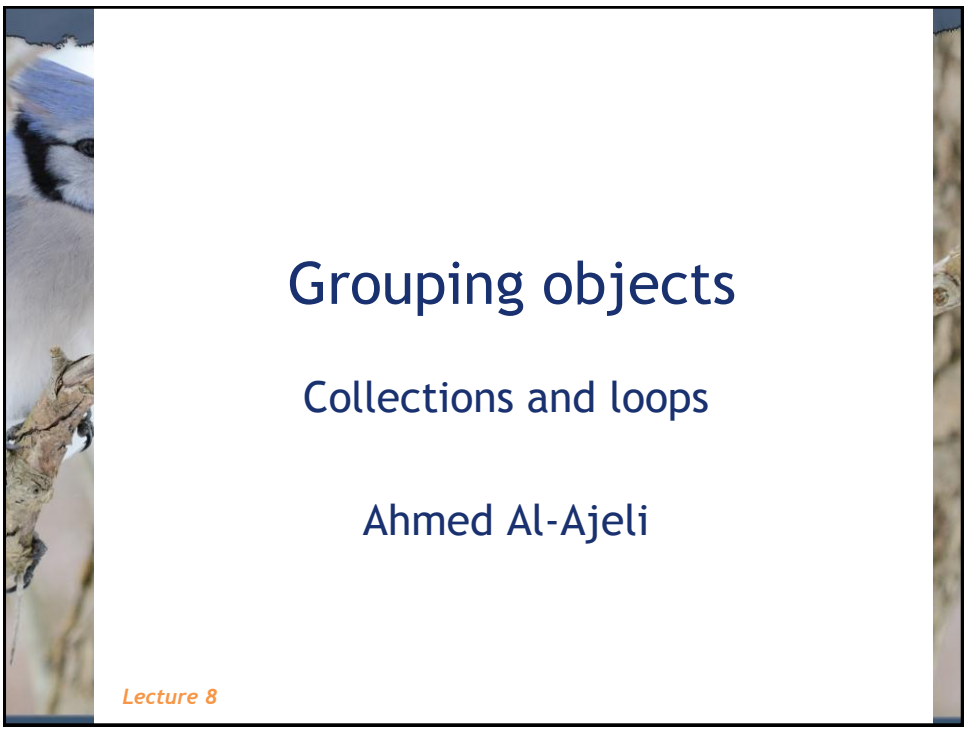
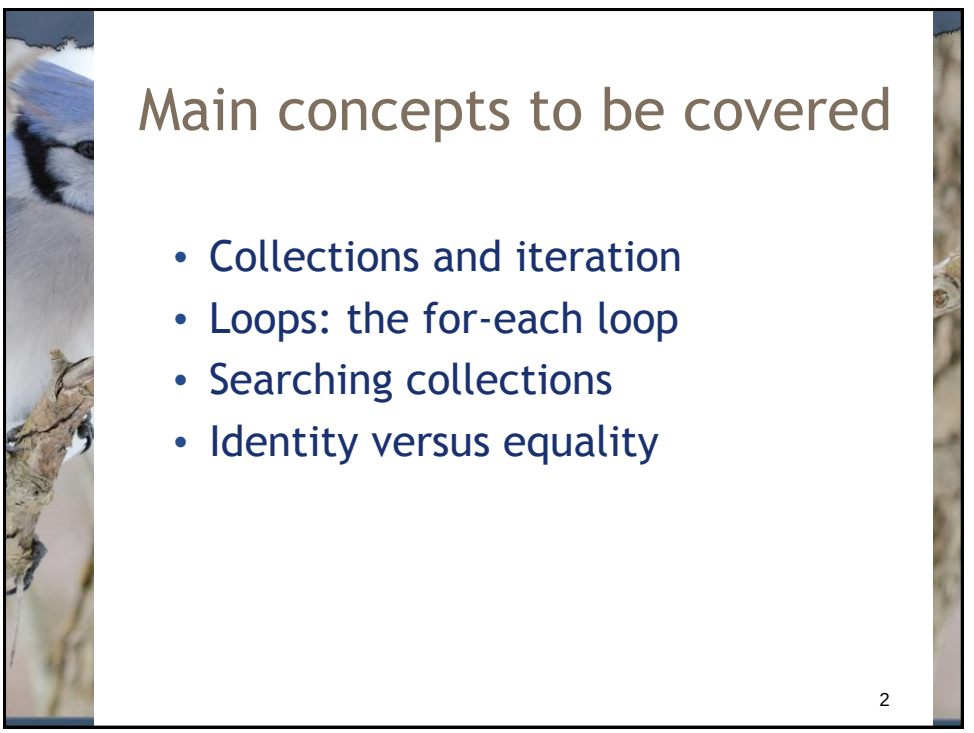


Grouping objects

Collections and loops

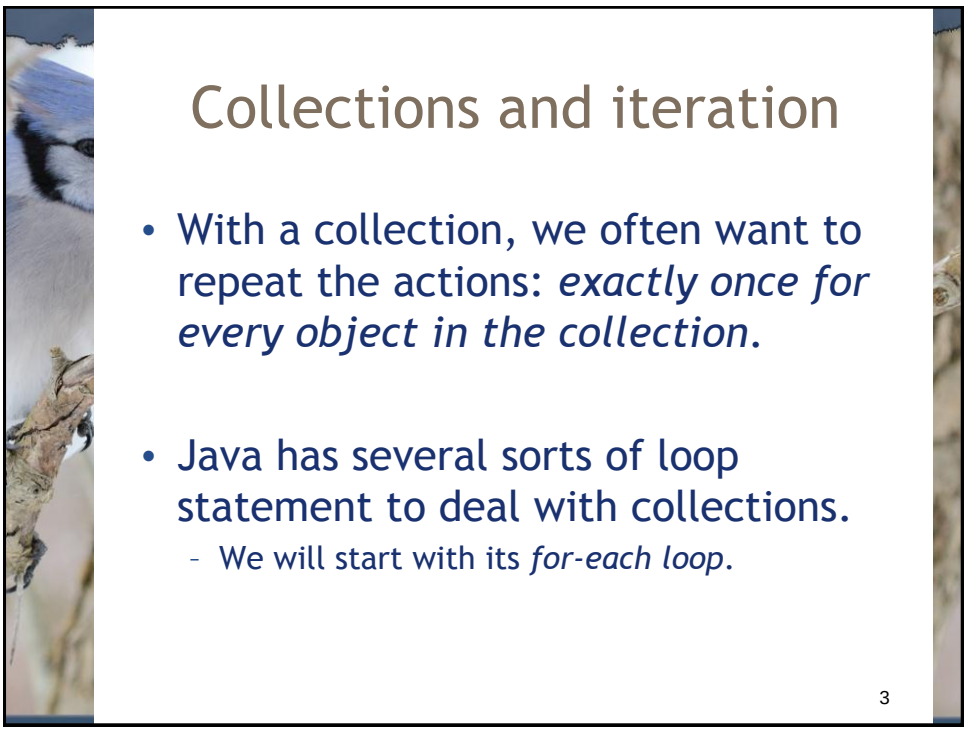
Ahmed Al-Ajeli

Lecture 8



Main concepts to be covered

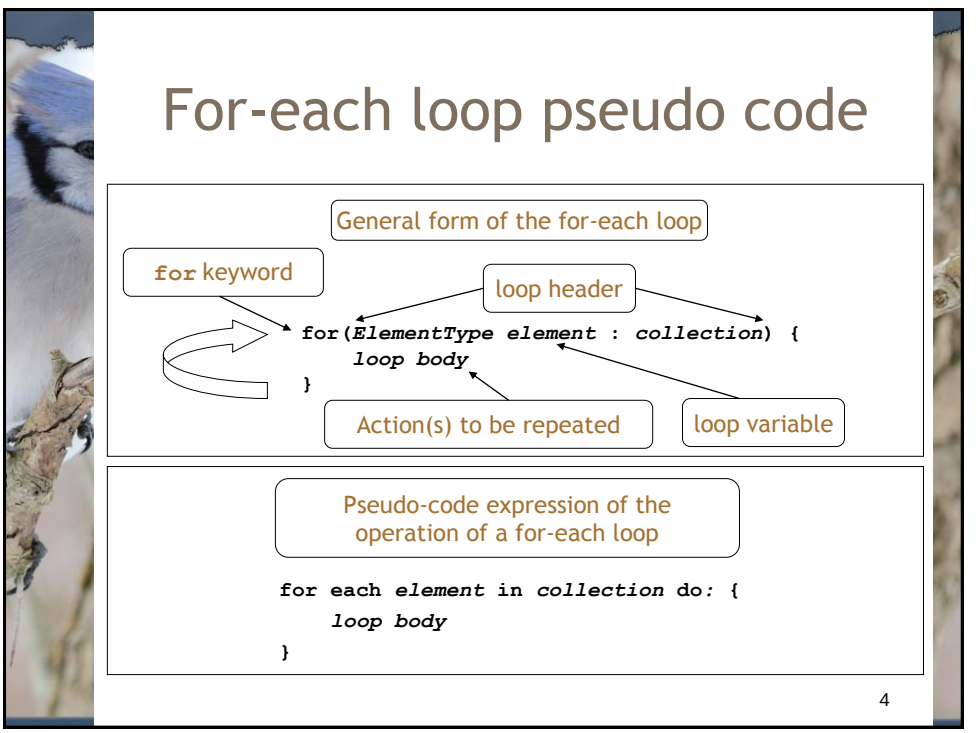
- Collections and iteration
- Loops: the for-each loop
- Searching collections
- Identity versus equality



Collections and iteration

- With a collection, we often want to repeat the actions: *exactly once for every object in the collection.*
- Java has several sorts of loop statement to deal with collections.
  - We will start with its *for-each loop.*

3



For-each loop pseudo code

General form of the for-each loop

`for` keyword

loop header

```
for(ElementType element : collection) {
    loop body
}
```

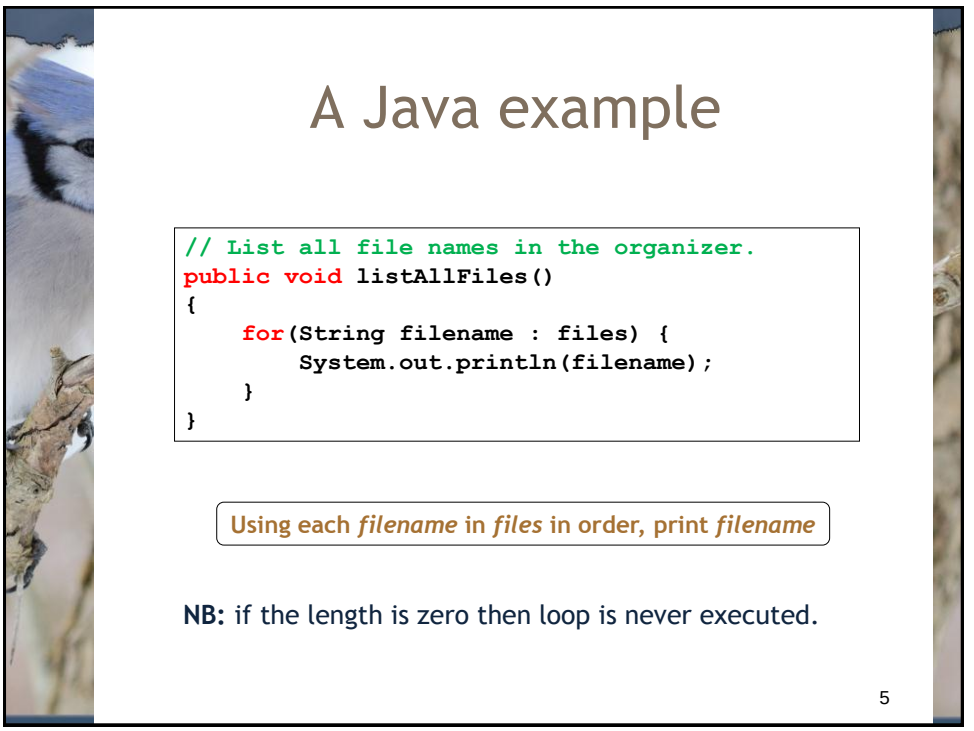
Action(s) to be repeated

loop variable

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

```
for each element in collection do: {
    loop body
}
```

4



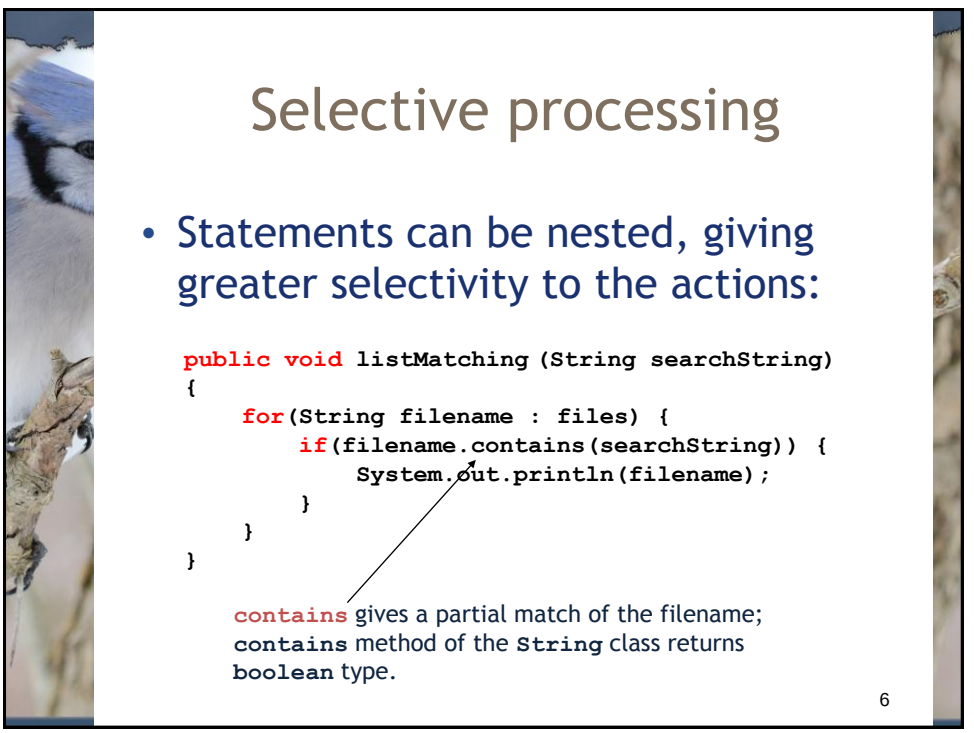
A Java example

```
// 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*

NB: if the length is zero then loop is never executed.

5



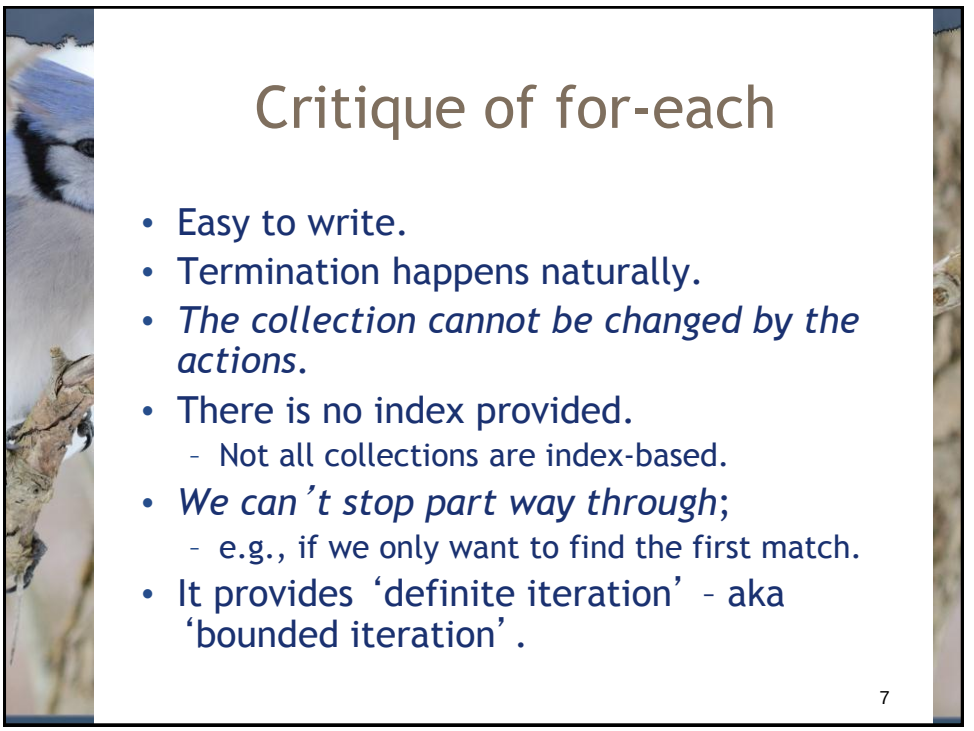
Selective processing

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

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

`contains` gives a partial match of the filename; `contains` method of the `String` class returns `boolean` type.

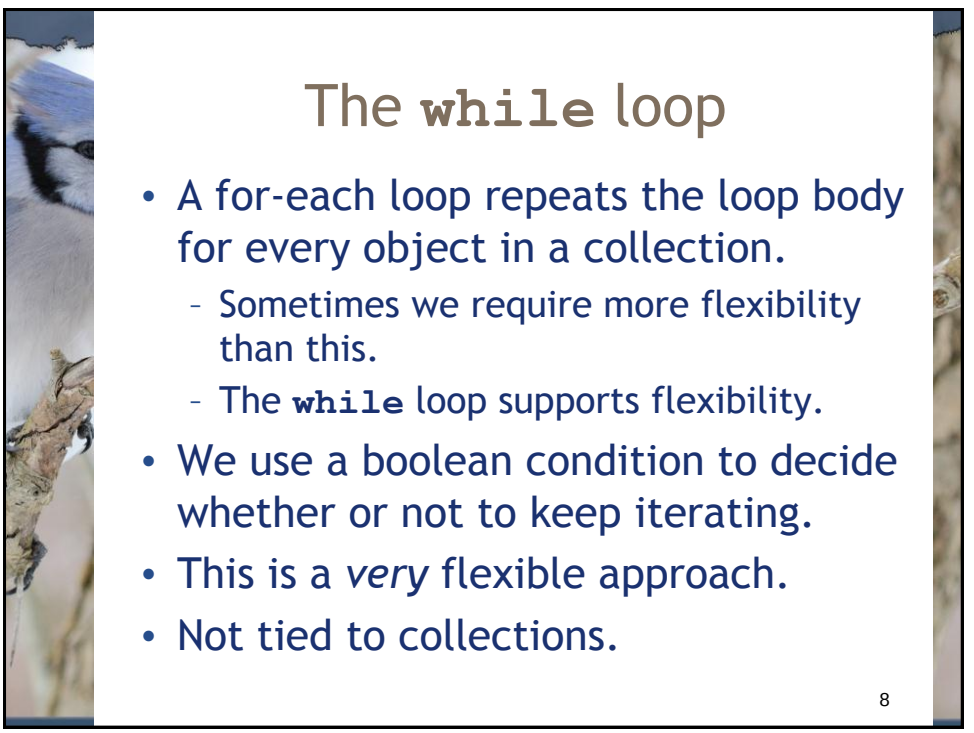
6



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'.

7



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.

8

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

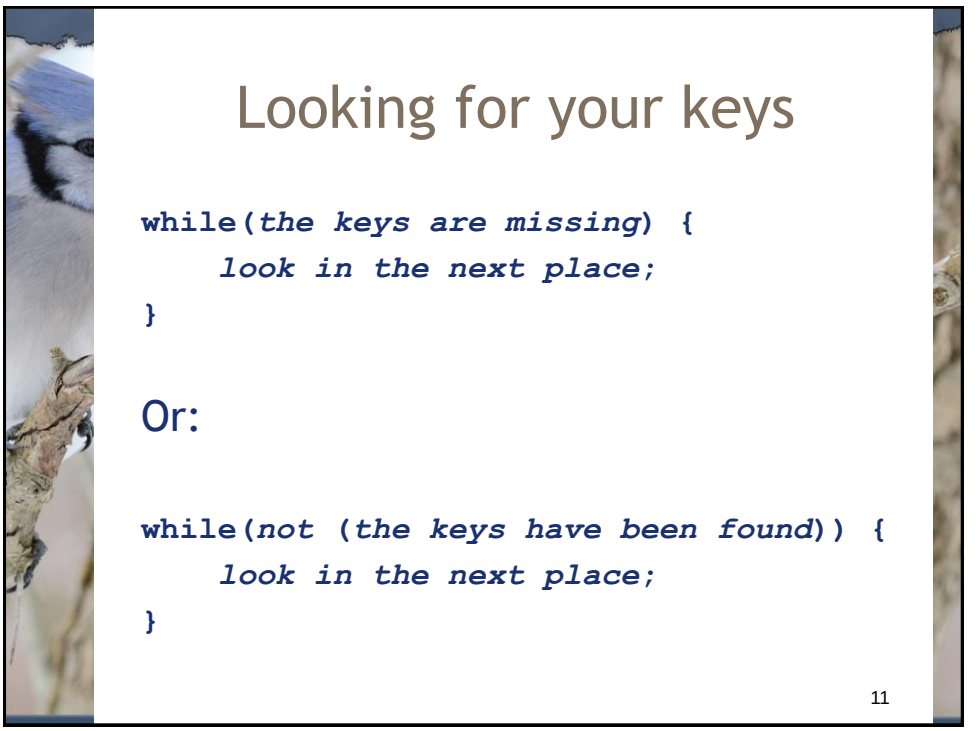
while the value of *index* is less than the size of the collection, get and print the next file name, and then increment *index*

9

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.

10



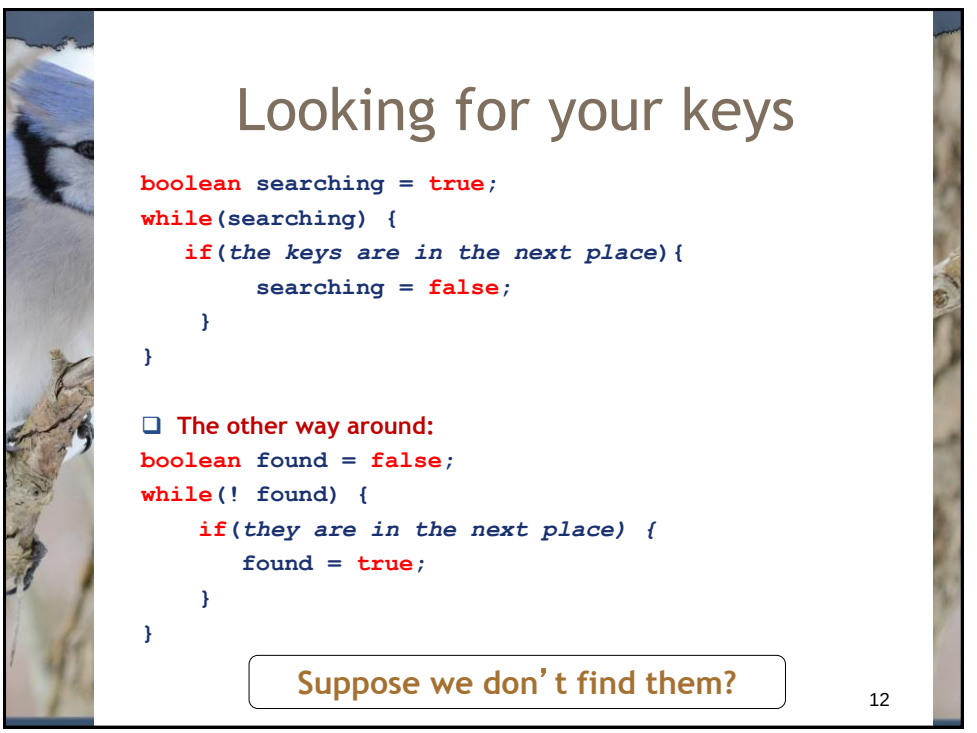
Looking for your keys

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

Or:

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

11



Looking for your keys

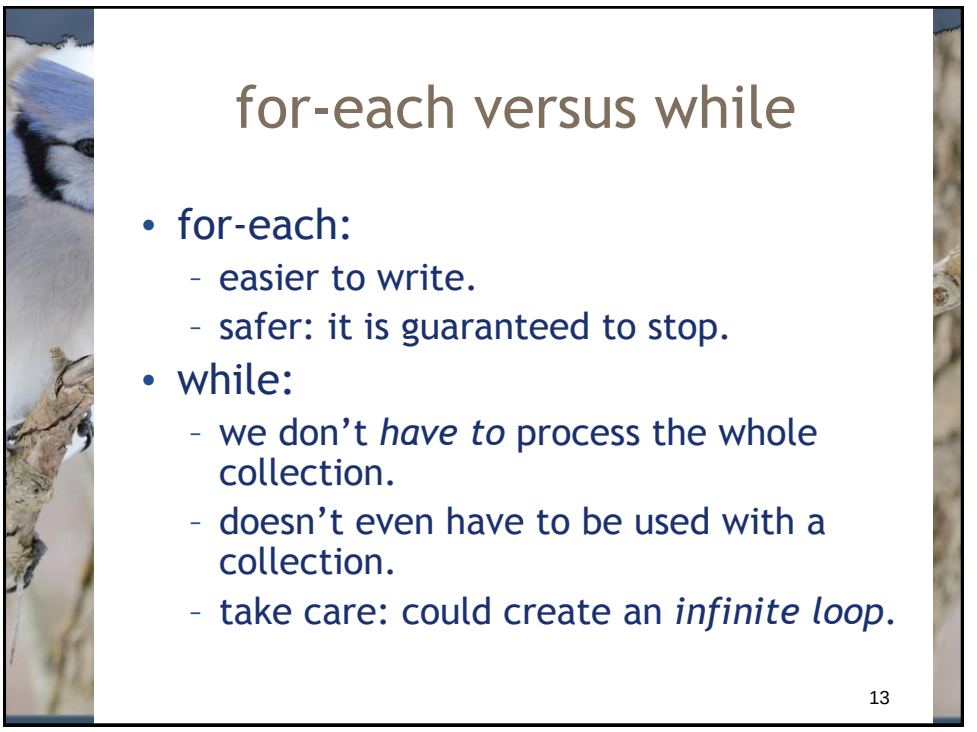
```
boolean searching = true;
while(searching) {
   if(the keys are in the next place){
        searching = false;
    }
}
```

- The other way around:

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

Suppose we don't find them?

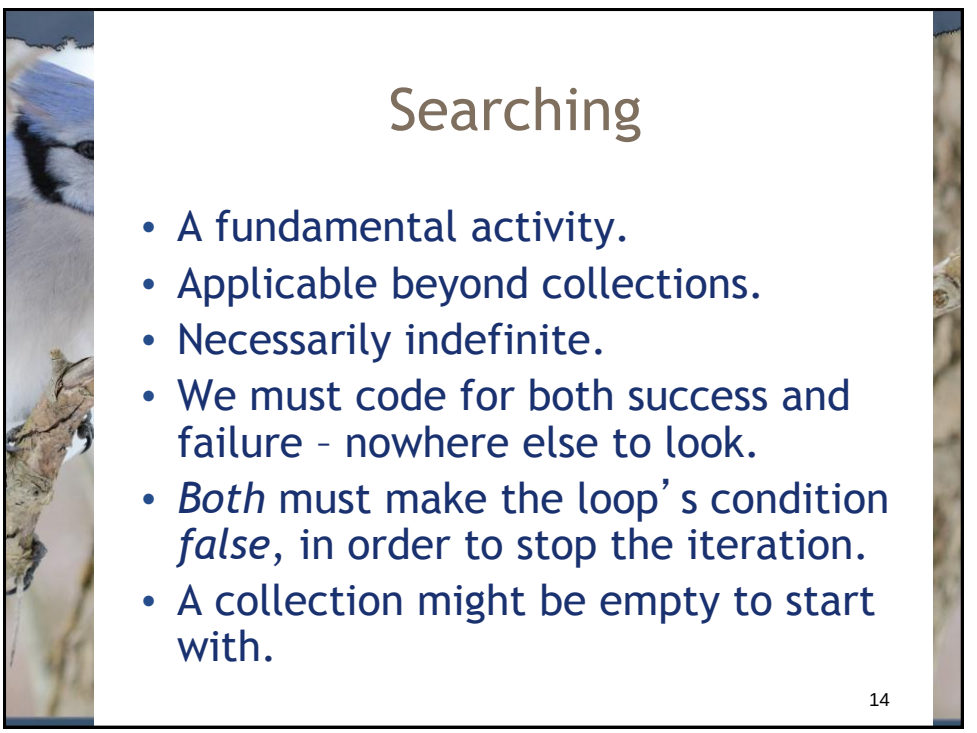
12



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*.

13



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.

14

Finishing a search

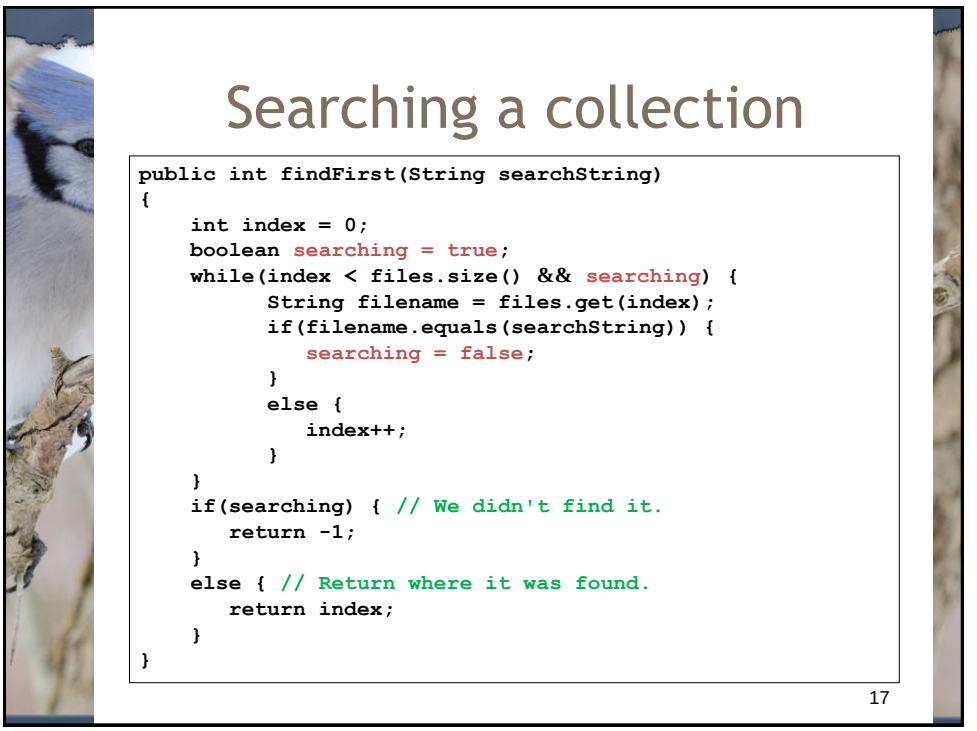
- How do we finish a search?
- *Either* there are no more items to check:
  `index >= files.size()`
- *Or* the item has been found:
  `found == true`
  `found`
  `! searching`

15

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.

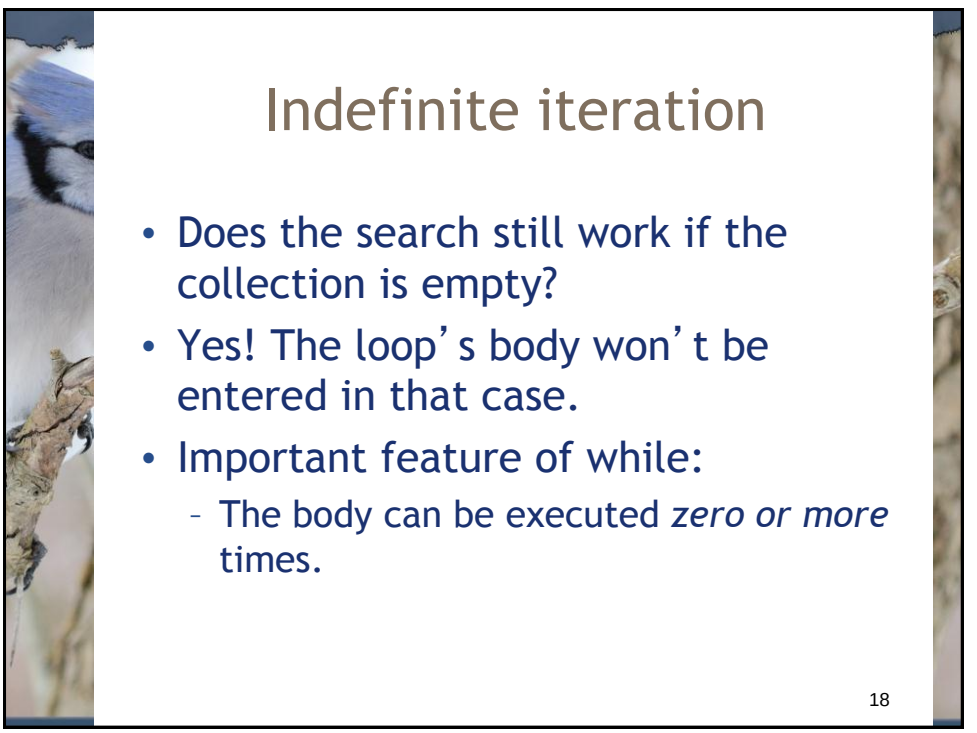
16



Searching a collection

```
public int findFirst(String searchString)
{
    int index = 0;
    boolean searching = true;
    while(index < files.size() && searching) {
          String filename = files.get(index);
          if(filename.equals(searchString)) {
             searching = false;
          }
          else {
             index++;
          }
    }
    if(searching) { // We didn't find it.
       return -1;
    }
    else { // Return where it was found.
       return index;
    }
}
```

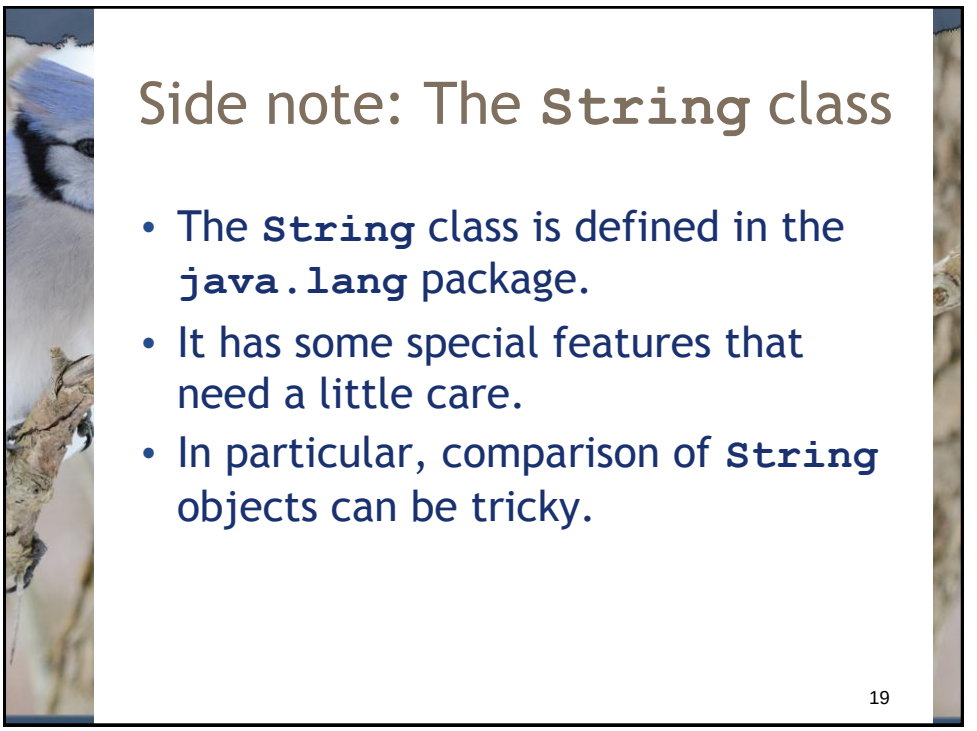
17



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.

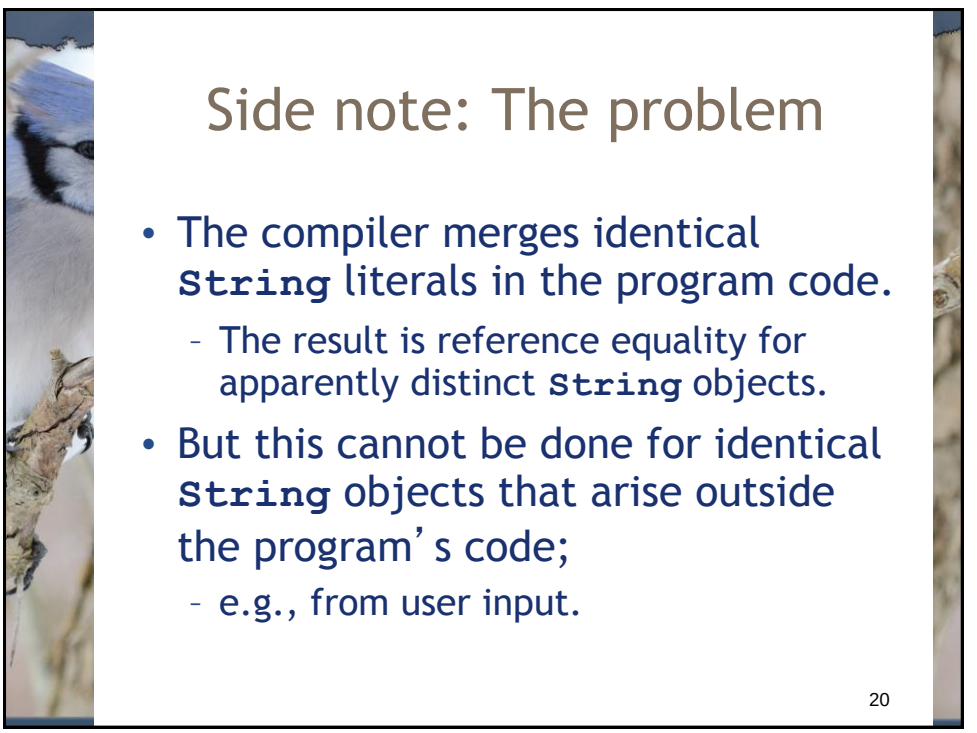
18



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.

19



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.

20

Side note: String equality

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

tests identity

Do not use!!

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

tests equality

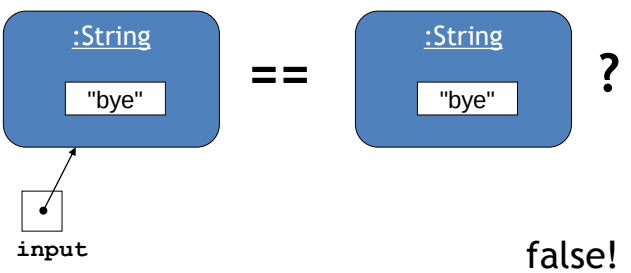
Important: Always use `.equals` for testing String equality!

21

Identity vs equality (Strings)

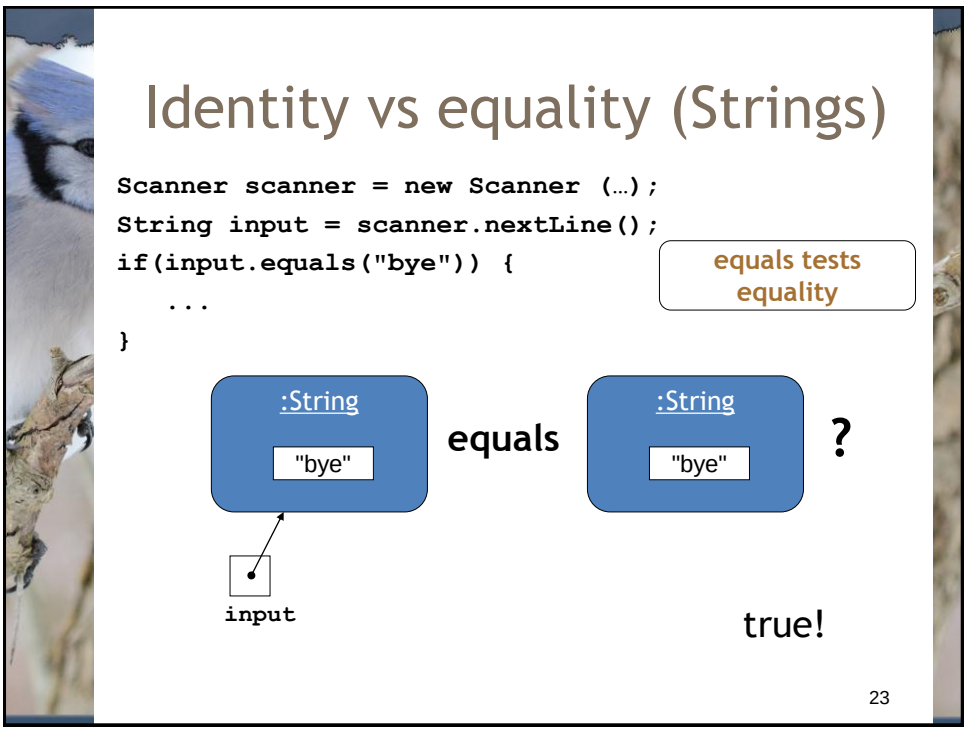
```
Scanner scanner = new Scanner (…);
String input = scanner.nextLine();
if(input == "bye") {
    ...
}
```

== tests identity



false!

22


Identity vs equality (Strings)
Scanner scanner = new Scanner (…);
String input = scanner.nextLine();
if(input.equals("bye")) {
...
}
equals tests equality
:String
"bye"
equals
:String
"bye"
?
input
true!
23