Capítulo 4

Agrupando objetos

Principais conceitos a serem abrangidos

- Coleções (especialmente ArrayList)

O requisito para agrupar objetos

- Muitas aplicações envolvem coleções de objetos:
  - Dispositivos do tipo PDA.
  - Catálogos de biblioteca.
  - Sistema de registro de estudantes.
- O número de itens a ser armazenado varia.
  - Itens adicionados.
  - Itens excluídos.

Um bloco de notas pessoal

- Anotações podem ser armazenadas.
- Anotações individuais podem ser visualizadas.
- Não há limite para o número de notas.
- Isso informa quantas anotações foram armazenadas.
- Explore o projeto *notebook1*.

Bibliotecas de classe

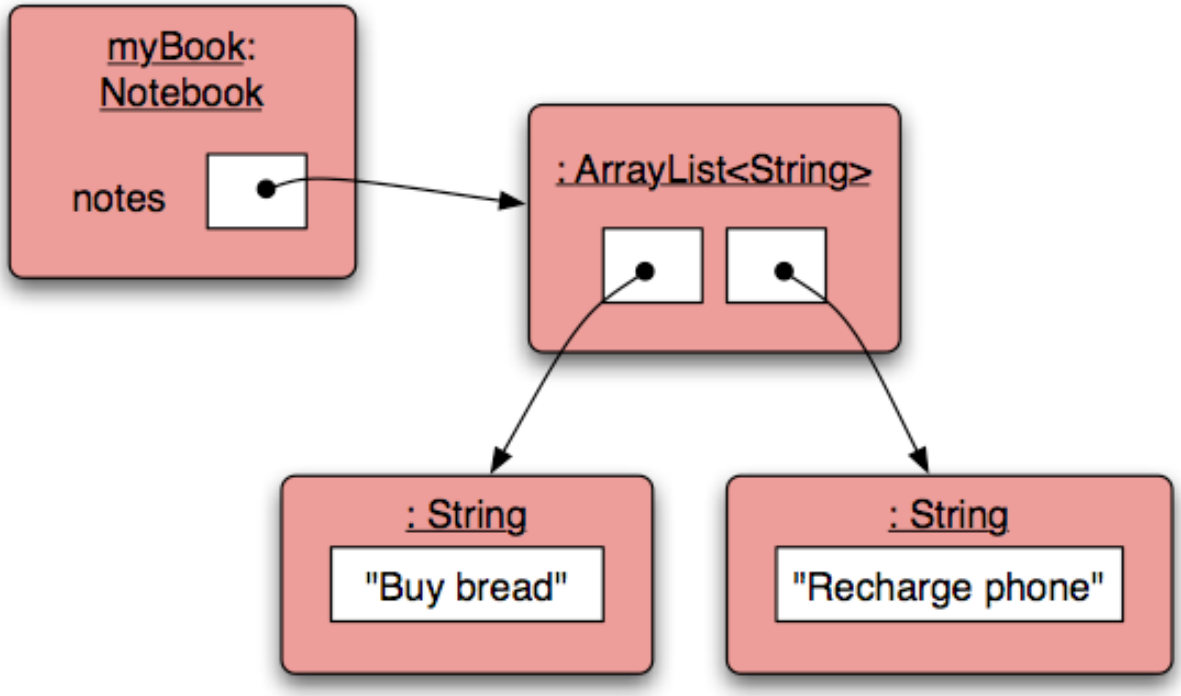
- Coleções de classes úteis.
- Não temos de escrever tudo do zero.
- O Java chama suas bibliotecas de *pacotes*.
- Agrupar objetos é um requisito recorrente.
  - O pacote `java.util` contém classes para fazer isso.

```
import java.util.ArrayList;

/**
 * ...
 */
public class Notebook
{
    // Armazena um número arbitrário de notas.
    private ArrayList<String> notes;

    /**
     * Executa qualquer inicialização necessária para o
     * bloco de notas.
     */
    public Notebook()
    {
        notes = new ArrayList<String>();
    }

    ...
}
```

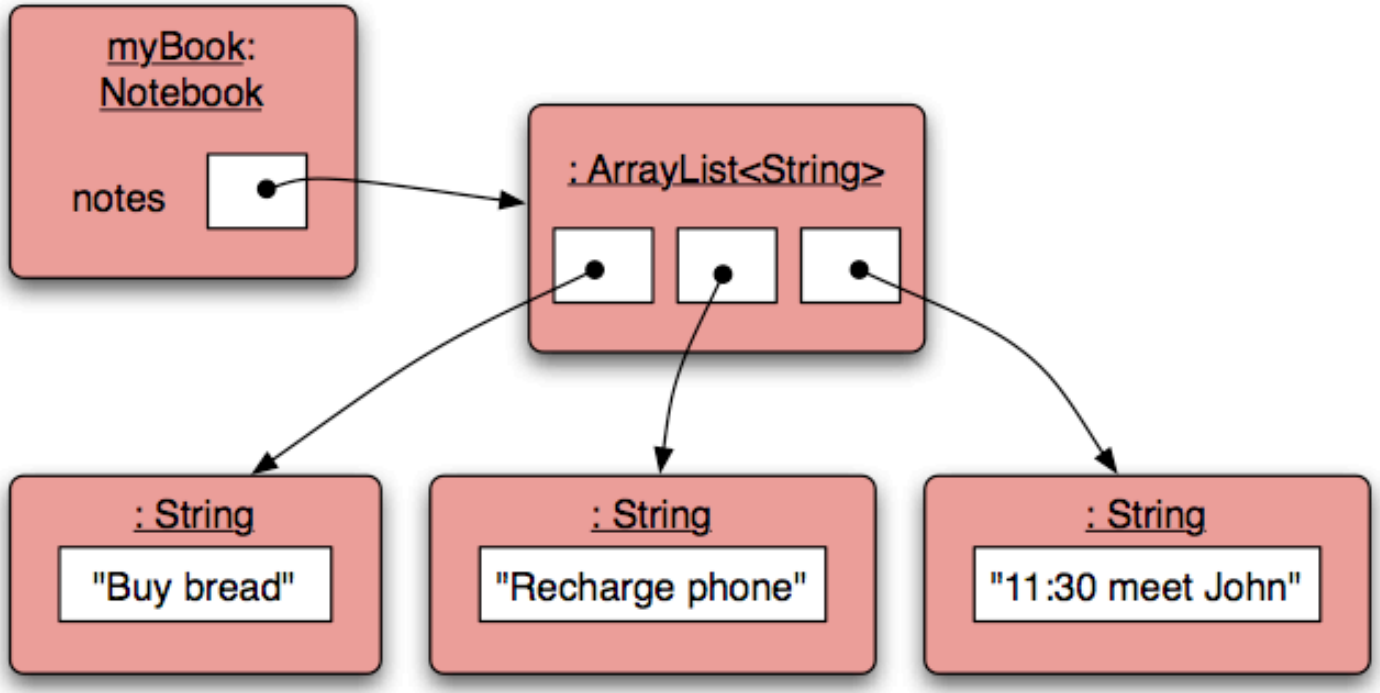
Coleções

- Especificamos:
  - o tipo de coleção: `ArrayList`
  - o tipo de objetos que ele conterá: `<String>`.
- Dizemos, “ArrayList de String”.

Estruturas de objeto com coleções



Adicionando uma terceira nota



Recursos da coleção

- Aumenta sua capacidade de acordo com a necessidade.
- Mantém uma contagem (método de acesso `size()`).
- Mantém os objetos em ordem.
- Os detalhes de como tudo isso é feito são ocultos.
  - Isso importa? Não saber como nos impedir de usá-lo?

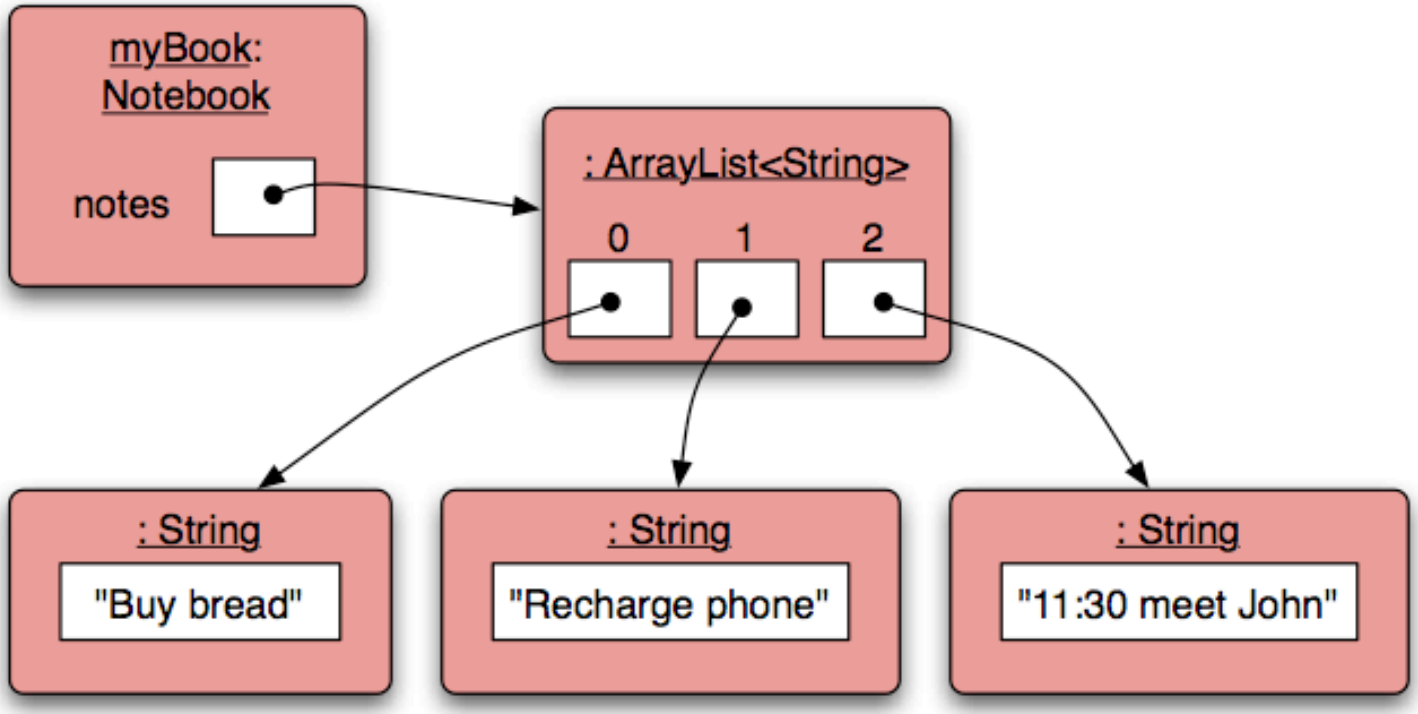
Utilizando a coleção

```
public class Notebook
{
    private ArrayList<String> notes;
    ...

    public void storeNote(String note)
    {
        notes.add(note);
    }

    public int numberOfNotes()
    {
        return notes.size();
    }

    ...
}
```

Adicionando uma terceira nota

Retornando o número de notas (*delegação*)

Numeração de índice



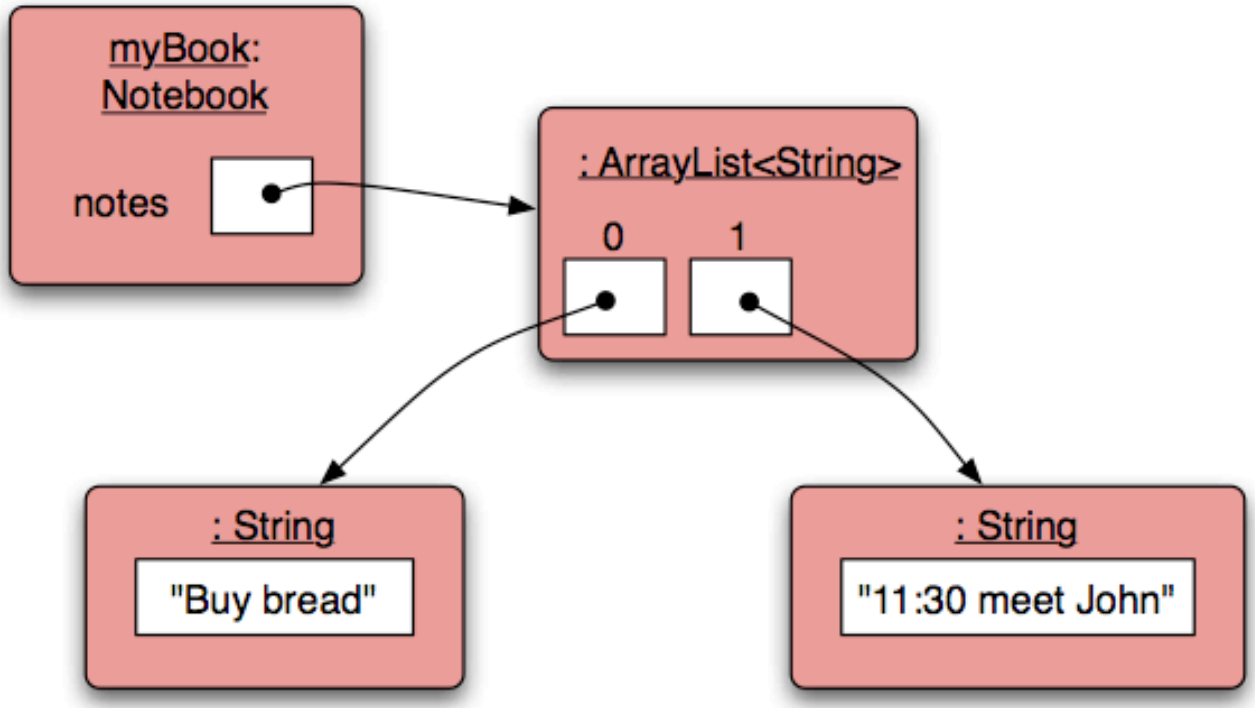
Recuperando um objeto

```
public void showNote(int noteNumber)
{
    if(noteNumber < 0) {
        // Esse não é um número de nota válido.
    }
    else if(noteNumber < numberOfNotes()) {
        System.out.println(notes.get(noteNumber));
    }
    else {
        // Esse não é um número de nota válido.
    }
}
```

Verificações de validade de índice

Recupera e imprime a nota

A remoção pode afetar a numeração



Classes genéricas

- Coleções são conhecidas como tipos *parametrizados* ou *genéricos*.
- `ArrayList` implementa a funcionalidade de lista:
  - **`add`**, **`get`**, **`size`**, etc.
- O parâmetro de tipo diz o que queremos de uma lista de:
  - **`ArrayList<Person>`**
  - **`ArrayList<TicketMachine>`**
  - etc.

Revisão

- As coleções permitem que um número arbitrário de objetos seja armazenado.
- As bibliotecas de classes normalmente contêm classes de coleção testadas e comprovadas.
- As bibliotecas de classes do Java são chamadas de *pacotes*.
- Usamos a classe `ArrayList` do pacote `java.util`.

Revisão

- Os itens podem ser adicionados e removidos.
- Cada item tem um índice.
- Os valores de índice podem mudar se os itens forem removidos (ou se mais itens forem adicionados).
- O métodos `ArrayList` principais são `add`, `get`, `remove` e `size`.
- `ArrayList` é um tipo parametrizado ou genérico.

Agrupando objetos

Coleções e o loop for-each

Interlúdio: Alguns erros populares...

O que está errado aqui?

```
/**
 * Imprime as informações do bloco de notas
   (número de entradas).
 */
public void showStatus()
{
   if(notes.size() == 0); {
      System.out.println("Notebook is empty");
   }
   else {
      System.out.print("Notebook holds ");
      System.out.println(notes.size() + " notes);
   }
}
```

Este é o mesmo de antes!

```
/**
 * Imprime as informações do bloco de notas
   (número de entradas).
 */
public void showStatus()
{
   if(notes.size() == 0);

   {
      System.out.println("Notebook is empty");
   }
   else {
      System.out.print("Notebook holds ");
      System.out.println(notes.size() + " notes);
   }
}
```

Esse é o mesmo de antes novamente

```
/**
 * Imprime as informações do bloco de notas
   (número de entradas).
 */
public void showStatus()
{
   if(notes.size() == 0)
      ;

   {
      System.out.println("Notebook is empty");
   }
   else {
      System.out.print("Notebook holds ");
      System.out.println(notes.size() + " notes);
   }
}
```

e o mesmo novamente...

```
/**
 * Imprime as informações do bloco de notas
 * (número de entradas).
 */
public void showStatus()
{
    if(notes.size() == 0) {
        ;
    }

    {
        System.out.println("Notebook is empty");
    }
    else {
        System.out.print("Notebook holds ");
        System.out.println(notes.size() + " notes);
    }
}
```

O que está errado aqui?

Desta vez tenho um campo booleano chamado 'isEmpty'...

```
/**
 * Imprime as informações do bloco de notas
 * (número de entradas).
 */
public void showStatus()
{
   if(isEmpty = true) {
      System.out.println("Notebook is empty");
   }
   else {
      System.out.print("Notebook holds ");
      System.out.println(notes.size() + " notes);
   }
}
```

A versão correta

Dessa vez tenho um campo booleano chamado 'isEmpty'...

```
/**
 * Imprime as informações do bloco de notas
   (número de entradas).
 */
public void showStatus()
{
   if(isEmpty == true) {
      System.out.println("Notebook is empty");
   }
   else {
      System.out.print("Notebook holds ");
      System.out.println(notes.size() + " notes);
   }
}
```

O que está errado aqui?

```
/**
 * Armazena uma nova nota no bloco de notas. Se o
 * bloco de notas estiver cheio, salva-o e inicia
   um novo.
 */
public void addNote(String note)
{
   if(notes.size() == 100)
       notes.save();
       // starting new notebook
       notes = new ArrayList<String>();

   notes.add(note);
}
```

Este é o mesmo

```
/**
 * Armazena uma nova nota no bloco de notas. Se o
 * bloco de notas estiver cheio, salva-o e
   inicia um novo.
 */
public void addNote(String note)
{
    if(notes.size == 100)
        notes.save();

    // starting new notebook
    notes = new ArrayList<String>();

    notes.add(note);
}
```

A versão correta

```
/**
 * Armazena uma nova nota no bloco de notas. Se o
 * bloco de notas estiver cheio, salva-o e
   inicia um novo.
 */
public void addNote(String note)
{
    if(notes.size() == 100) {
        notes.save();
        // starting new notebook
        notes = new ArrayList<String>();
    }

    notes.add(note);
}
```

Principais conceitos a serem abrangidos

- Coleções
- Loops: o loop for-each

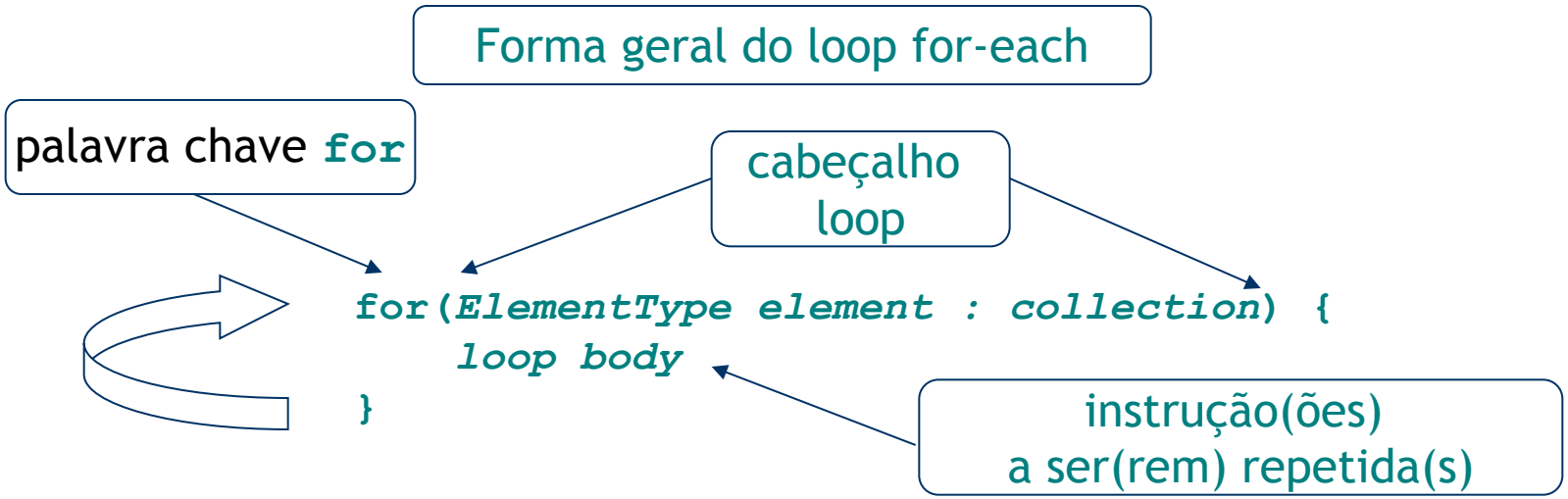
Iteração

- Em geral, queremos realizar algumas ações em um número arbitrário de vezes.
  - Por exemplo, imprimir todas as notas do bloco de notas. Há quantas?
- A maioria das linguagens de programação incluem *instruções de loop* para tornar isso possível.
- O Java tem vários tipos de instrução de loop.
  - Começaremos com seu *loop for-each*.

Princípios básicos da iteração

- Em geral, queremos repetir algumas ações inúmeras vezes.
- Loops fornecem uma forma de controlar quantas vezes repetimos essas ações.
- Com coleções, muitas vezes queremos repetir coisas uma vez para cada objeto em uma coleção particular.

Pseudocódigo do loop for-each



Expressão de pseudocódigo de ações de um loop for-each

Para cada *elemento* na *coleção*, faça as coisas no *corpo do loop*.

Um exemplo do Java

```
/**
 * Lista todas as notas no bloco de notas.
 */
public void listNotes()
{
    for(String note : notes) {
        System.out.println(note);
    }
}
```

para cada *nota* em *notes*, imprime a *nota*

Revisão

- Instruções de loop permitem que um bloco de instruções seja repetido.
- O loop for-each permite iteração sobre toda uma coleção.

Agrupando objetos

O loop while

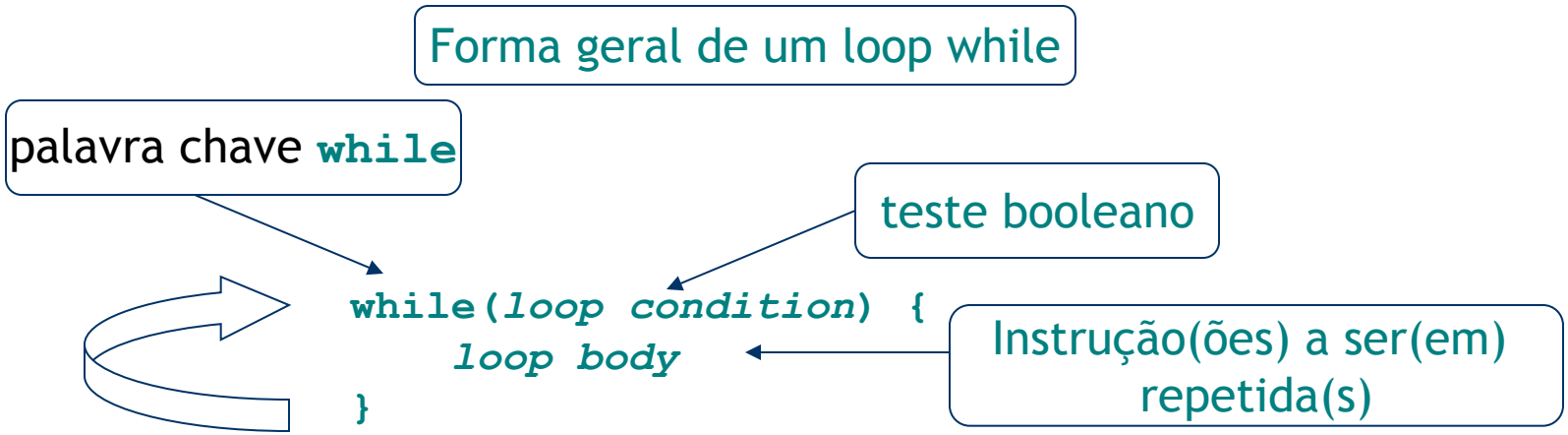
Principal conceito a ser abrangido

- O loop while

O loop while

- Um loop for-each repete o corpo do loop para cada objeto em uma coleção.
- Às vezes precisamos de mais variação do que esta.
- Podemos usar uma condição booleana para decidir se continuamos ou não.
- Um loop while fornece esse controle.

Pseudocódigo do loop while



Expressão de pseudocódigo de ações de um loop while

enquanto desejamos continuar, fazemos as coisas no corpo do loop

Um exemplo do Java

```
/**
 * Lista todas as notas no bloco de notas.
 */
public void listNotes()
{
    int index = 0;
    while(index < notes.size()) {
        System.out.println(notes.get(index));
        index++;
    }
}
```

Incrementa *index* por 1

enquanto o valor de *index* é menor do que o tamanho da coleção, imprime a próxima nota, e, então, incrementa *index*

for-each *versus* while

- for-each:
  - fácil de escrever.
  - mais seguro: oferece a garantia de parar.
- while:
  - *não precisamos* processar a coleção inteira.
  - não precisa nem mesmo ser usado com uma coleção.
  - cuidado: poderia entrar em um *loop infinito*.

While sem uma coleção

```
// Imprime todos os números pares de 0 a 30.
int index = 0;
while(index <= 30) {
    System.out.println(index);
    index = index + 2;
}
```

Pesquisando uma coleção

```
int index = 0;
boolean found = false;
while(index < notes.size() && !found) {
    String note = notes.get(index);
    if(note.contains(searchString)) {
        // Não precisamos continuar a busca.
        found = true;
    }
    else {
        index++;
    }
}
// Se o encontrarmos, ou pesquisarmos a
// coleção inteira.
```

Agrupando objetos

Iteradores

Principais conceitos a serem abrangidos

- Comparação de strings
- Iteradores

Nota extra: Igualdade de string

```
if(input == "bye") {
    ...
}
```

testa a identidade

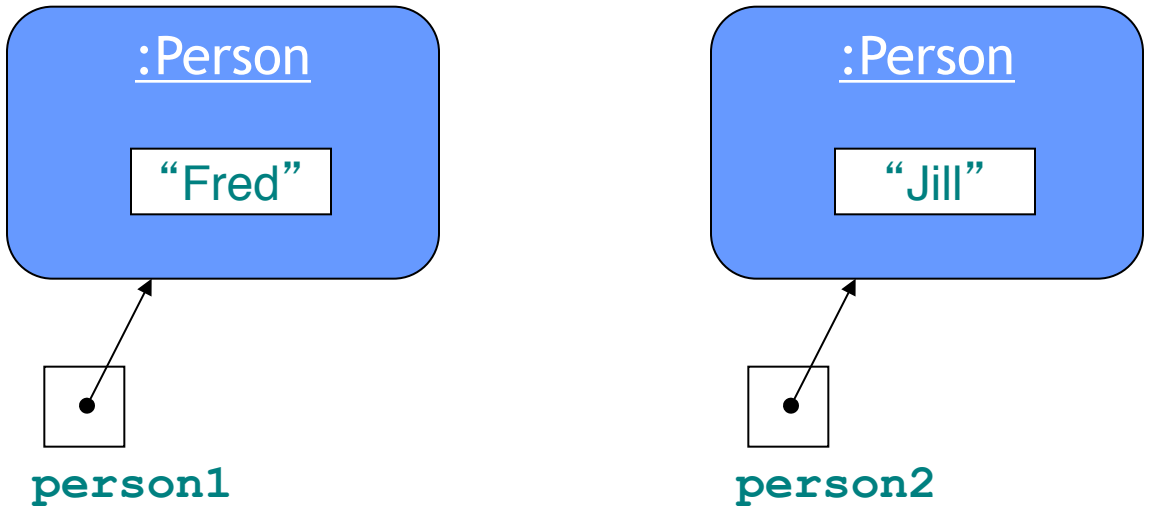
```
if(input.equals("bye")) {
    ...
}
```

testa a igualdade

Strings devem ser sempre comparadas com **`.equals`**

Identidade *versus* igualdade (1)

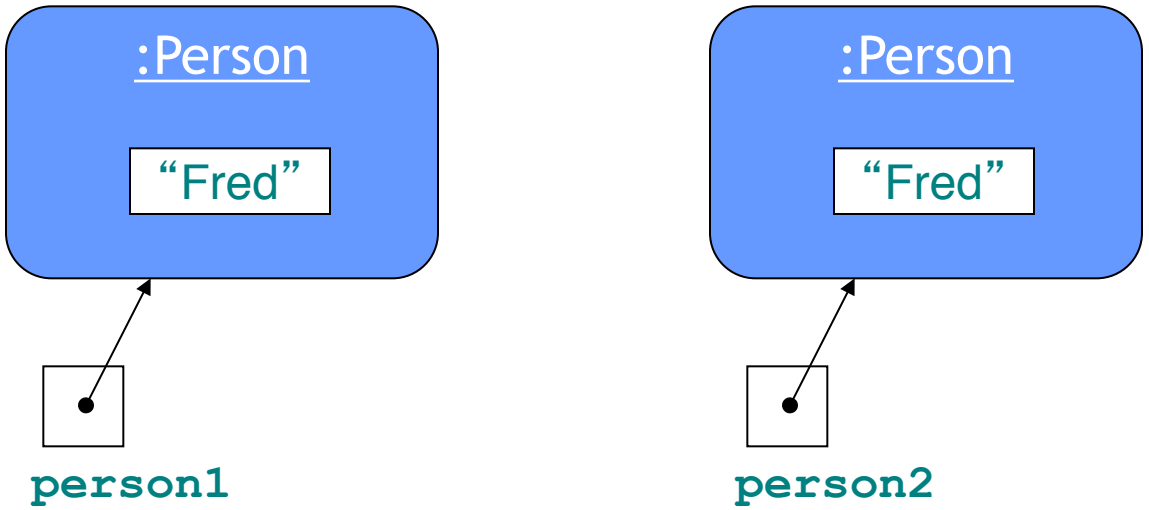
Outros objetos (não-String):




```
person1 == person2 ?
```

Identidade *versus* igualdade (2)

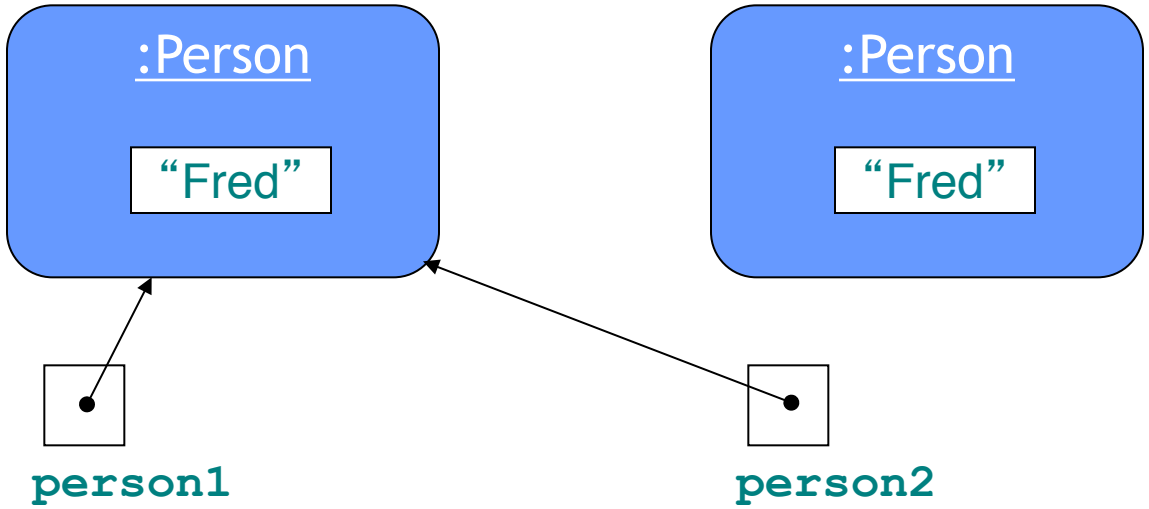
Outros objetos (não-String):




```
person1 == person2 ?
```

Identidade *versus* igualdade (3)

Outros objetos (não-String):

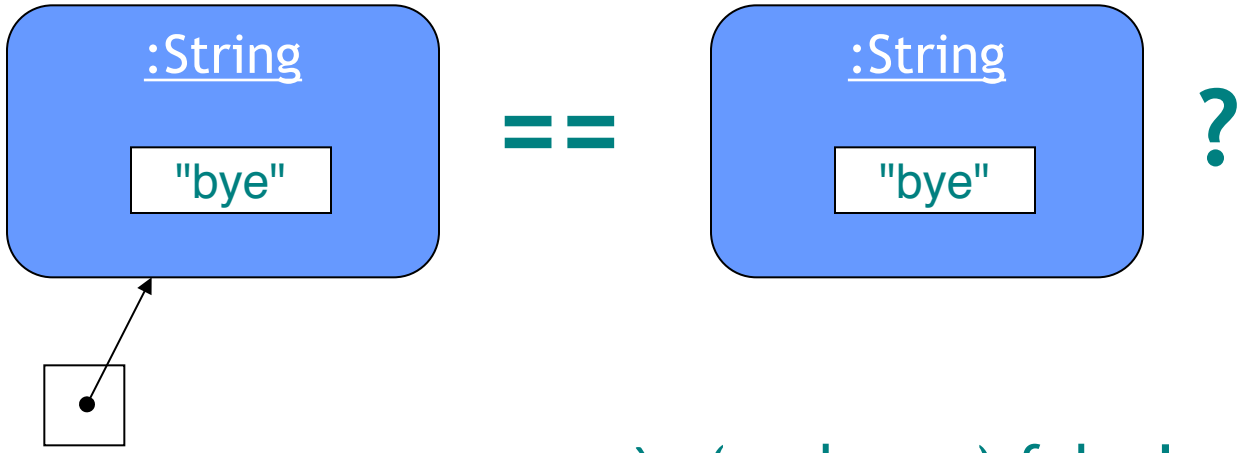



```
person1 == person2 ?
```

Identidade *versus* igualdade (Strings)

```
String input = reader.getInput();
if(input == "bye") {
    ...
}
```

== testa a identidade

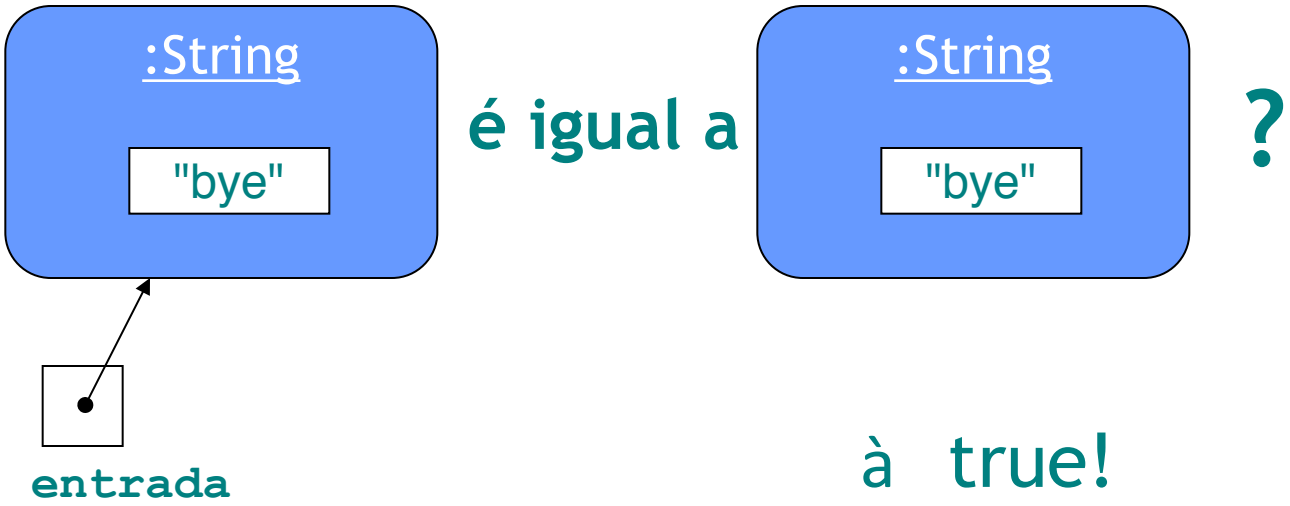


à (pode ser) false!

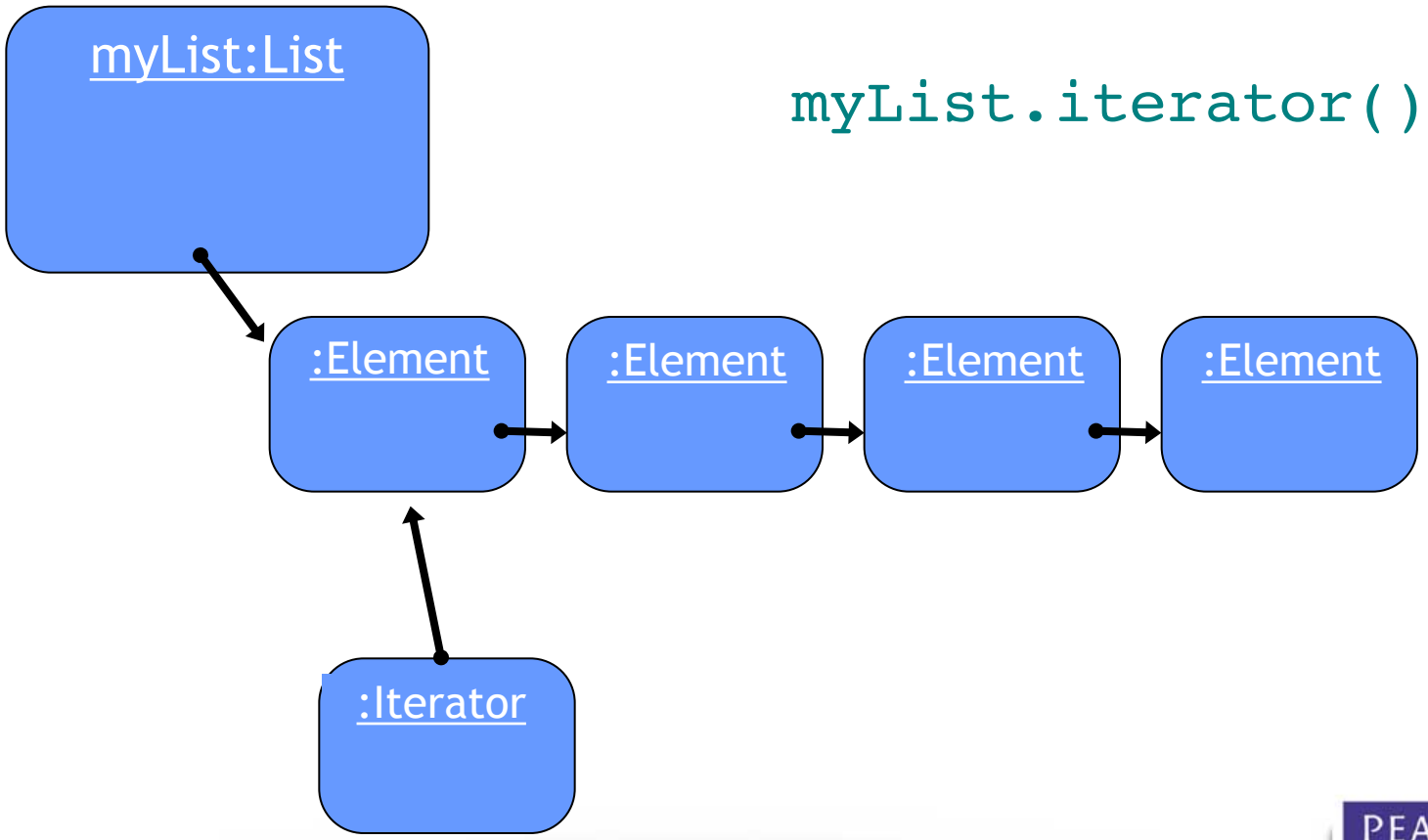
Identidade *versus* igualdade (Strings)

```
String input = reader.getInput();
if(input == "bye") {
    ...
}
```

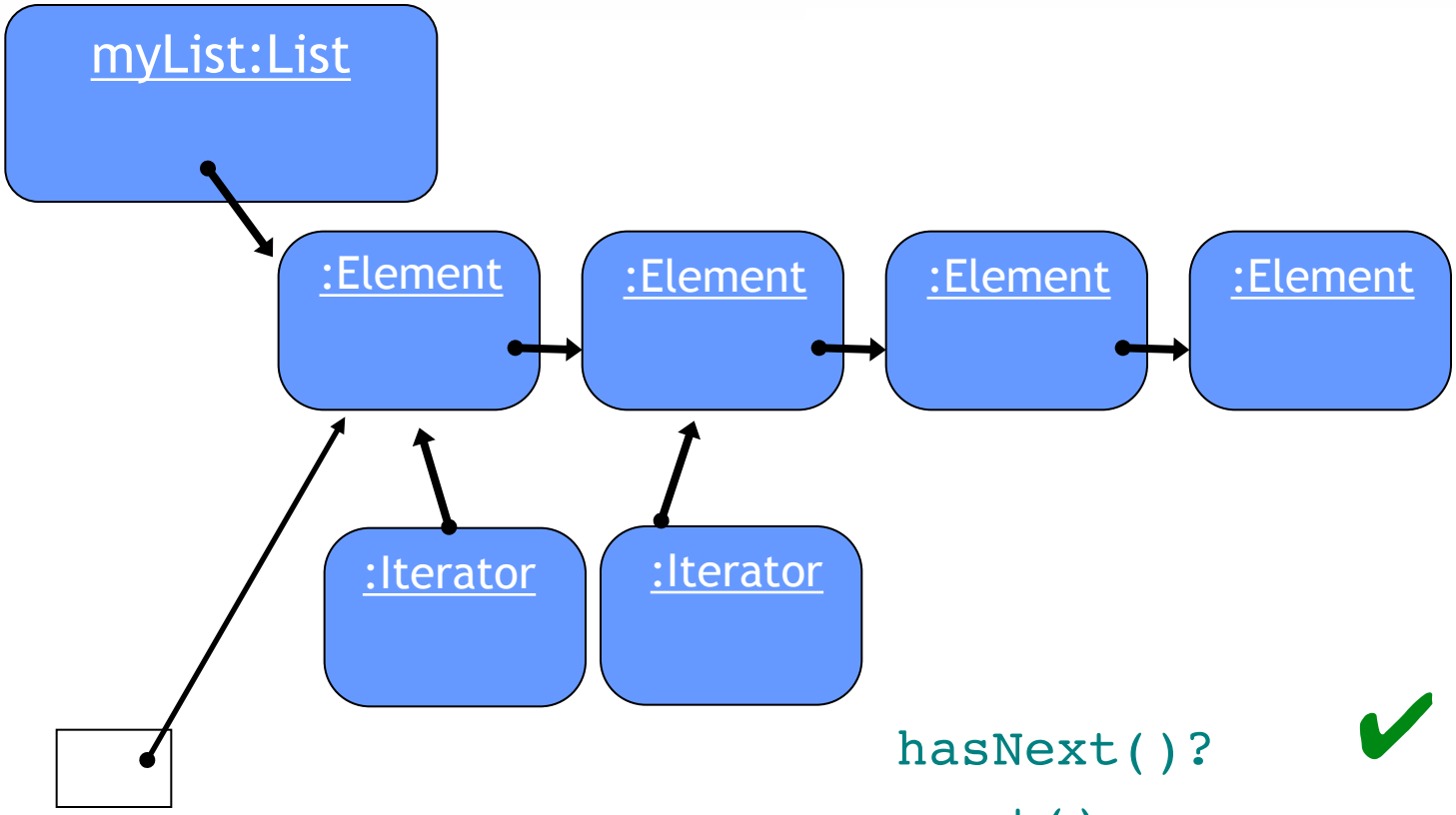
equals testa a igualdade



Iteradores



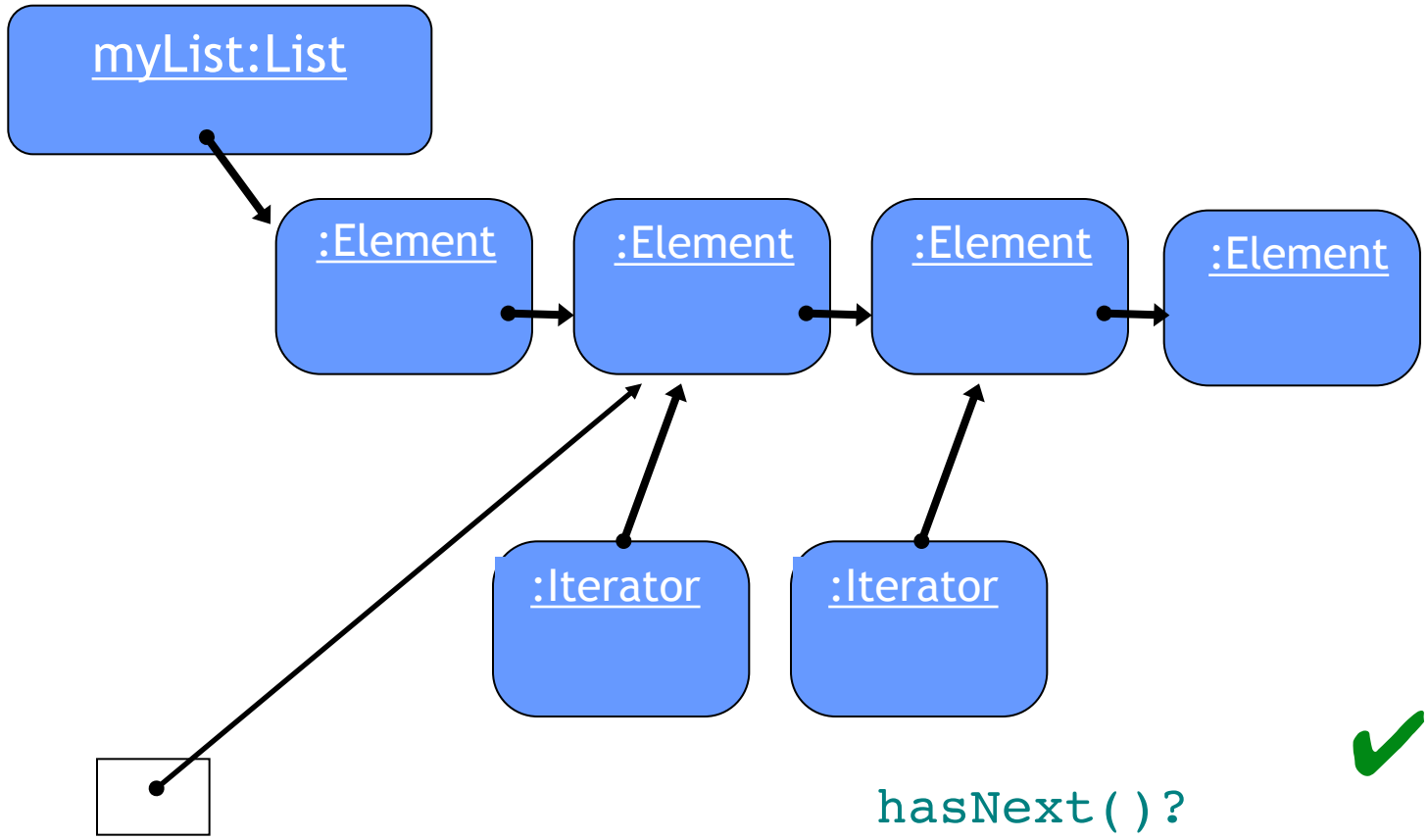

```
myList.iterator()
```



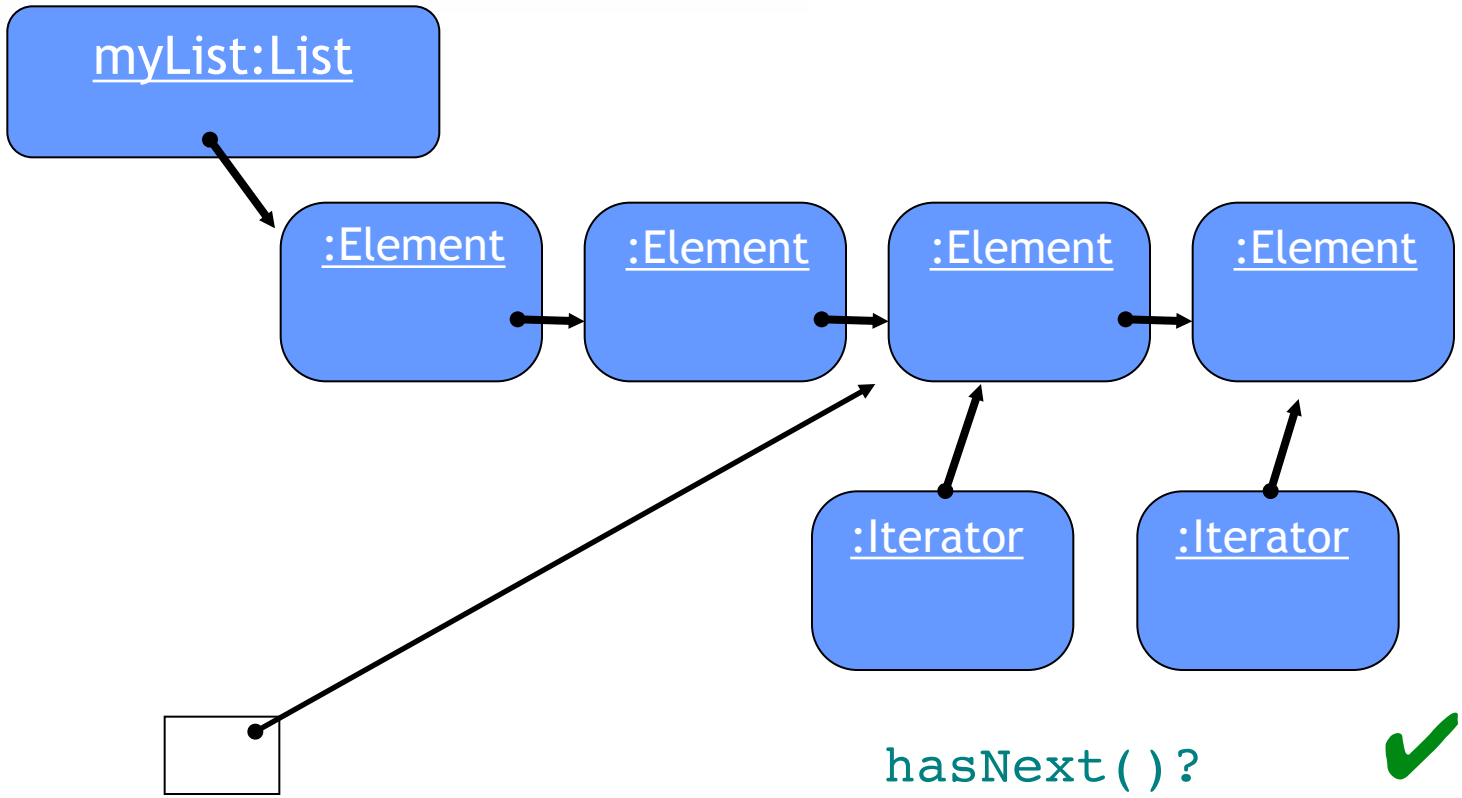

```
hasNext()?
next()
```

✔

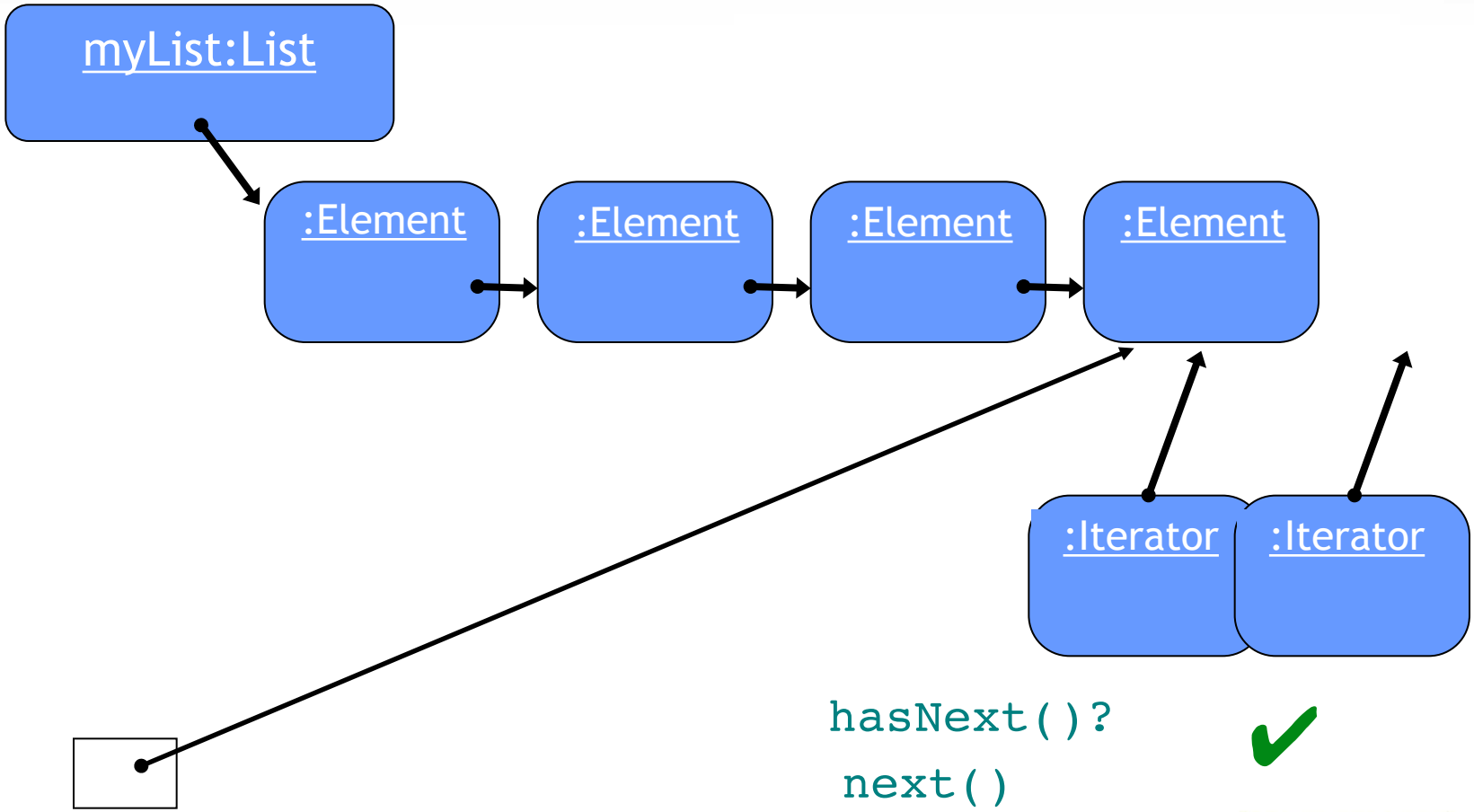
```
Element e = iterator.next();
```

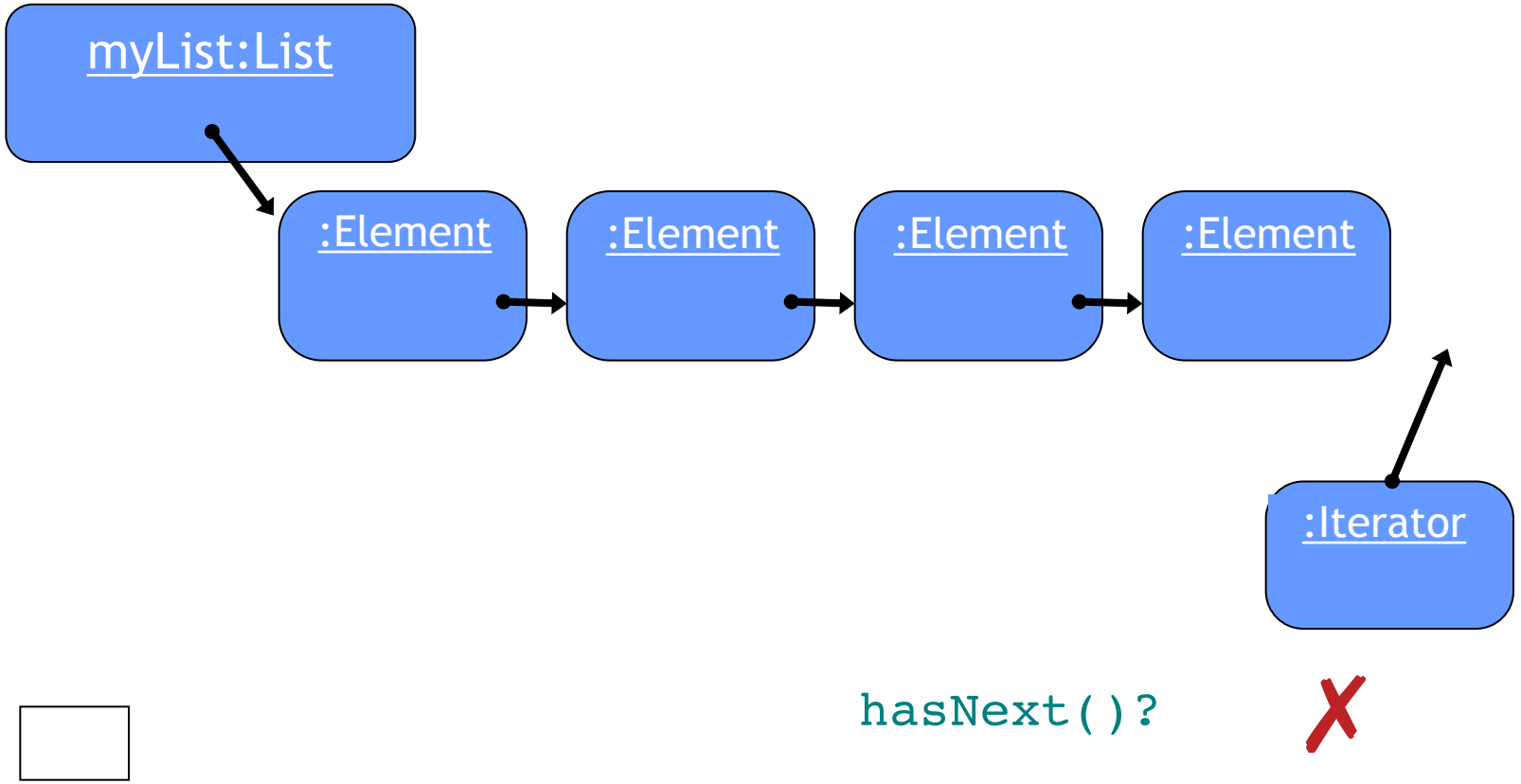



```
hasNext()?
next()
```




```
hasNext()?
next()
```





Utilizando um objeto Iterator

`java.util.Iterator`

retorna um objeto `Iterator`

```
Iterator<ElementType> it = myCollection.iterator();
while(it.hasNext()) {
    call it.next() para obter o próximo objeto
    faz algo com esse objeto
}
```

```
public void listNotes()
{
    Iterator<String> it = notes.iterator();
    while(it.hasNext()) {
        System.out.println(it.next());
    }
}
```

Índice *versus* iteradores

- Maneiras de interar em uma coleção:
  - loop for-each, loop.
    - Usamos se queremos processar cada elemento.
  - loop while.
    - Usamos se quisermos parar no meio do caminho.
    - Usamos para repetição que não envolva uma coleção.
  - Objeto Iterator.
    - Usamos se quisermos parar no meio do caminho.
    - Muitas vezes usado com coleções onde um acesso indexado não é muito eficiente, ou é impossível.
- Iteração é um importante *padrão* de programação.

O projeto *auction*

- O projeto *auction* fornece explicação adicional sobre as coleções e iteração.
- Mais um ponto a ser discutido: o valor `null`.
  - Usado para indicar 'no object'.
  - Podemos testar se uma variável de objeto detém a variável null .

Revisão

- Instruções de loop permitem que um bloco de instruções seja repetido.
- O loop for-each permite iteração sobre toda uma coleção.
- O loop while permite que a repetição seja controlada por uma expressão booleana.
- Todas as classes coleção fornecem objetos `Iterator` especiais que oferecem acesso seqüencial a uma coleção inteira.

Agrupando objetos

Arrays

Coleções de tamanho fixo

- Às vezes o tamanho máximo da coleção pode ser pré-determinado.
- Linguagens de programação normalmente têm um tipo de coleção especial de tamanho fixo: um *array*.
- Arrays Java podem armazenar objetos ou valores de tipo primitivo.
- Arrays usam uma sintaxe especial.

O projeto *weblog-analyzer*

- O servidor Web registra detalhes de cada acesso.
- Suporta tarefas do webmaster.
  - Páginas mais populares.
  - Períodos mais ocupados.
  - Quantos dados estão sendo entregues.
  - Referências que não funcionam.
- Analisa acessos por hora.

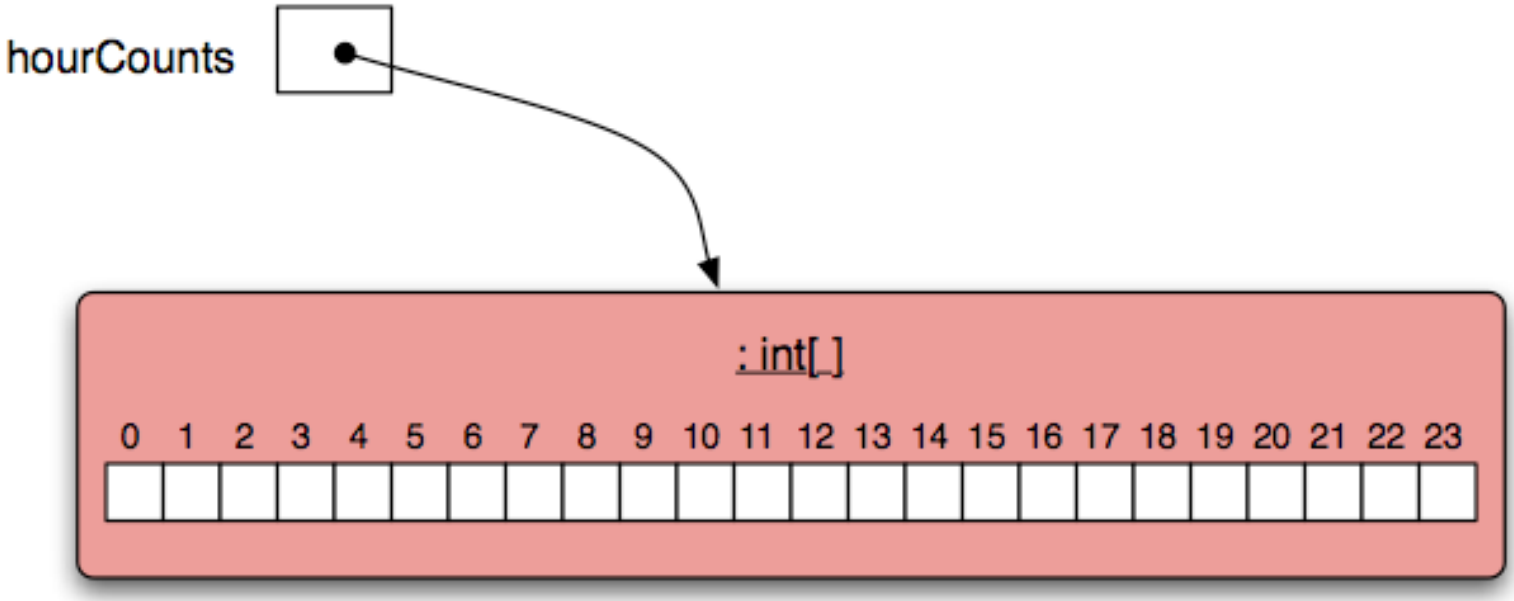
Criando um objeto array

```
public class LogAnalyzer
{
    private int[] hourCounts;
    private LogfileReader reader;

    public LogAnalyzer()
    {
        hourCounts = new int[24];
        reader = new LogfileReader();
    }
    ...
}
```

Instrução de variável de array

Criação de objeto array

O array hourCounts



Utilizando um array

- A notação entre colchetes é usada para acessar um elemento array: **`hourCounts[...]`**
- Elementos são utilizados como variáveis comuns:
  - À esquerda de uma atribuição:

```
hourCounts[hour] = ...;
```

  - Em uma expressão:

```
adjusted = hourCounts[hour] - 3;
hourCounts[hour]++;
```

Uso de array padrão

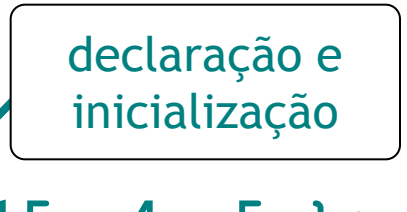
```
private int[] hourCounts;
private String[] names;

...

hourCounts = new int[24];

...

hourcounts[i] = 0;
hourcounts[i]++;
System.out.println(hourcounts[i]);
```

declaração

criação

uso

Literais de array




```
private int[] numbers = { 3, 15, 4, 5 };
```

```
System.out.println(numbers[i]);
```

- Os literais podem ser utilizados apenas em inicializações.

Comprimento do array

```
private int[] numbers = { 3,
15, 4, 5 };
```

```
int n = numbers.length;
```

sem colchetes!

- Nota: 'comprimento' não é um método.

O loop for

- Há duas variações do loop, *for-each* e *for*.
- O loop for é muito usado para iterar um número fixo de vezes.
- Muitas vezes usado com uma variável que muda uma quantidade fixa de cada iteração.

Pseudocódigo do loop while

Forma geral de um loop while

```
for(initialization; condition; post-body action) {
    statements to be repeated
}
```

Equivalente à forma de loop while

```
initialization;
while(condition) {
    statements to be repeated
    post-body action
}
```

Um exemplo do Java

para versão de loop

```
for(int hour = 0; hour < hourCounts.length; hour++) {
    System.out.println(hour + ": " + hourCounts[hour]);
}
```

versão do loop while

```
int hour = 0;
while(hour < hourCounts.length) {
    System.out.println(hour + ": " + hourCounts[hour]);
    hour++;
}
```

Prática

- Considerando um array de números, imprima todos os números no array, usando um loop for.

```
int[] numbers = { 4, 1, 22, 9, 14, 3, 9};

for ...
```

Prática

- Preencha um array com a sequência de Fibonacci.

0 1 1 2 3 5 8 13 21 34 ...

```
int[] fib = new int[100];

fib[0] = 0;
fib[1] = 1;

for ...
```

loop for com um passo maior

```
// Imprime múltiplos de 3 que estão abaixo de 40.
for(int num = 3; num < 40; num = num + 3) {
    System.out.println(num);
}
```

Revisão

- Arrays são apropriados onde uma coleção de tamanho fixo é necessária.
- Arrays usam uma sintaxe especial.
- Loops for oferecem uma alternativa aos loops while quando o número de repetições é conhecido.
- Os loops for são usados quando uma variável índice é necessária.