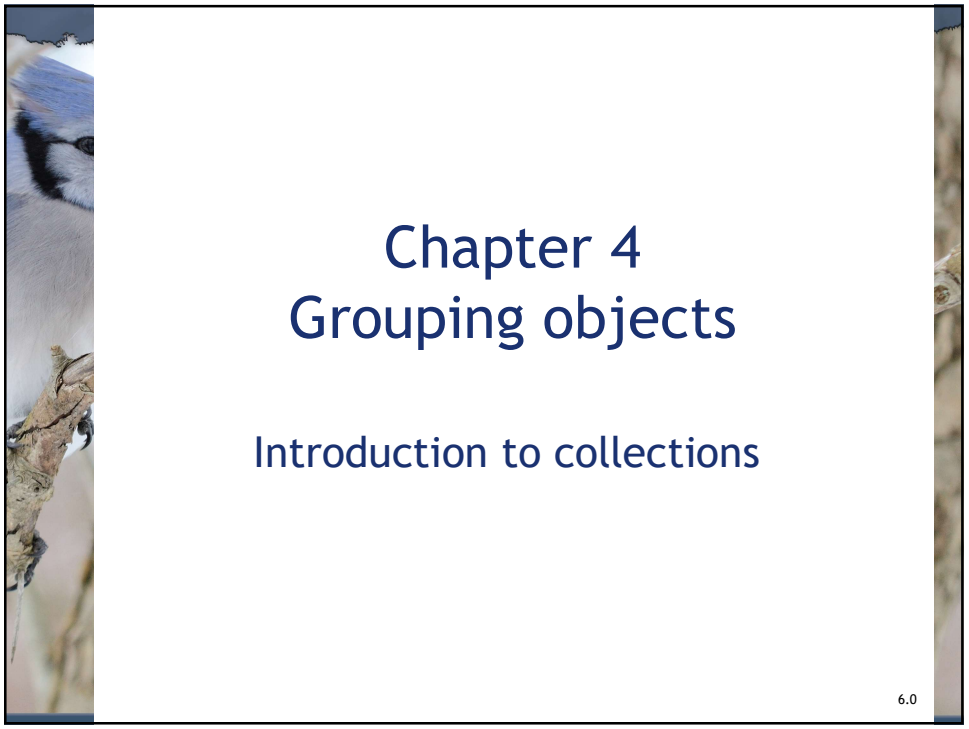
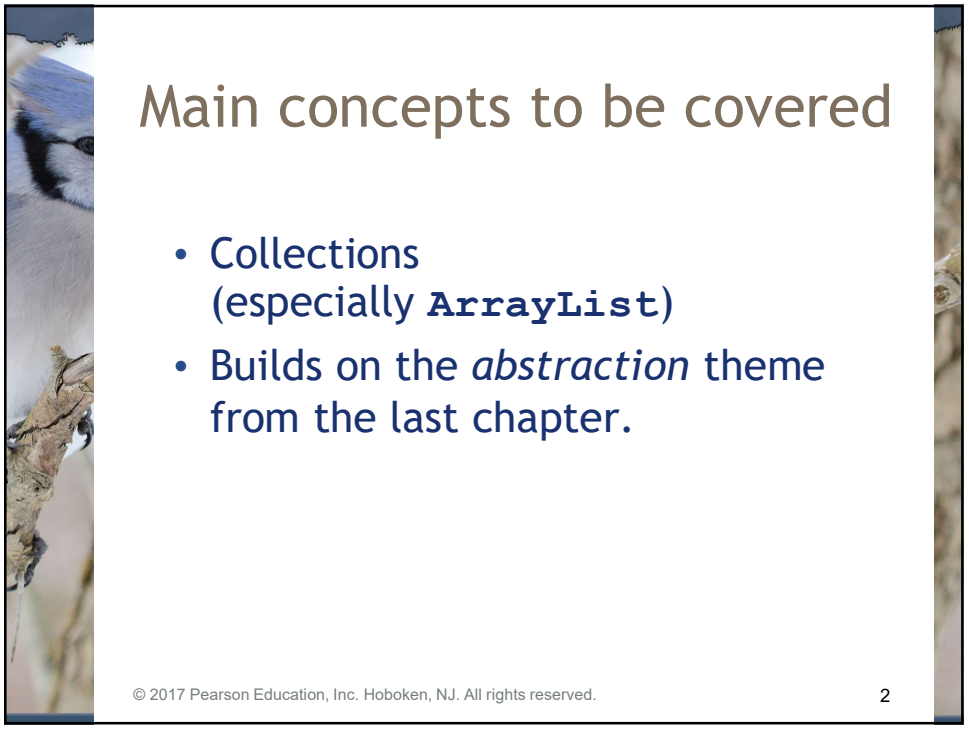


Chapter 4
Grouping objects

Introduction to collections

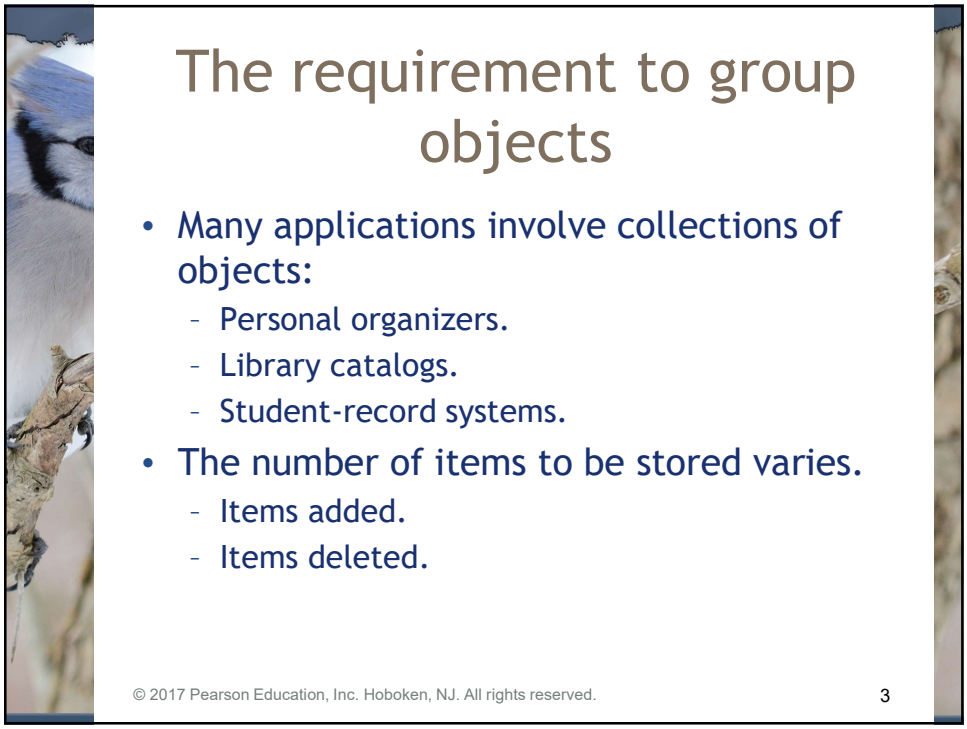
6.0

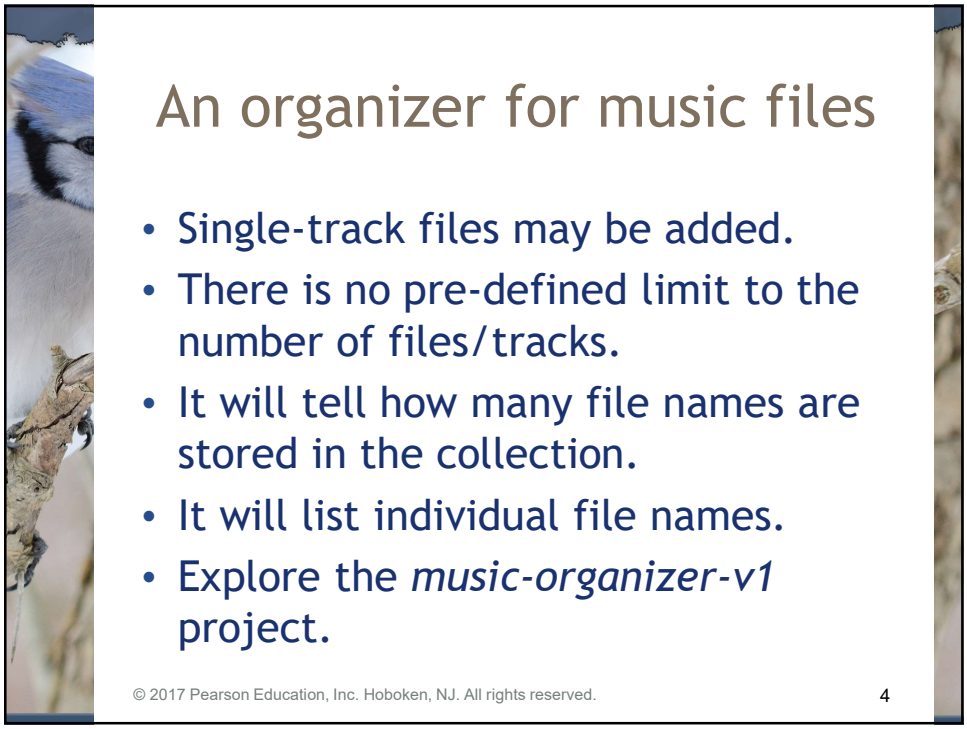


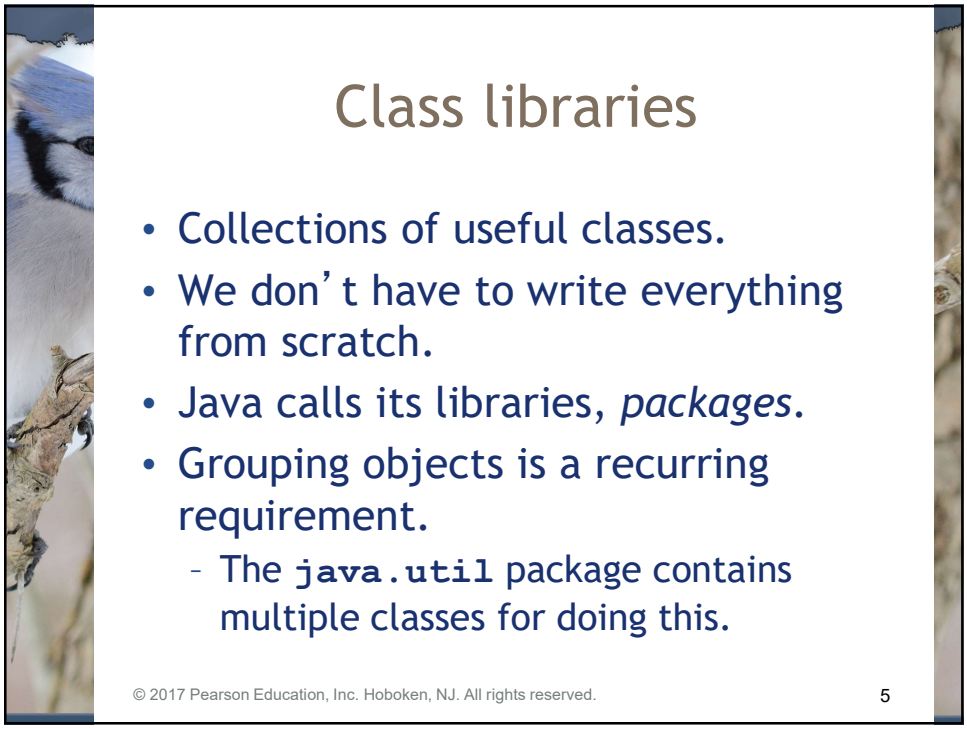
Main concepts to be covered

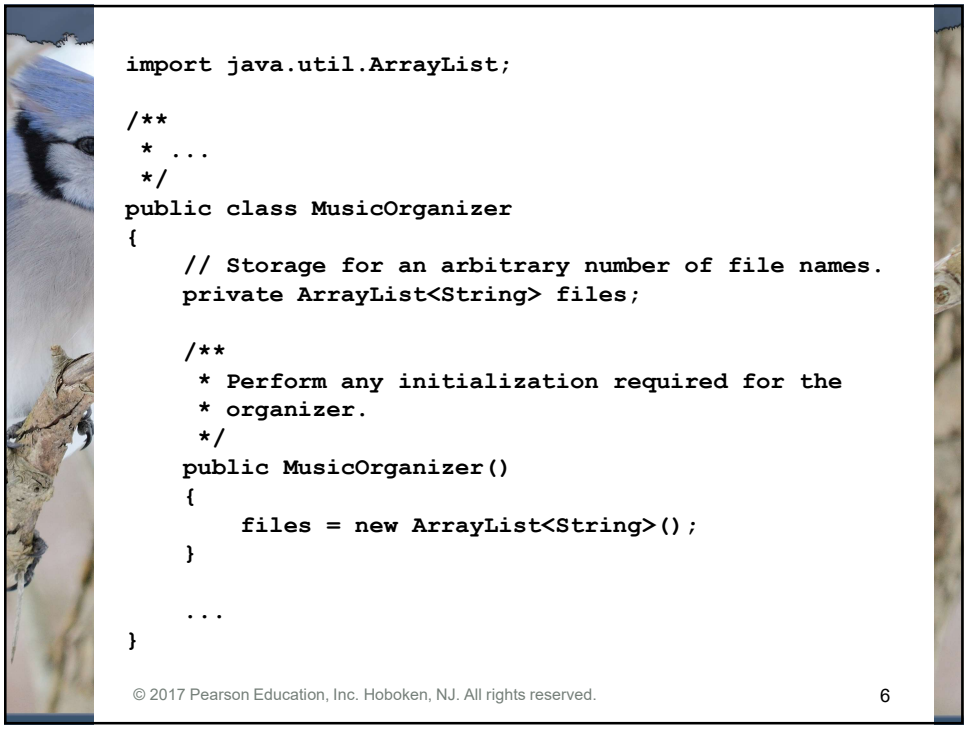
- Collections (especially **`ArrayList`**)
- Builds on the *abstraction* theme from the last chapter.

2

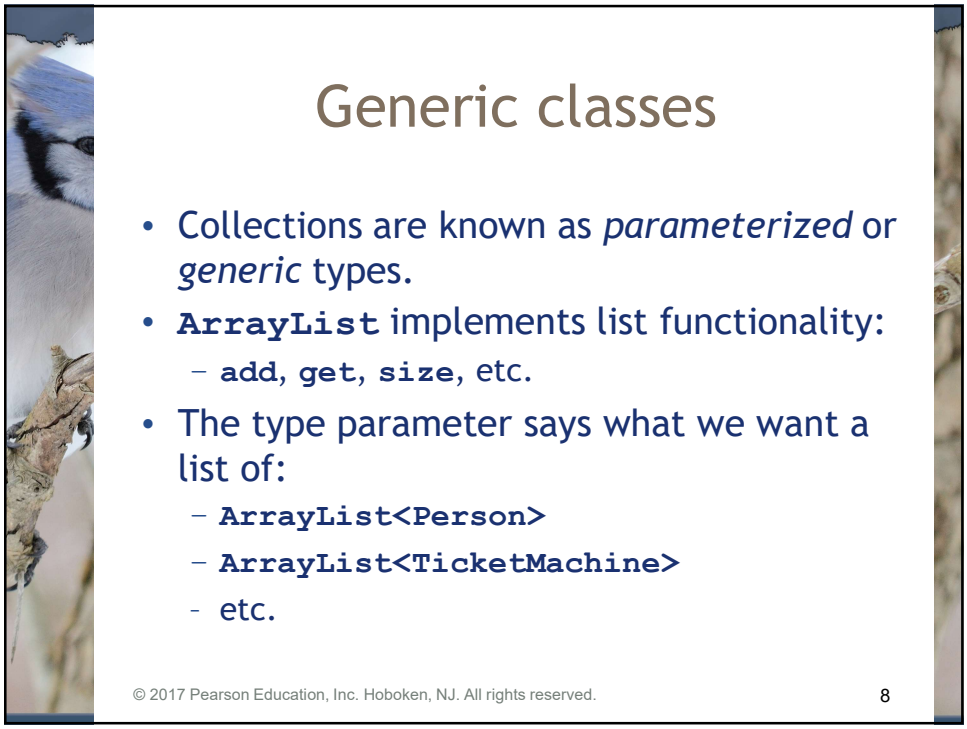

The requirement to group objects
• Many applications involve collections of objects:
– Personal organizers.
– Library catalogs.
– Student-record systems.
• The number of items to be stored varies.
– Items added.
– Items deleted.
© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.
3


An organizer for music files
• Single-track files may be added.
• There is no pre-defined limit to the number of files/tracks.
• It will tell how many file names are stored in the collection.
• It will list individual file names.
• Explore the music-organizer-v1 project.
© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.
4


Class libraries
• Collections of useful classes.
• We don't have to write everything from scratch.
• Java calls its libraries, *packages*.
• Grouping objects is a recurring requirement.
– The **java.util** package contains multiple classes for doing this.
© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.
5


import java.util.ArrayList;
/**
* ...
*/
public class MusicOrganizer
{
// Storage for an arbitrary number of file names.
private ArrayList<String> files;
/**
* Perform any initialization required for the
* organizer.
*/
public MusicOrganizer()
{
files = new ArrayList<String>();
}
...
}
© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.
6


Collections
• We specify:
– the type of collection: ArrayList
– the type of objects it will contain: <String>
– private ArrayList<String> files;
• We say, “ArrayList of String”.
© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.
7

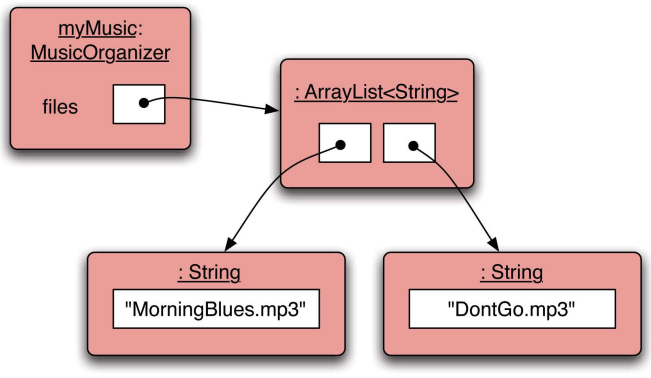

Generic classes
• Collections are known as parameterized or generic types.
• ArrayList implements list functionality:
– add, get, size, etc.
• The type parameter says what we want a list of:
– ArrayList<Person>
– ArrayList<TicketMachine>
– etc.
© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.
8

Creating an ArrayList object

- In versions of Java prior to version 7:
  - `files = new ArrayList<String>();`
- Java 7 introduced 'diamond notation'
  - `files = new ArrayList<>();`
- The type parameter can be inferred from the variable being assigned to.
  - A convenience we will use.

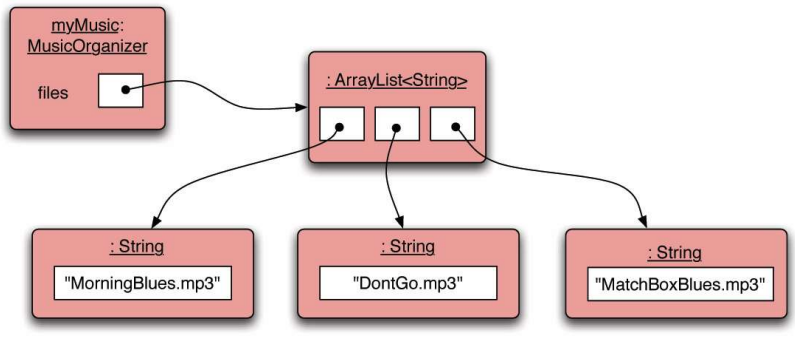
 9

Object structures with collections



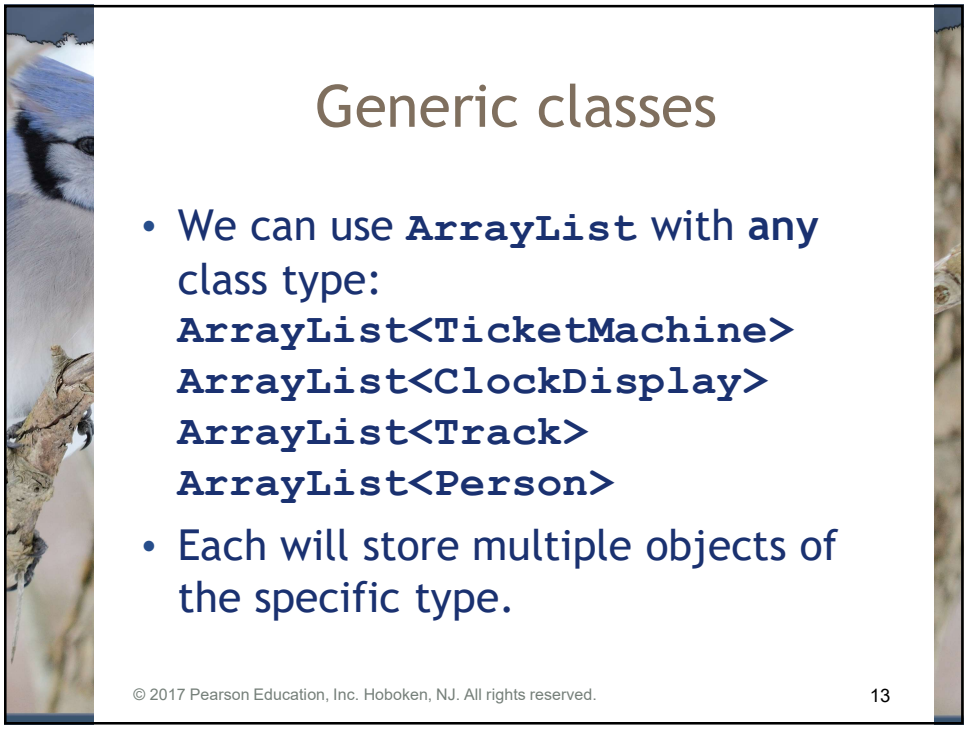
 10

Adding a third file



Features of the collection

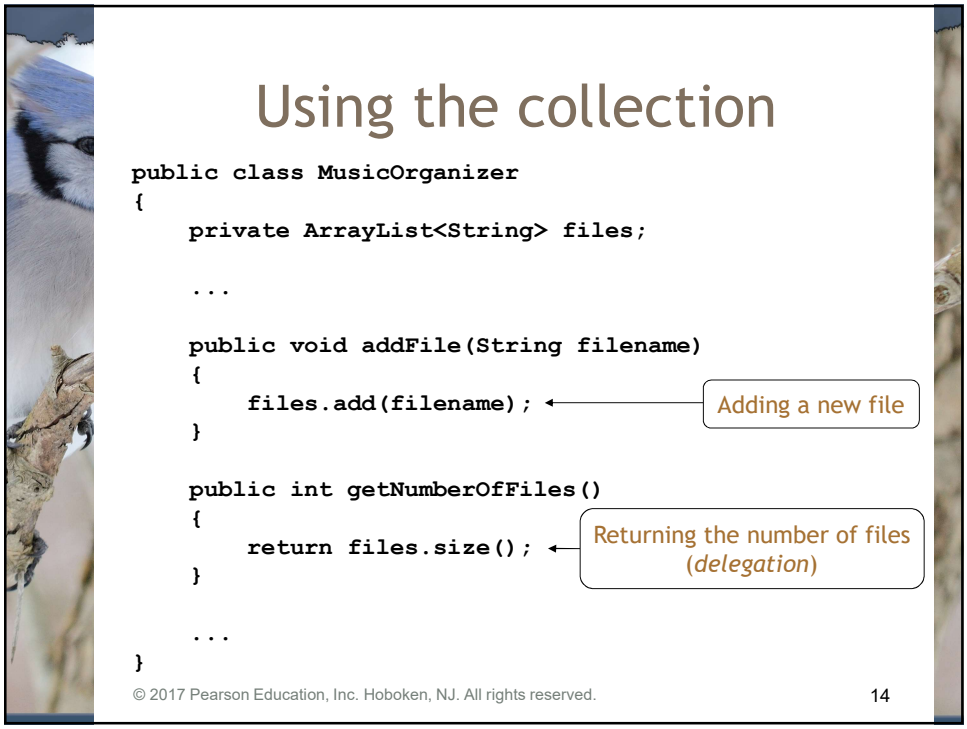
- It increases its capacity as necessary.
- It keeps a private count:
  - `size()` accessor.
- It keeps the objects in order.
- Details of how all this is done are hidden.
  - Does that matter? Does not knowing how prevent us from using it?



Generic classes

- We can use **`ArrayList`** with **any** class type:
  `ArrayList<TicketMachine>`
  `ArrayList<ClockDisplay>`
  `ArrayList<Track>`
  `ArrayList<Person>`
- Each will store multiple objects of the specific type.

13



Using the collection

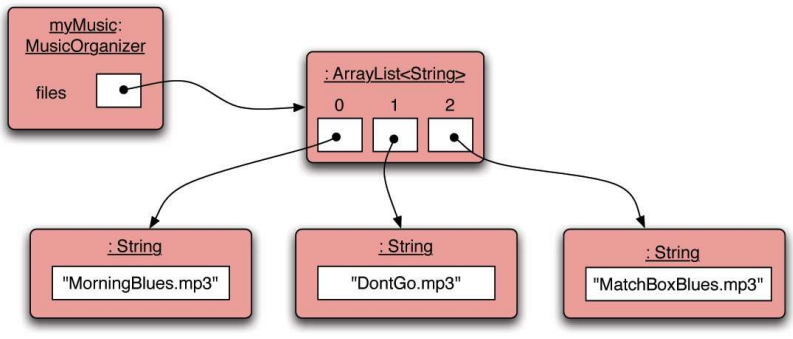
```
public class MusicOrganizer
{
    private ArrayList<String> files;

    ...

    public void addFile(String filename)
    {
        files.add(filename);
    }

    public int getNumberOfFiles()
    {
        return files.size();
    }

    ...
}
```

Adding a new file

Returning the number of files (*delegation*)

14

Index numbering



15

Retrieving from the collection

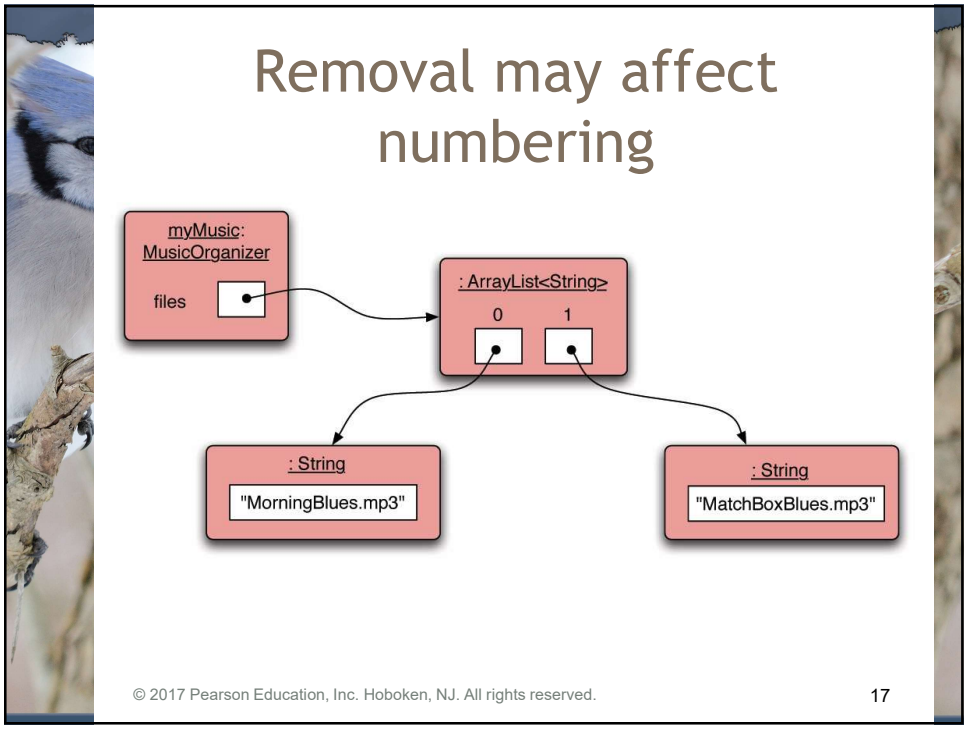
```
public void listFile(int index)
{
    if(index >= 0 && index < files.size()) {
        String filename = files.get(index);
        System.out.println(filename);
    }
    else {
        // This is not a valid index.
    }
}
```

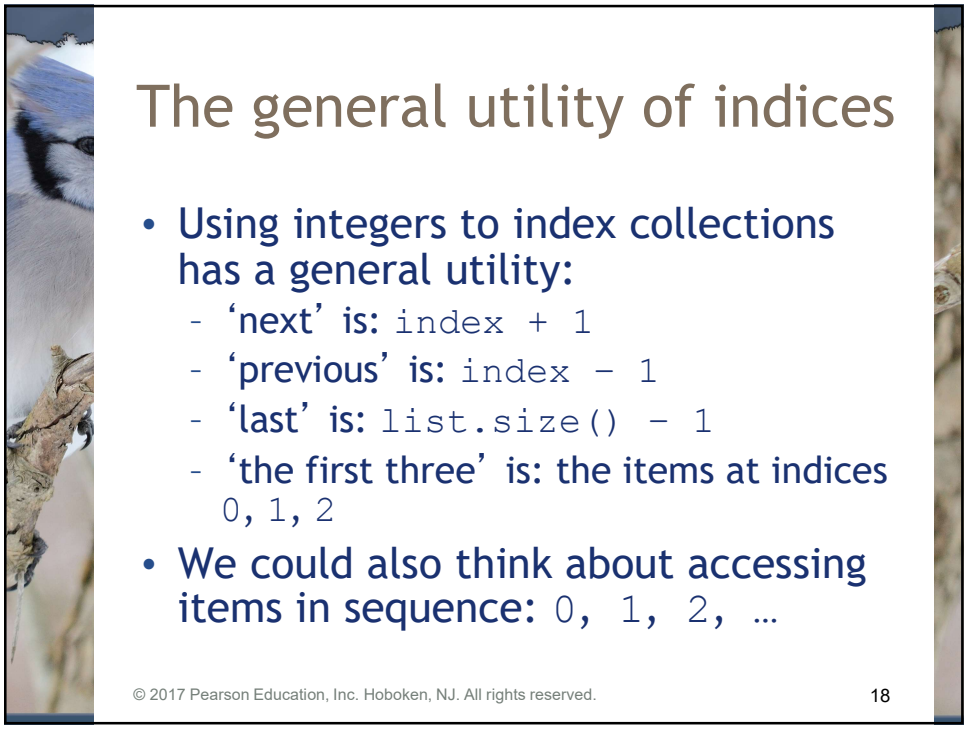
Index validity checks

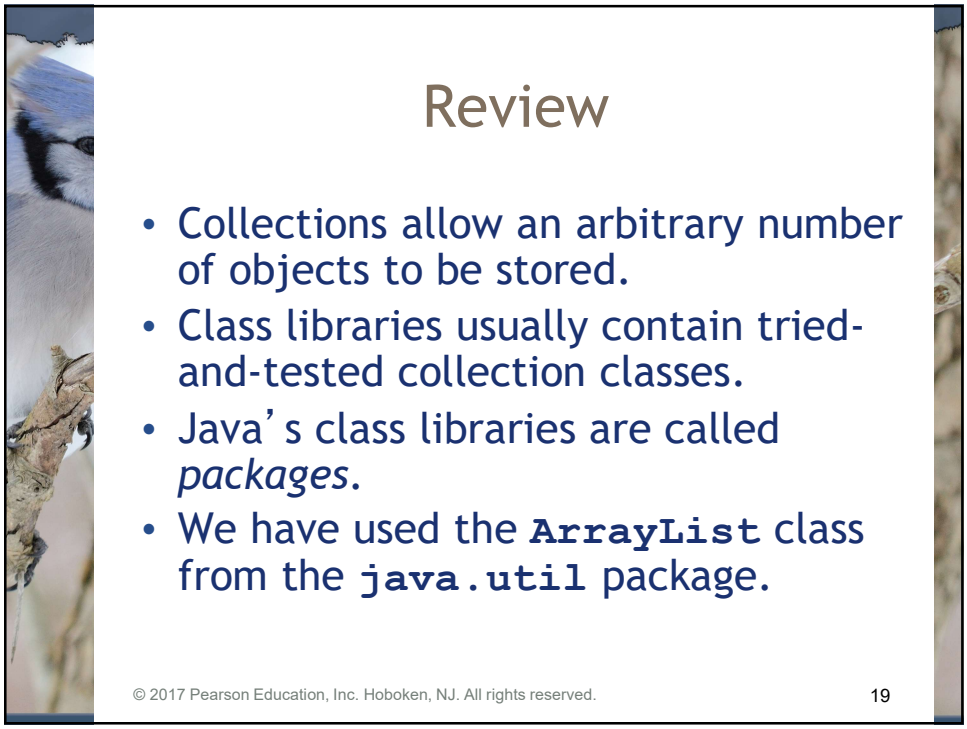
Retrieve and print the file name

Needed? (Error message?)

16


Removal may affect numbering
myMusic:
MusicOrganizer
files
: ArrayList<String>
0
1
: String
"MorningBlues.mp3"
: String
"MatchBoxBlues.mp3"
© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.
17

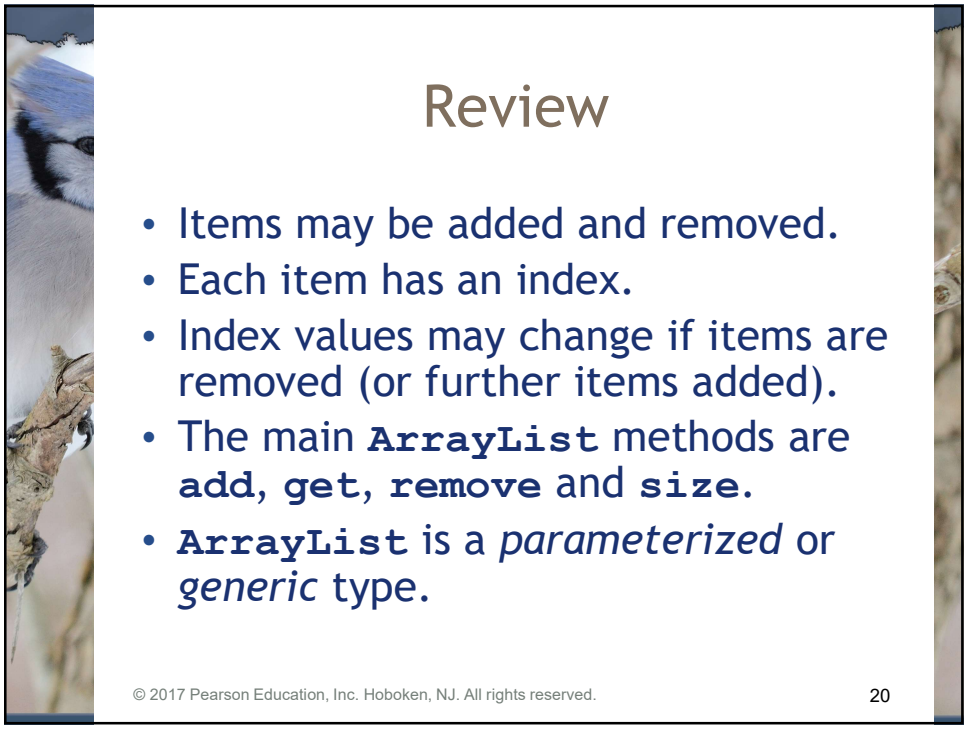

The general utility of indices
• Using integers to index collections has a general utility:
- 'next' is: index + 1
- 'previous' is: index - 1
- 'last' is: list.size() - 1
- 'the first three' is: the items at indices 0, 1, 2
• We could also think about accessing items in sequence: 0, 1, 2, …
© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.
18



Review

- Collections allow an arbitrary number of objects to be stored.
- Class libraries usually contain tried-and-tested collection classes.
- Java's class libraries are called *packages*.
- We have used the **`ArrayList`** class from the **`java.util`** package.

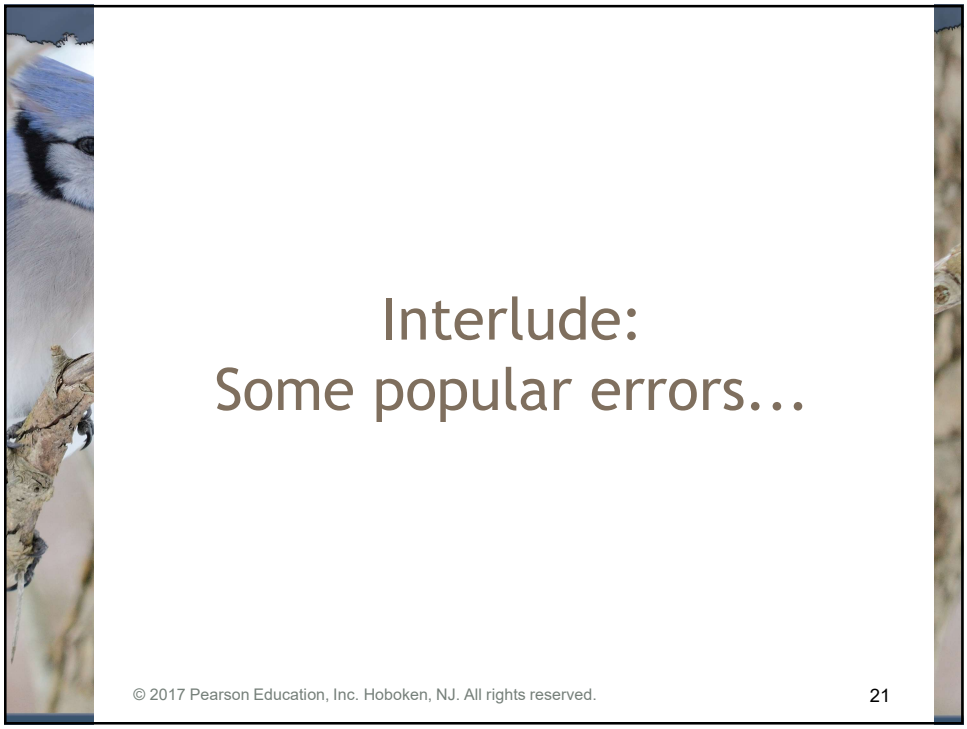
 19

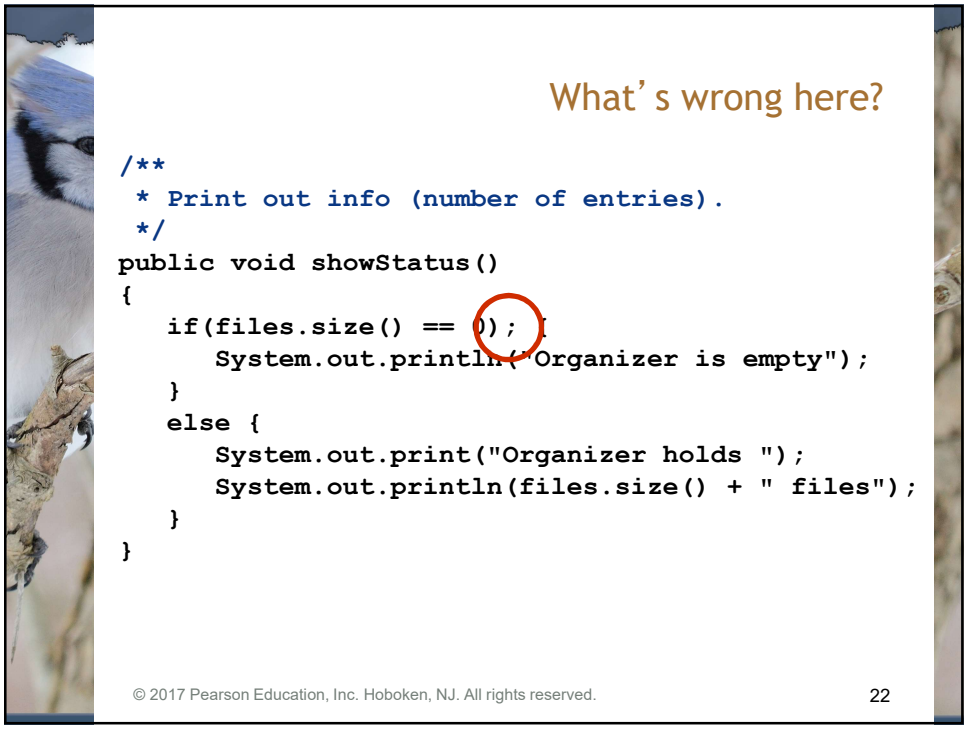


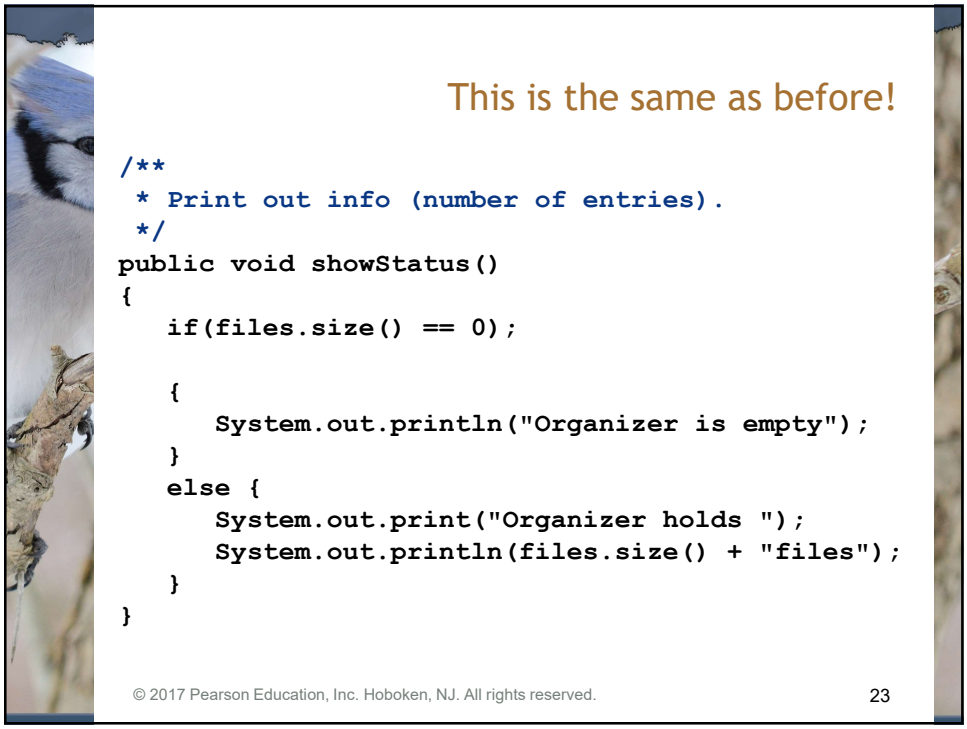
Review

- Items may be added and removed.
- Each item has an index.
- Index values may change if items are removed (or further items added).
- The main **`ArrayList`** methods are **`add`**, **`get`**, **`remove`** and **`size`**.
- **`ArrayList`** is a *parameterized* or *generic* type.

 20


Interlude:
Some popular errors...
© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.
21

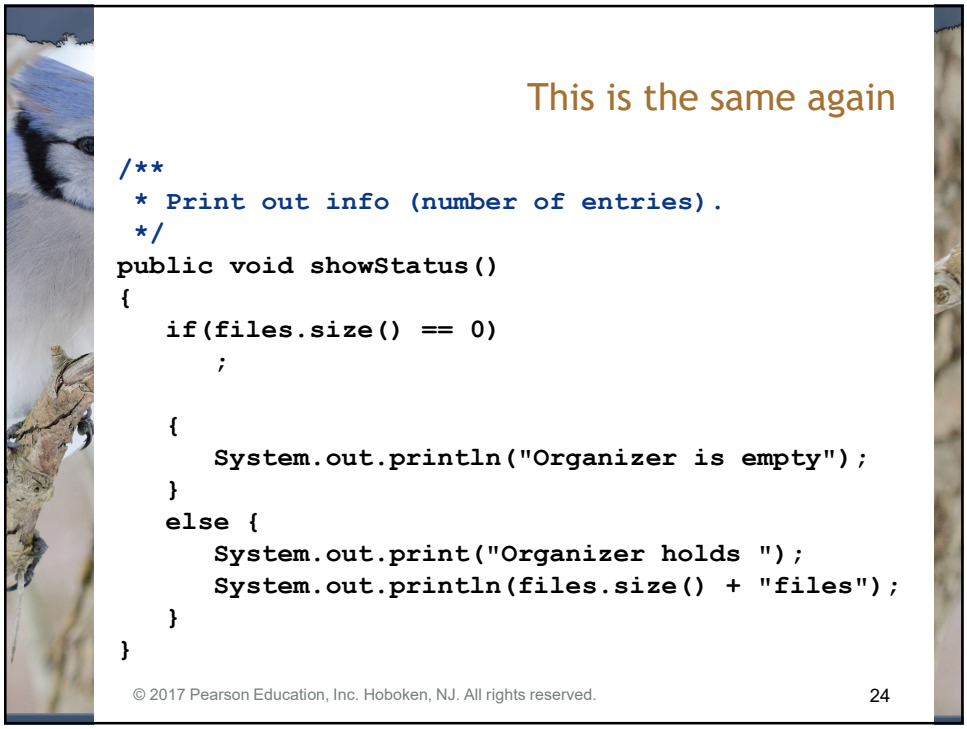

What's wrong here?
/**
* Print out info (number of entries).
*/
public void showStatus()
{
if(files.size() == 0);
System.out.println("Organizer is empty");
}
else {
System.out.print("Organizer holds ");
System.out.println(files.size() + " files");
}
}
© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.
22



This is the same as before!

```
/**
 * Print out info (number of entries).
 */
public void showStatus()
{
   if(files.size() == 0);

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

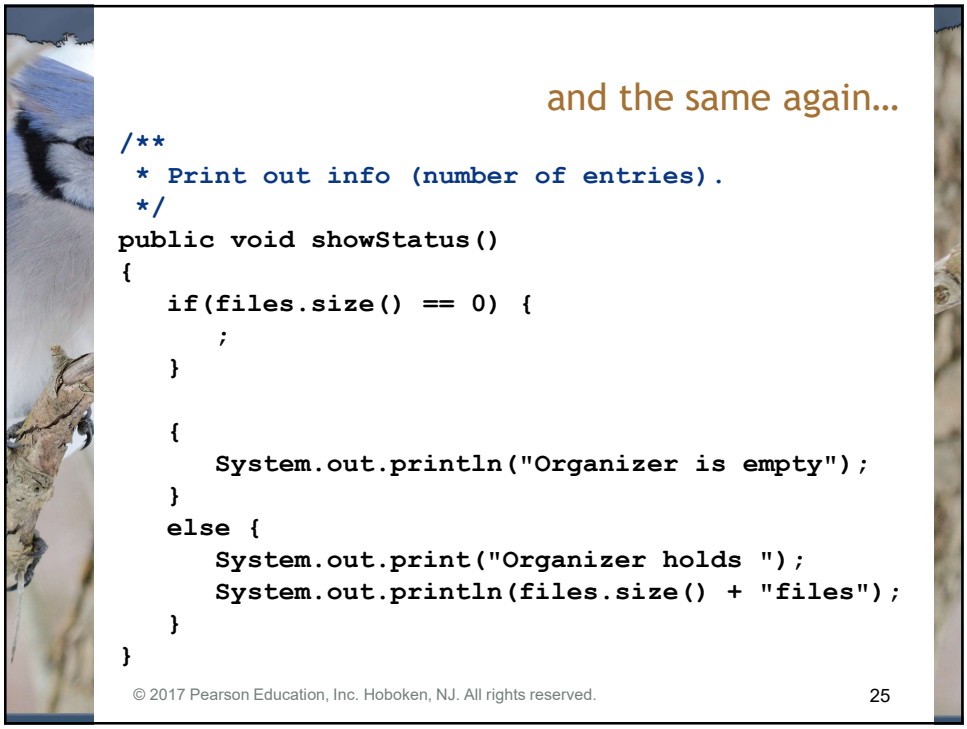
 23



This is the same again

```
/**
 * Print out info (number of entries).
 */
public void showStatus()
{
   if(files.size() == 0)
      ;

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

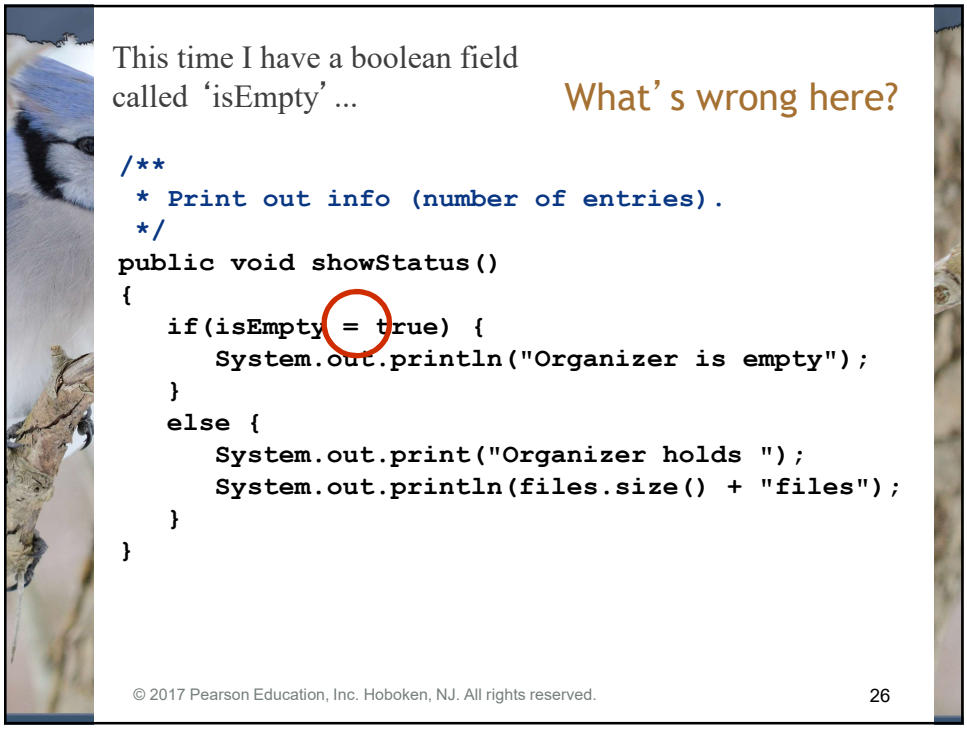
 24



and the same again…

```
/**
 * Print out info (number of entries).
 */
public void showStatus()
{
   if(files.size() == 0) {
      ;
   }

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


25

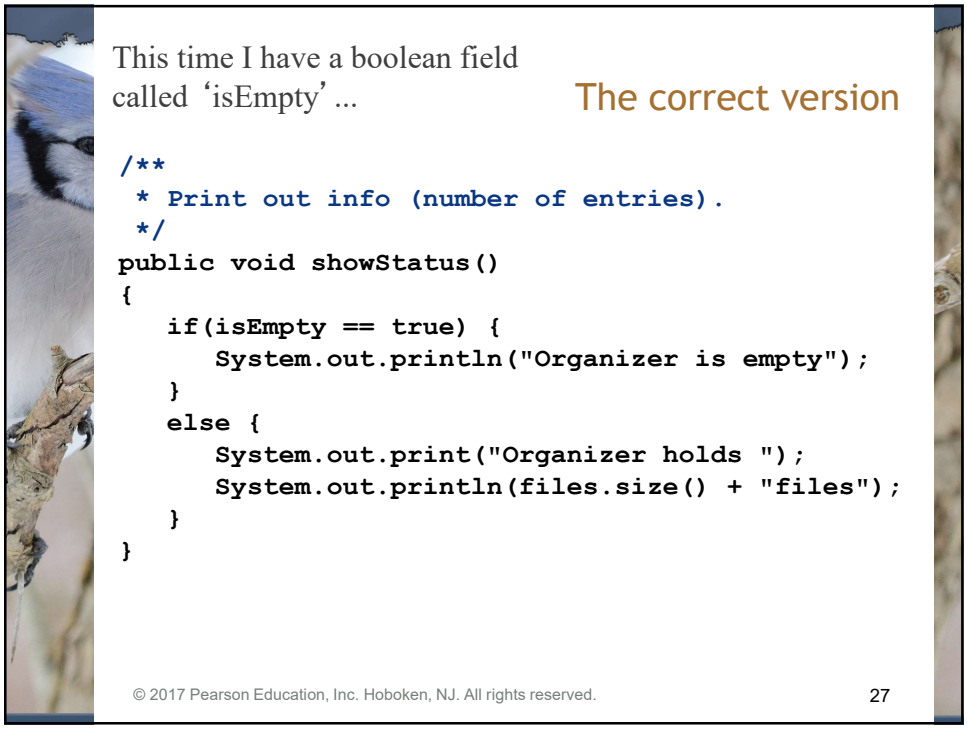


This time I have a boolean field called 'isEmpty' ...

What's wrong here?

```
/**
 * Print out info (number of entries).
 */
public void showStatus()
{
   if(isEmpty = true) {
      System.out.println("Organizer is empty");
   }
   else {
      System.out.print("Organizer holds ");
      System.out.println(files.size() + "files");
   }
}
```


26

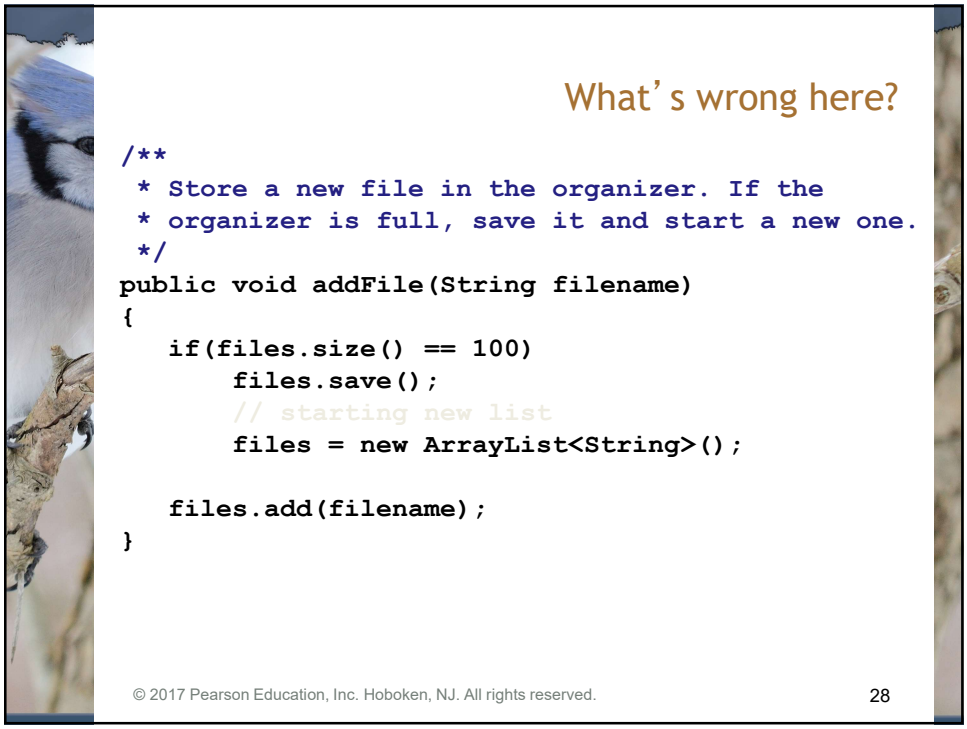


This time I have a boolean field called 'isEmpty' ...

The correct version

```
/**
 * Print out info (number of entries).
 */
public void showStatus()
{
   if(isEmpty == true) {
      System.out.println("Organizer is empty");
   }
   else {
      System.out.print("Organizer holds ");
      System.out.println(files.size() + "files");
   }
}
```

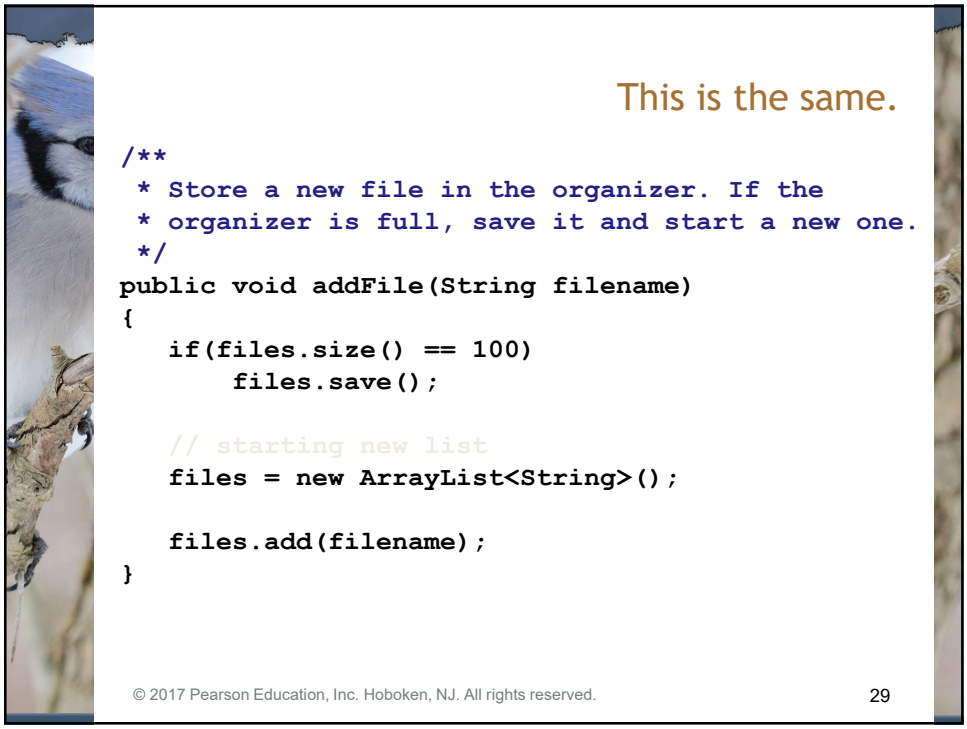
 27

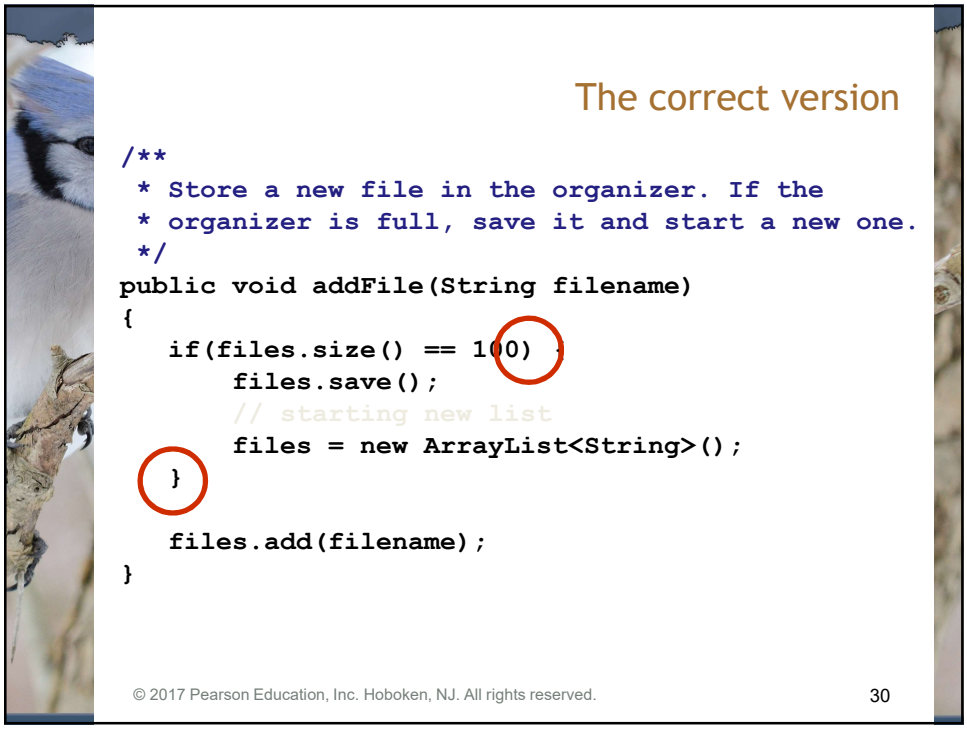


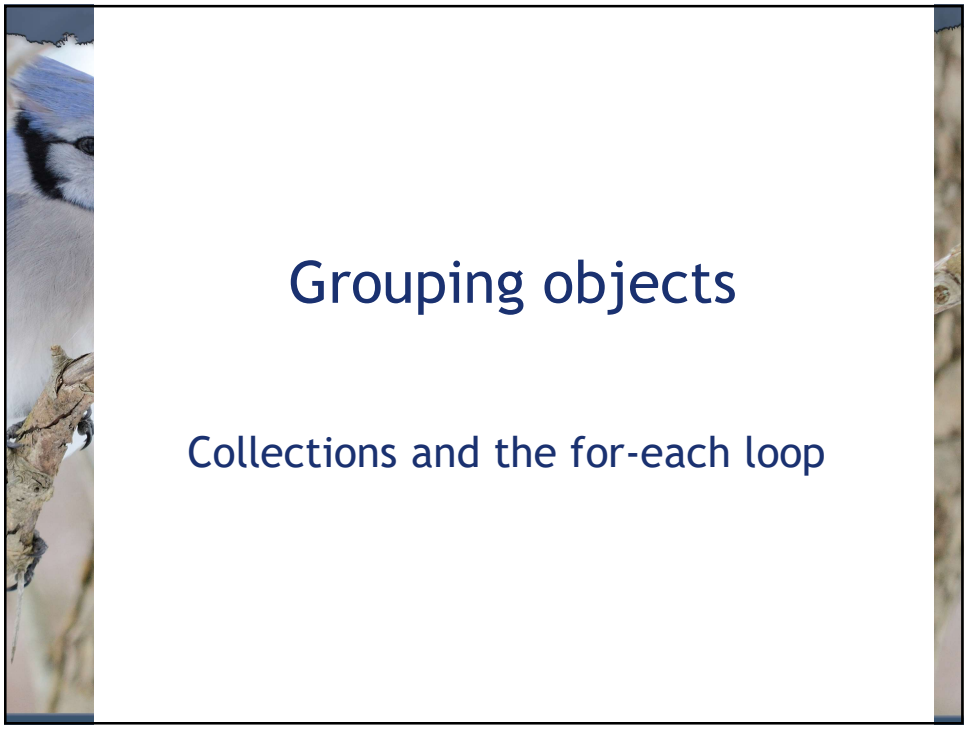
What's wrong here?

```
/**
 * Store a new file in the organizer. If the
 * organizer is full, save it and start a new one.
 */
public void addFile(String filename)
{
   if(files.size() == 100)
       files.save();
       // starting new list
       files = new ArrayList<String>();

   files.add(filename);
}
```

 28

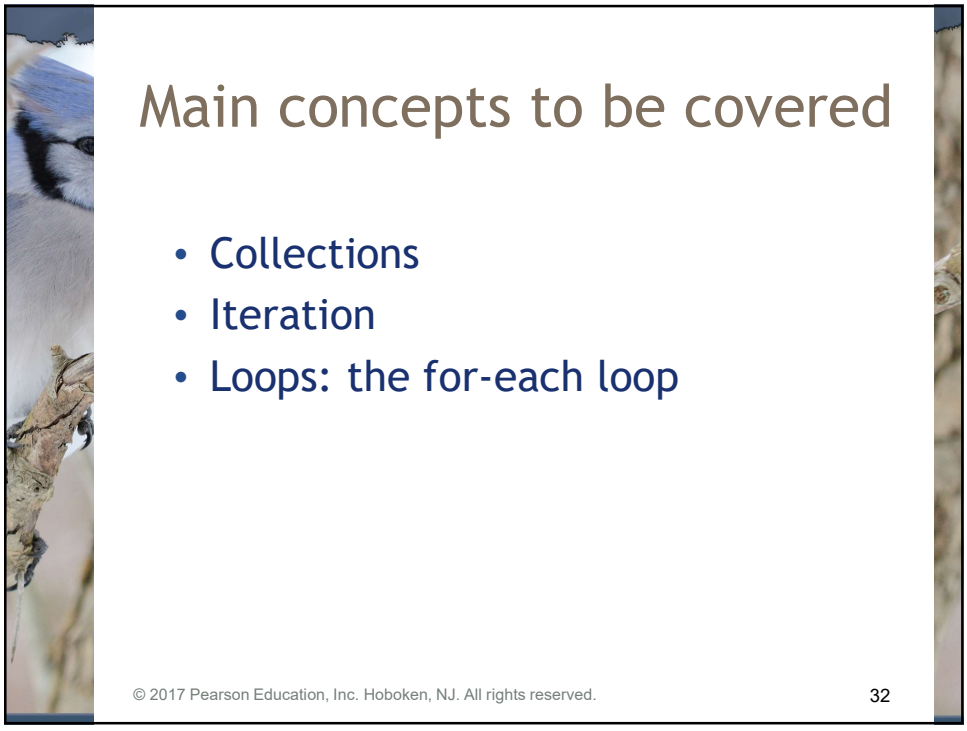

This is the same.
/**
* Store a new file in the organizer. If the
* organizer is full, save it and start a new one.
*/
public void addFile(String filename)
{
if(files.size() == 100)
files.save();
// starting new list
files = new ArrayList<String>();
files.add(filename);
}
© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.
29


The correct version
/**
* Store a new file in the organizer. If the
* organizer is full, save it and start a new one.
*/
public void addFile(String filename)
{
if(files.size() == 100) {
files.save();
// starting new list
files = new ArrayList<String>();
}
files.add(filename);
}
© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.
30



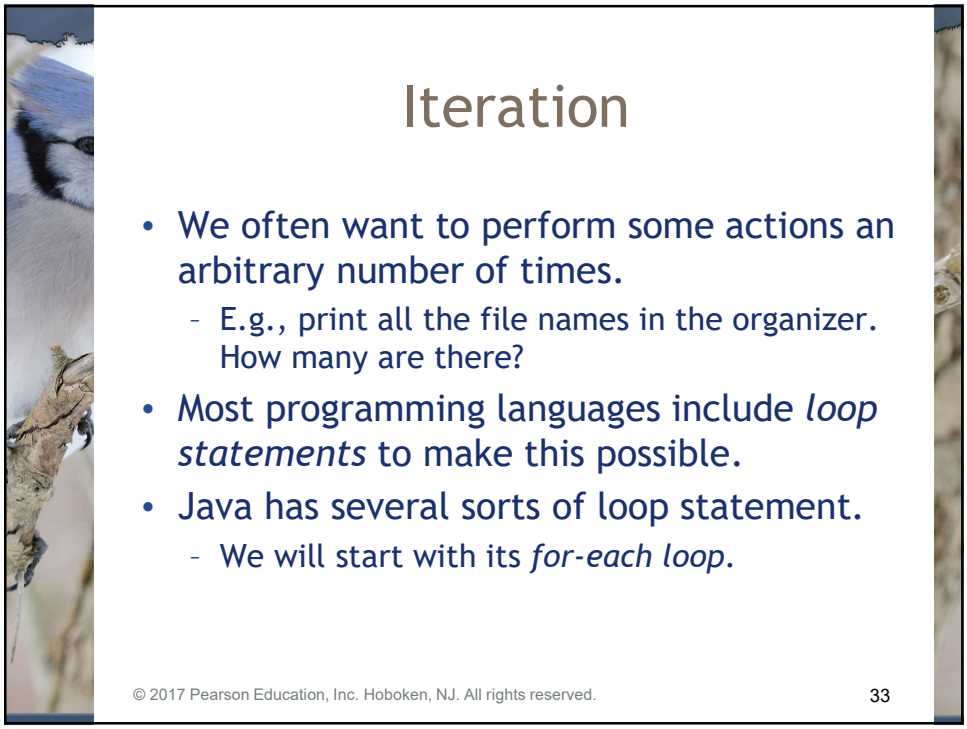
Grouping objects

Collections and the for-each loop



Main concepts to be covered

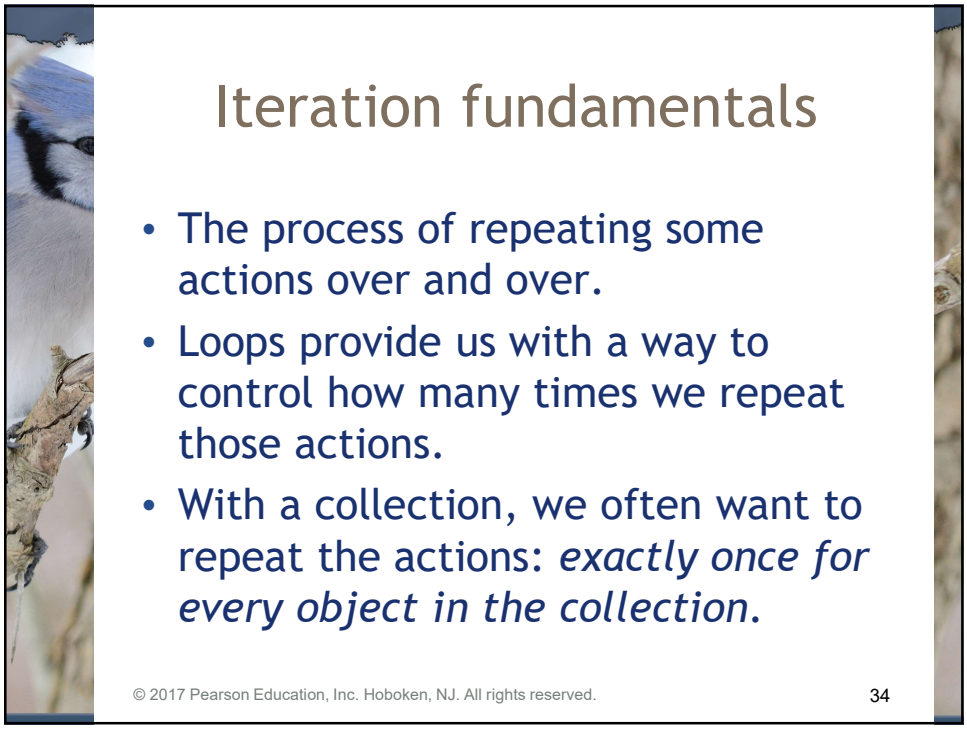
- Collections
- Iteration
- Loops: the for-each loop



Iteration

- We often want to perform some actions an arbitrary number of times.
  - E.g., print all the file names in the organizer. How many are there?
- Most programming languages include *loop statements* to make this possible.
- Java has several sorts of loop statement.
  - We will start with its *for-each loop*.

 33

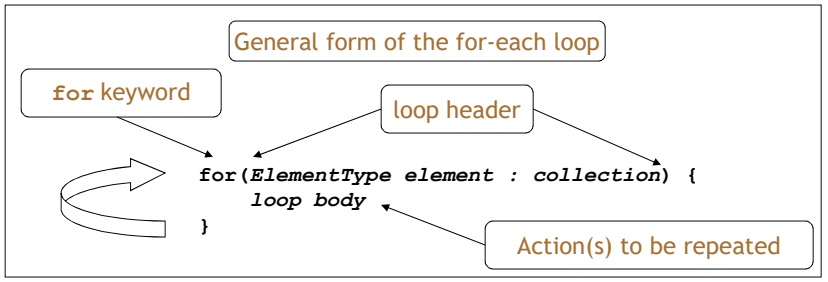


Iteration fundamentals

- The process of repeating some actions over and over.
- Loops provide us with a way to control how many times we repeat those actions.
- With a collection, we often want to repeat the actions: *exactly once for every object in the collection*.

 34

For-each loop pseudo code



Pseudo-code expression of the operation of a for-each loop

Using each *element* in *collection* in order, do the things in the *loop body* with that *element*.

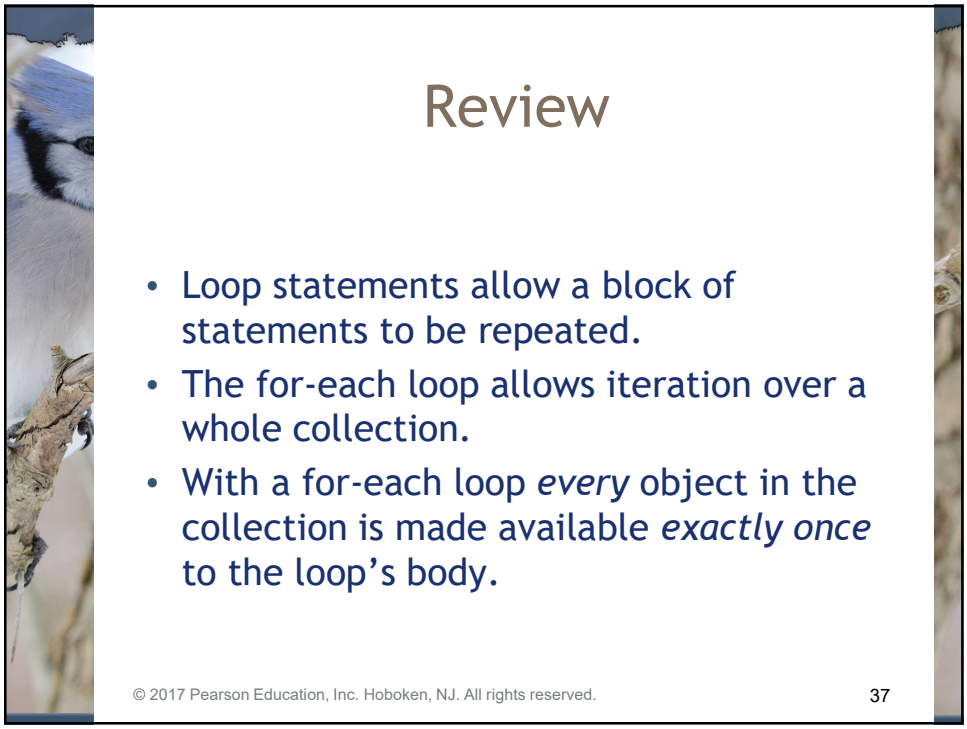
35

A Java example

```
/**
 * List all file names in the organizer.
 */
public void listAllFiles()
{
    for(String filename : files) {
        System.out.println(filename);
    }
}
```

Using each *filename* in *files* in order, print *filename*

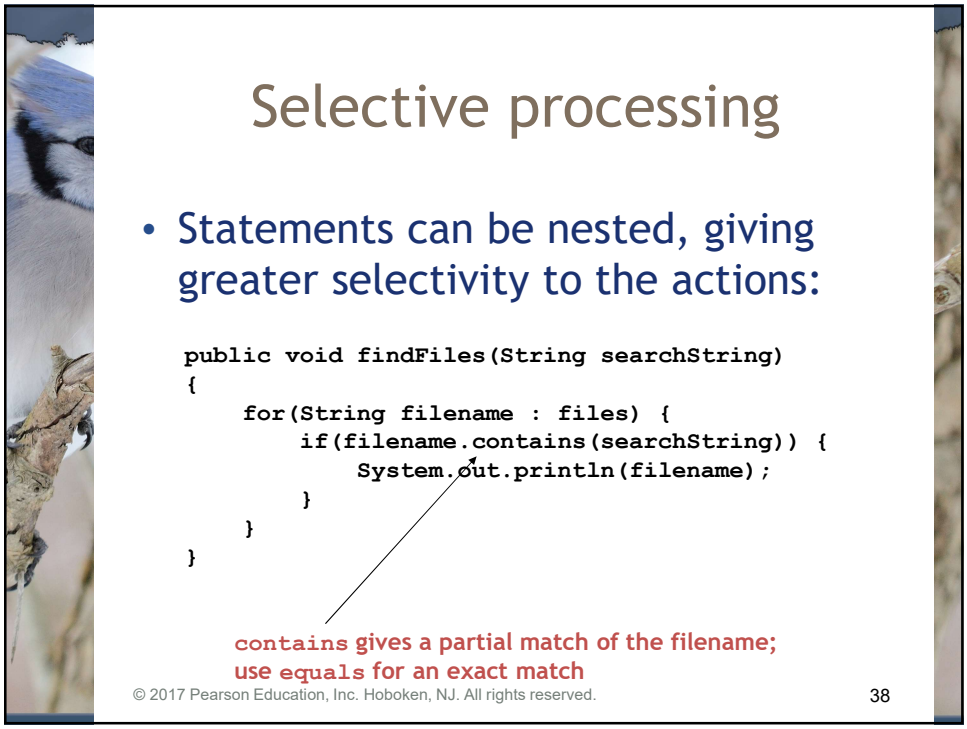
36



Review

- Loop statements allow a block of statements to be repeated.
- The for-each loop allows iteration over a whole collection.
- With a for-each loop *every* object in the collection is made available *exactly once* to the loop's body.

 37



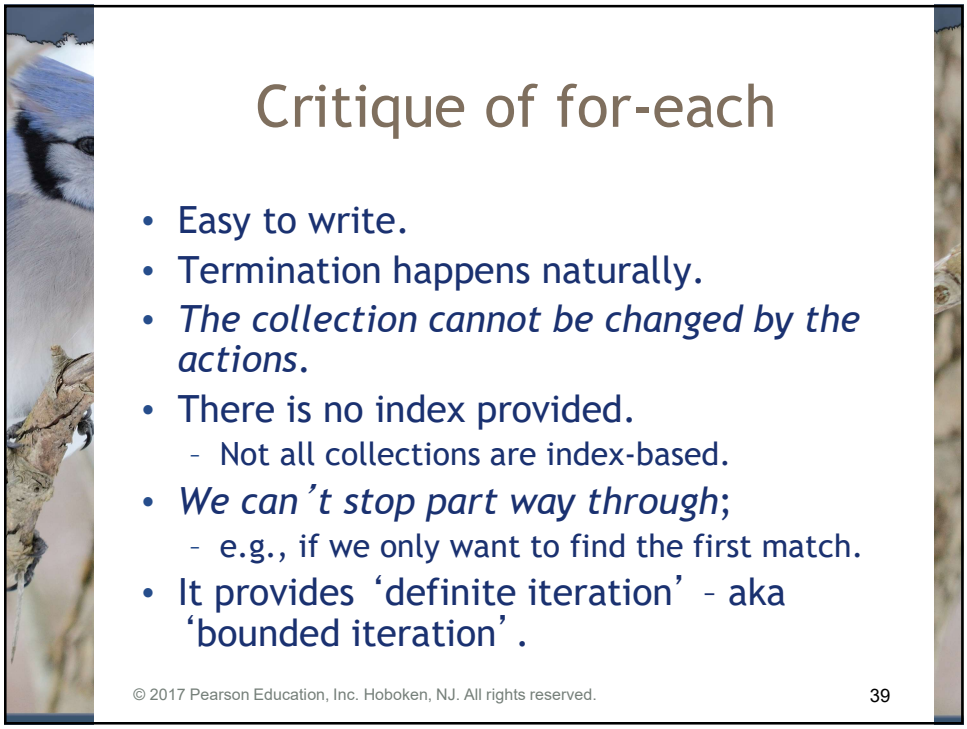
Selective processing

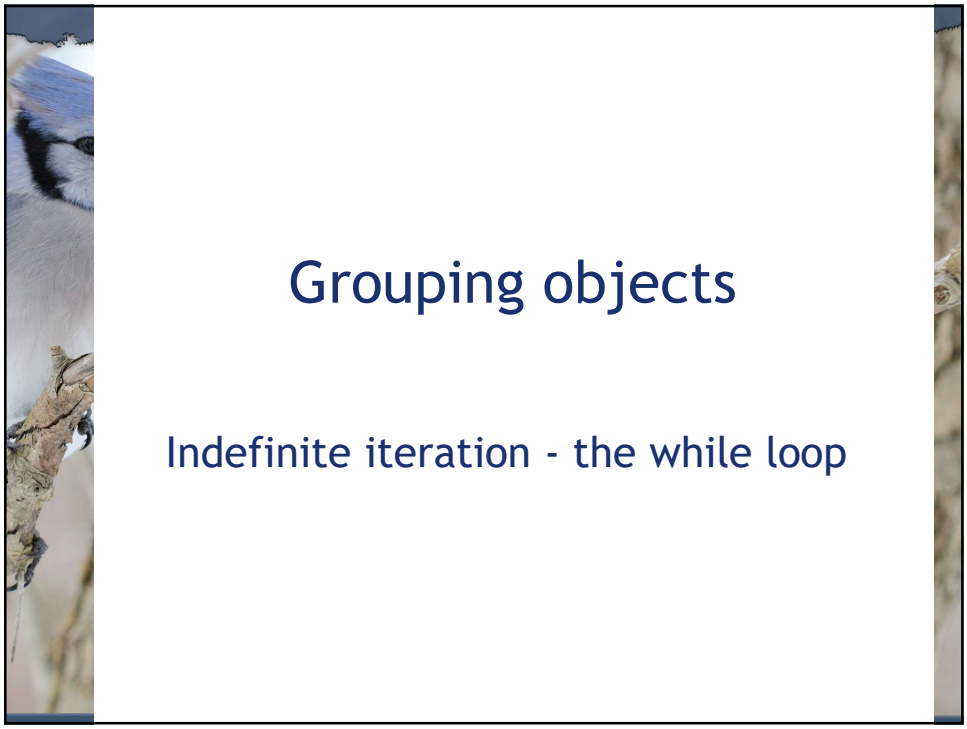
- Statements can be nested, giving greater selectivity to the actions:

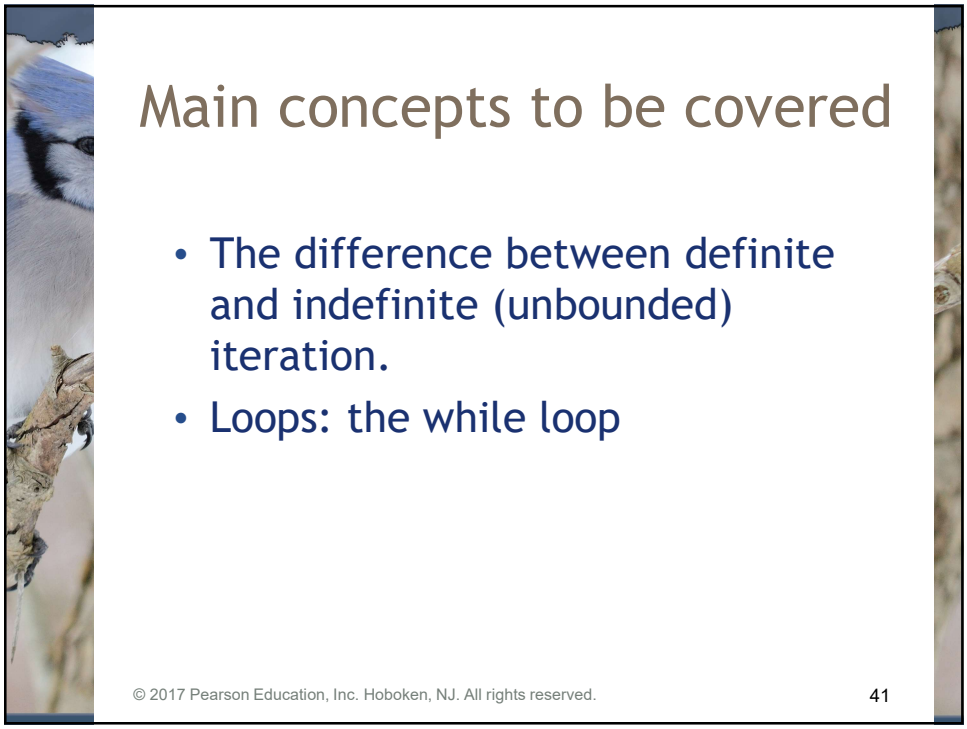
```
public void findFiles(String searchString)
{
    for(String filename : files) {
        if(filename.contains(searchString)) {
            System.out.println(filename);
        }
    }
}
```

`contains` gives a partial match of the filename; use `equals` for an exact match

 38


Critique of for-each
• Easy to write.
• Termination happens naturally.
• The collection cannot be changed by the actions.
• There is no index provided.
– Not all collections are index-based.
• We can't stop part way through;
– e.g., if we only want to find the first match.
• It provides 'definite iteration' – aka 'bounded iteration'.
© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.
39

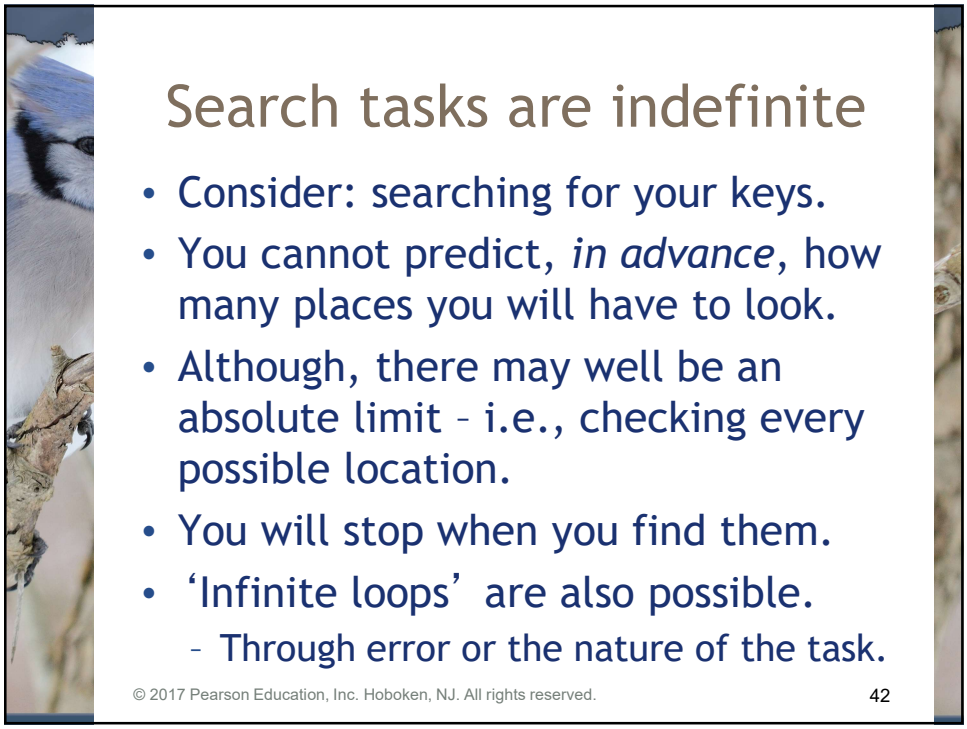

Grouping objects
Indefinite iteration - the while loop



Main concepts to be covered

- The difference between definite and indefinite (unbounded) iteration.
- Loops: the while loop

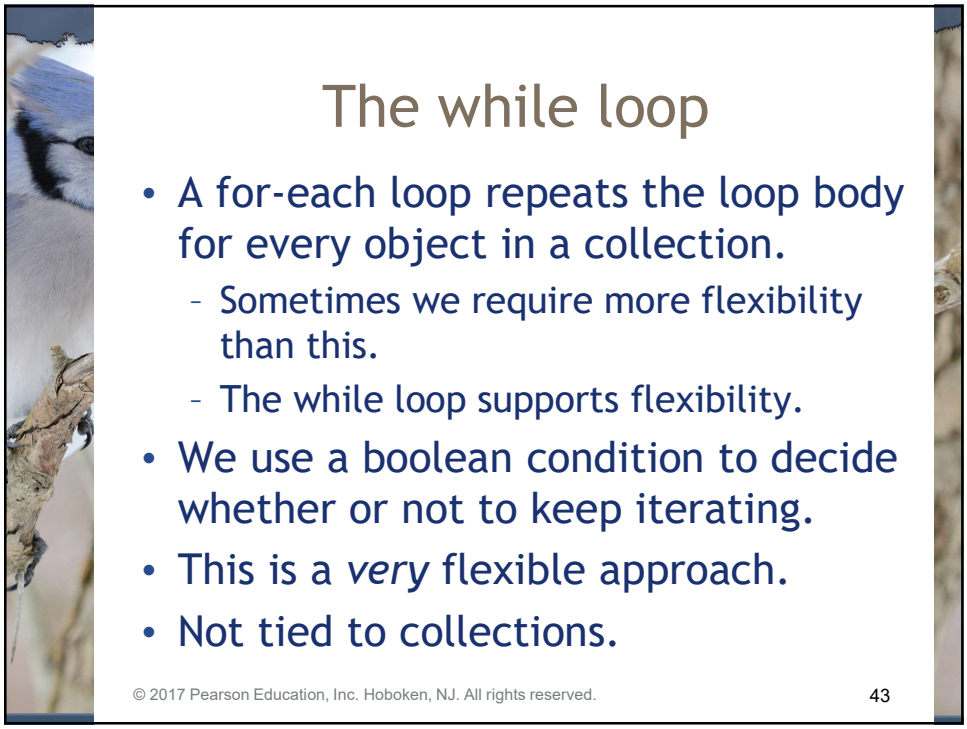
 41



Search tasks are indefinite

- Consider: searching for your keys.
- You cannot predict, *in advance*, how many places you will have to look.
- Although, there may well be an absolute limit – i.e., checking every possible location.
- You will stop when you find them.
- 'Infinite loops' are also possible.
  - Through error or the nature of the task.

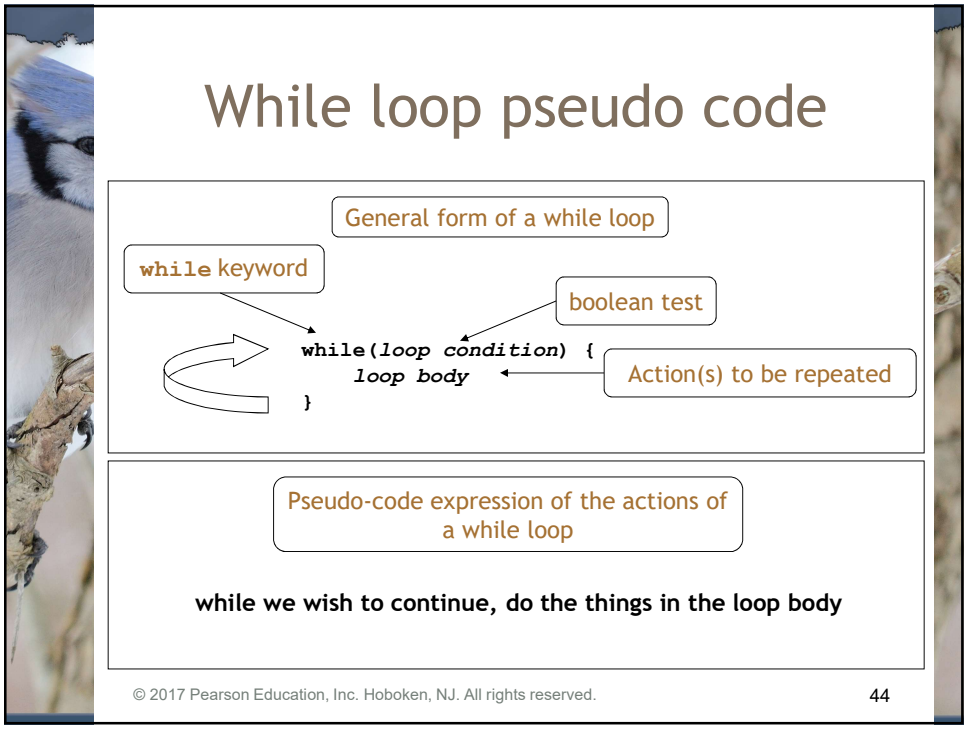
 42



The while loop

- A for-each loop repeats the loop body for every object in a collection.
  - Sometimes we require more flexibility than this.
  - The while loop supports flexibility.
- We use a boolean condition to decide whether or not to keep iterating.
- This is a *very* flexible approach.
- Not tied to collections.

 43



While loop pseudo code

General form of a while loop

`while` keyword

boolean test

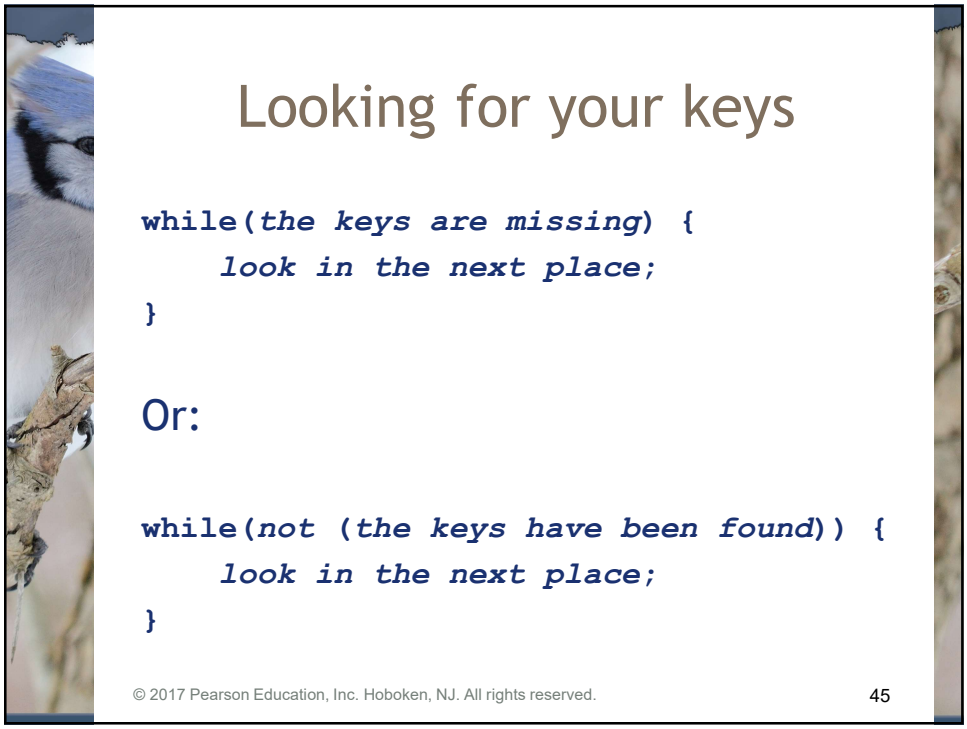
```
while(loop condition) {
    loop body
}
```

Action(s) to be repeated

Pseudo-code expression of the actions of a while loop

while we wish to continue, do the things in the loop body

 44



Looking for your keys

```
while(the keys are missing) {
    look in the next place;
}
```

Or:

```
while(not (the keys have been found)) {
    look in the next place;
}
```

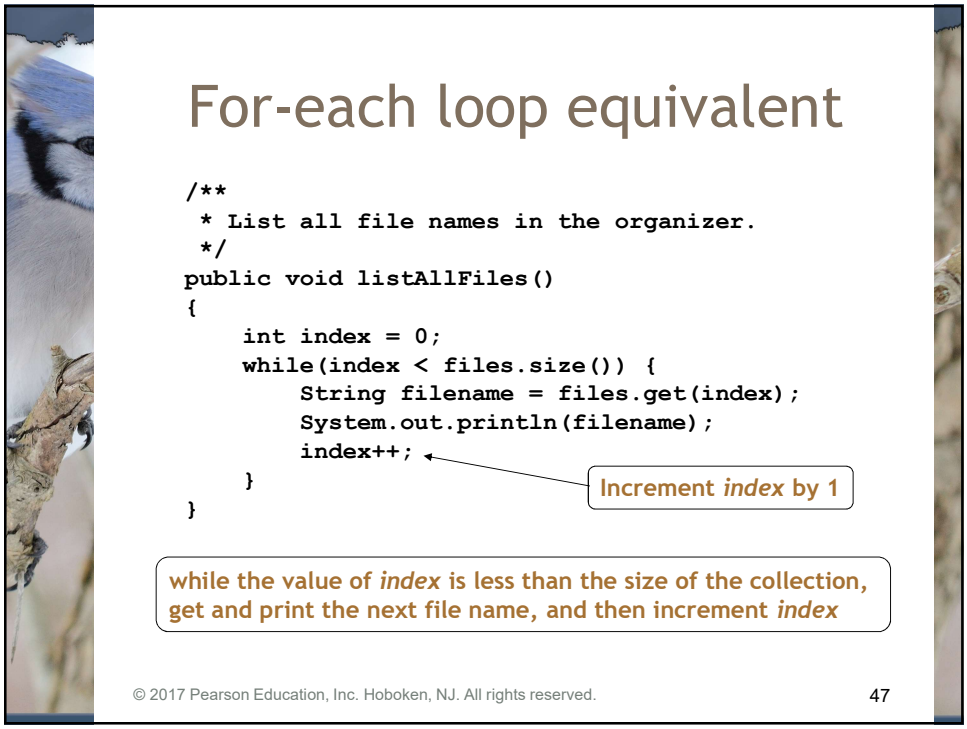
 45

Looking for your keys

```
boolean searching = true;
while(searching) {
    if(they are in the next place) {
        searching = false;
    }
}
```

Suppose we don't find them?

 46



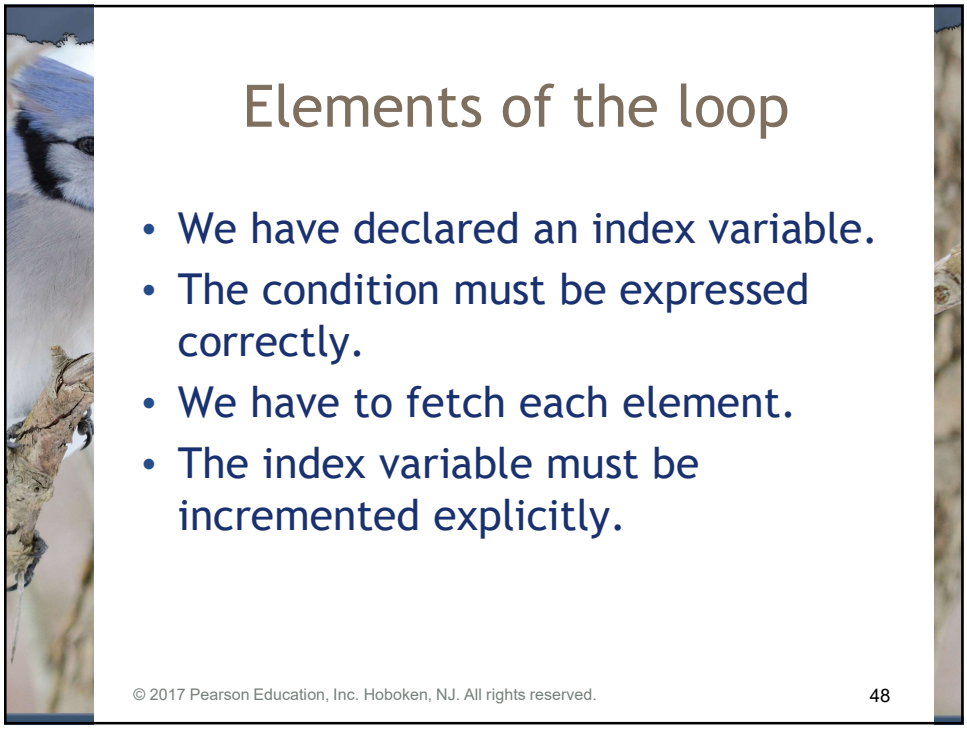
For-each loop equivalent

```
/**
 * List all file names in the organizer.
 */
public void listAllFiles()
{
    int index = 0;
    while(index < files.size()) {
        String filename = files.get(index);
        System.out.println(filename);
        index++;
    }
}
```

Increment *index* by 1

while the value of *index* is less than the size of the collection, get and print the next file name, and then increment *index*

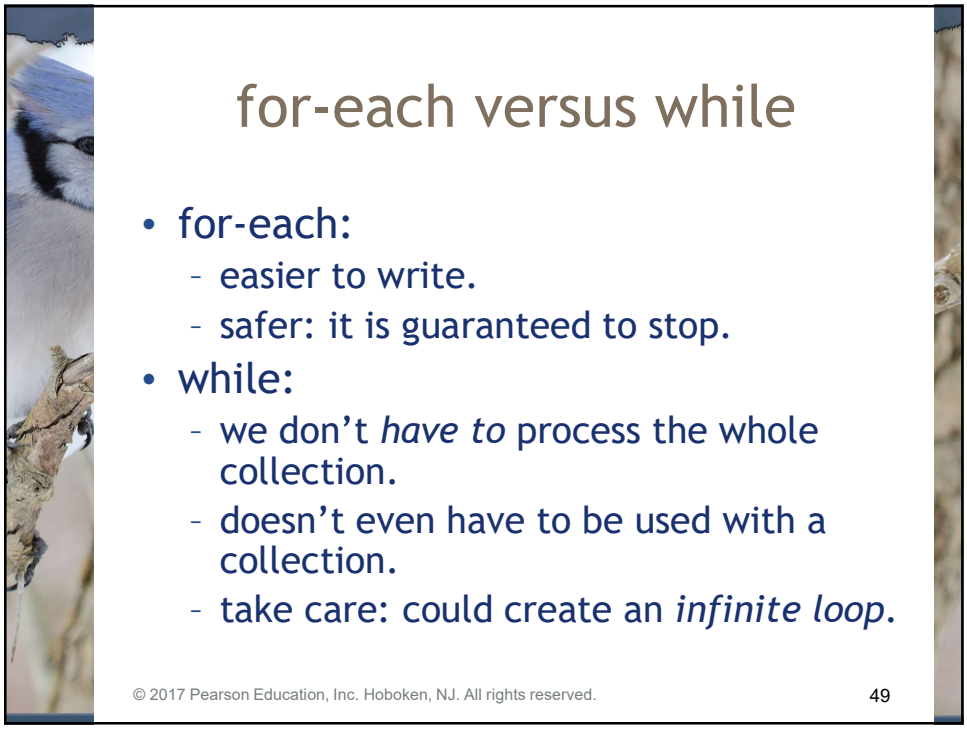
47



Elements of the loop

- We have declared an index variable.
- The condition must be expressed correctly.
- We have to fetch each element.
- The index variable must be incremented explicitly.

48



for-each versus while

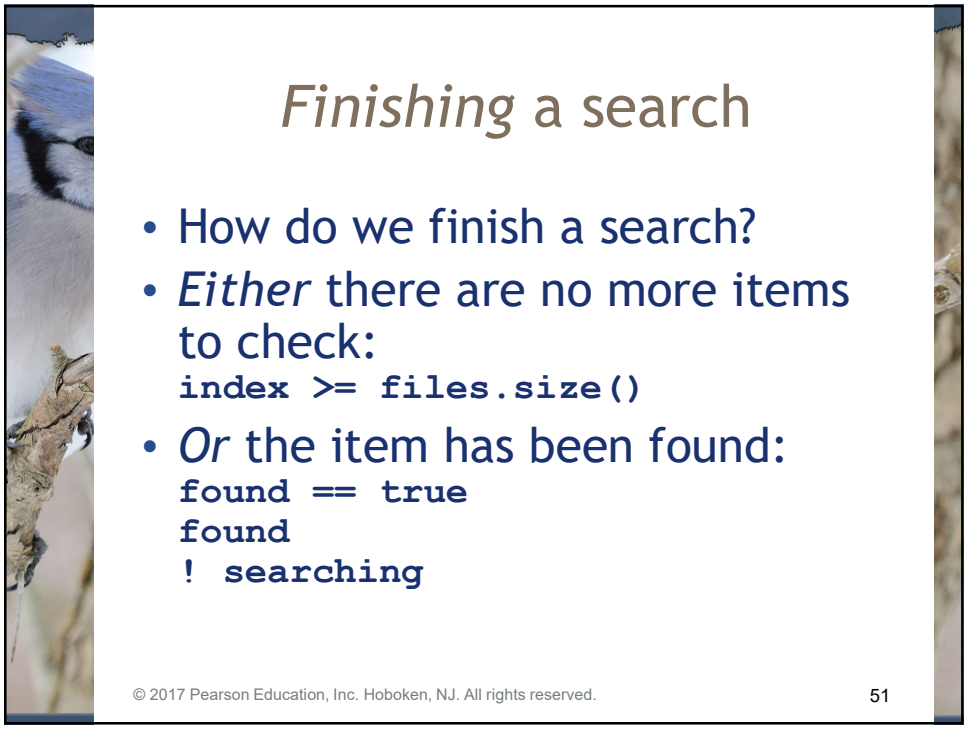
- for-each:
  - easier to write.
  - safer: it is guaranteed to stop.
- while:
  - we don't *have to* process the whole collection.
  - doesn't even have to be used with a collection.
  - take care: could create an *infinite loop*.

 49

Searching

- A fundamental activity.
- Applicable beyond collections.
- Necessarily indefinite.
- We must code for both success and failure – nowhere else to look.
- *Both* must make the loop's condition *false*, in order to stop the iteration.
- A collection might be empty to start with.

 50



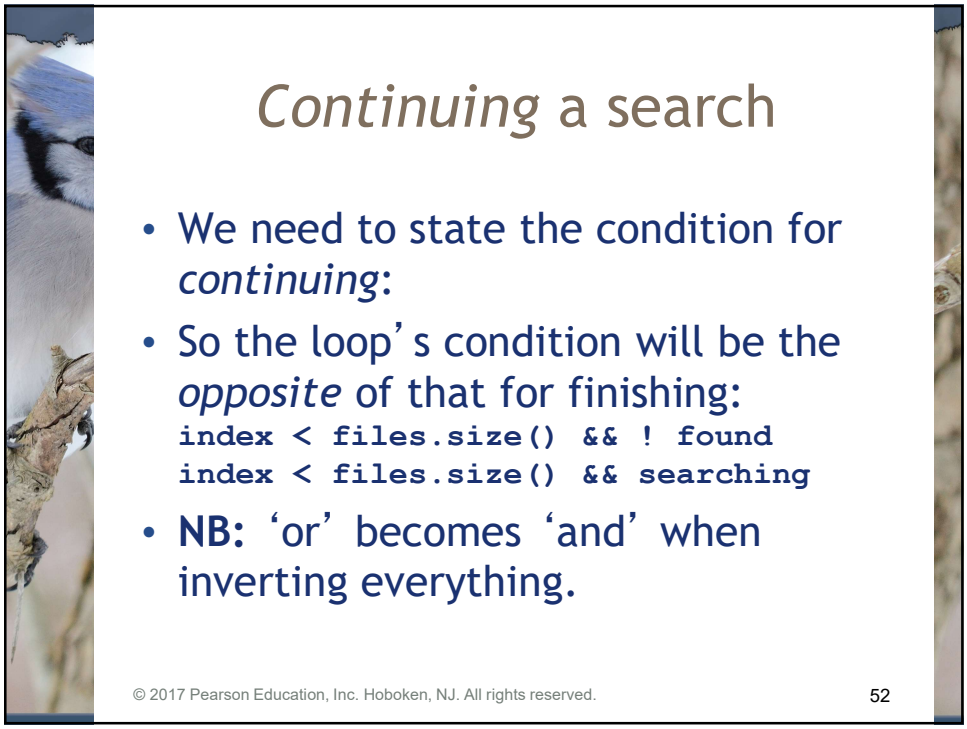
Finishing a search

- How do we finish a search?
- *Either* there are no more items to check:

```
index >= files.size()
```

- *Or* the item has been found:

```
found == true
found
! searching
```

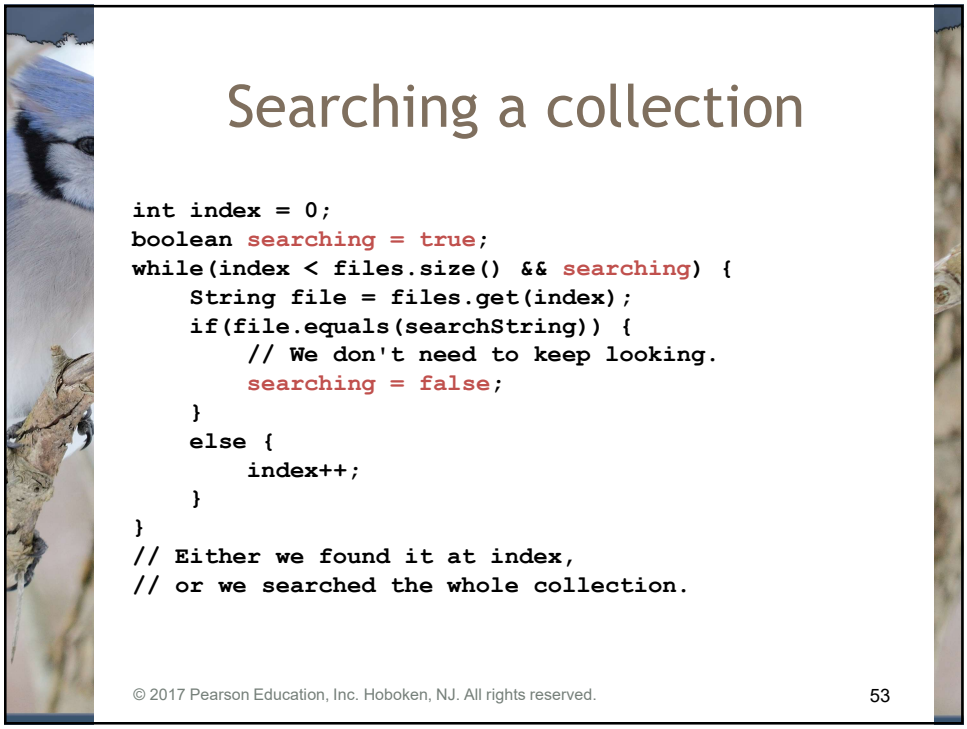



Continuing a search

- We need to state the condition for *continuing*:
- So the loop's condition will be the *opposite* of that for finishing:

```
index < files.size() && ! found
index < files.size() && searching
```

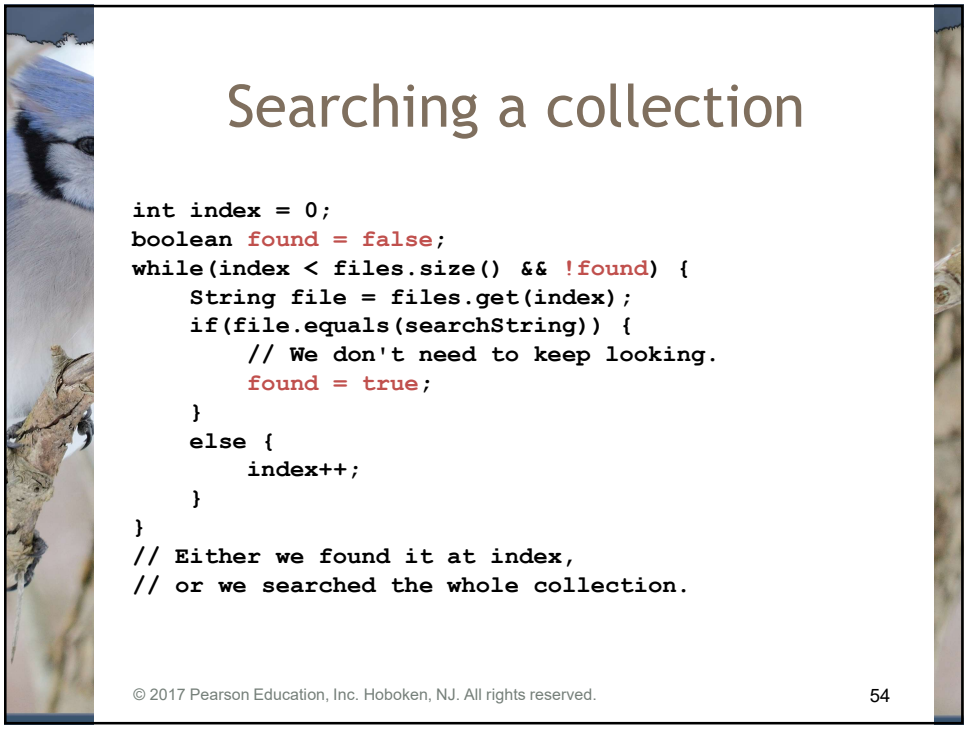
- **NB:** 'or' becomes 'and' when inverting everything.



Searching a collection

```
int index = 0;
boolean searching = true;
while(index < files.size() && searching) {
    String file = files.get(index);
    if(file.equals(searchString)) {
        // We don't need to keep looking.
        searching = false;
    }
    else {
        index++;
    }
}
// Either we found it at index,
// or we searched the whole collection.
```

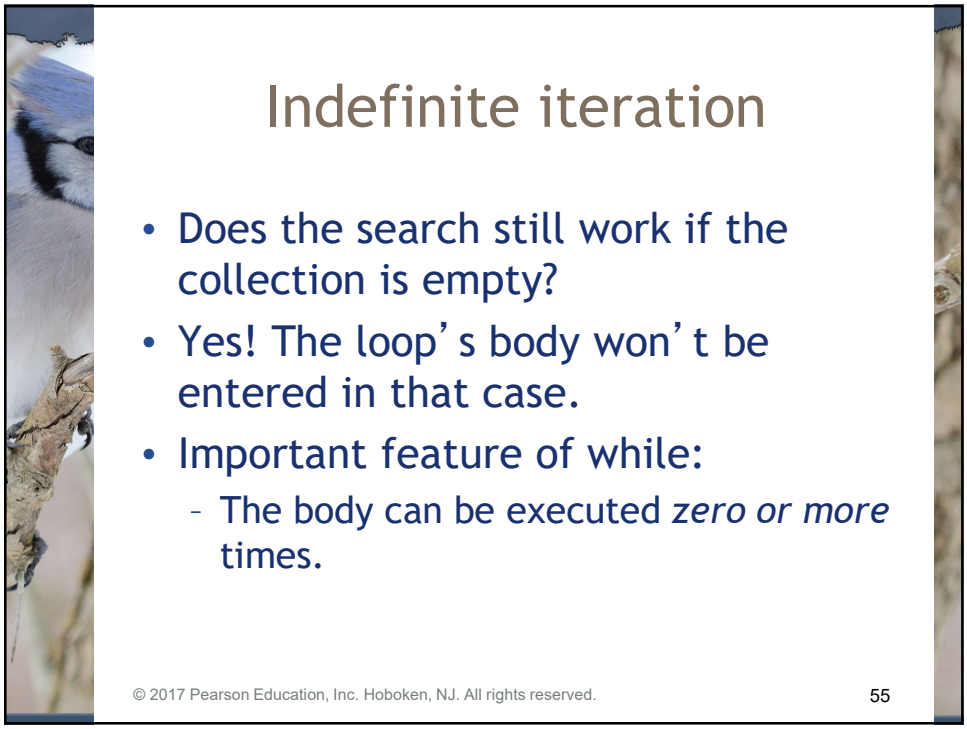

53



Searching a collection

```
int index = 0;
boolean found = false;
while(index < files.size() && !found) {
    String file = files.get(index);
    if(file.equals(searchString)) {
        // We don't need to keep looking.
        found = true;
    }
    else {
        index++;
    }
}
// Either we found it at index,
// or we searched the whole collection.
```

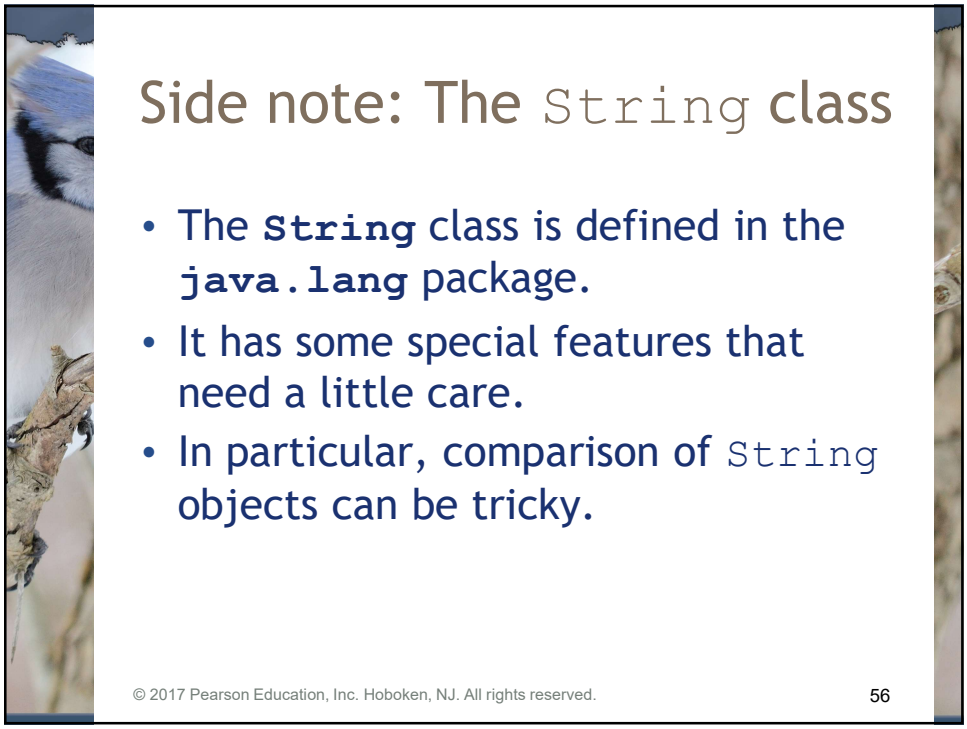

54



Indefinite iteration

- Does the search still work if the collection is empty?
- Yes! The loop's body won't be entered in that case.
- Important feature of while:
  - The body can be executed *zero or more* times.

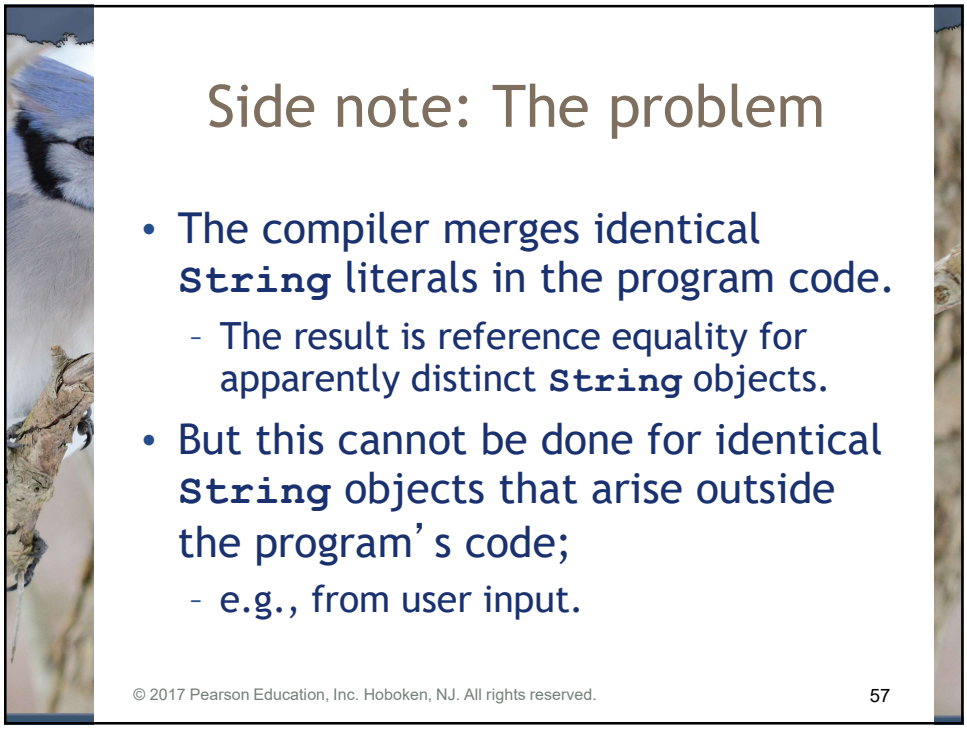
 55



Side note: The `String` class

- The **`String`** class is defined in the **`java.lang`** package.
- It has some special features that need a little care.
- In particular, comparison of `String` objects can be tricky.

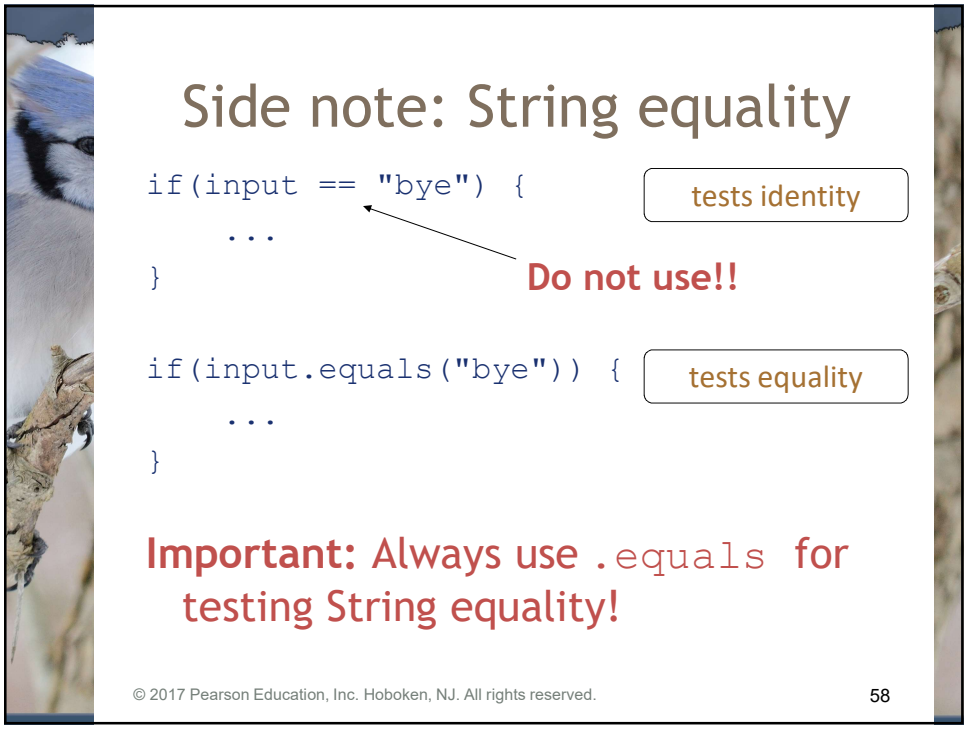
 56



Side note: The problem

- The compiler merges identical **`String`** literals in the program code.
  - The result is reference equality for apparently distinct **`String`** objects.
- But this cannot be done for identical **`String`** objects that arise outside the program's code;
  - e.g., from user input.

 57



Side note: String equality

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

tests identity

Do not use!!

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

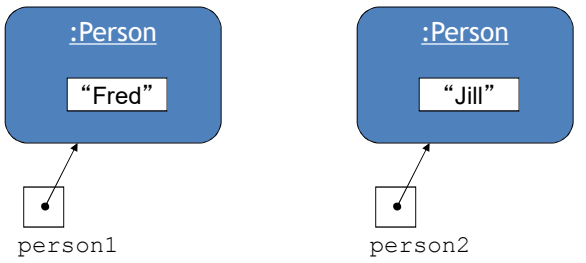
tests equality

Important: Always use `.equals` for testing String equality!

 58

Identity vs equality 1

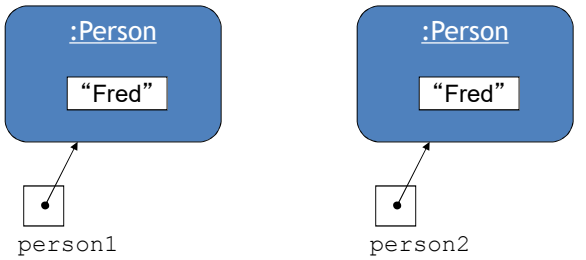
Other (non-String) objects:




```
person1 == person2 ?
```


Identity vs equality 2

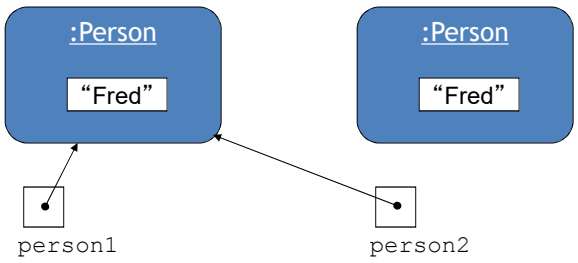
Other (non-String) objects:




```
person1 == person2 ?
```

Identity vs equality 3

Other (non-String) objects:

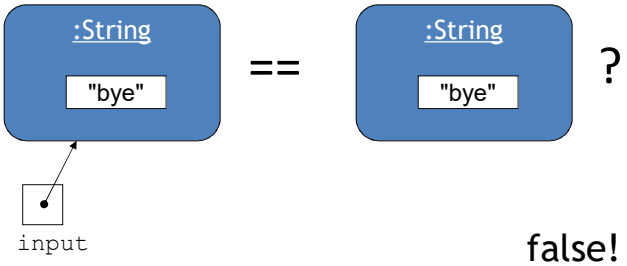



```
person1 == person2 ?
```


Identity vs equality (Strings)

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

== tests identity

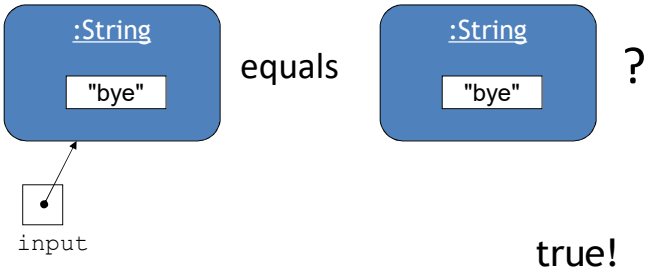


false!

Identity vs equality (Strings)

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

equals tests equality



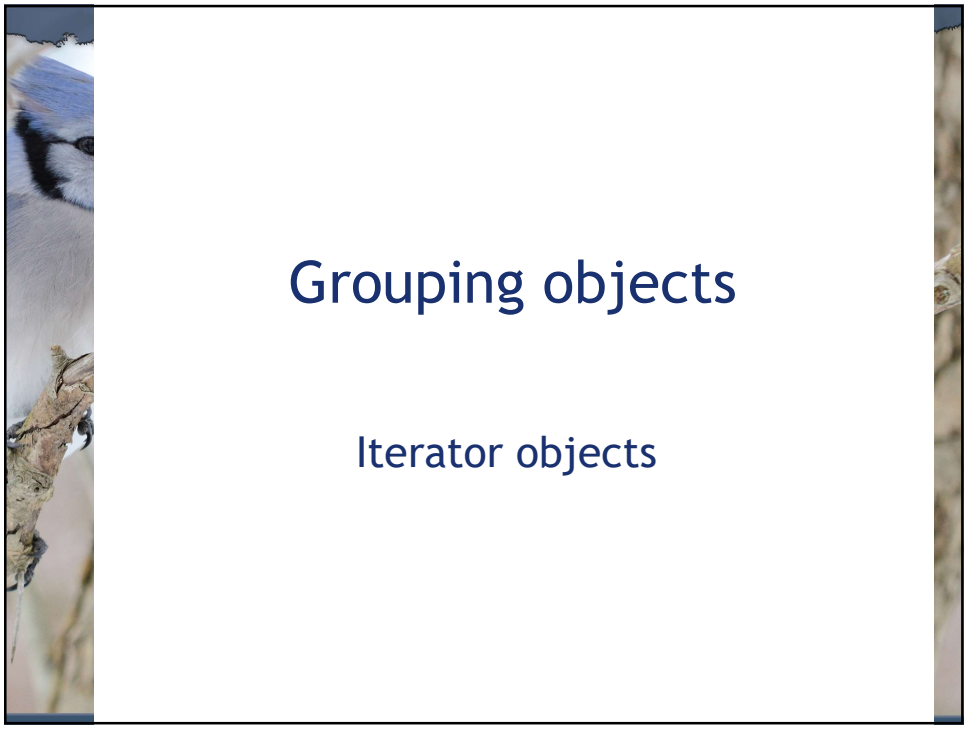
true!

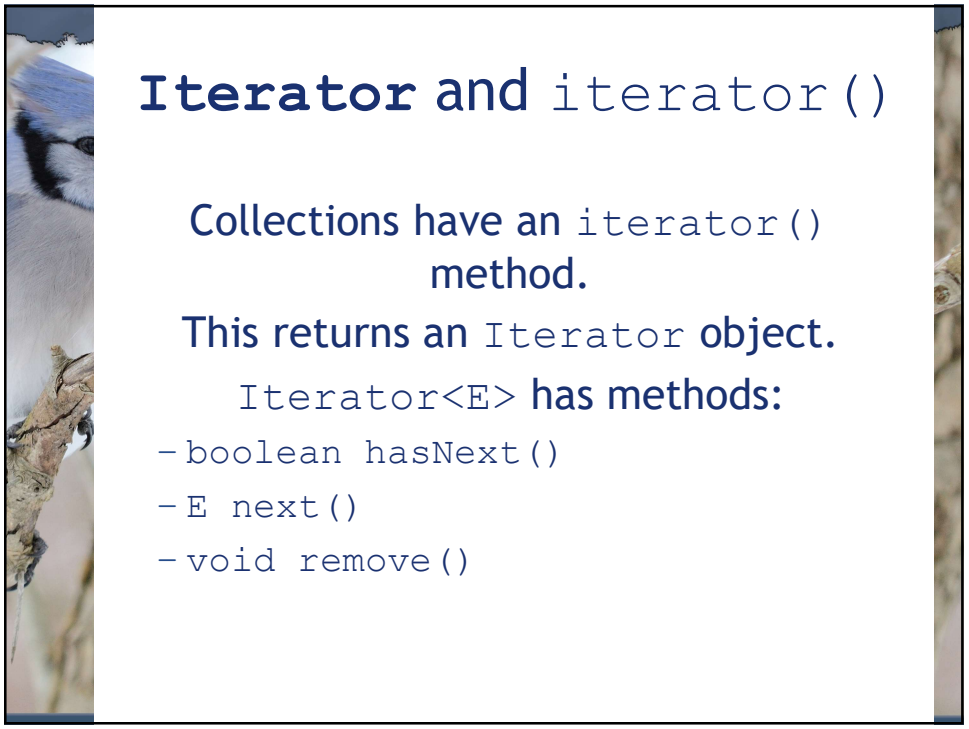
While without a collection

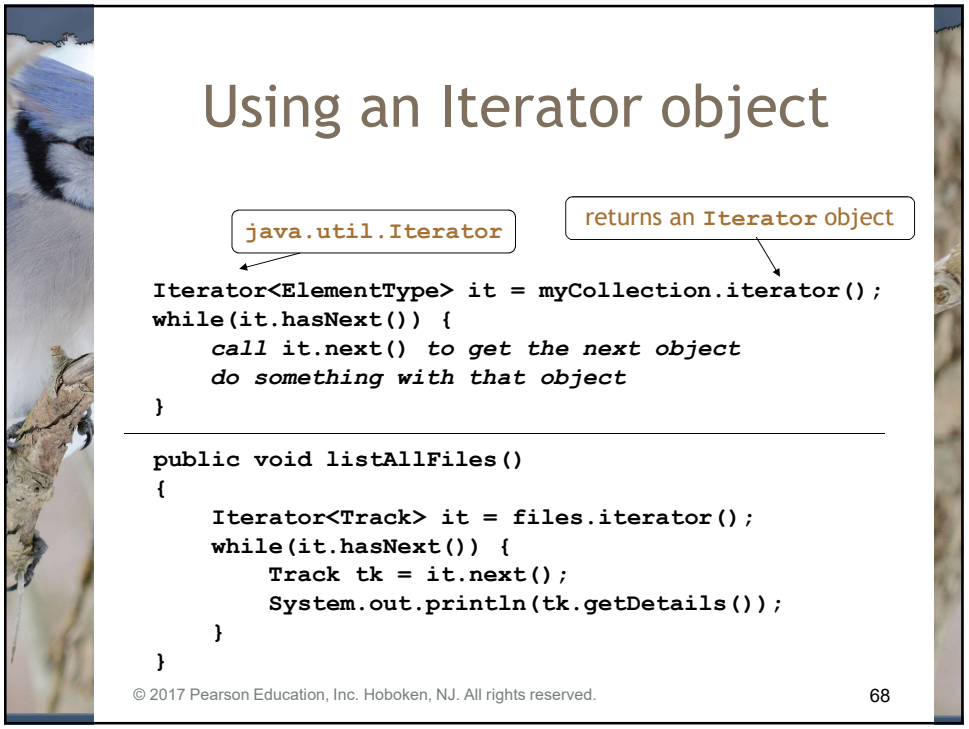
```
// Print all even numbers from 2 to 30.
int index = 2;
while(index <= 30) {
    System.out.println(index);
    index = index + 2;
}
```

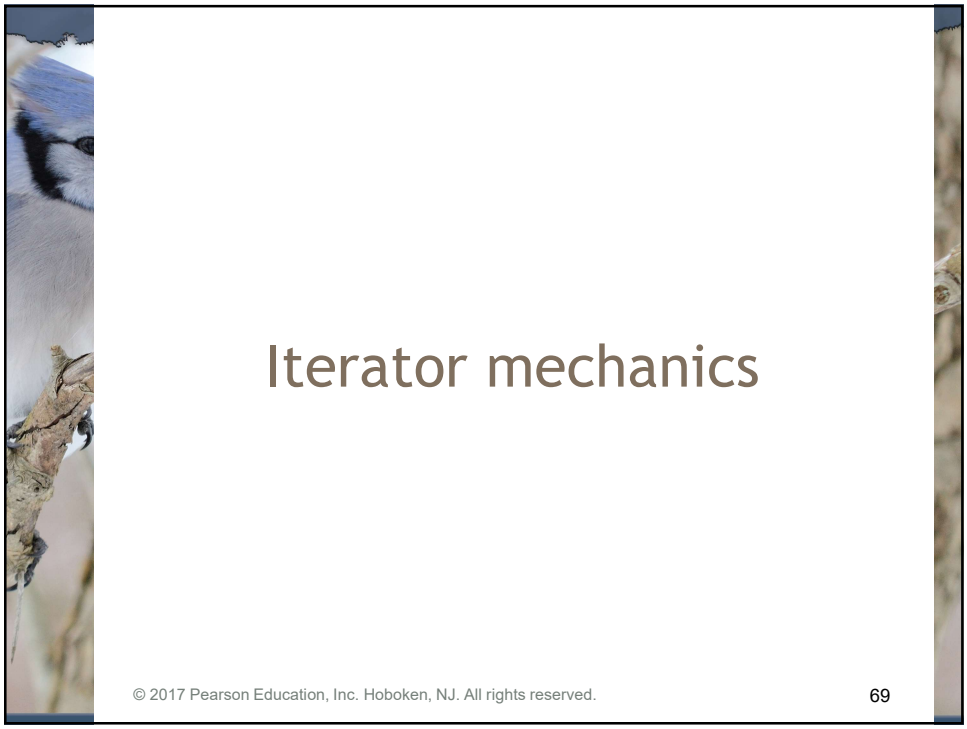
Any **boolean expression can be used to control a while loop.**

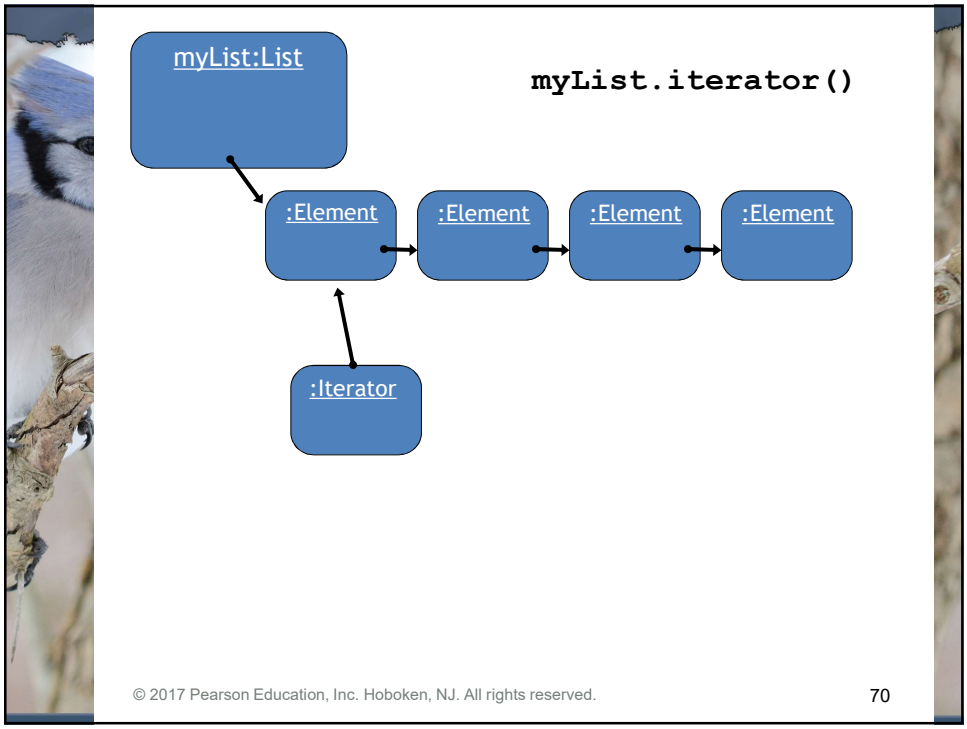

Moving away from String
• Our collection of String objects for music tracks is limited.
• No separate identification of artist, title, etc.
• A Track class with separate fields:
– artist
– title
– filename
© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.
65

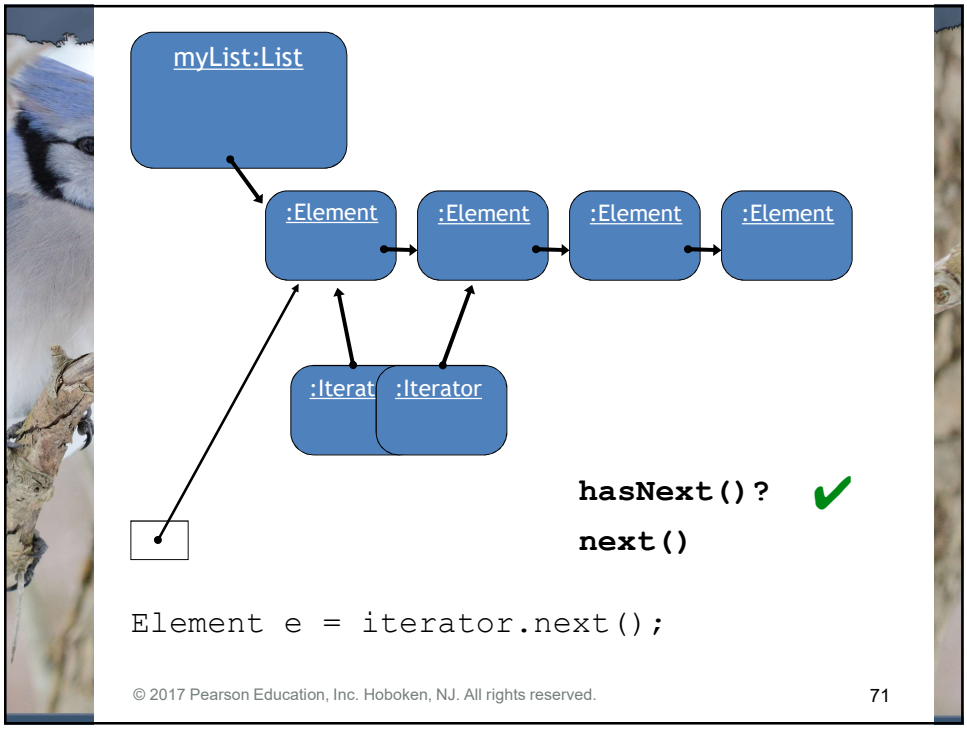

Grouping objects
Iterator objects

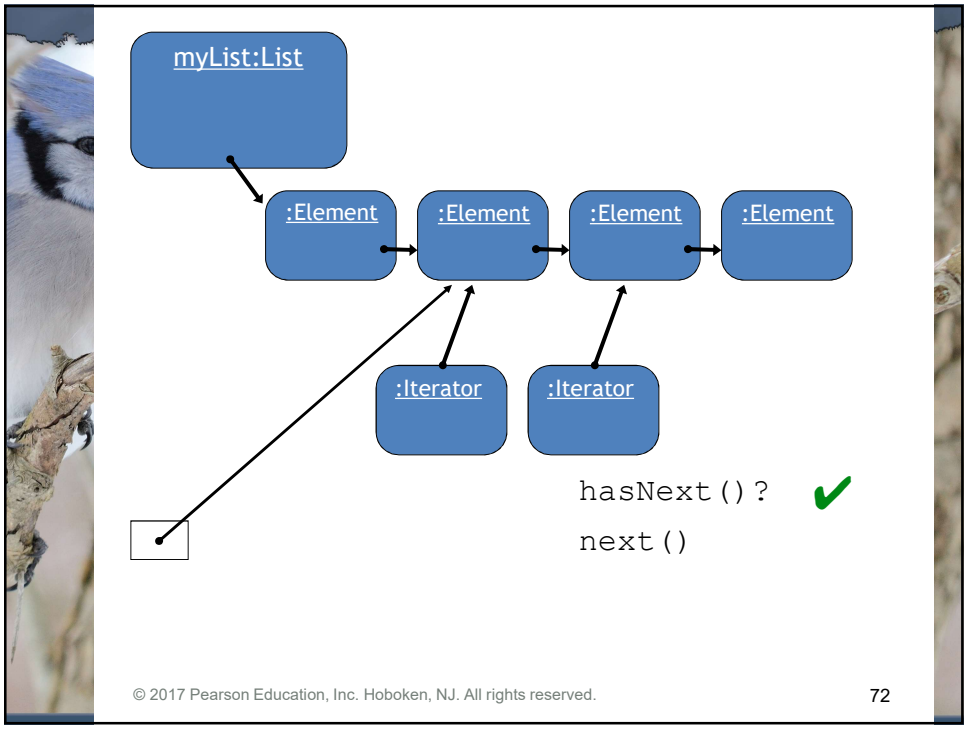

Iterator and iterator()
Collections have an iterator() method.
This returns an Iterator object.
Iterator<E> has methods:
– boolean hasNext()
– E next()
– void remove()

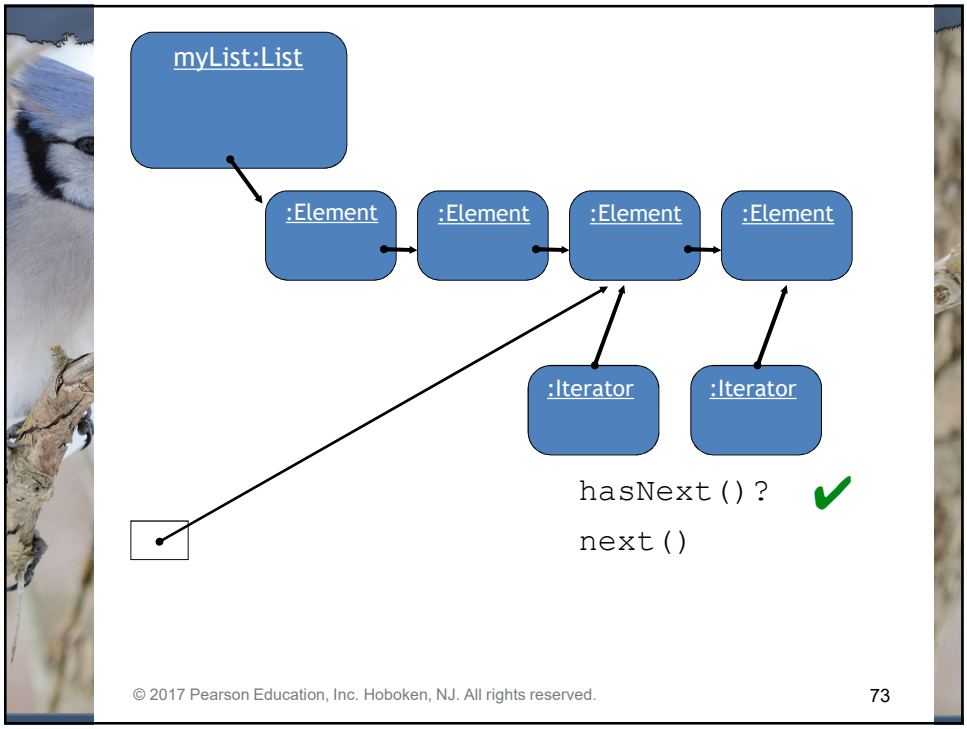

Using an Iterator object
java.util.Iterator
returns an Iterator object
Iterator<ElementType> it = myCollection.iterator();
while(it.hasNext()) {
call it.next() to get the next object
do something with that object
}
public void listAllFiles()
{
Iterator<Track> it = files.iterator();
while(it.hasNext()) {
Track tk = it.next();
System.out.println(tk.getDetails());
}
}
© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.
68

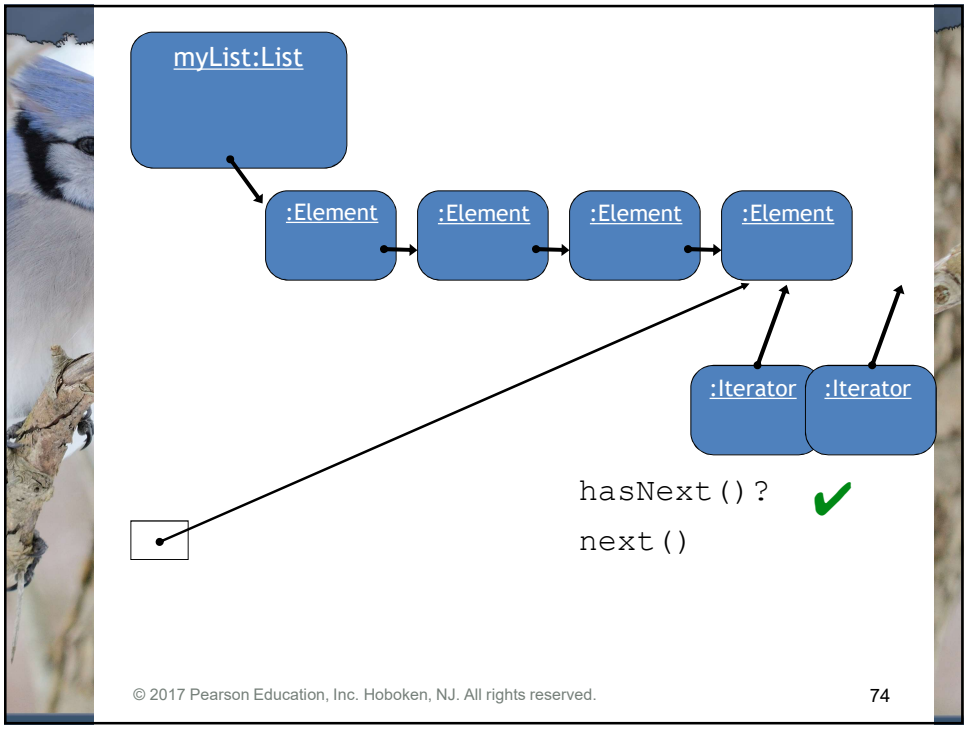

Iterator mechanics
© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.
69

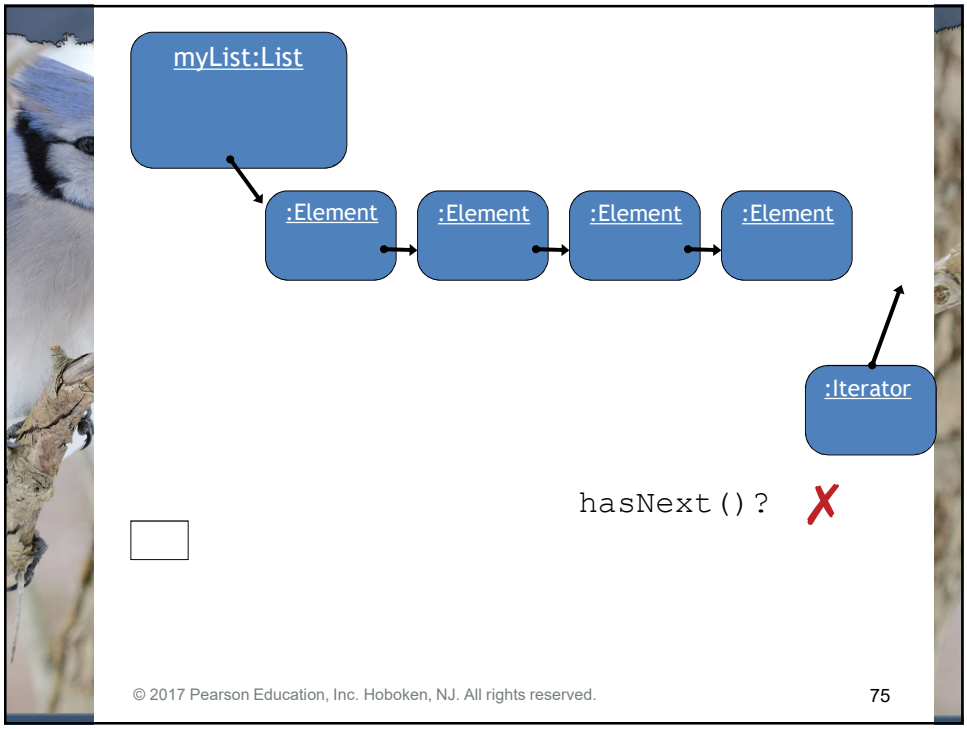

myList:List
myList.iterator()
:Element
:Element
:Element
:Element
:Iterator
© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.
70


myList:List
:Element
:Element
:Element
:Element
:Iterat
:Iterator
hasNext()?
next()
Element e = iterator.next();
© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.
71


myList:List
:Element
:Element
:Element
:Element
:Iterator
:Iterator
hasNext()?
next()
© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.
72


myList:List
:Element
:Element
:Element
:Element
:Iterator
:Iterator
hasNext()?
next()
© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.
73


myList:List
:Element
:Element
:Element
:Element
:Iterator
:Iterator
hasNext()?
next()
© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.
74

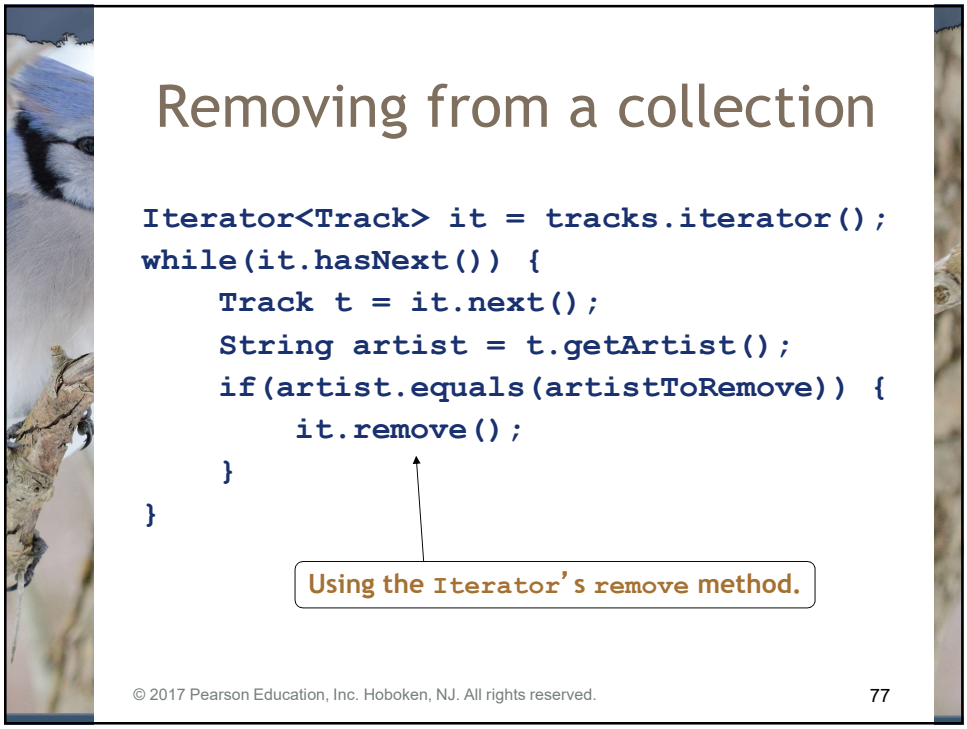


 75

Index versus Iterator

- Ways to iterate over a collection:
  - for-each loop.
    - Use if we want to process every element.
  - while loop.
    - Use if we might want to stop part way through.
    - Use for repetition that doesn't involve a collection.
  - **`Iterator`** object.
    - Use if we might want to stop part way through.
    - Often used with collections where indexed access is not very efficient, or impossible.
    - *Use to remove from a collection.*
- Iteration is an important programming *pattern*.

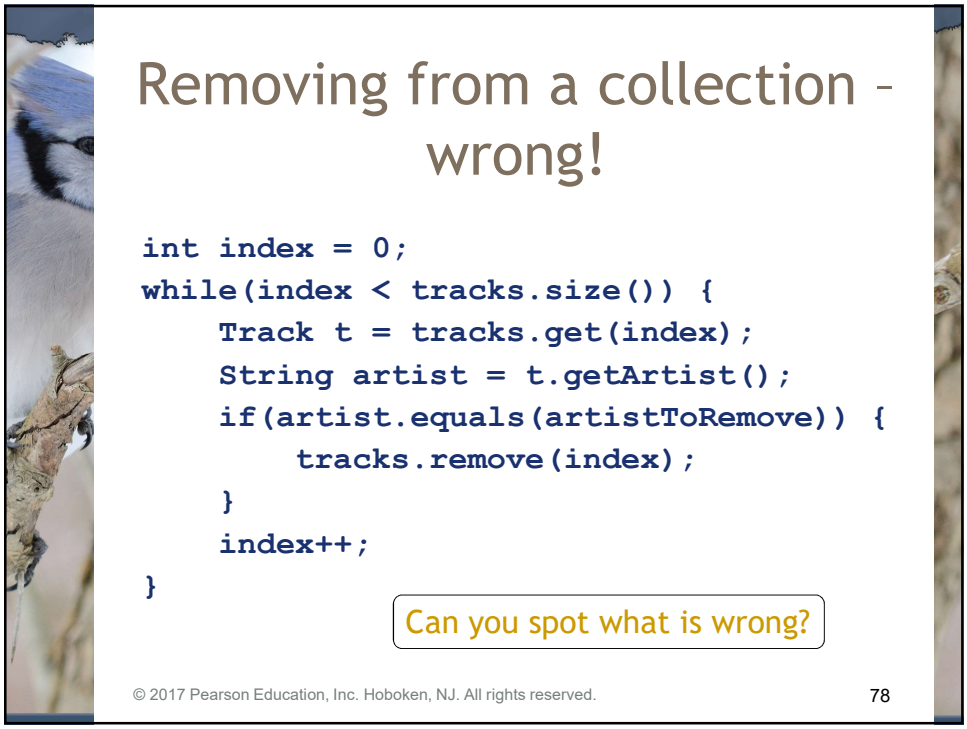
 76



Removing from a collection

```
Iterator<Track> it = tracks.iterator();
while(it.hasNext()) {
    Track t = it.next();
    String artist = t.getArtist();
    if(artist.equals(artistToRemove)) {
        it.remove();
    }
}
```

Using the `Iterator`'s `remove` method.



Removing from a collection – wrong!

```
int index = 0;
while(index < tracks.size()) {
    Track t = tracks.get(index);
    String artist = t.getArtist();
    if(artist.equals(artistToRemove)) {
        tracks.remove(index);
    }
    index++;
}
```

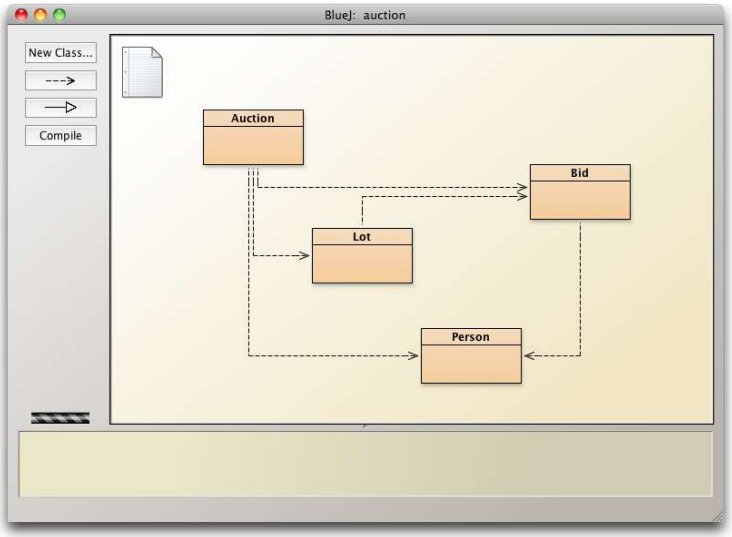
Can you spot what is wrong?

Review

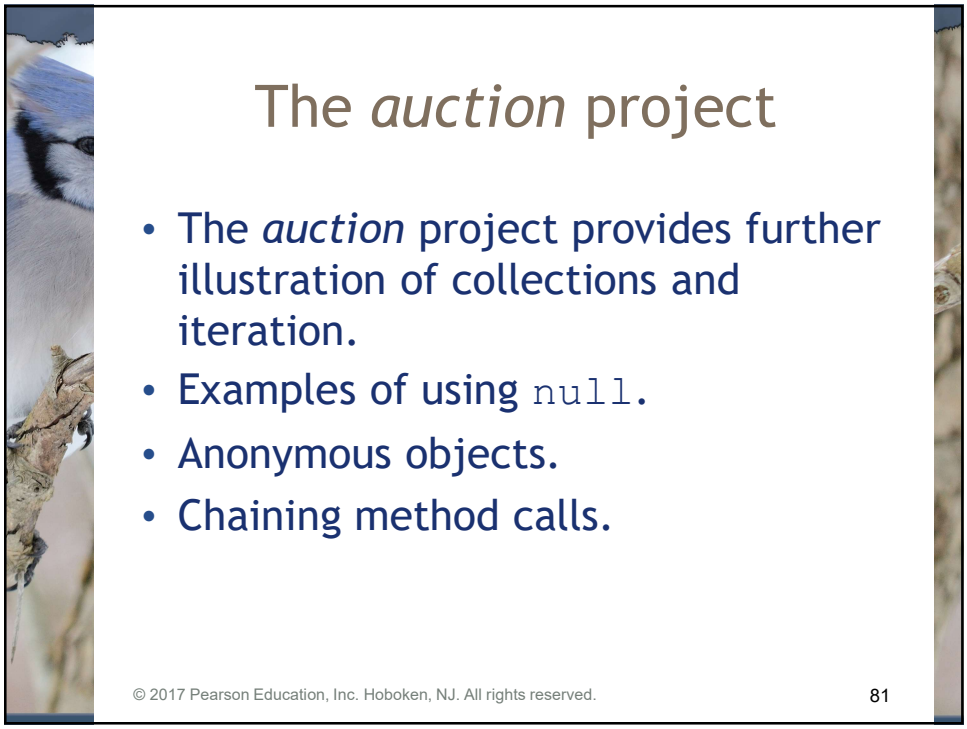
- Loop statements allow a block of statements to be repeated.
- The for-each loop allows iteration over a whole collection.
- The while loop allows the repetition to be controlled by a boolean expression.
- All collection classes provide special `Iterator` objects that provide sequential access to a whole collection.

 79

The auction project



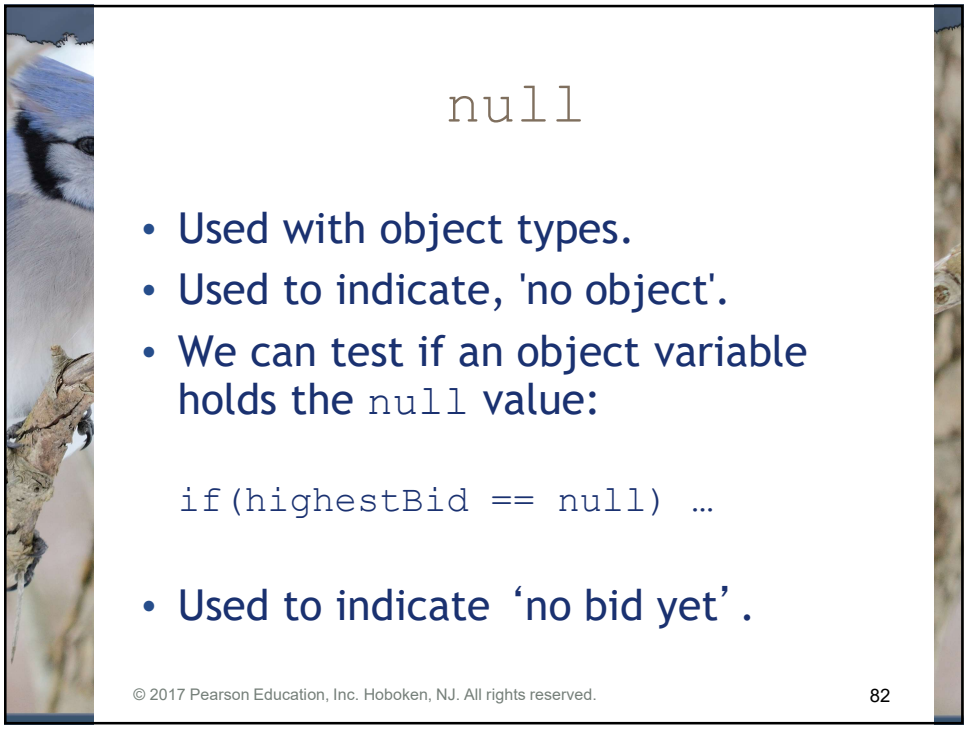
 80



The *auction* project

- The *auction* project provides further illustration of collections and iteration.
- Examples of using `null`.
- Anonymous objects.
- Chaining method calls.

 81



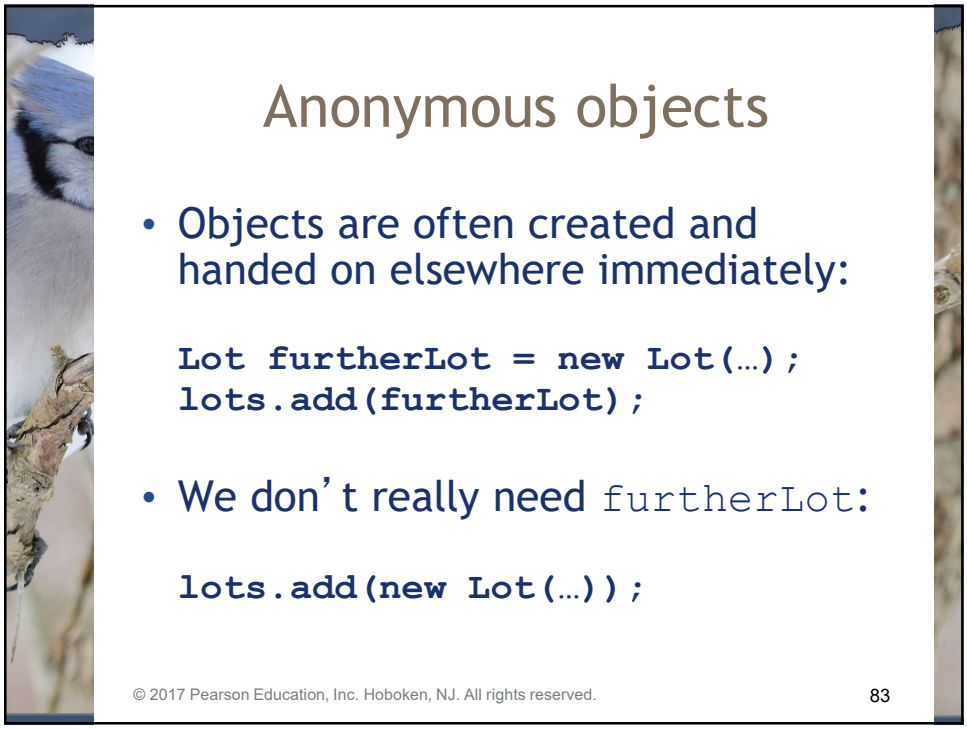
`null`

- Used with object types.
- Used to indicate, 'no object'.
- We can test if an object variable holds the `null` value:

```
if(highestBid == null) …
```

- Used to indicate ‘no bid yet’.

 82



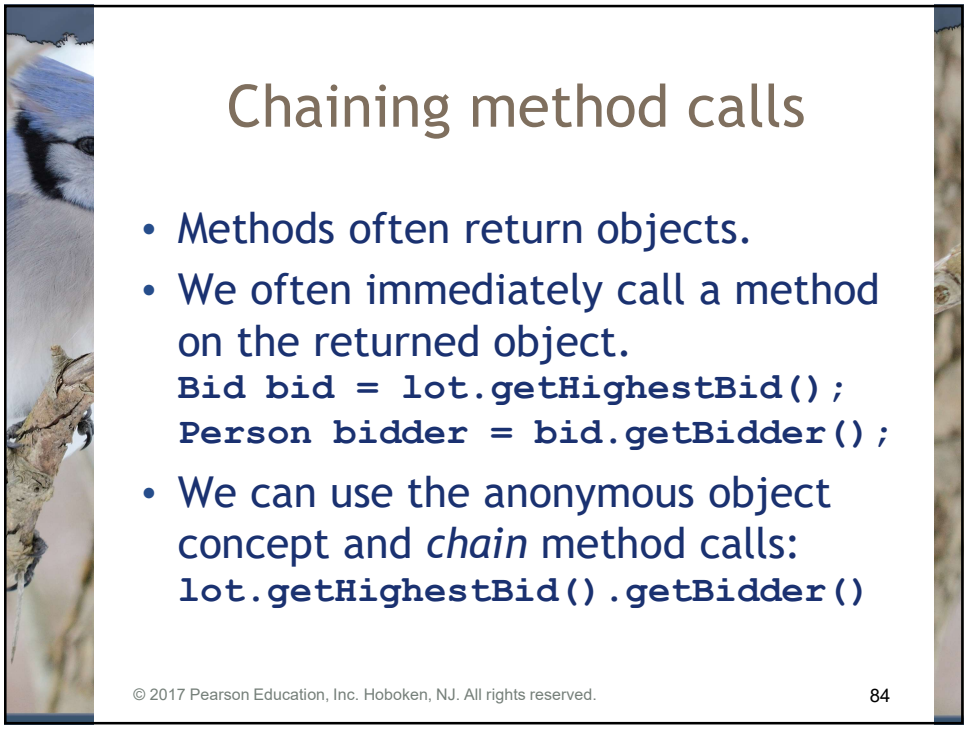
Anonymous objects

- Objects are often created and handed on elsewhere immediately:

```
Lot furtherLot = new Lot(…);
lots.add(furtherLot);
```

- We don't really need `furtherLot`:

```
lots.add(new Lot(…));
```

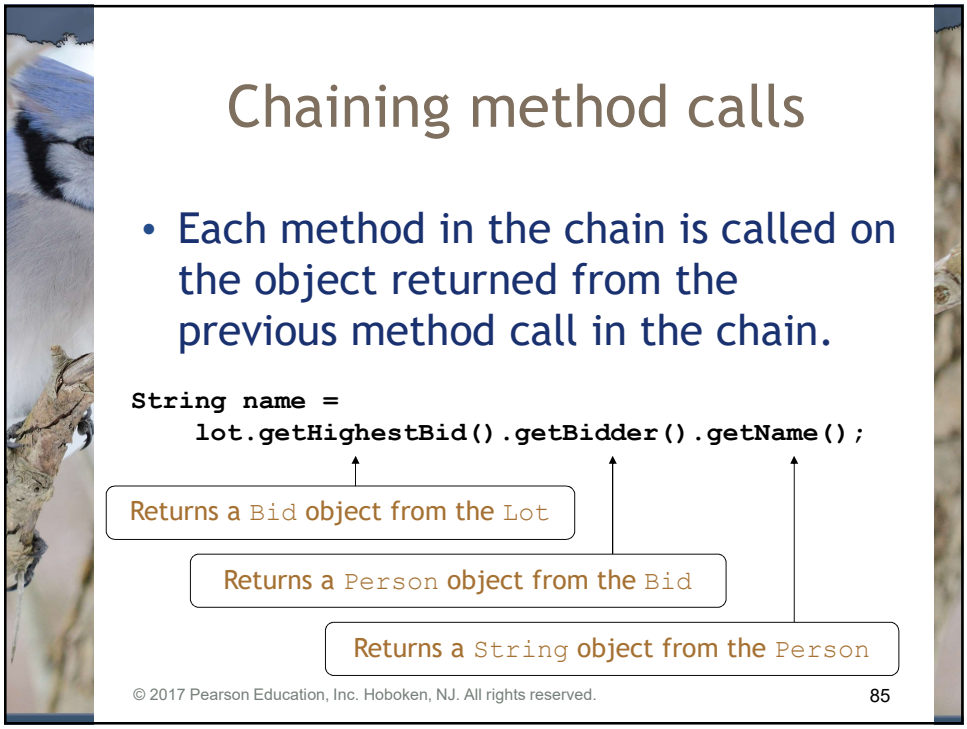
Chaining method calls

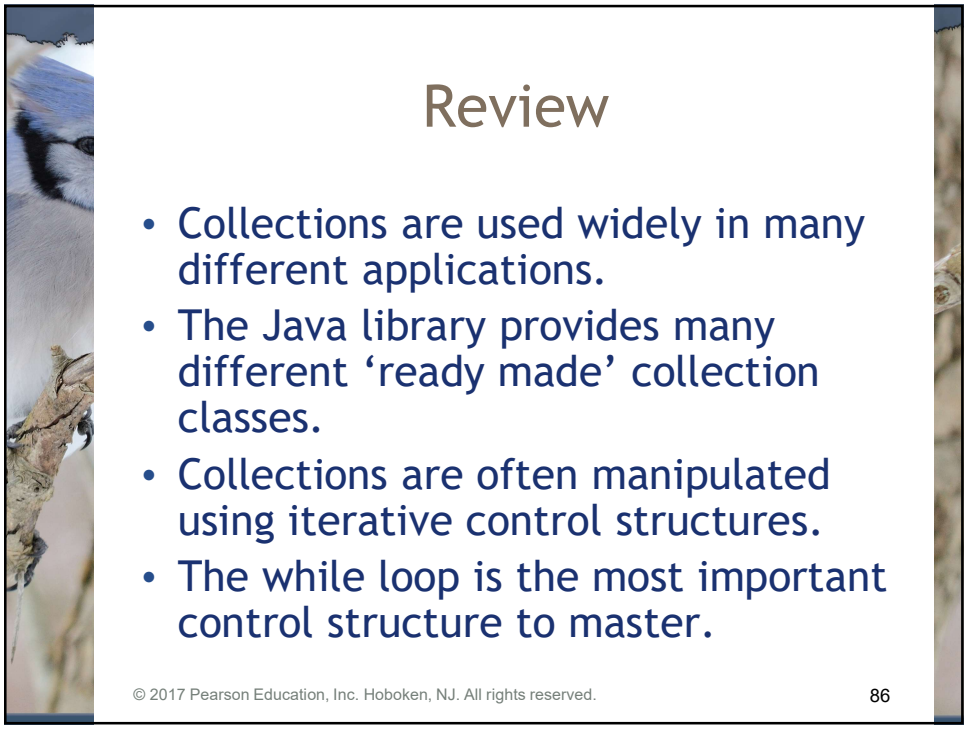
- Methods often return objects.
- We often immediately call a method on the returned object.

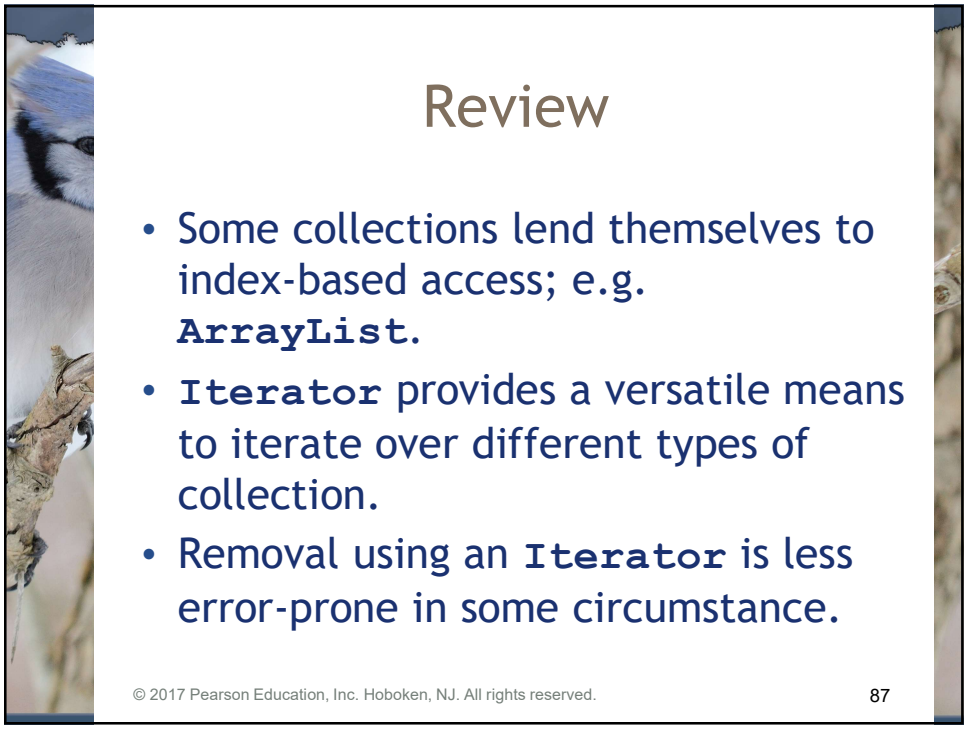
```
Bid bid = lot.getHighestBid();
Person bidder = bid.getBidder();
```

- We can use the anonymous object concept and *chain* method calls:

```
lot.getHighestBid().getBidder()
```


Chaining method calls
• Each method in the chain is called on the object returned from the previous method call in the chain.
String name =
lot.getHighestBid().getBidder().getName();
Returns a Bid object from the Lot
Returns a Person object from the Bid
Returns a String object from the Person
© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.
85


Review
• Collections are used widely in many different applications.
• The Java library provides many different 'ready made' collection classes.
• Collections are often manipulated using iterative control structures.
• The while loop is the most important control structure to master.
© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.
86


Review
• Some collections lend themselves to index-based access; e.g. ArrayList.
• Iterator provides a versatile means to iterate over different types of collection.
• Removal using an Iterator is less error-prone in some circumstance.
© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.
87