _____ de iteração do loop for



Soluções dos Exercícios

Seja a JVM:

```
class Output {
    public static void main(String [] args) {
        Output o = new Output();
        o.go();
    }
    void go() {
        int y = 7;
        for(int x = 1; x < 8; x++) {
            y++;
            if (x > 4) {
                System.out.print(++y + " ");
            }
            if (y > 14) {
                System.out.println(" x = " + x);
                break;
            }
        }
    }
}
```

Você se lembrou
de considerar a
instrução break?
Como isso
afetou a saída?

```
Arquivo Editar Janela Ajuda ManutençãoMotocicleta
% java Output
13 15 x = 6
```

Ímãs com Código

```
class MultiFor {
    public static void main(String [] args) {
        for(int x = 0; x < 4; x++) {
            for(int y = 4; y > 2; y--) {
                System.out.println(x + " " + y);
            }
            if (x == 1) {
                x++;
            }
        }
    }
}
```

O que aconteceria
se esse bloco de
código viesse antes
do loop for de 'y'?

Arquivo Editar Janela Ajuda Monopólio

```
% java MultiFor
0 4
0 3
1 4
1 3
3 4
3 3
```

Soluções dos quebra-cabeças



Candidatos:

Saídas possíveis:

<code>x = x + 3;</code>	45 6
<code>x = x + 6;</code>	36 6
<code>x = x + 2;</code>	54 6
<code>x++;</code>	60 10
<code>x--;</code>	18 6
<code>x = x + 0;</code>	6 14
	12 14

