Grouping objects
Part 3

The String Class,
Iterators and the Auction project

suggested reading:

Textbook, Ch. 4

Main concepts to be covered

- the String class
- Iterators
- The Auction project

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: String equality

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

tests identity

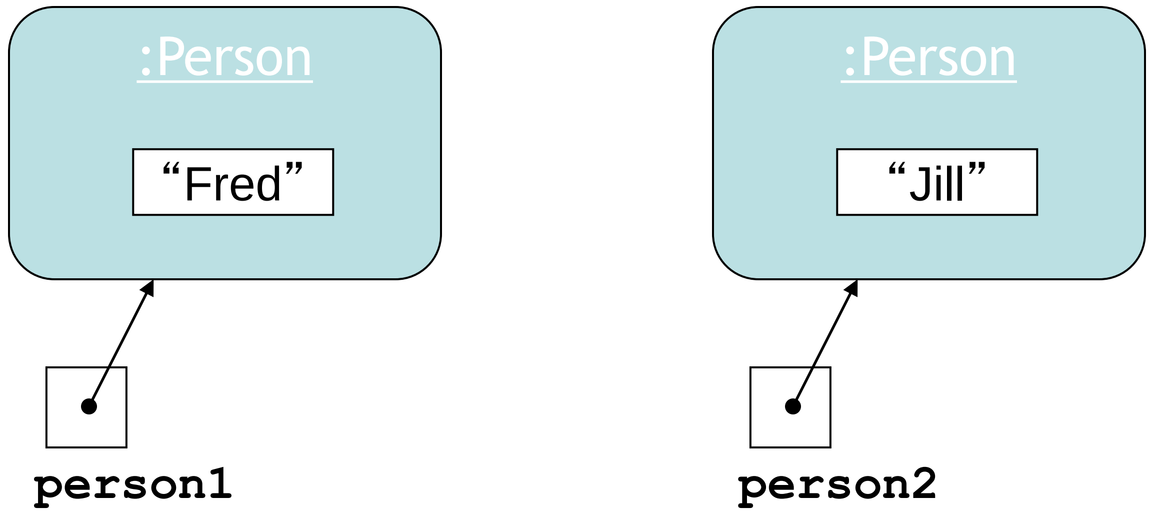
```
if(input.equals("bye")) {
    ...
}
```

tests equality

Always use `.equals` for text equality.

Identity vs equality 1

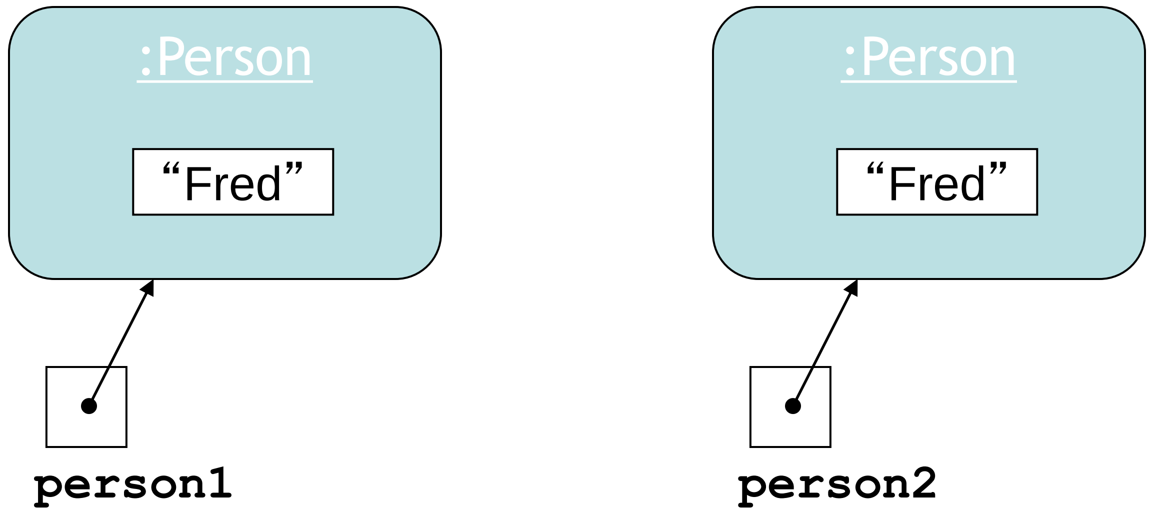
Other (non-String) objects:



`person1 == person2 ?`

Identity vs equality 2

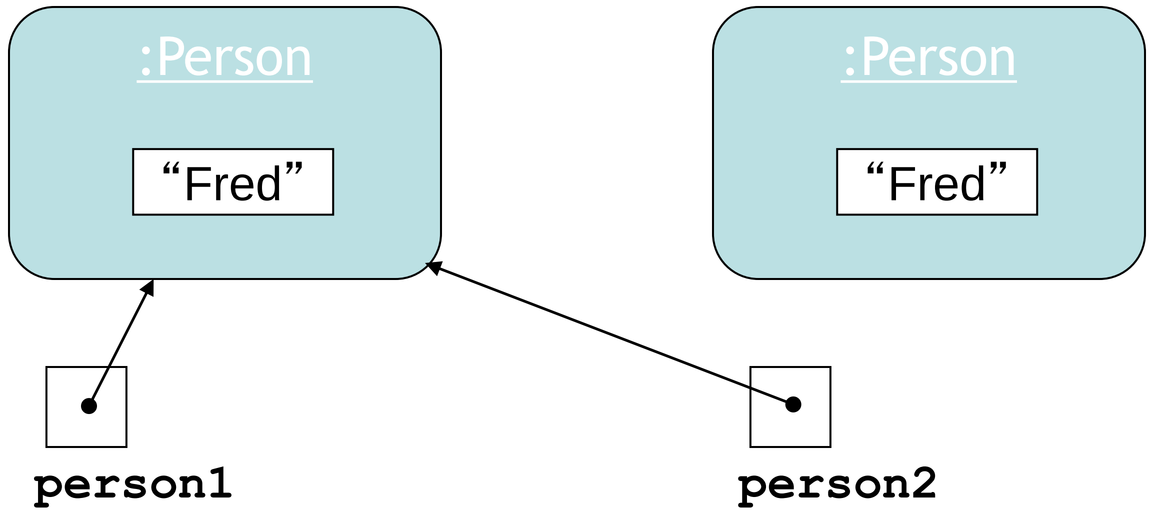
Other (non-String) objects:



`person1 == person2 ?`

Identity vs equality 3

Other (non-String) objects:

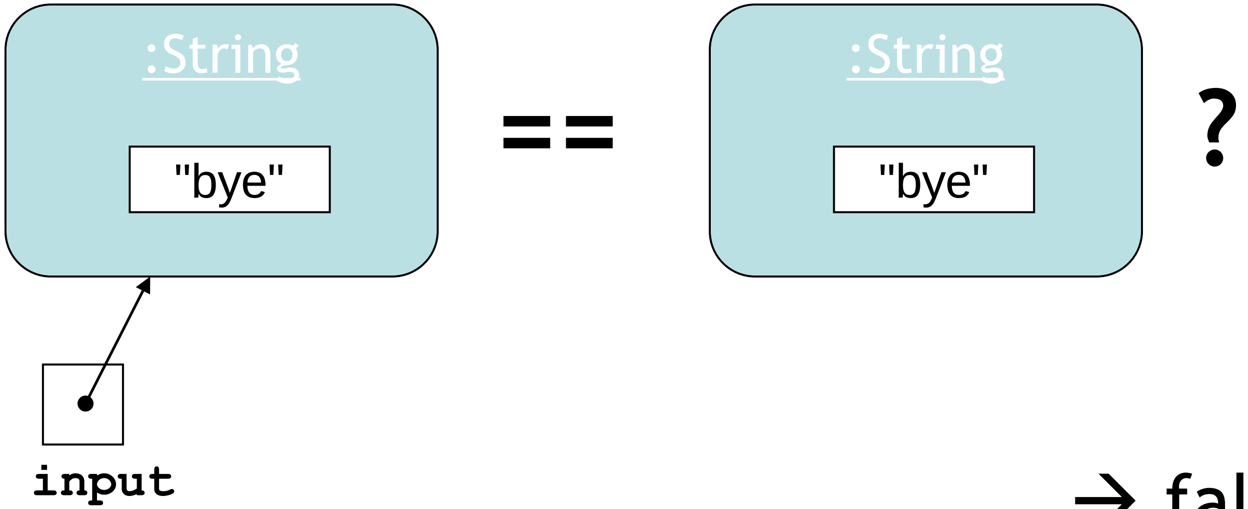


`person1 == person2 ?`

Identity vs equality (Strings)

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

== tests identity

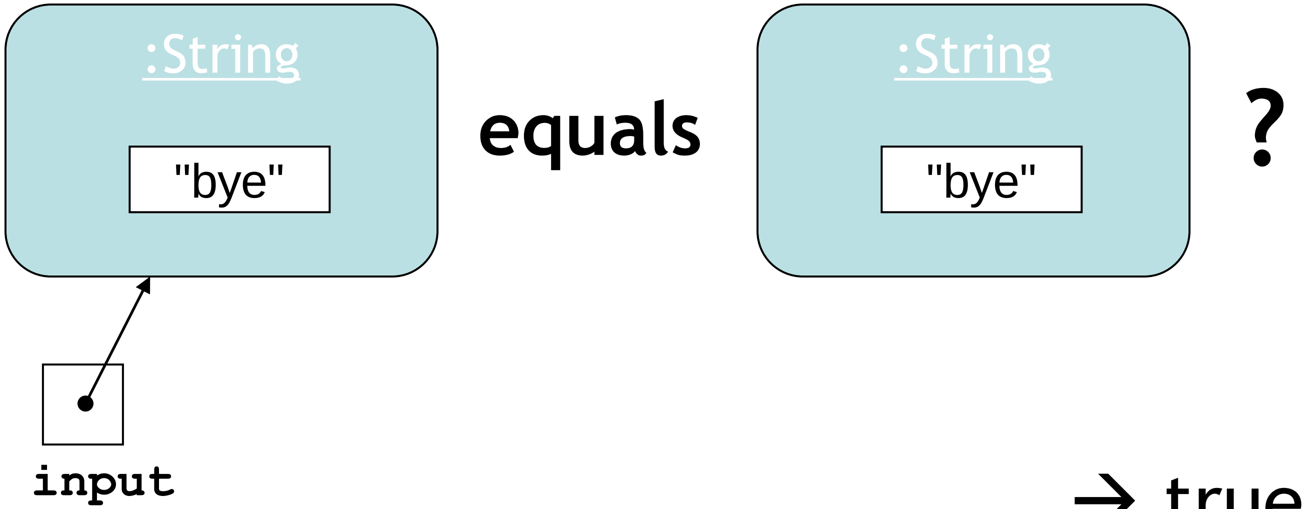


→ false!

Identity vs equality (Strings)

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

equals tests equality



→ true!

The problem with Strings

- 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 strings that arise outside the program's code;
  - e.g., from user input.

Grouping objects

Iterators

`Iterator` and `iterator()`

- Collections have an **`iterator()`** method.
- This returns an **`Iterator`** object.
- **`Iterator<E>`** has three methods:
  - **`boolean hasNext()`**
  - **`E next()`**
  - **`void remove()`**

Using an Iterator object

java.util.Iterator

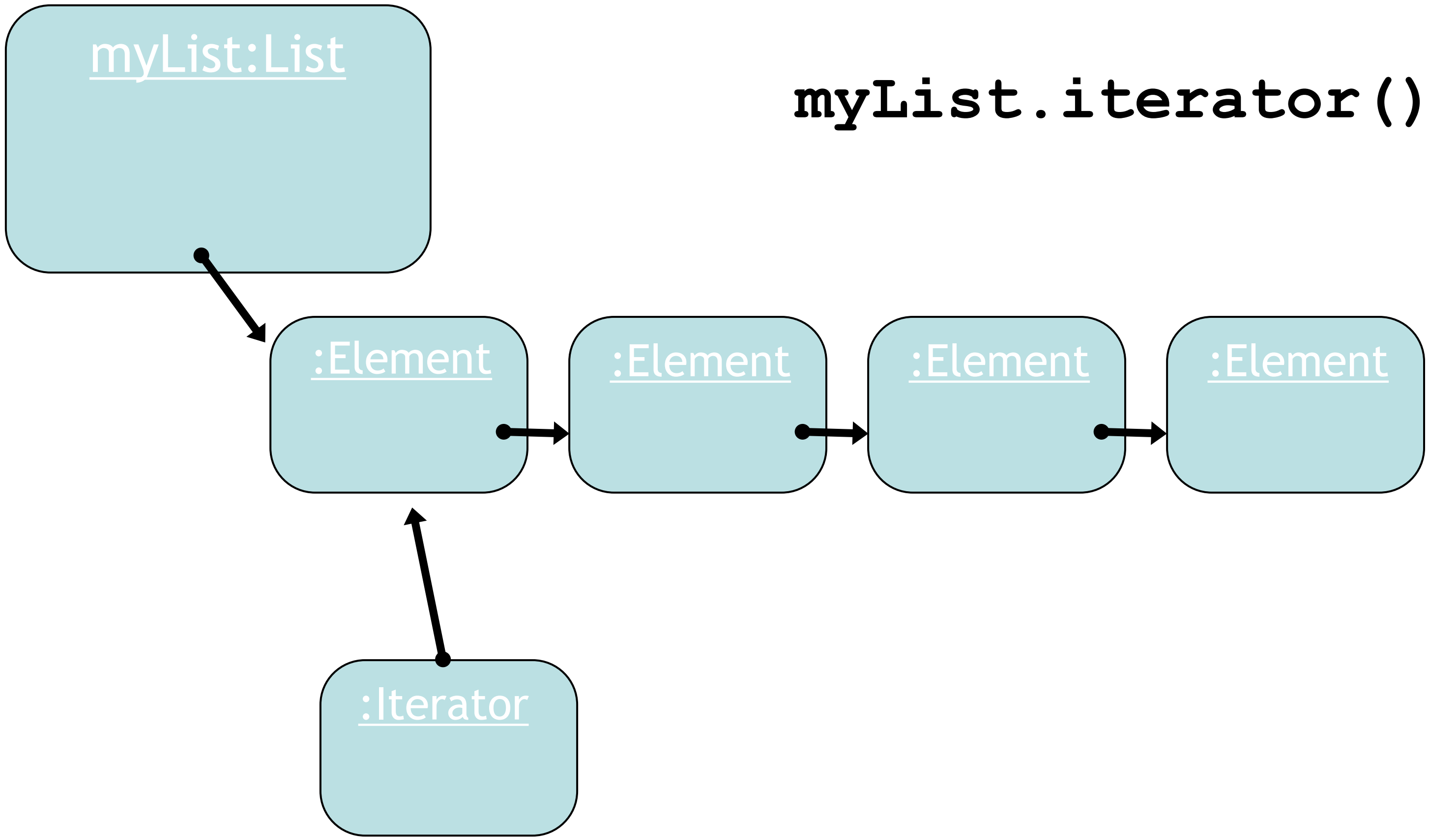
returns an Iterator object

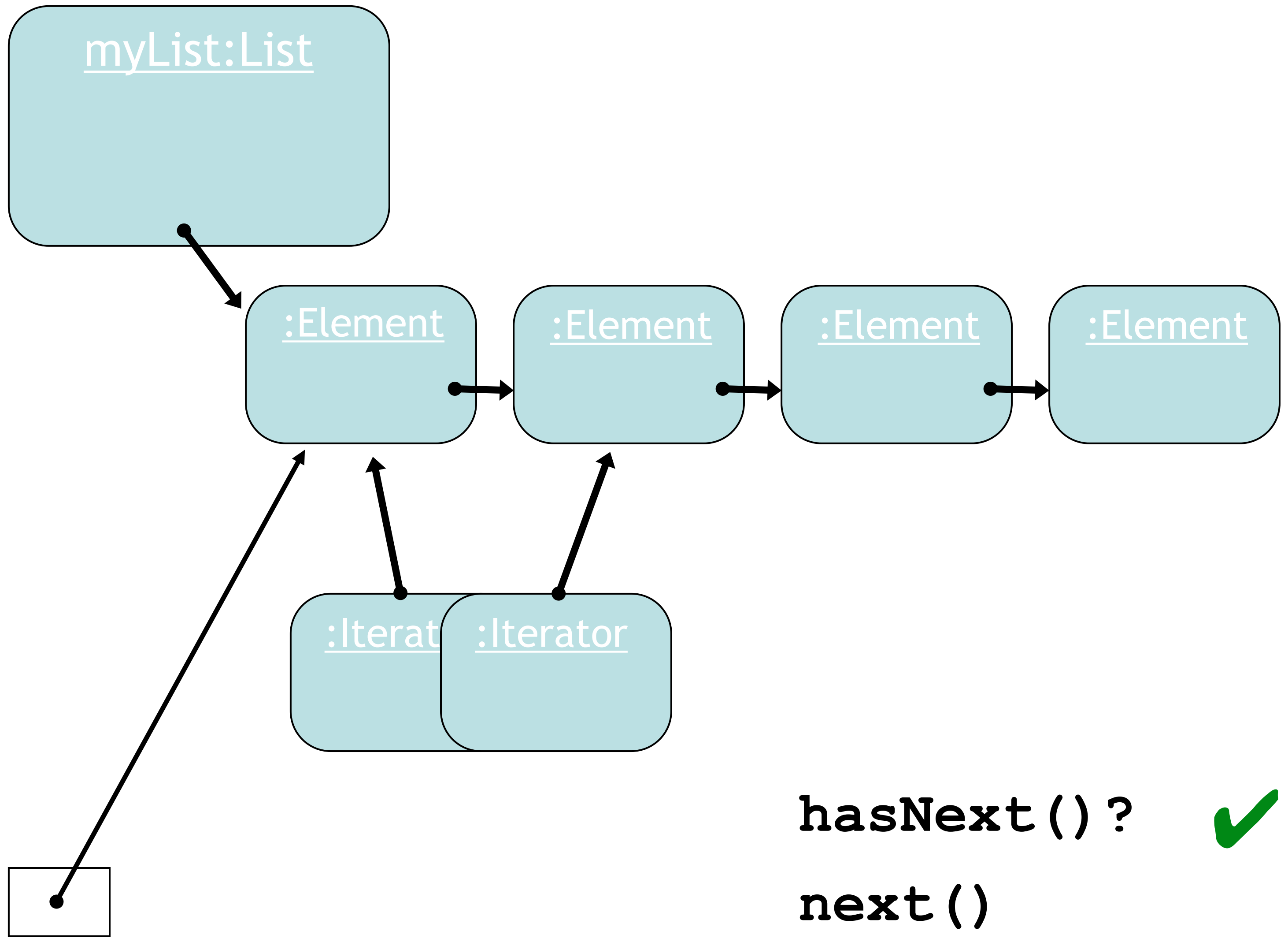
```
Iterator<ElementType> it = myCollection.iterator();
while(it.hasNext()) {
    call it.next() to get the next object
    do something with that object
}

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

Iterator mechanics

`myList.iterator()`

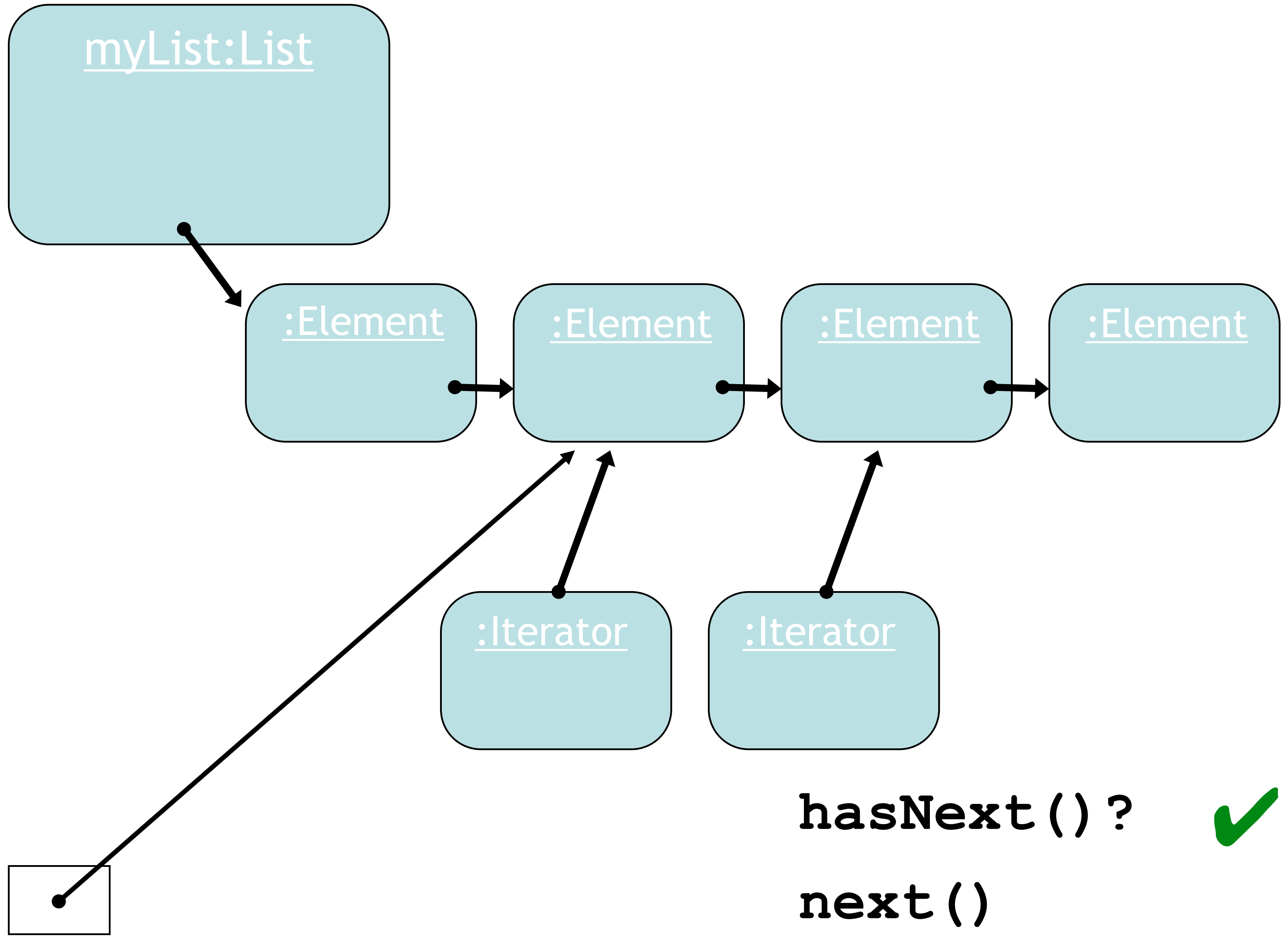




hasNext()? ✔

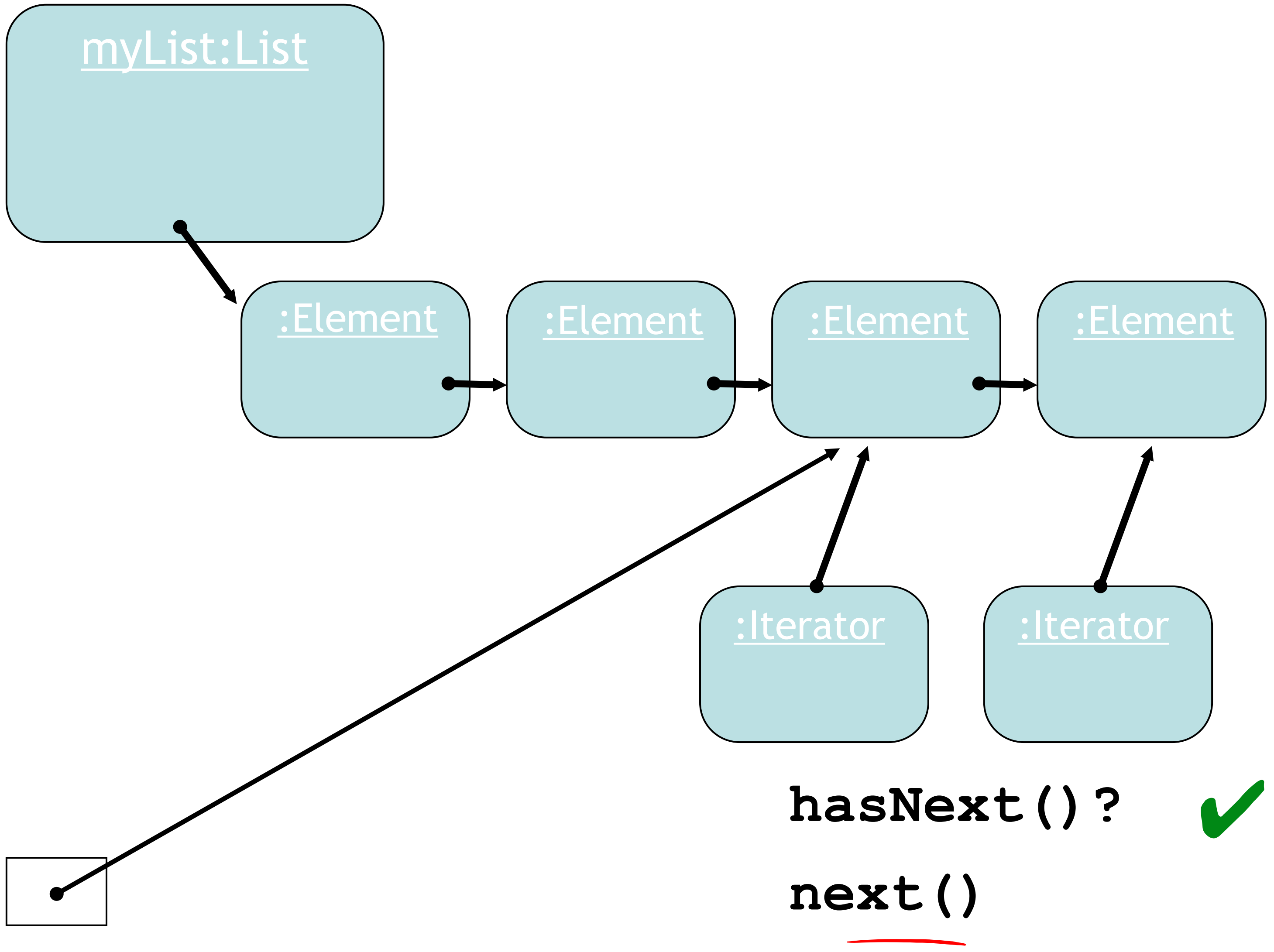
next()

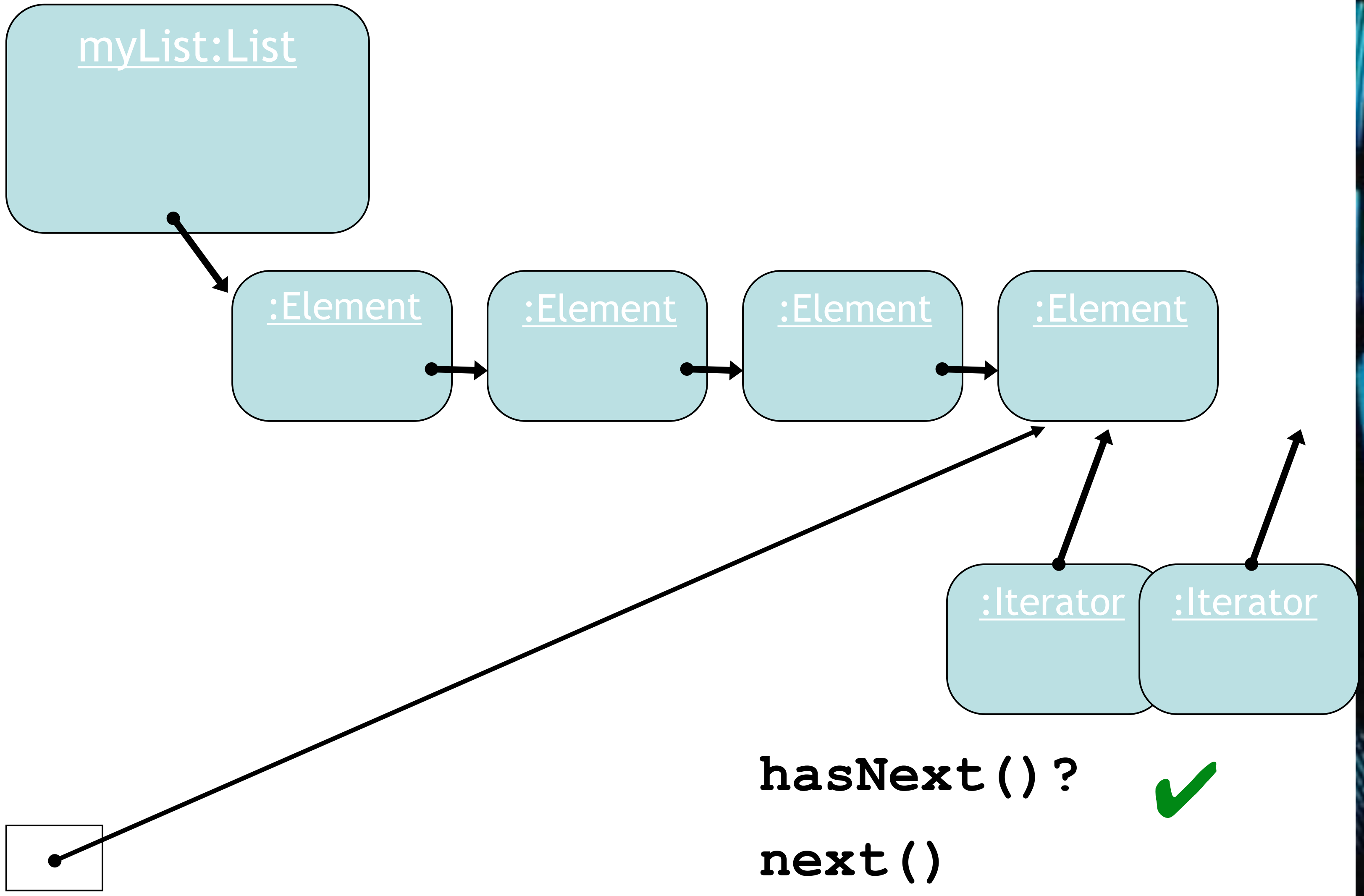
```
Element e = iterator.next();
```



hasNext()? ✓

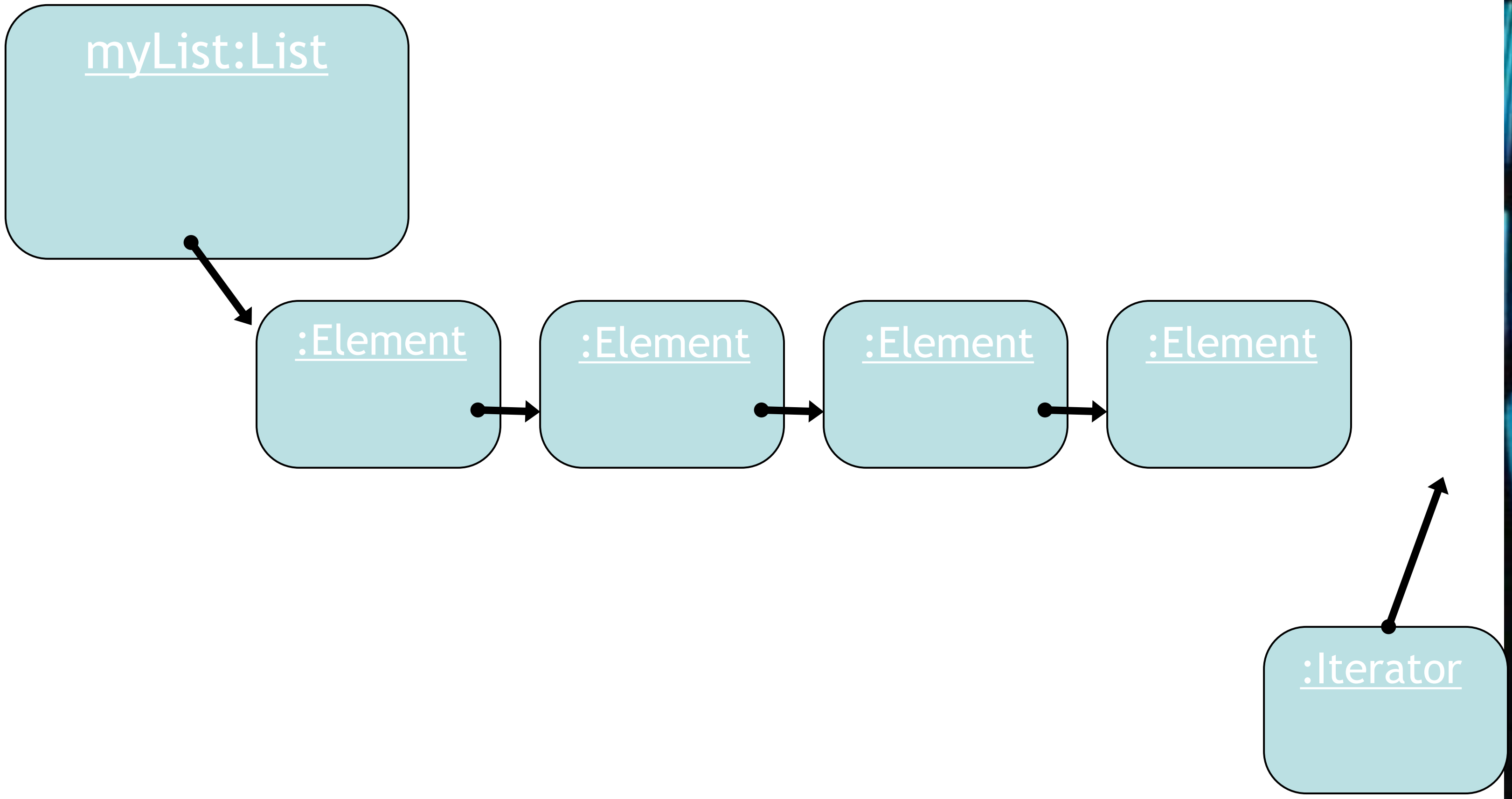
next()


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



hasNext()? ✓

next()



`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();
    }
}
```

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

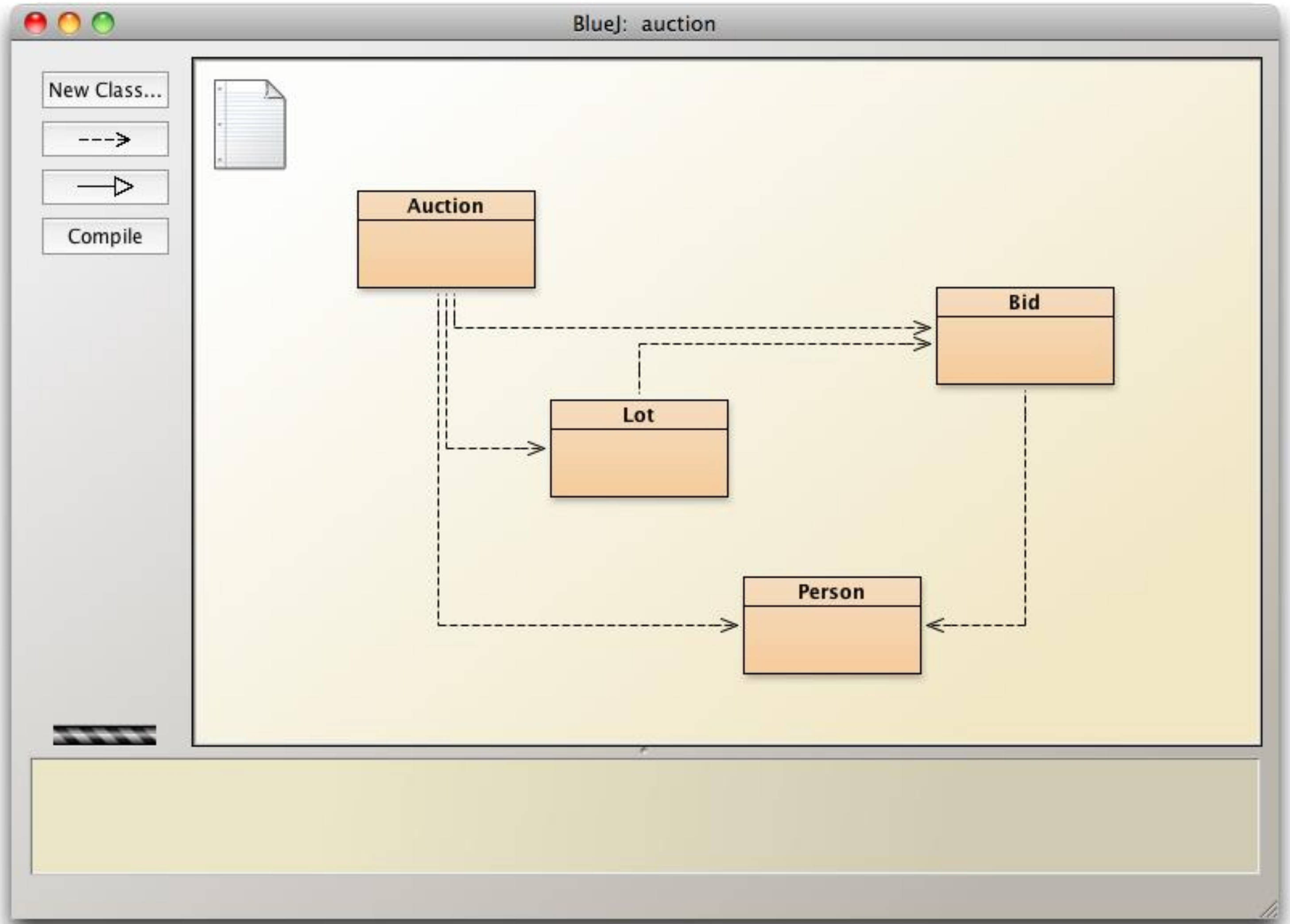
Review

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

The auction project



`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**