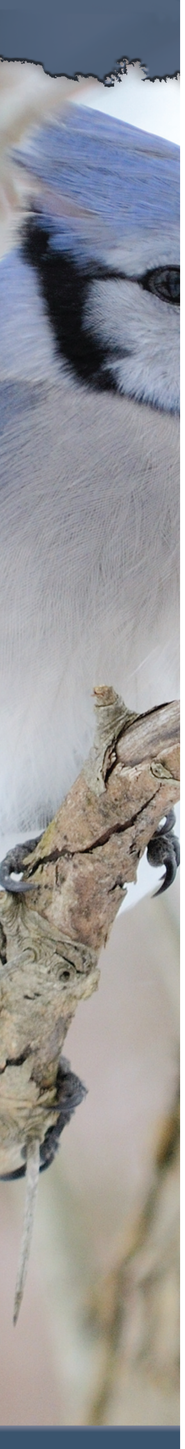
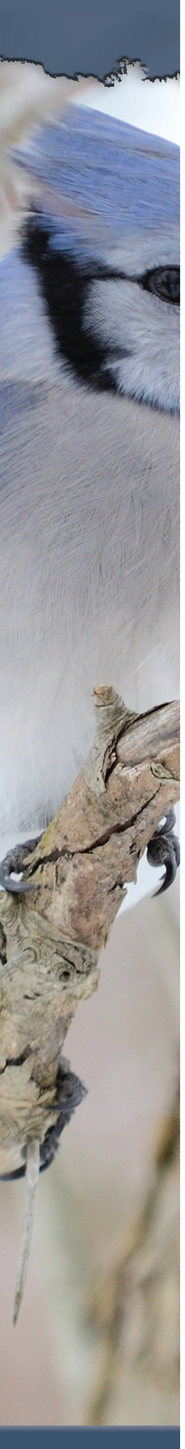


KING'S College LONDON

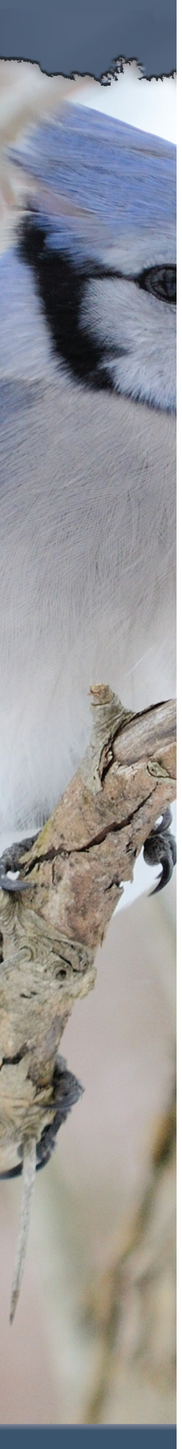
Programming Practice and Applications

Grouping objects - Part II

Michael Kölling

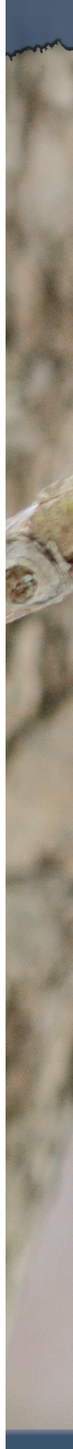


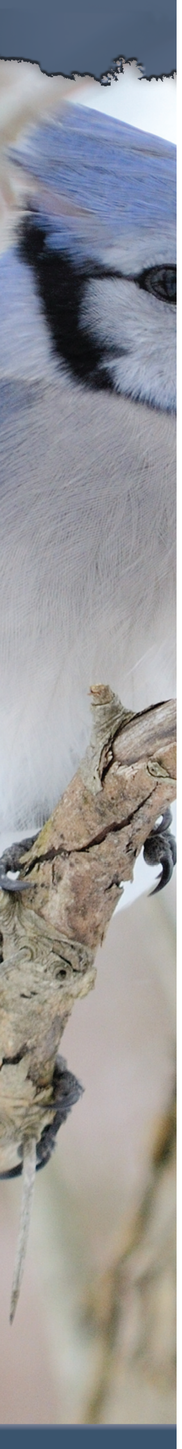
Indefinite iteration - the while loop



Main concepts to be covered

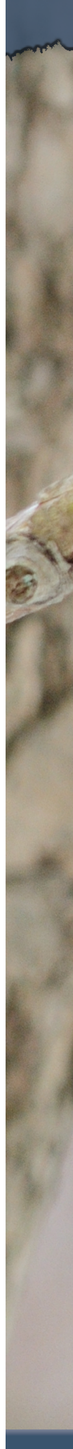
- The difference between definite and indefinite (unbounded) iteration.
- Loops: the while loop

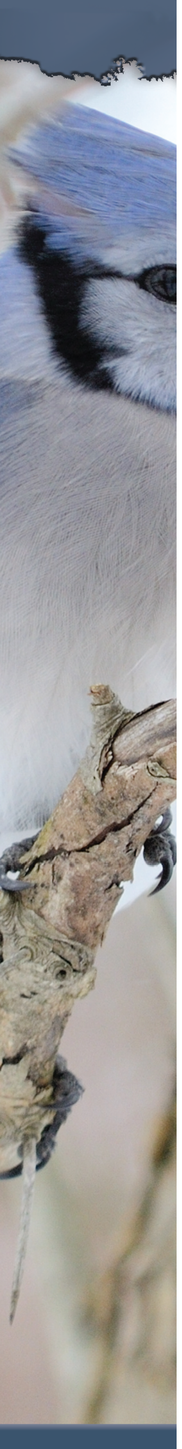




Search tasks are indefinite

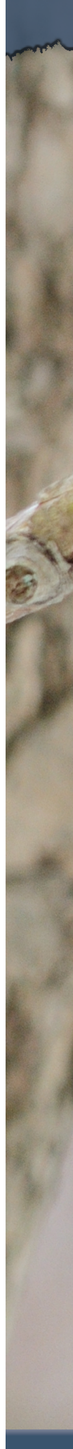
- Consider: searching for your keys.
- You cannot predict, *in advance*, how many places you will have to look.
- Although, there may well be an absolute limit – i.e., checking every possible location.
- You will stop when you find them.
- ‘Infinite loops’ are also possible.
  - Through error or the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



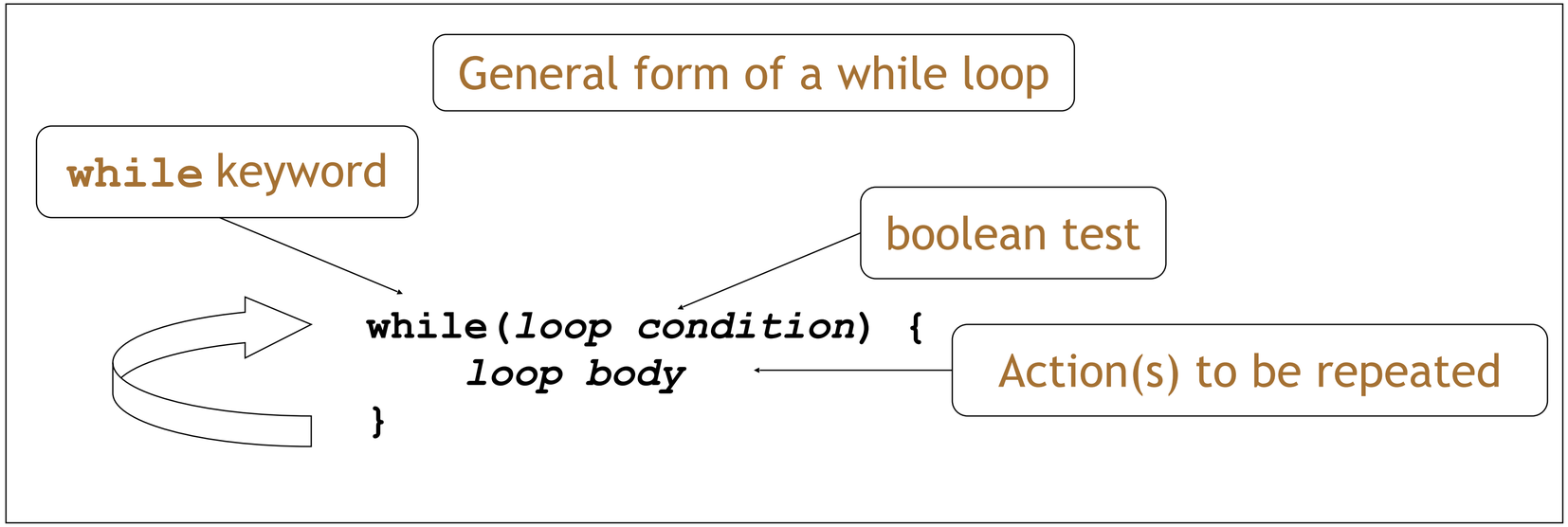


The while loop

- 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.

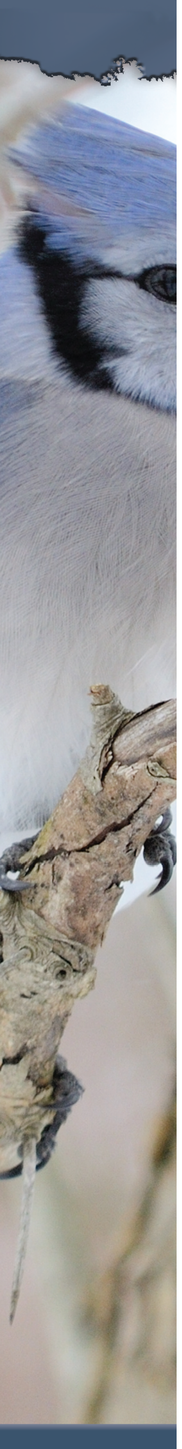


While loop pseudo code



Pseudo-code expression of the actions of a while loop

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

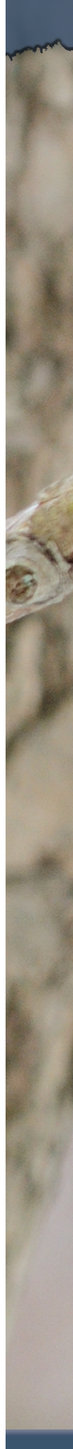


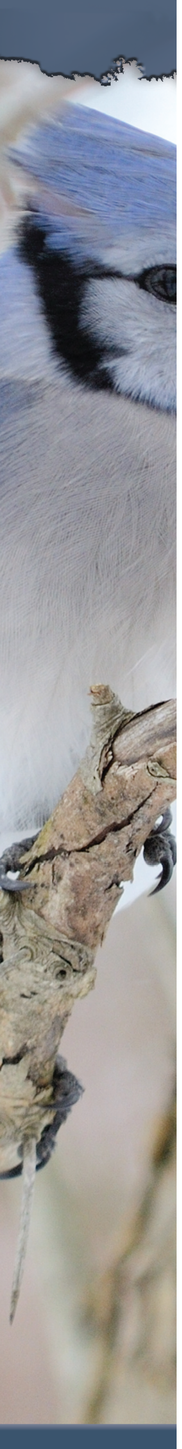
Looking for your keys

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

Or:

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





Looking for your keys

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

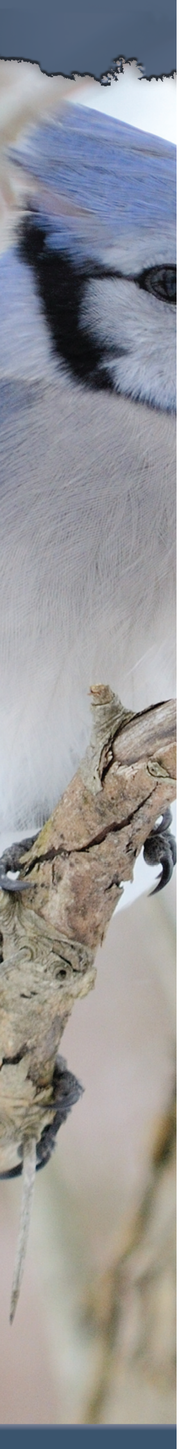
Suppose we don't find them?

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++;
    }
}
```

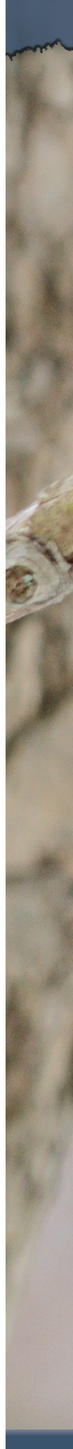
Increment *index* by 1

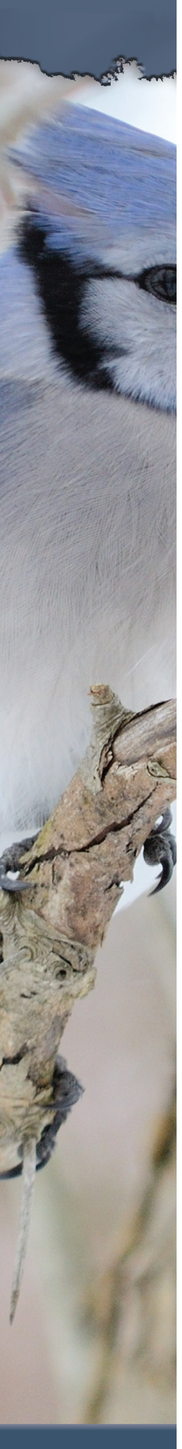
while the value of *index* is less than the size of the collection, get and print the next file name, and then increment *index*



Elements of the loop

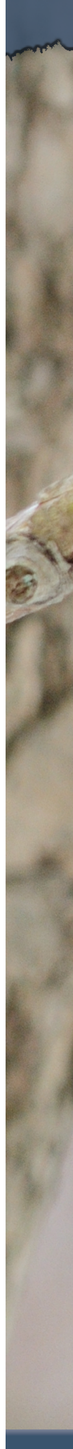
- We have declared an index variable.
- The condition must be expressed correctly.
- We have to fetch each element.
- The index variable must be incremented explicitly.

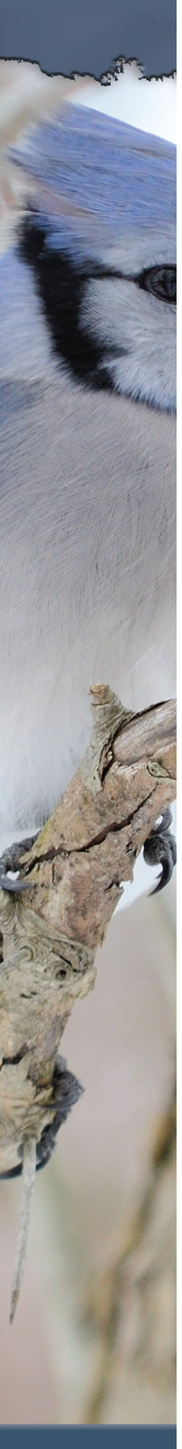




for-each versus while

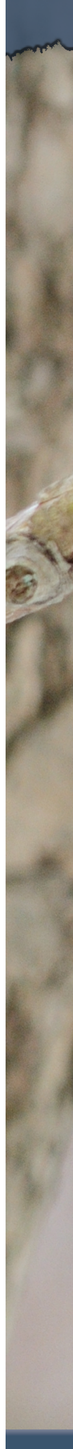
- for-each:
  - easier to write.
  - safer: it is guaranteed to stop.
- while:
  - we don't *have to* process the whole collection.
  - doesn't even have to be used with a collection.
  - take care: could create an *infinite loop*.





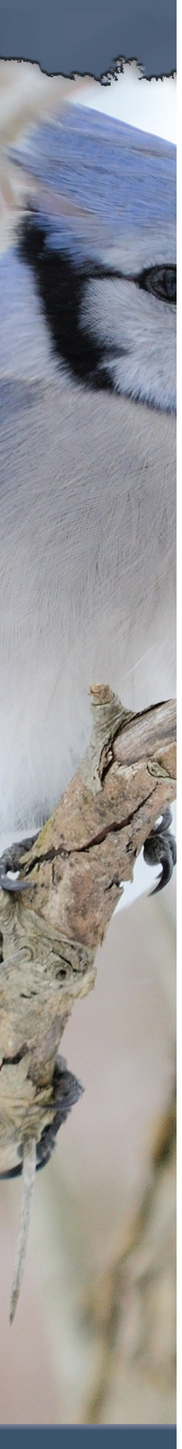
Searching

- 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.



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`

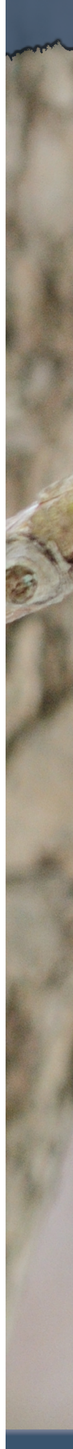


Continuing a search

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

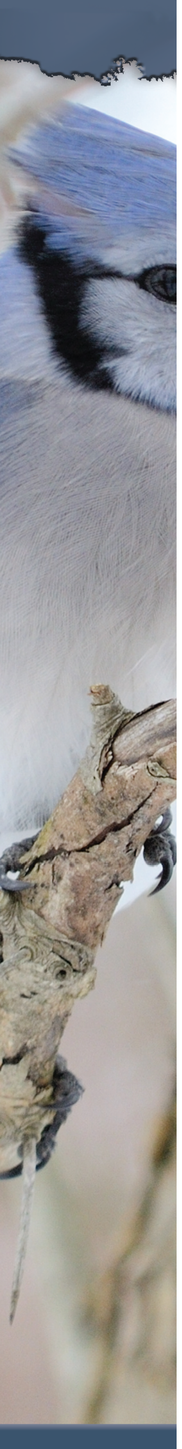
```
index < files.size() && ! found
index < files.size() && searching
```

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



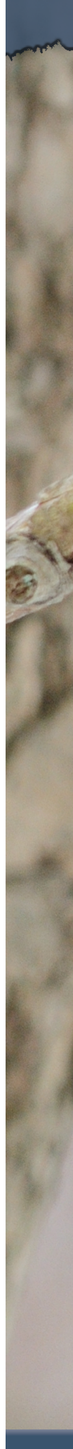
Searching a collection

```
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.
```



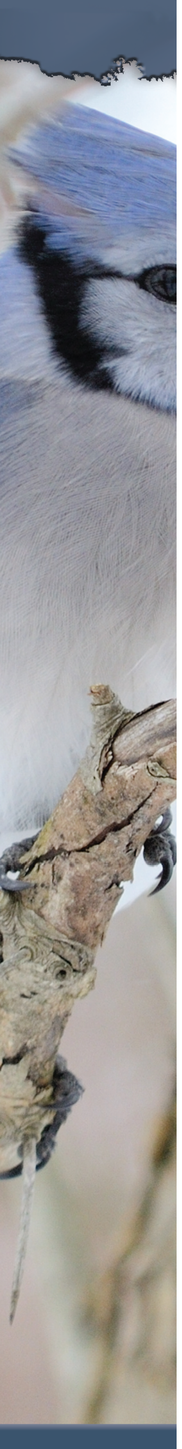
Searching a collection

```
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.
```



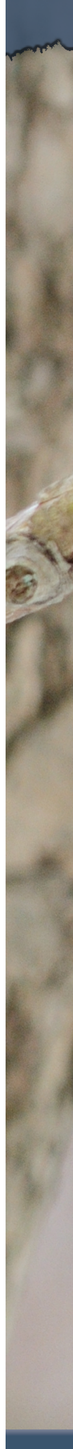
Indefinite iteration

- 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.



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.



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.

Side note: String equality

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

tests identity

Do not use!!

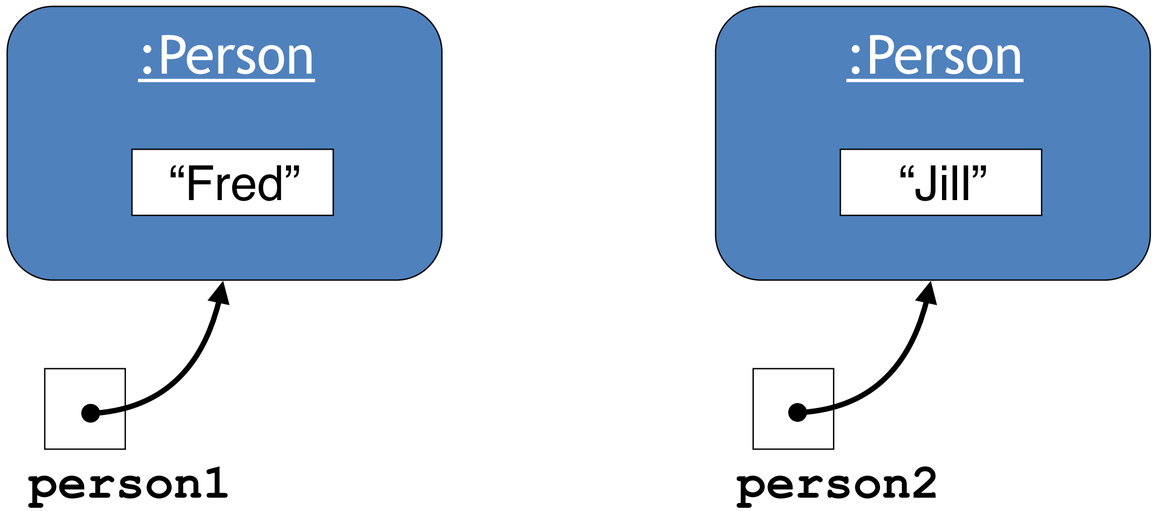
```
if(input.equals("bye")) {
    ...
}
```

tests equality

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

Identity vs equality 1

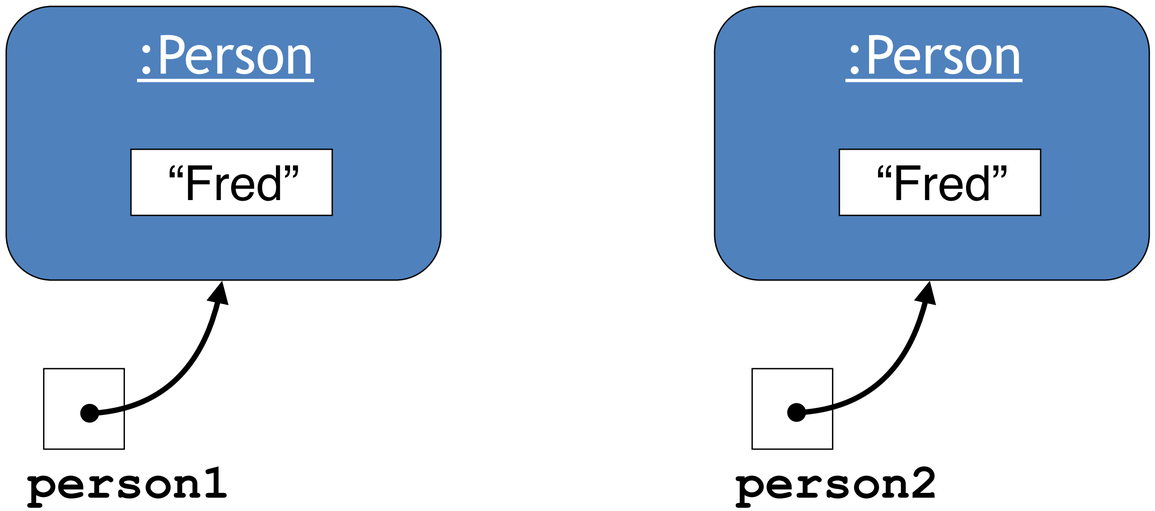
Other (non-String) objects:




```
person1 == person2 ?
```

Identity vs equality 2

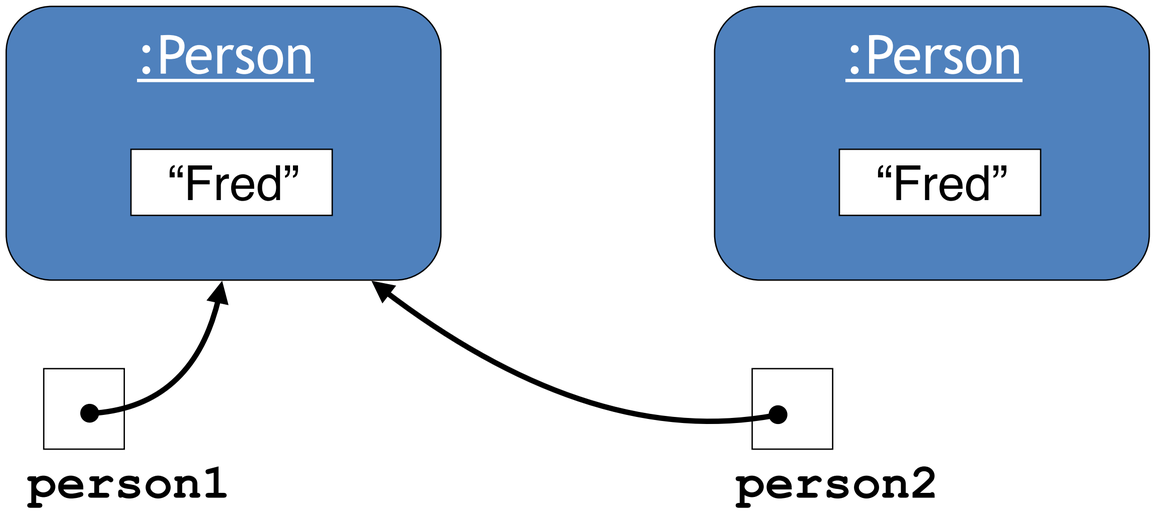
Other (non-String) objects:




```
person1 == person2 ?
```

Identity vs equality 3

Other (non-String) objects:

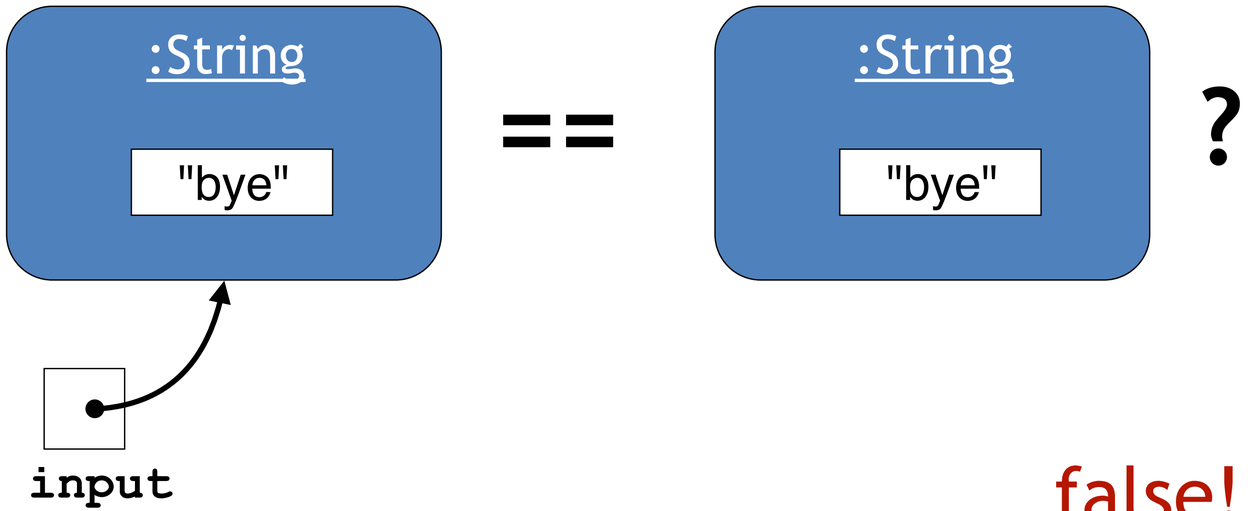



```
person1 == person2 ?
```

Identity vs equality (Strings)

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

== tests identity

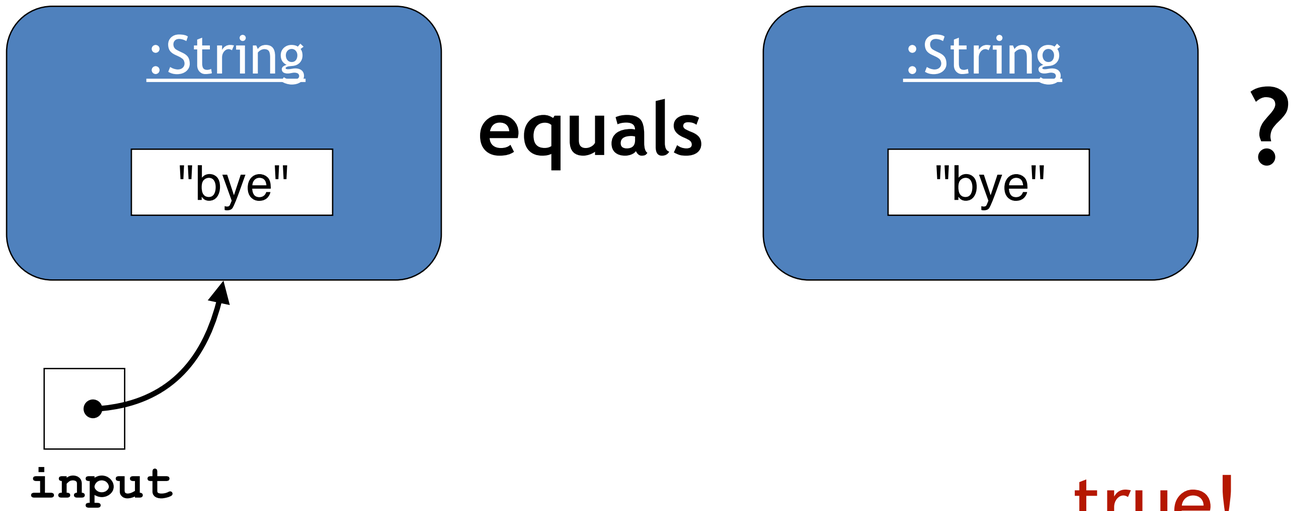


false!

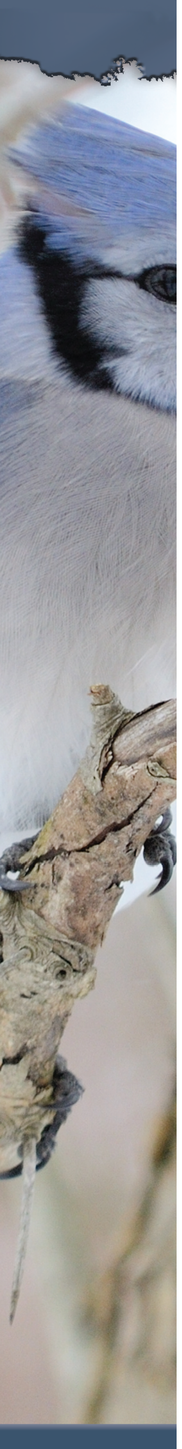
Identity vs equality (Strings)

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

equals tests equality

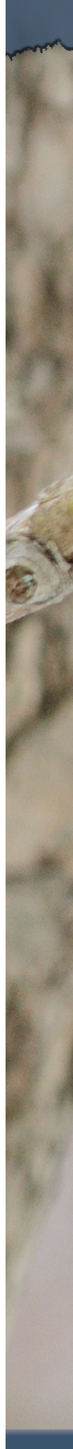


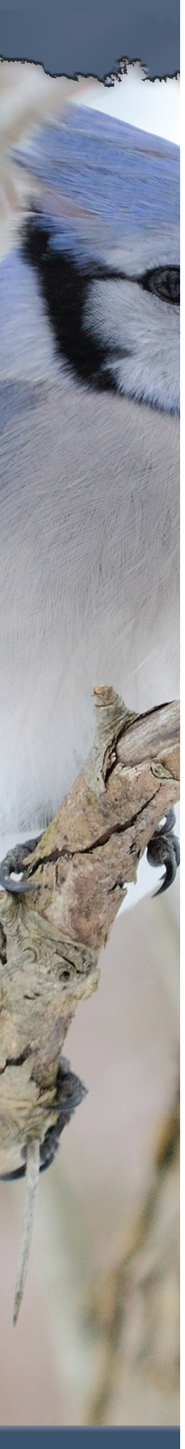
true!



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`**

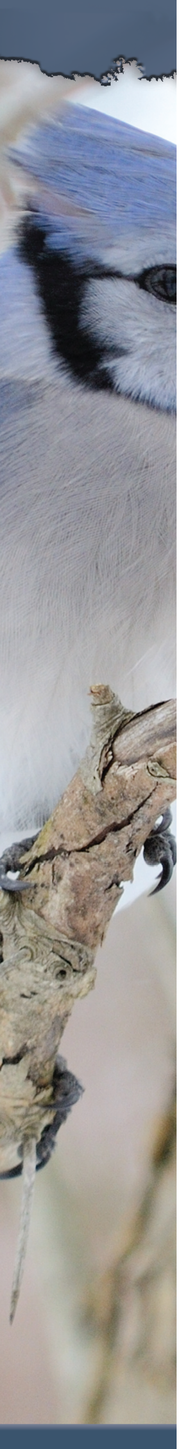




KING'S College LONDON

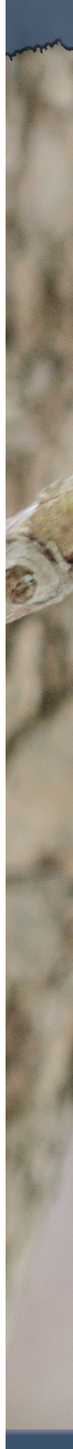
Grouping objects

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()



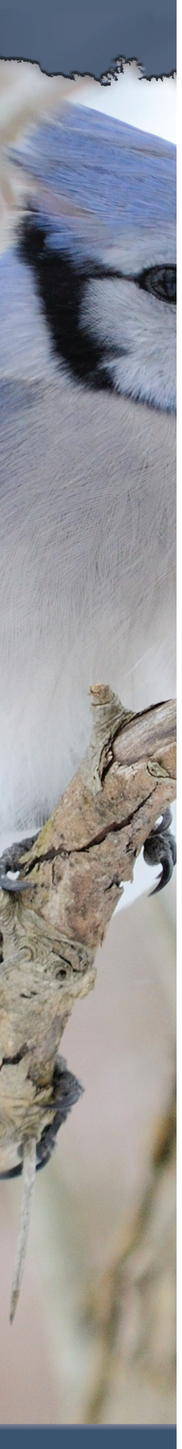
Using an Iterator object

java.util.Iterator

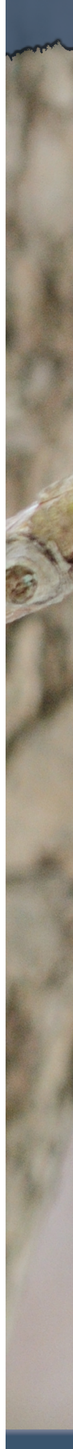
returns an Iterator object

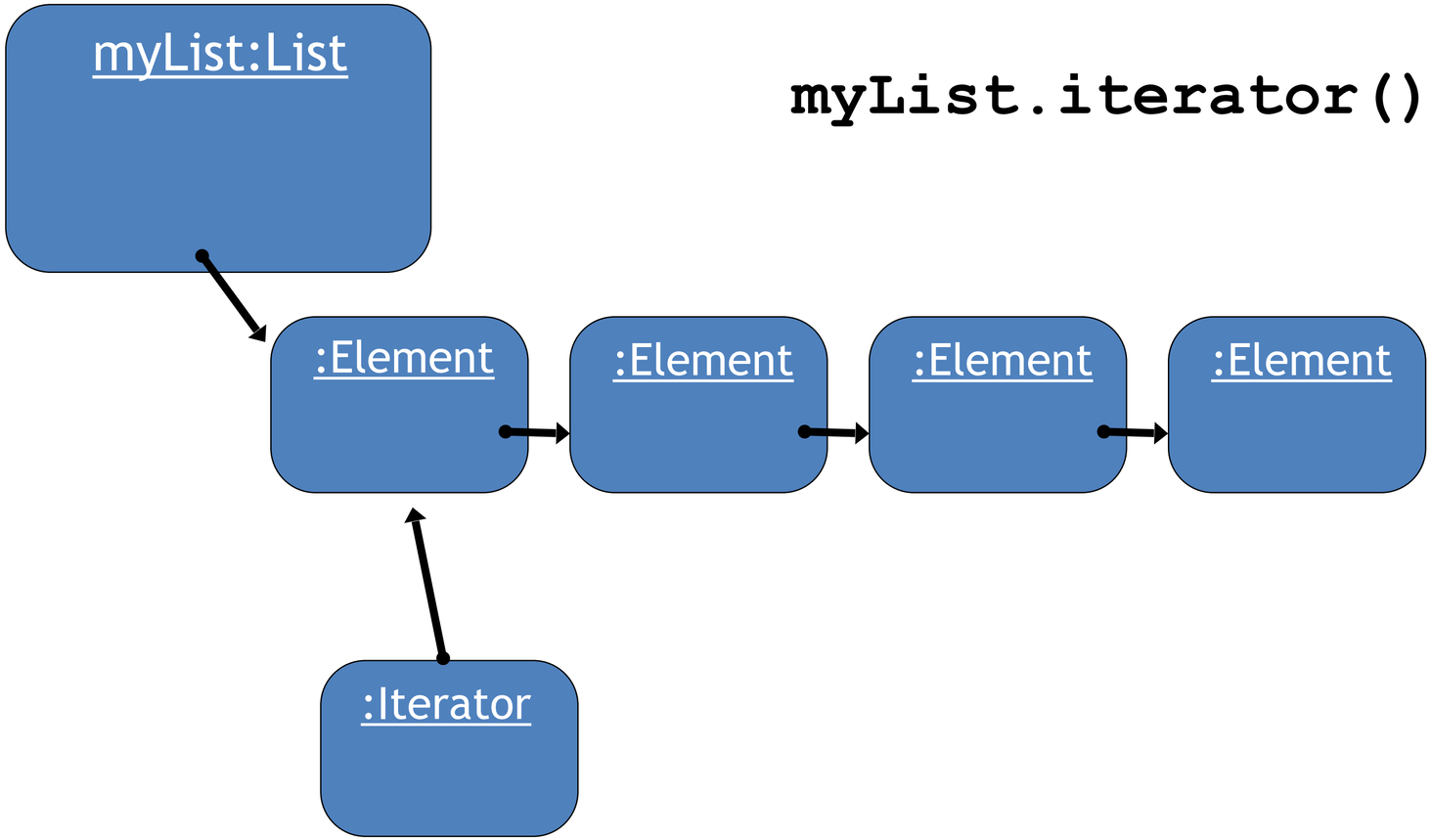
```
Iterator<ElementType> it = myCollection.iterator();
while(it.hasNext()) {
    call it.next() to get the next object
    do something with that object
}
```

```
public void listAllFiles()
{
    Iterator<Track> it = files.iterator();
    while(it.hasNext()) {
        Track tk = it.next();
        System.out.println(tk.getDetails());
    }
}
```

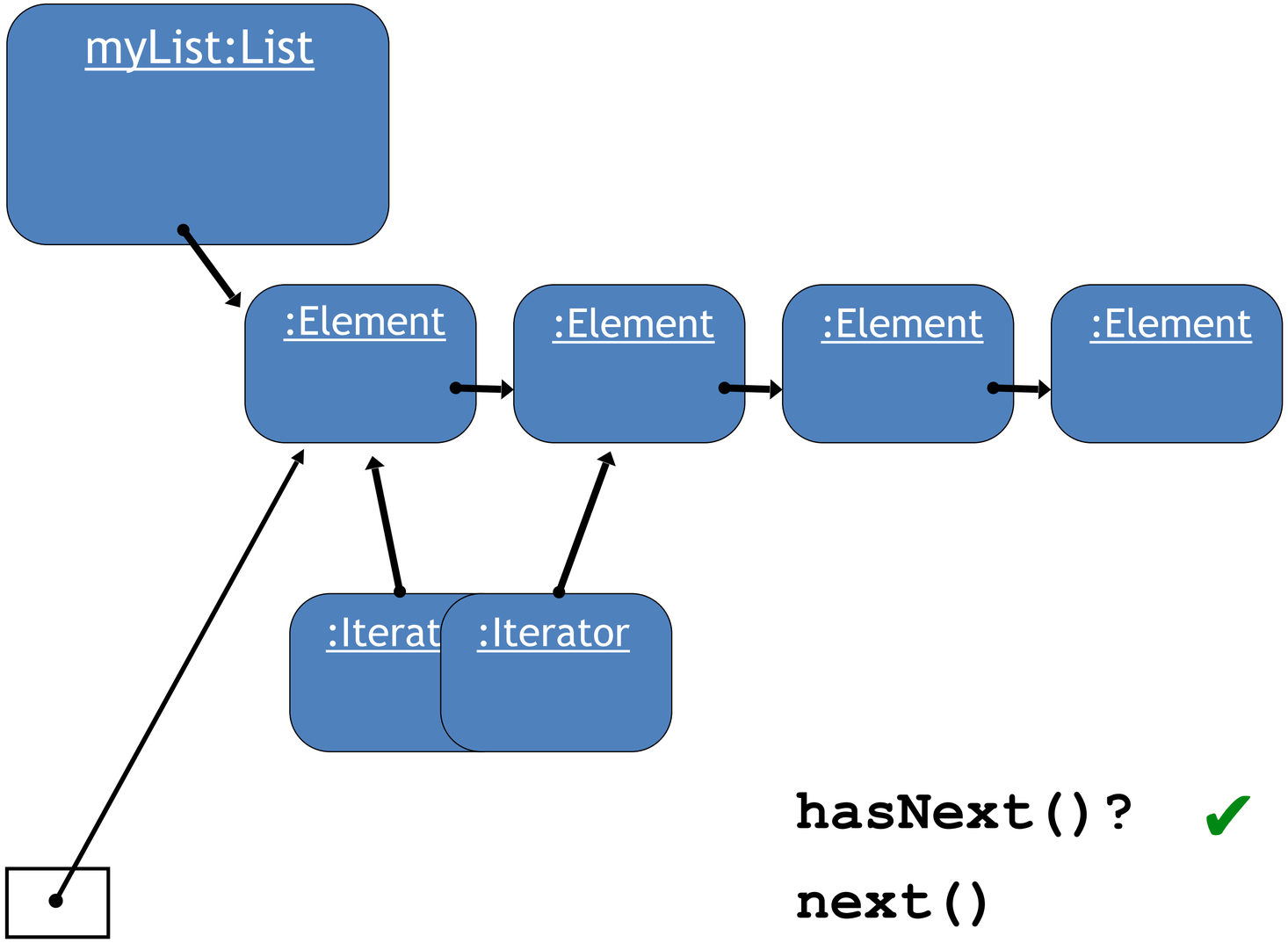


Iterator mechanics

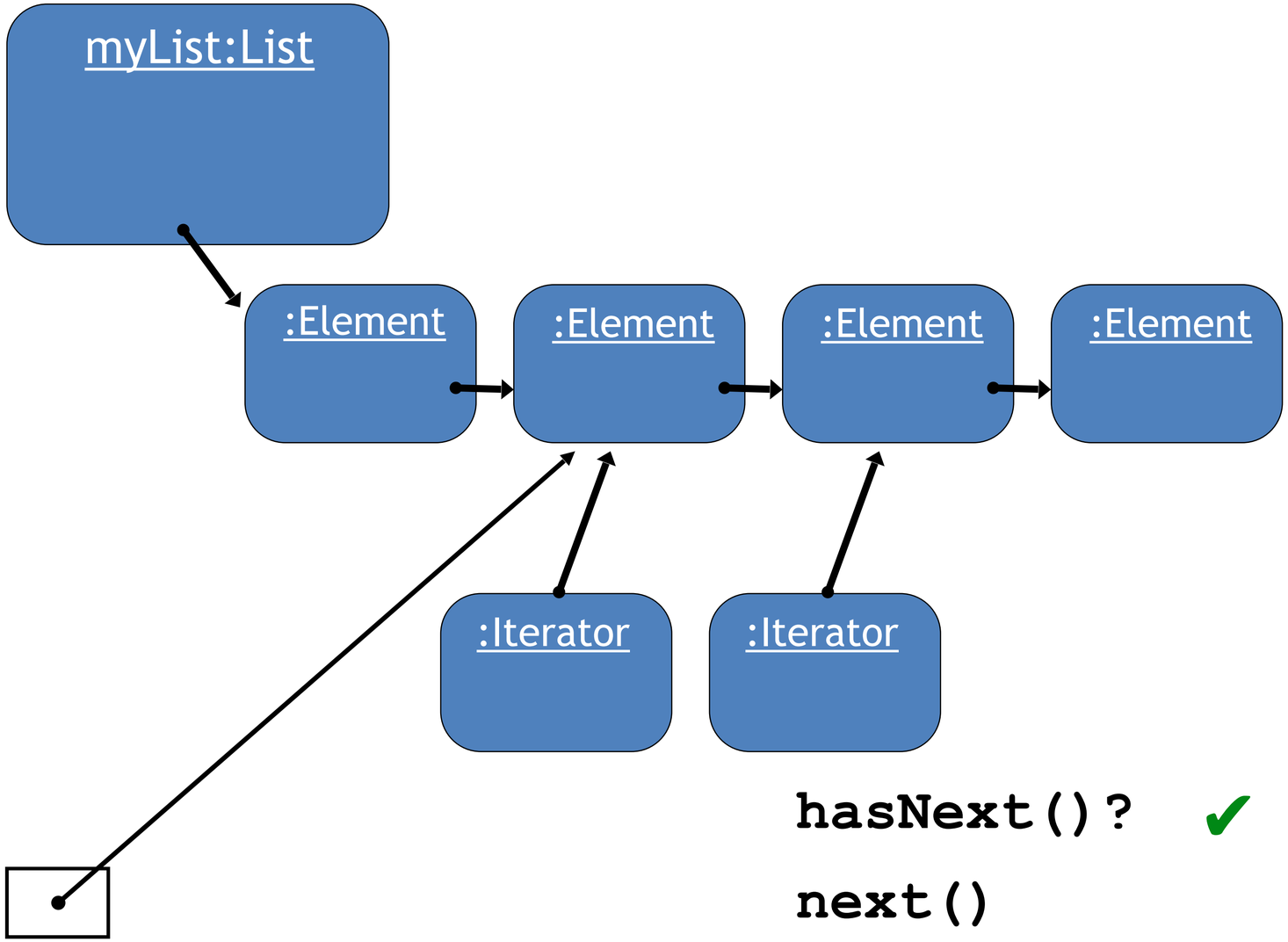


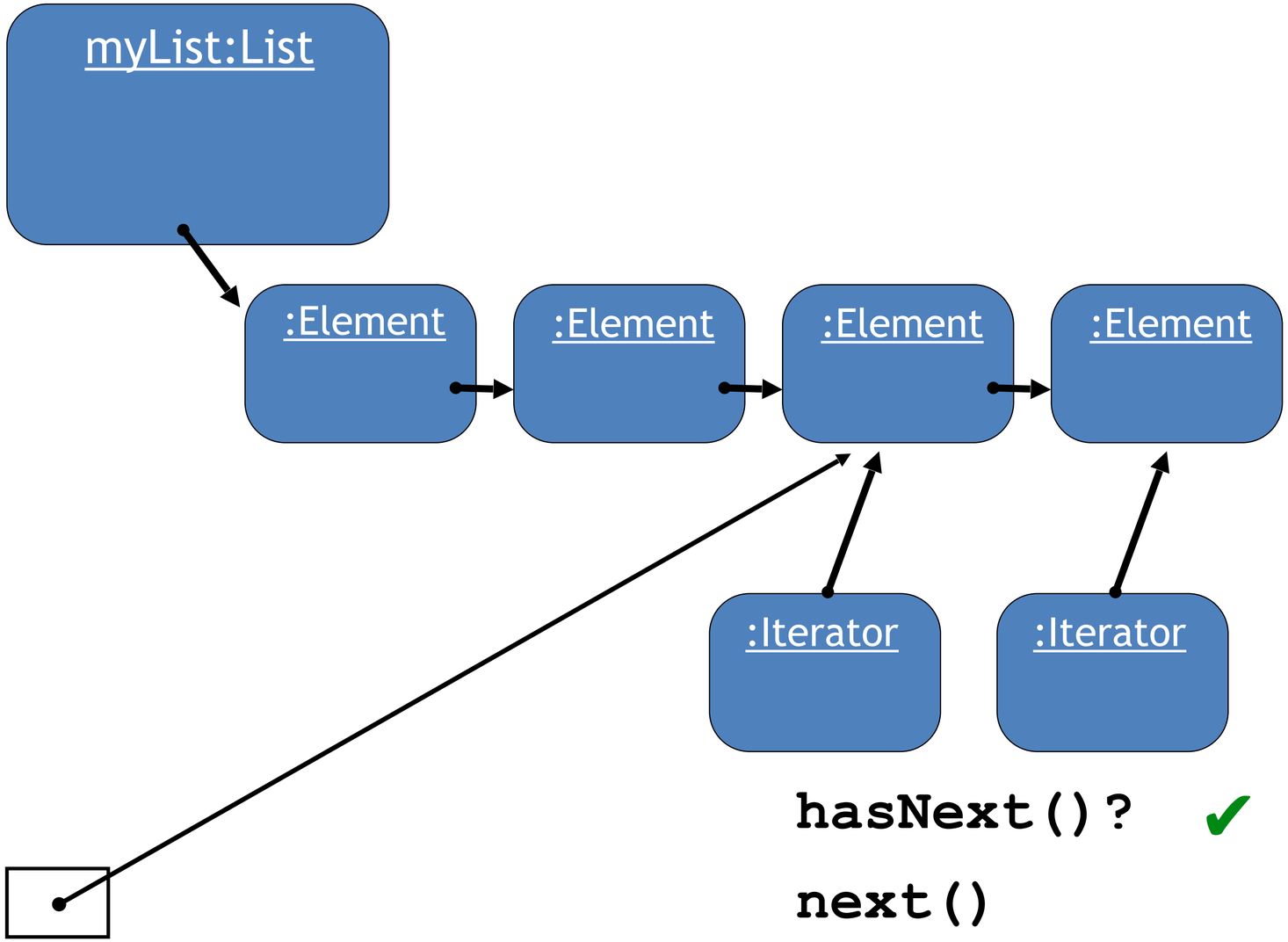


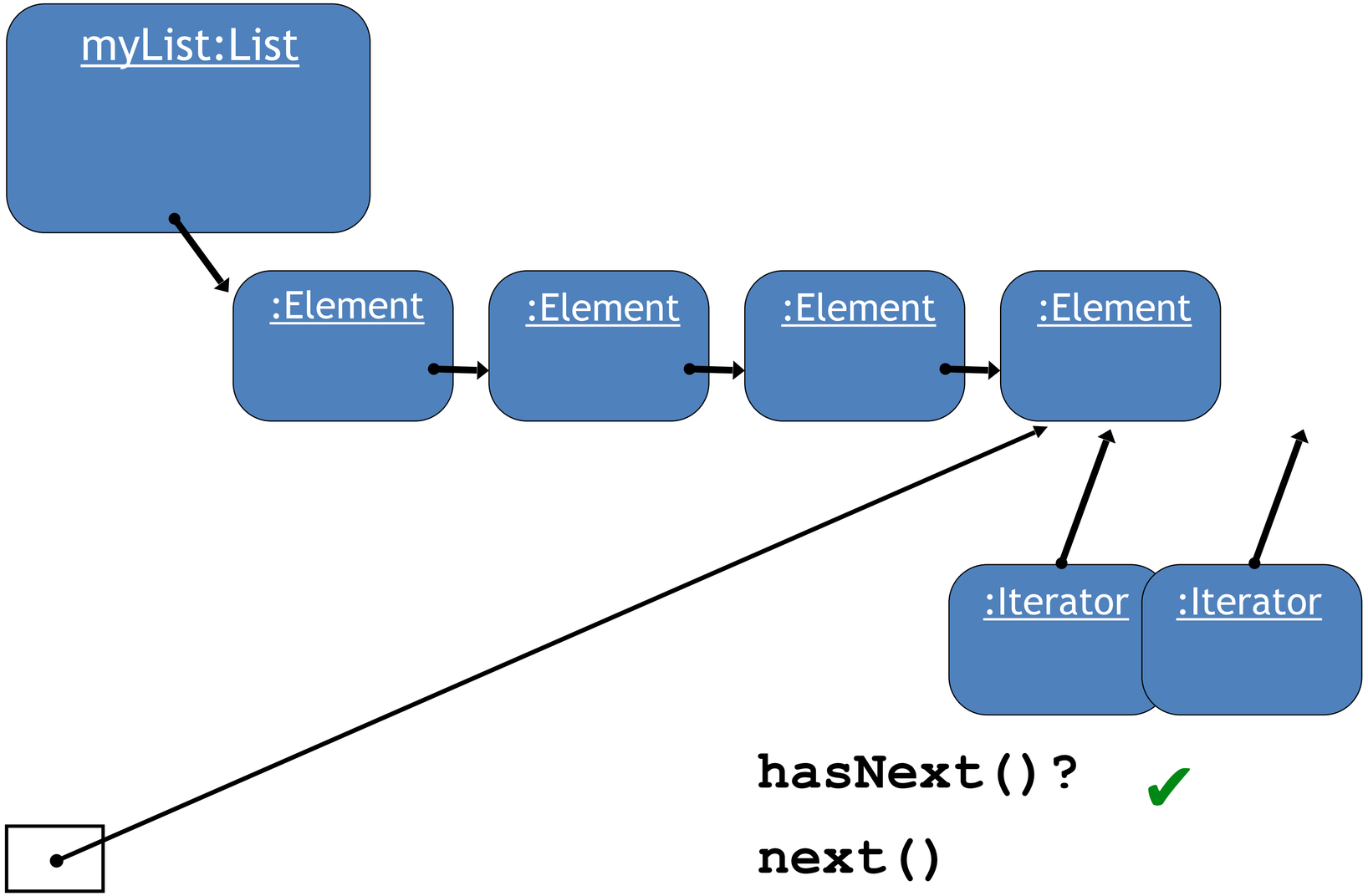

```
myList.iterator()
```

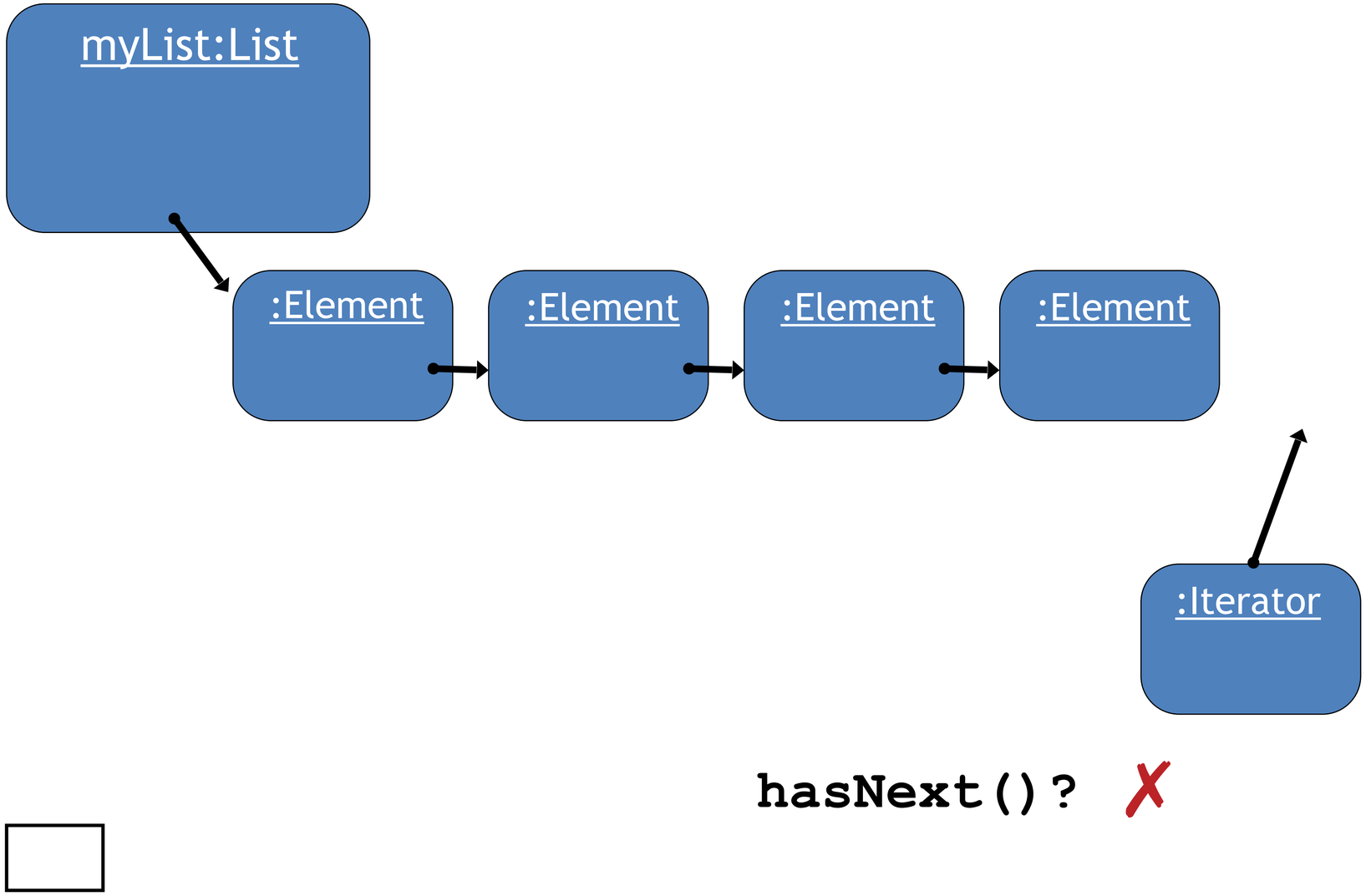


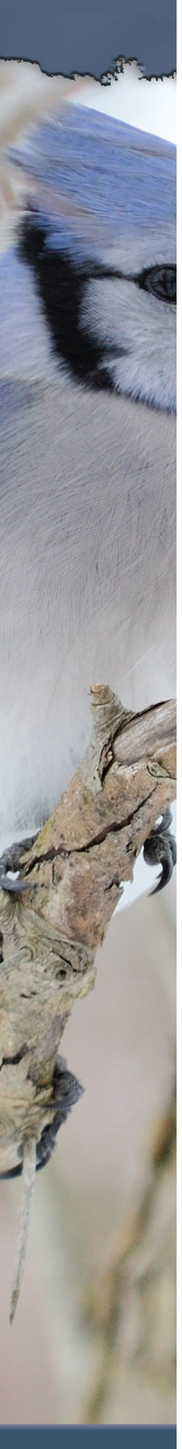

```
Element e = iterator.next();
```


myList:List
:Element
:Element
:Element
:Element
:Iterator
:Iterator
hasNext()? ✔
next()


myList:List
:Element
:Element
:Element
:Element
:Iterator
:Iterator
hasNext()?
next()

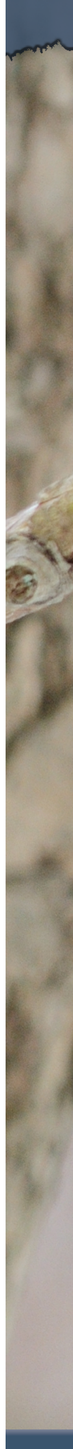

myList:List
:Element
:Element
:Element
:Element
:Iterator
:Iterator
hasNext()?
next()


myList:List
:Element
:Element
:Element
:Element
:Iterator
hasNext()?



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*.



Removing from a collection

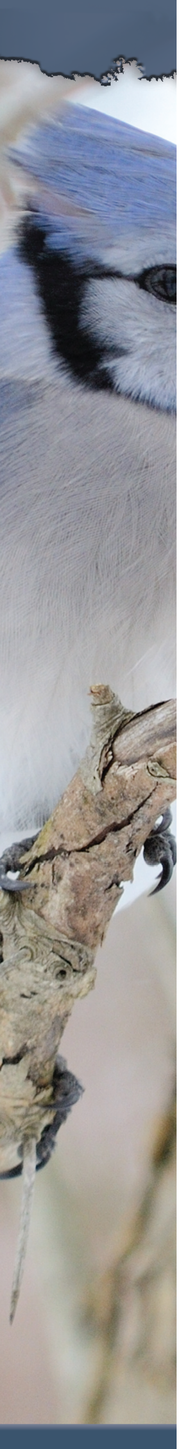
```
Iterator<Track> it = tracks.iterator();
while(it.hasNext()) {
    Track t = it.next();
    String artist = t.getArtist();
    if(artist.equals(artistToRemove)) {
        it.remove();
    }
}
```

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

Removing from a collection - wrong!

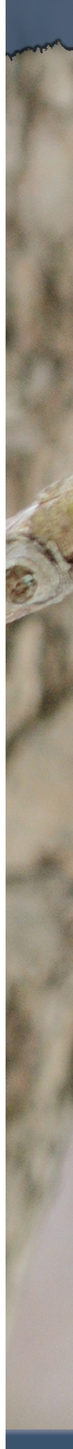
```
int index = 0;
while(index < tracks.size()) {
    Track t = tracks.get(index);
    String artist = t.getArtist();
    if(artist.equals(artistToRemove)) {
        tracks.remove(index);
    }
    index++;
}
```

Can you spot what is wrong?

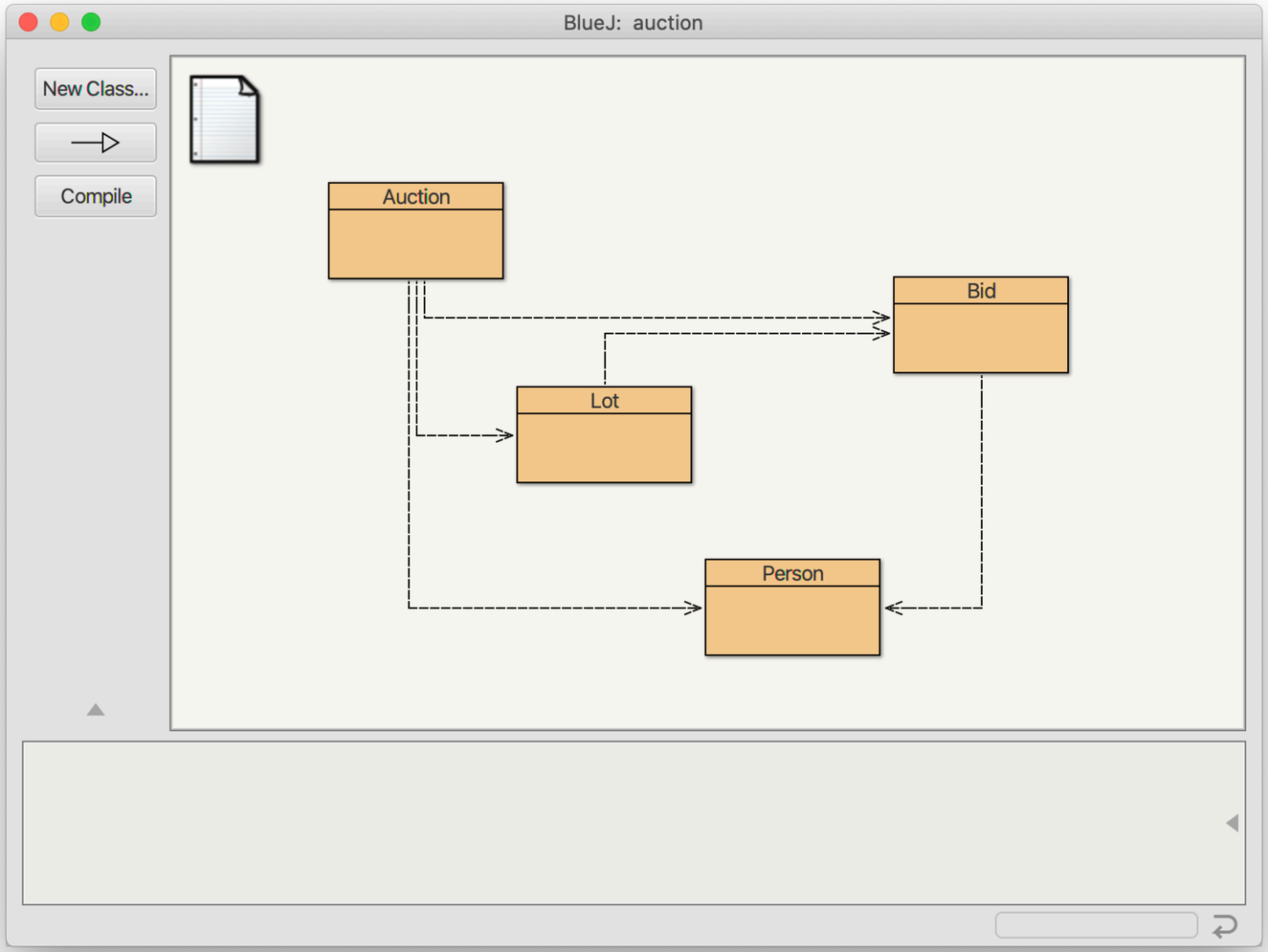


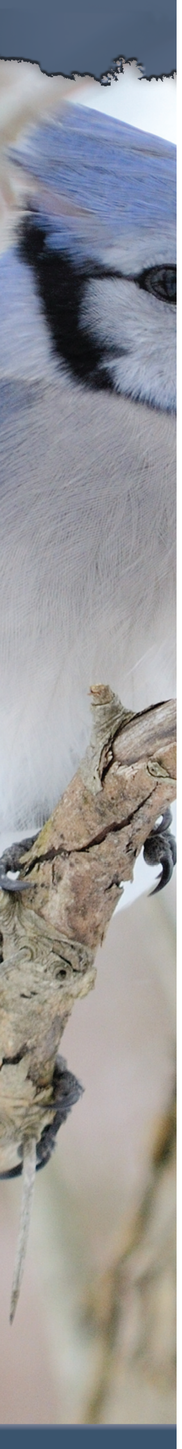
Review

- 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.



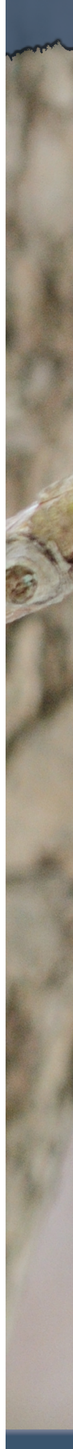
The auction project





The *auction* project

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

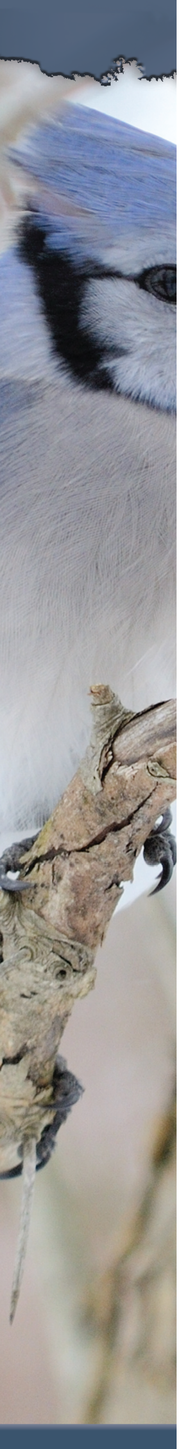


null

- 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’.



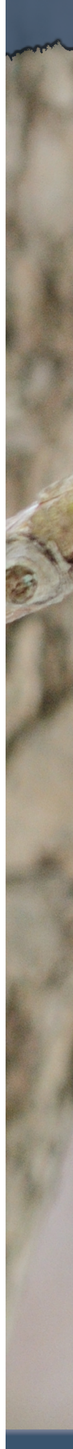
Anonymous objects

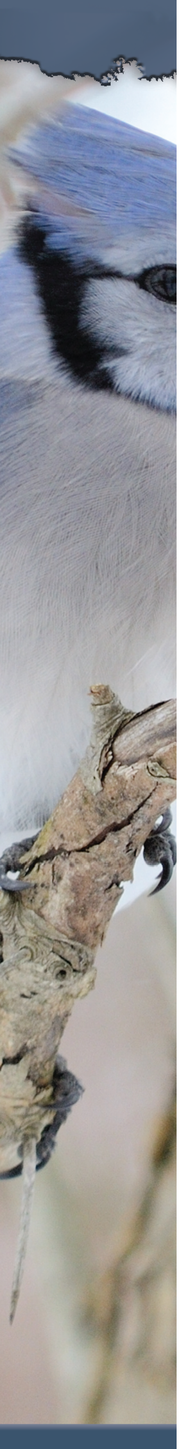
- Objects are often created and handed on elsewhere immediately:

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

- We don't really need `furtherLot`:

```
lots.add(new Lot(…));
```





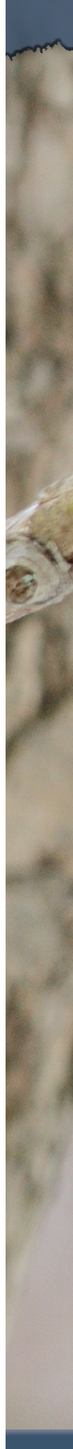
Chaining method calls

- 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

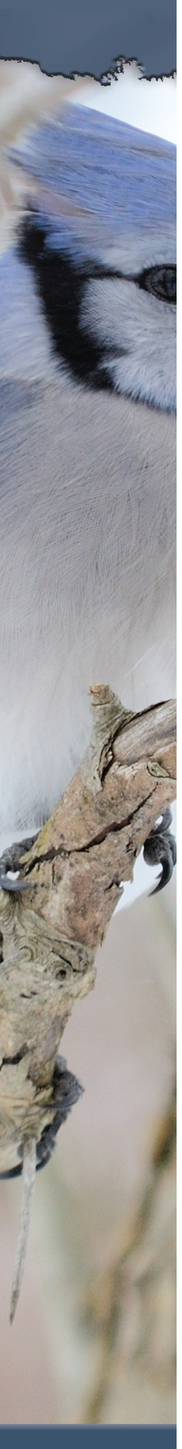
- Each method in the chain is called on the object returned from the previous method call in the chain.

```
String name =
    lot.getHighestBid().getBidder().getName();
```

Returns a `Bid` object from the `Lot`

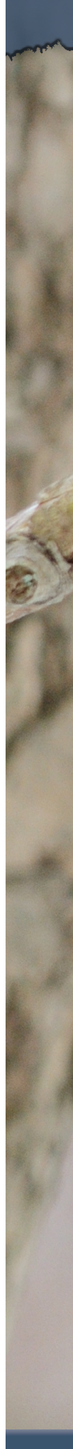
Returns a `Person` object from the `Bid`

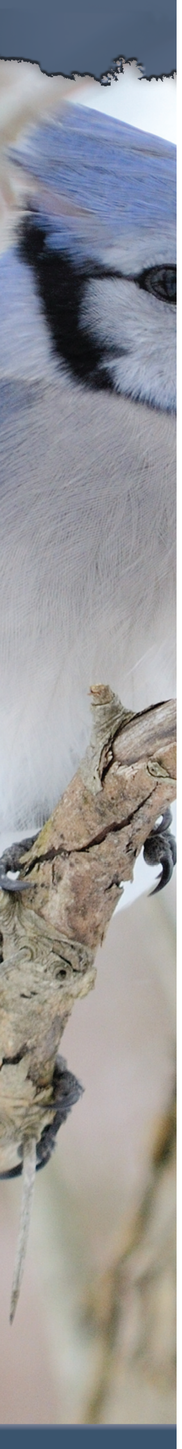
Returns a `String` object from the `Person`



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.





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.