Grouping objects

Introduction to collections

6.0

Main concepts to be covered

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

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.

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 *musicorganizer.v1* project.

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.

```java
import java.util.ArrayList;

/**
 * ...
 */
class MusicOrganizer {
    // Storage for an arbitrary number of file names.
    private final ArrayList<String> files;

    /**
     * Perform any initialization required for the
     * organizer.
     */
    MusicOrganizer() {
        files = new ArrayList<>();
    }

    ...
}
```

Collections

- We specify:
  - the type of collection: **ArrayList**
  - the type of objects it will contain: **<String>**
  - **ArrayList<String> files;**
- We say, “ArrayList of String”.

Generic classes

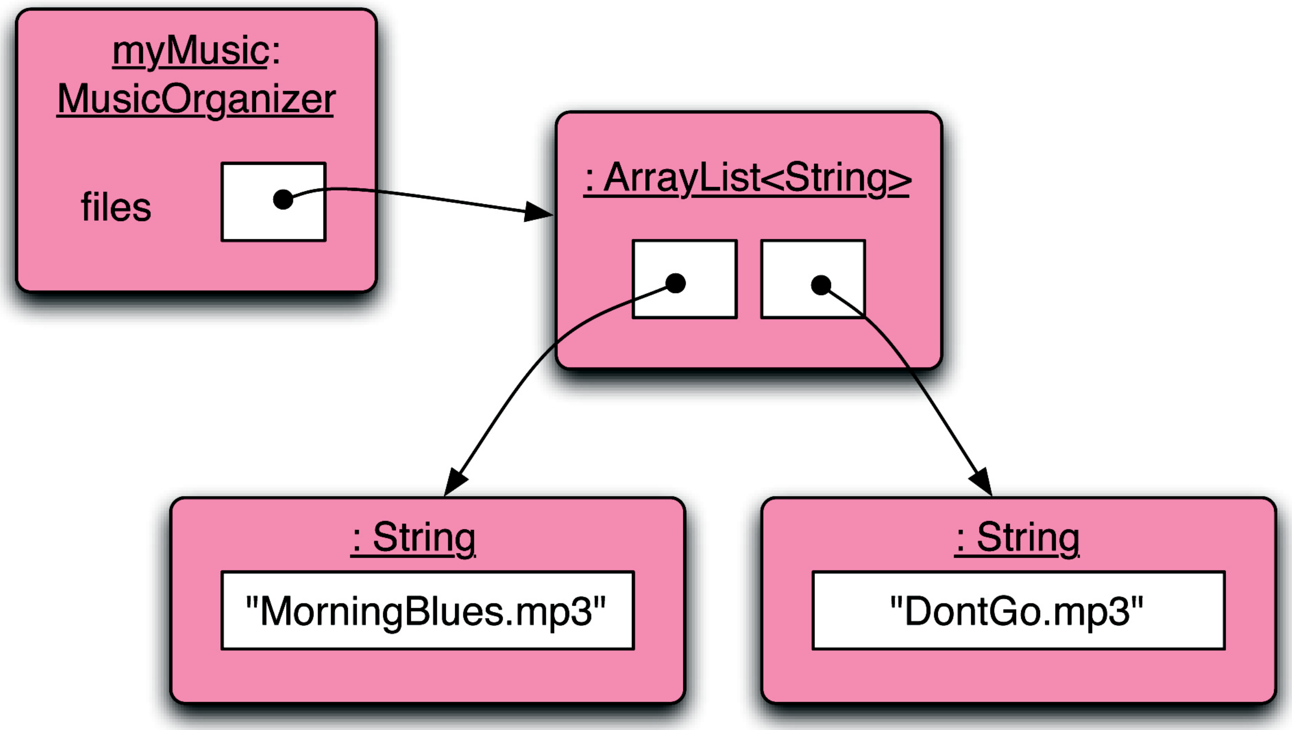
- Collections are known as *parameterized* or *generic* types.
- **`ArrayList`** implements list functionality:
  - **`add`**, **`get`**, **`size`**, etc.
- The type parameter says what we want a list of:
  - **`ArrayList<Person>`**
  - **`ArrayList<TicketMachine>`**
  - etc.

Creating an ArrayList object

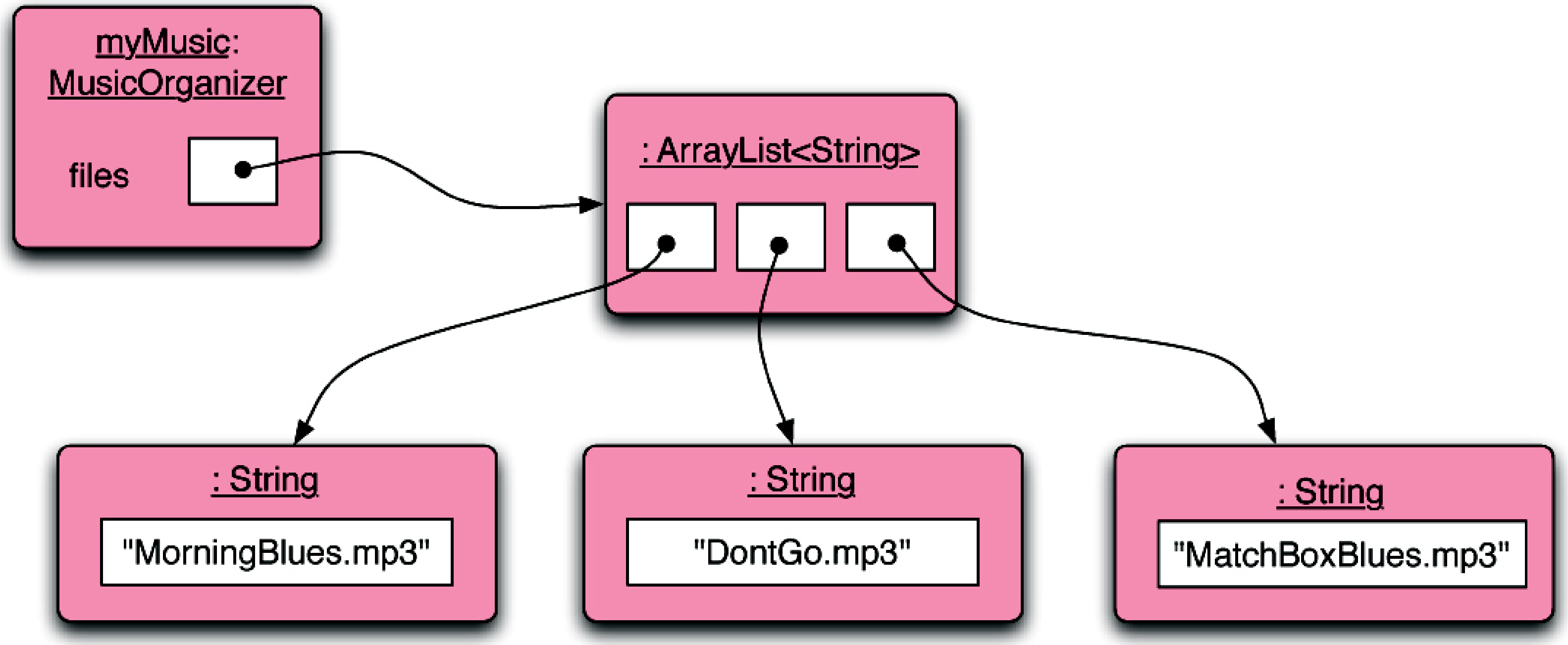
- The type parameter can be inferred from the variable being assigned to.

```
ArrayList<String> files = new ArrayList<>();
```

Object structures with collections



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 (encapsulated).
  - 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.

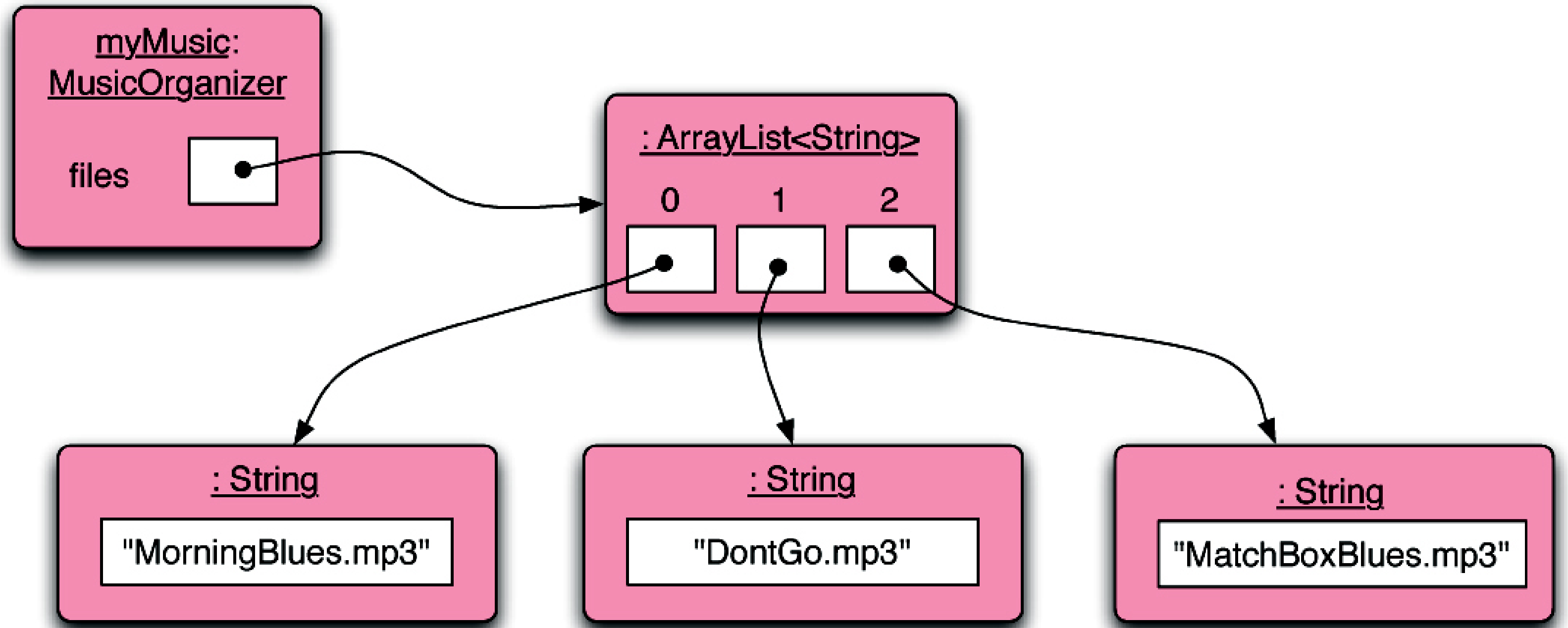
Using the collection

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

    ...

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

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

    ...
}
```

Adding a new file

Returning the number of files

Index numbering



Retrieving from the collection

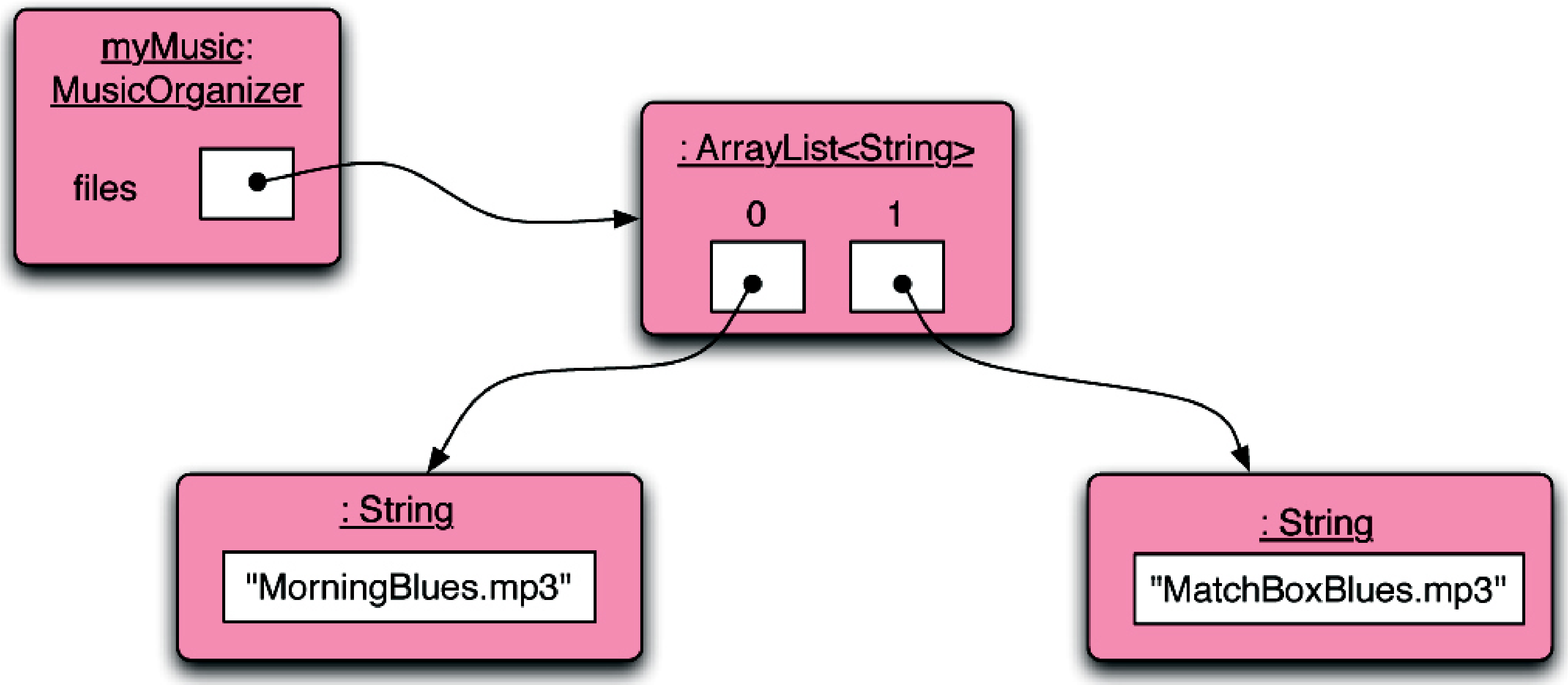
```
void listFile(int index) {
    if (index >= 0 && index < files.size()) {
        String filename = files.get(index);
        System.out.println(filename);
    } else {
        // This is not a valid index.
    }
}
```

Index validity checks

Needed? (Error message?)

Retrieve and print the file name

Removal may affect numbering



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, …

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.

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.

Interlude: Some 4-o’clock-in-the-morning errors…

What's wrong here?

```
/**
 * Print out info (number of entries).
 */
void showStatus() {
    if (files.size() == 0); {
        System.out.println("Organizer is empty");
    } else {
        System.out.print("Organizer holds ");
        System.out.println(files.size() + " files");
    }
}
```

What’s wrong here?

```
/**
 * Print out info (number of entries).
 */
void showStatus() {
   if (files.size() == 0); {
      System.out.println("Organizer is empty");
   } else {
      System.out.print("Organizer holds ");
      System.out.println(files.size() + " files");
   }
}
```

This is the same as before!

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

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

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

What's wrong here?

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

This time I have a boolean field called ‘isEmpty’...

What’s wrong here?

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

This time I have a boolean field called ‘isEmpty’...

The correct version

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

This time I have a boolean field called ‘isEmpty’...

Better version

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

What’s wrong here?

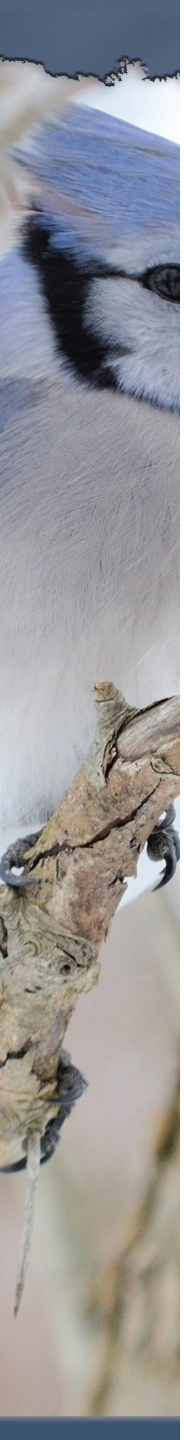
```
/**
 * Store a new file in the organizer. If the
 * organizer is full, save it and start a new one.
 */
void addFile(String filename) {
   if(files.size() == 100)
       files.save();
       files = new ArrayList<String>();
   files.add(filename);
}
```

This is the same.

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

The correct version

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



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*.

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*.

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

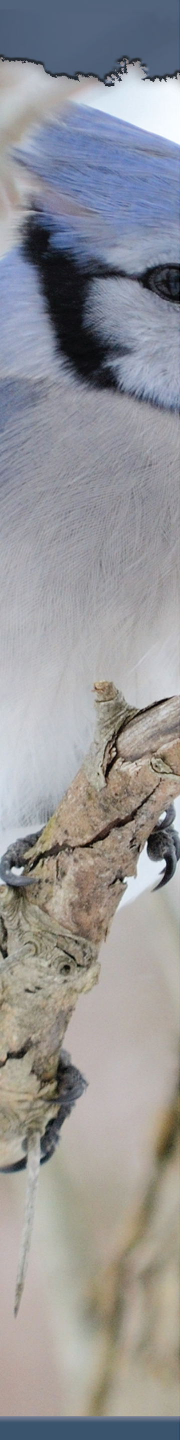
Pseudo-code expression of the operation of a for-each loop

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

A Java example

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

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



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.

Selective processing

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

```
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

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'.

Grouping objects

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

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

Looking for your keys

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

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.
 */
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`
  `!searching`

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
  !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
```

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.

When are Strings the same?

- The same object:
  - String me = "Fred";
  - String myself = "Fred";
- Two objects, same value:
  - String me = new String("Fred");
  - String myself = new String("Fred");
- Same question applies to objects more generally.

Side note: The problem

- The compiler merges identical `String` literals ("Fred") 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

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

tests value equality

Side note: String equality

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

tests identity

Do not use!!

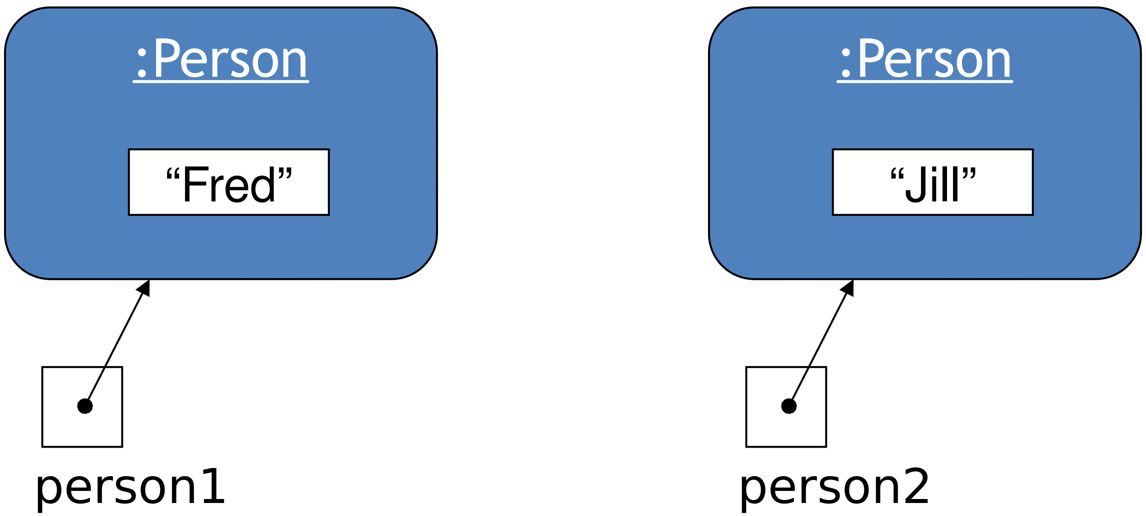
```
if (input.equals("bye")) {
    ...
}
```

tests value equality

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

Identity vs value equality

Other (non-String) objects:

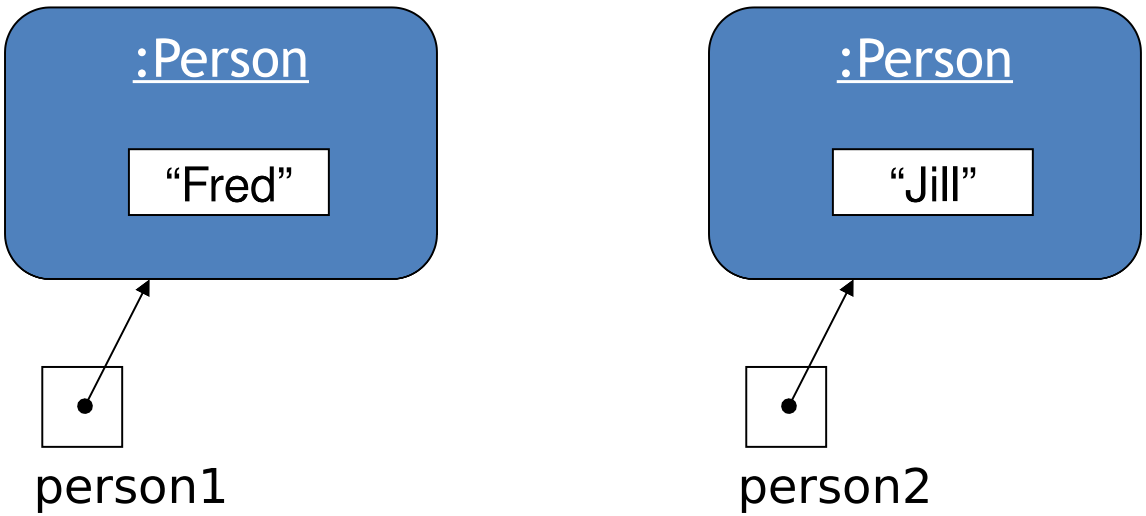


?

person1 == person2

Identity vs value equality

Other (non-String) objects:



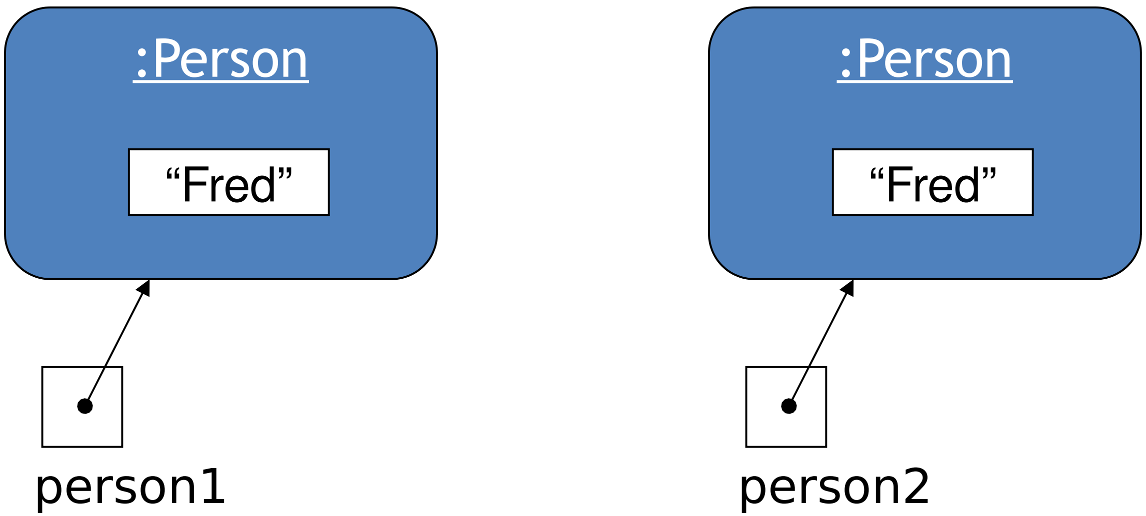
?

person1 == person2

`false`

Identity vs value equality

Other (non-String) objects:

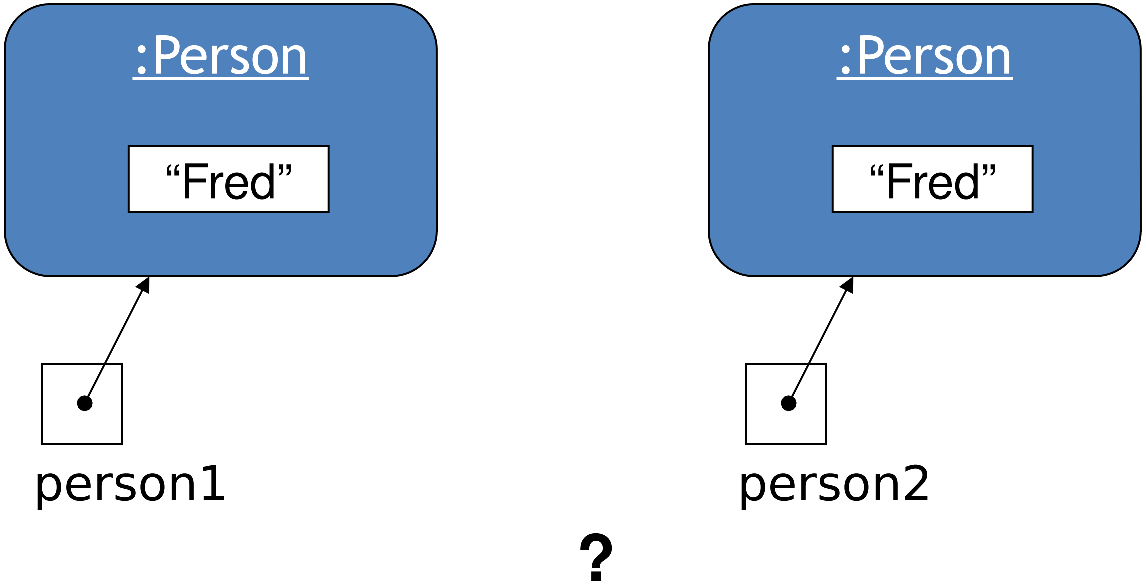


?

person1 == person2

Identity vs value equality

Other (non-String) objects:

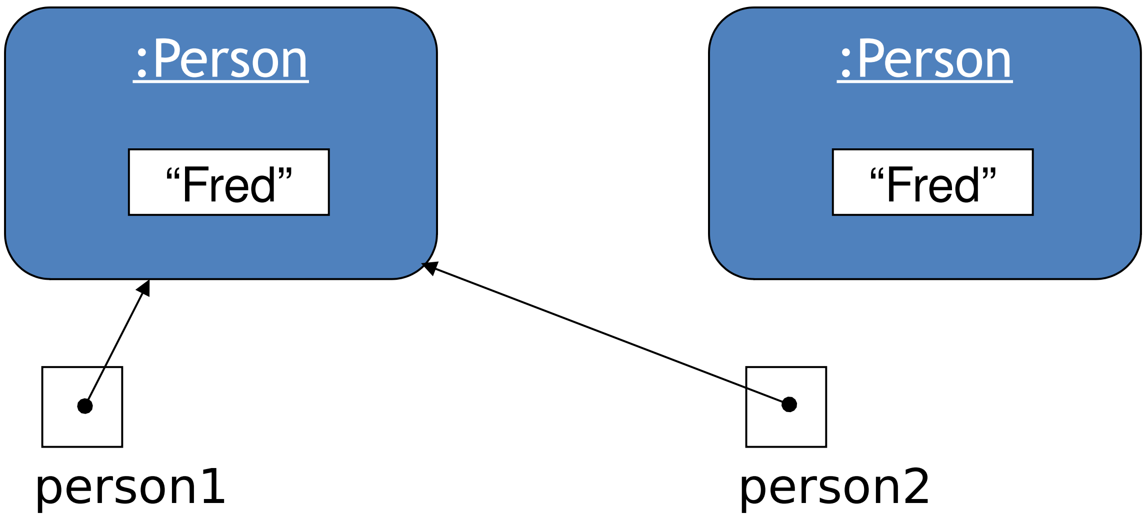


person1 == person2

false

Identity vs value equality

Other (non-String) objects:

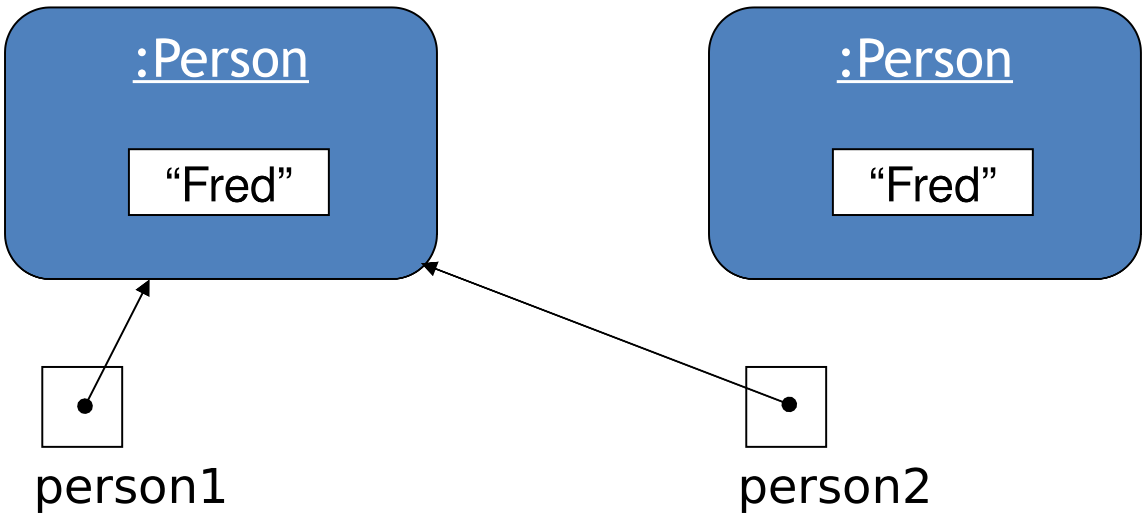


?

person1 == person2

Identity vs value equality

Other (non-String) objects:



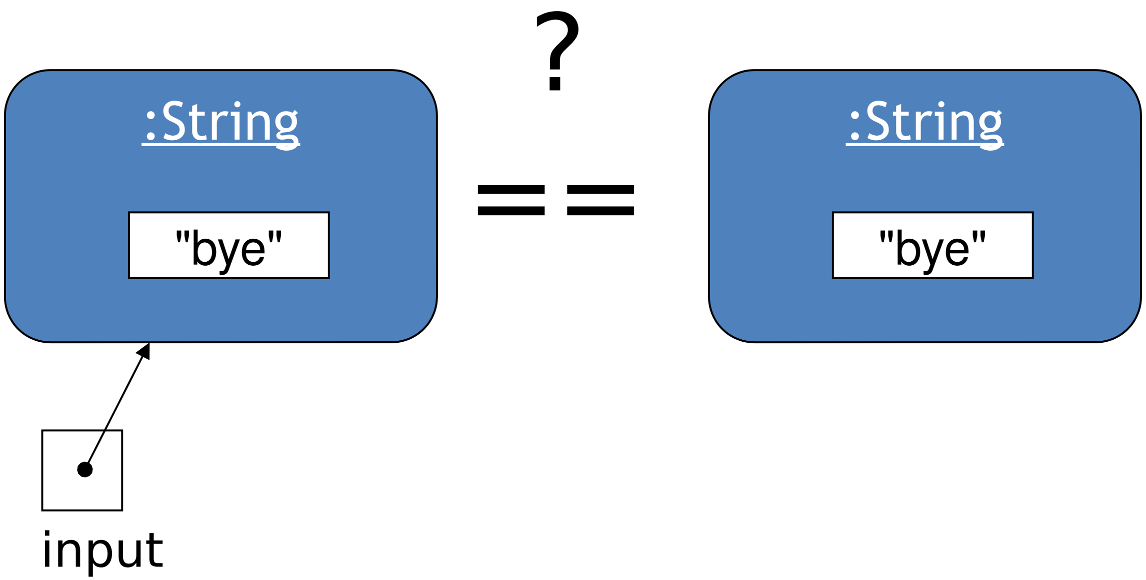
?

person1 == person2

`true`

Identity vs value equality (Strings)

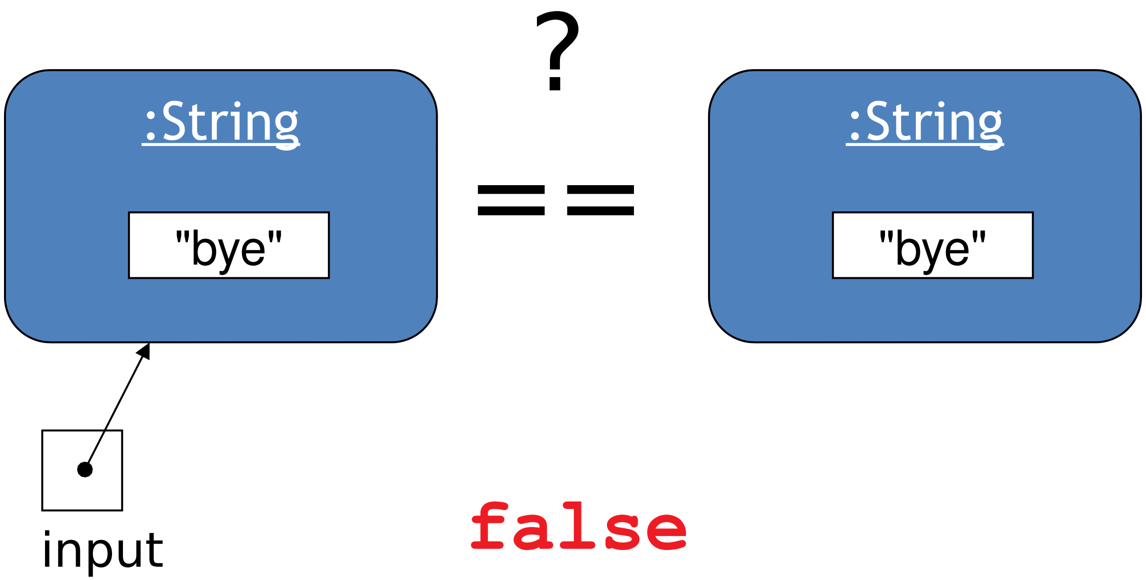
```
String input = reader.getInput();
if  (input == "bye") {
   ...
}
```



Identity vs value equality (Strings)

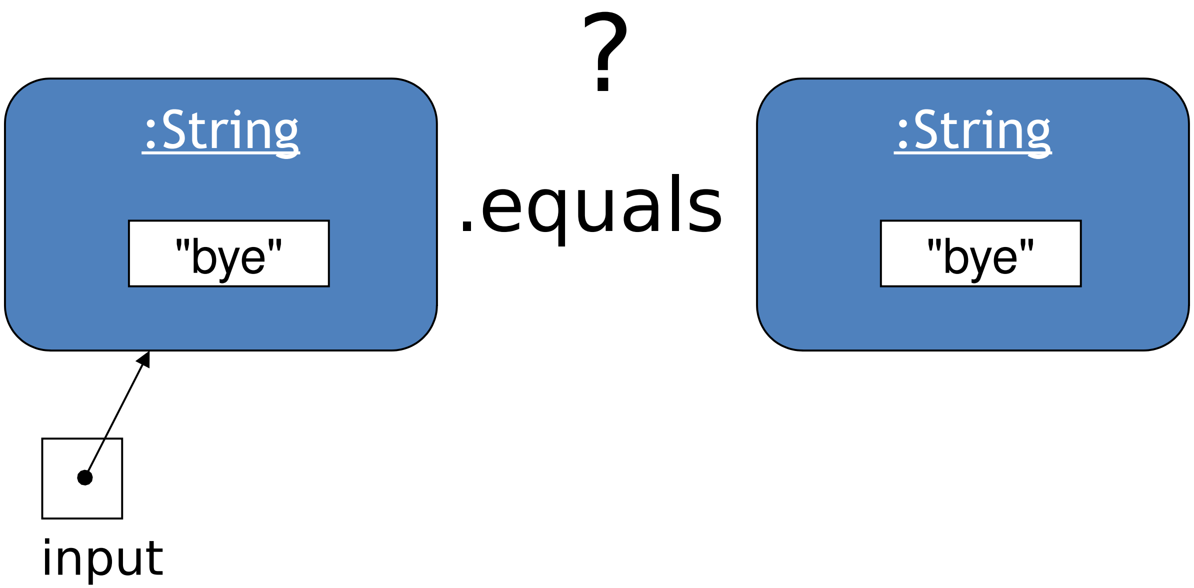
```
String input = reader.getInput();
if  (input == "bye") {
   ...
}
```

== tests identity



Identity vs value equality (Strings)

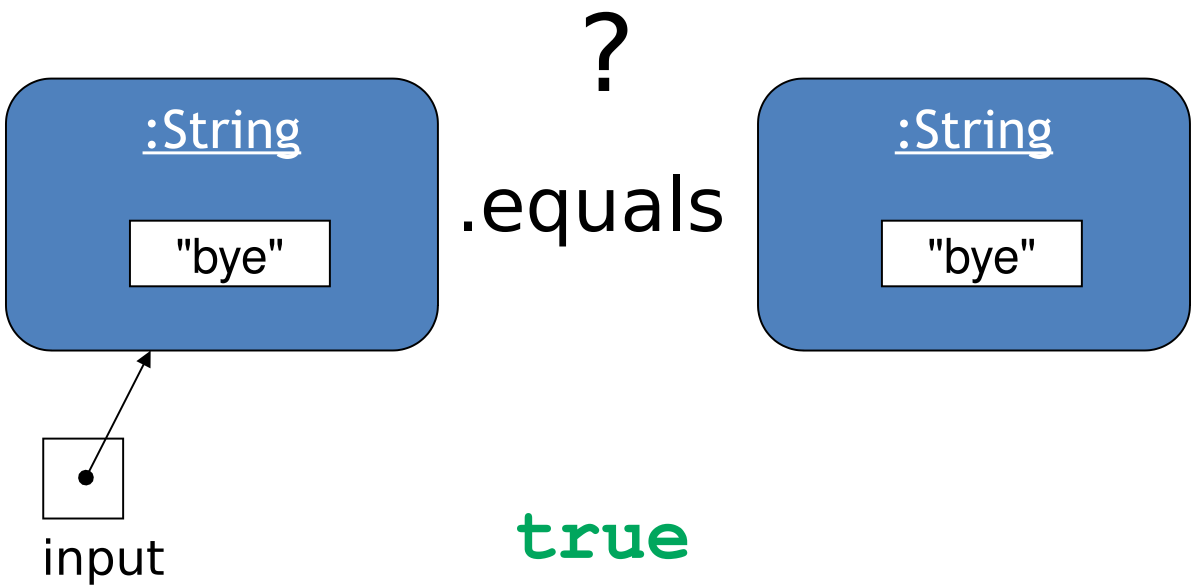
```
String input = reader.getInput();
if (input.equals("bye")) {
   ...
}
```



Identity vs value equality (Strings)

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

.equals tests value equality



Value equality for objects

- **`equals`** determines equivalence relation:
- Reflexive: **`x.equals(x)`** → **`true`**
- Symmetric: **`x.equals(y)`** → **`y.equals(x)`**

  Transitive: **`x.equals(y)`** & **`y.equals(z)`** → **`x.equals(z)`**
- Consistent: **`x.equals(y)`** always same result
- Must ensure all of the above when overriding* **`equals`**

* (Much) more on this later in the course

Value equality for objects

- Never override `equals` without also overriding `hashCode` (`equal`'s evil twin).
- Otherwise collections may not work properly.
- Consistent:
- `hashCode(x)` → always same `int`
- `x.equals(y)` → `true`
  `=> hashCode(x) == hashCode(y)`
- Never do

```
int hashCode() {return 42;}
```

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

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`

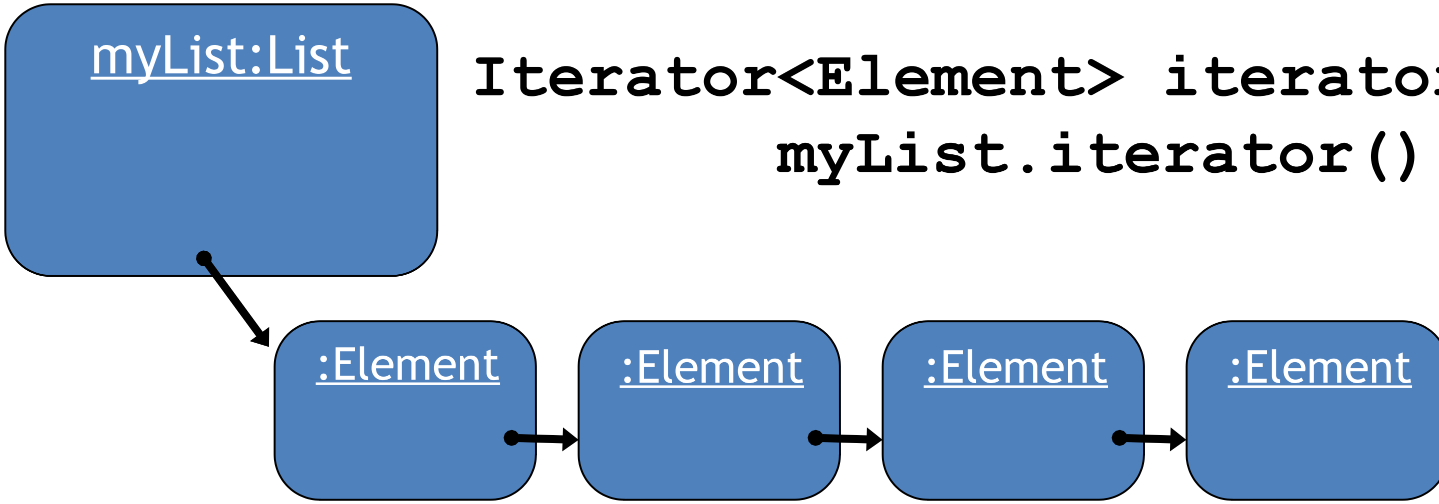
returns an `Iterator` object

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

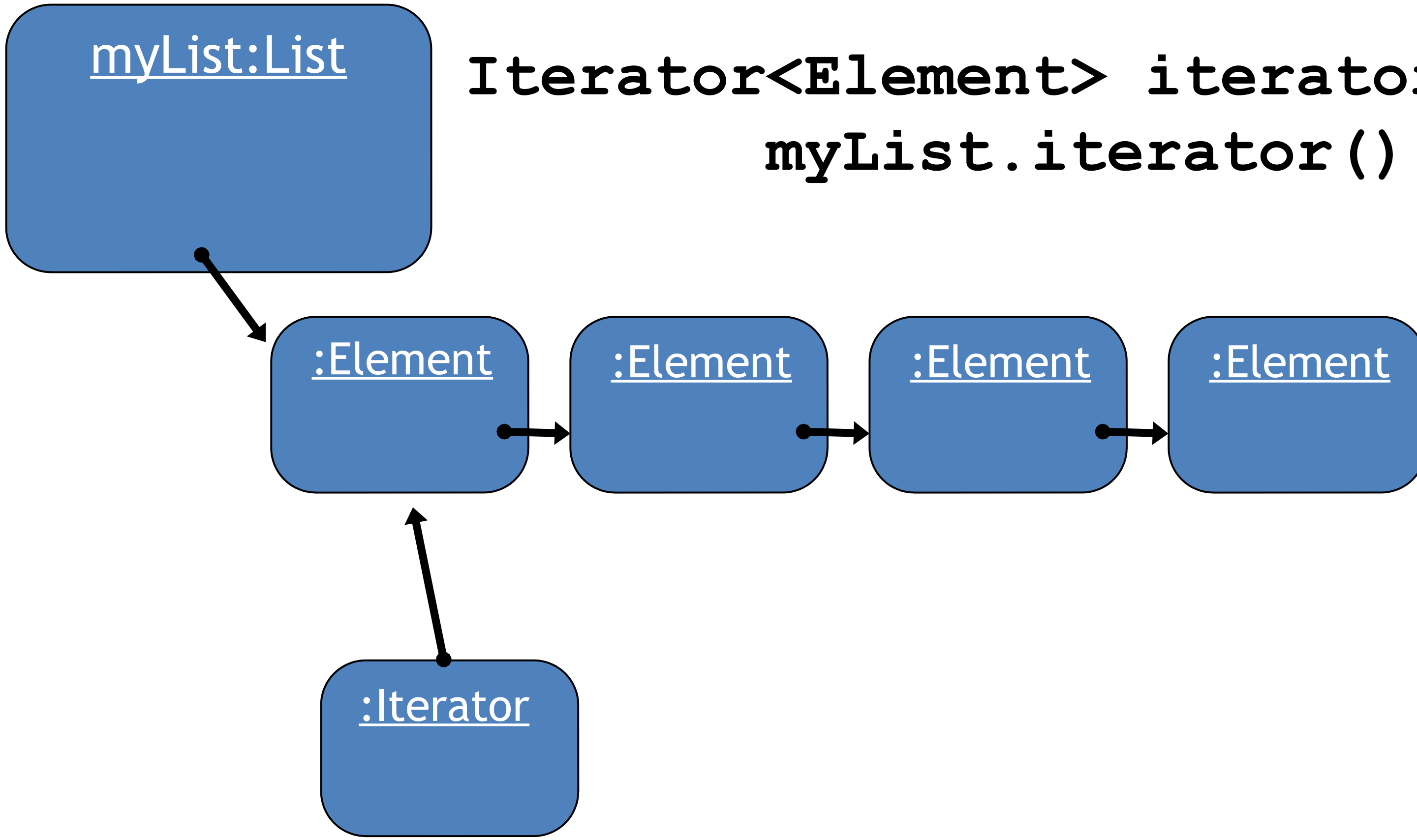
```
public void listAllFiles() {
    Iterator<Track> it = files.iterator();
    while (it.hasNext()) {
        Track tk = it.next();
        System.out.println(tk.getDetails());
    }
}
```

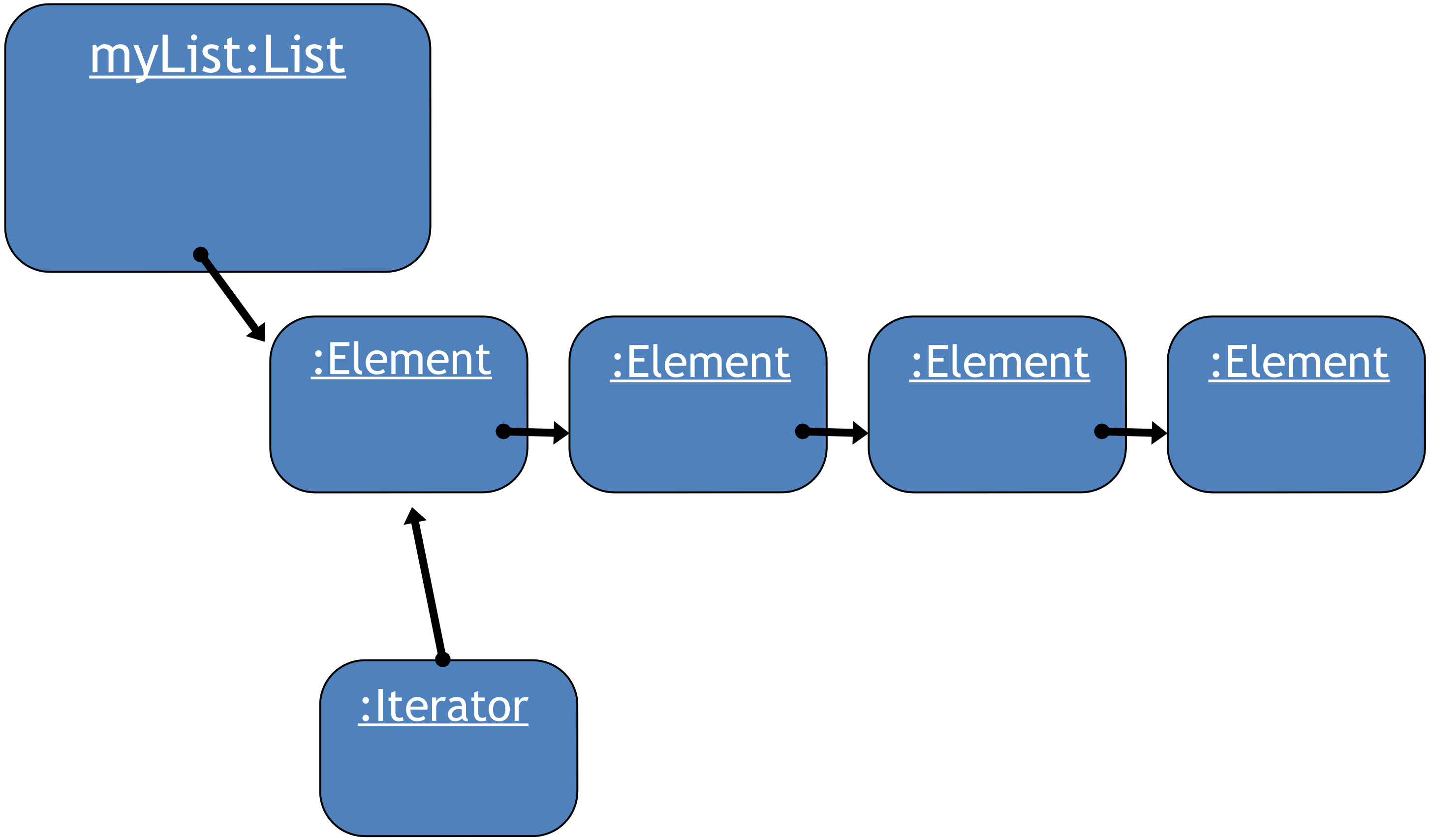
Iterator mechanics

```
Iterator<Element> iterator =
        myList.iterator();
```

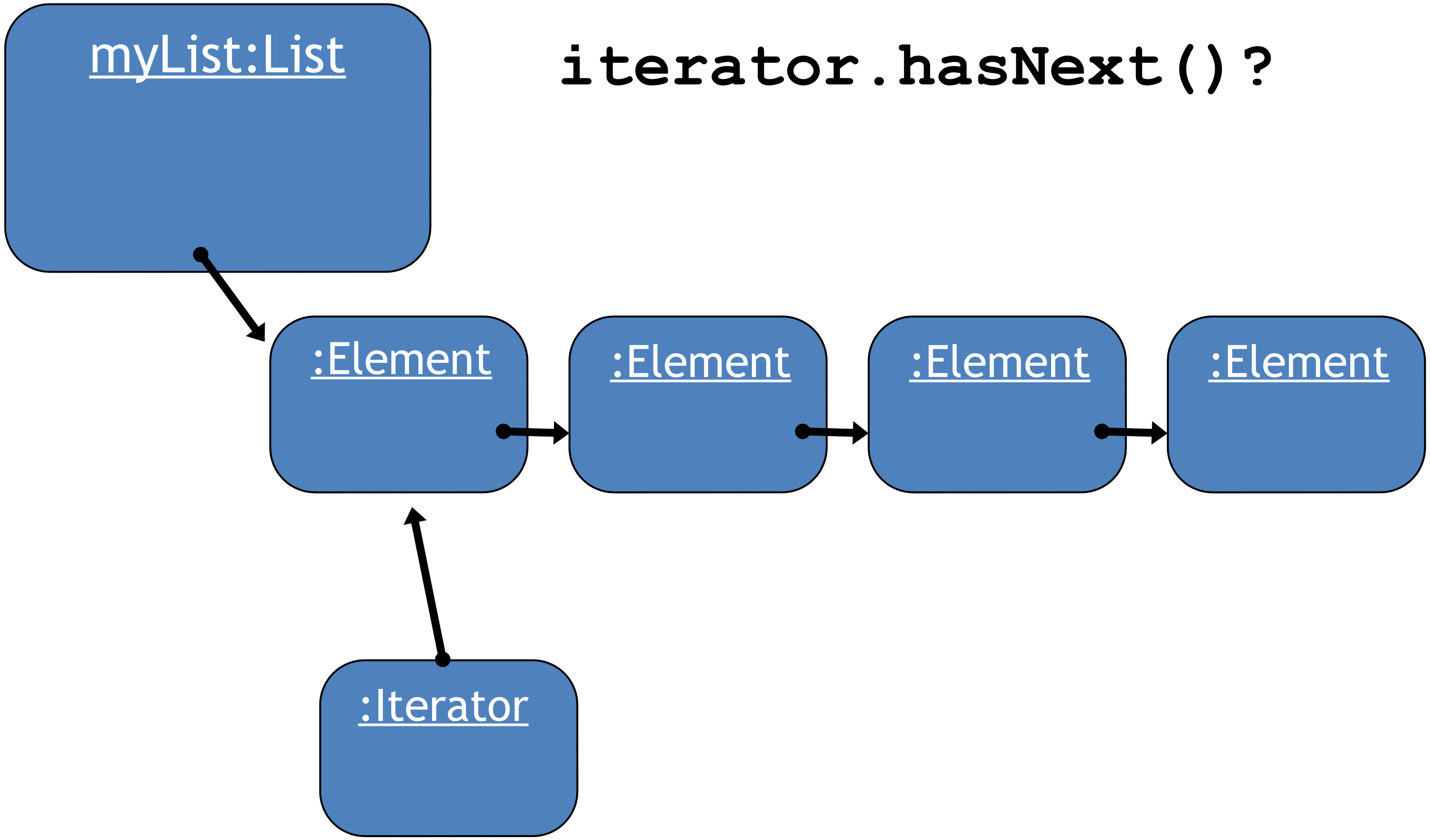


```
Iterator<Element> iterator =
          myList.iterator();
```

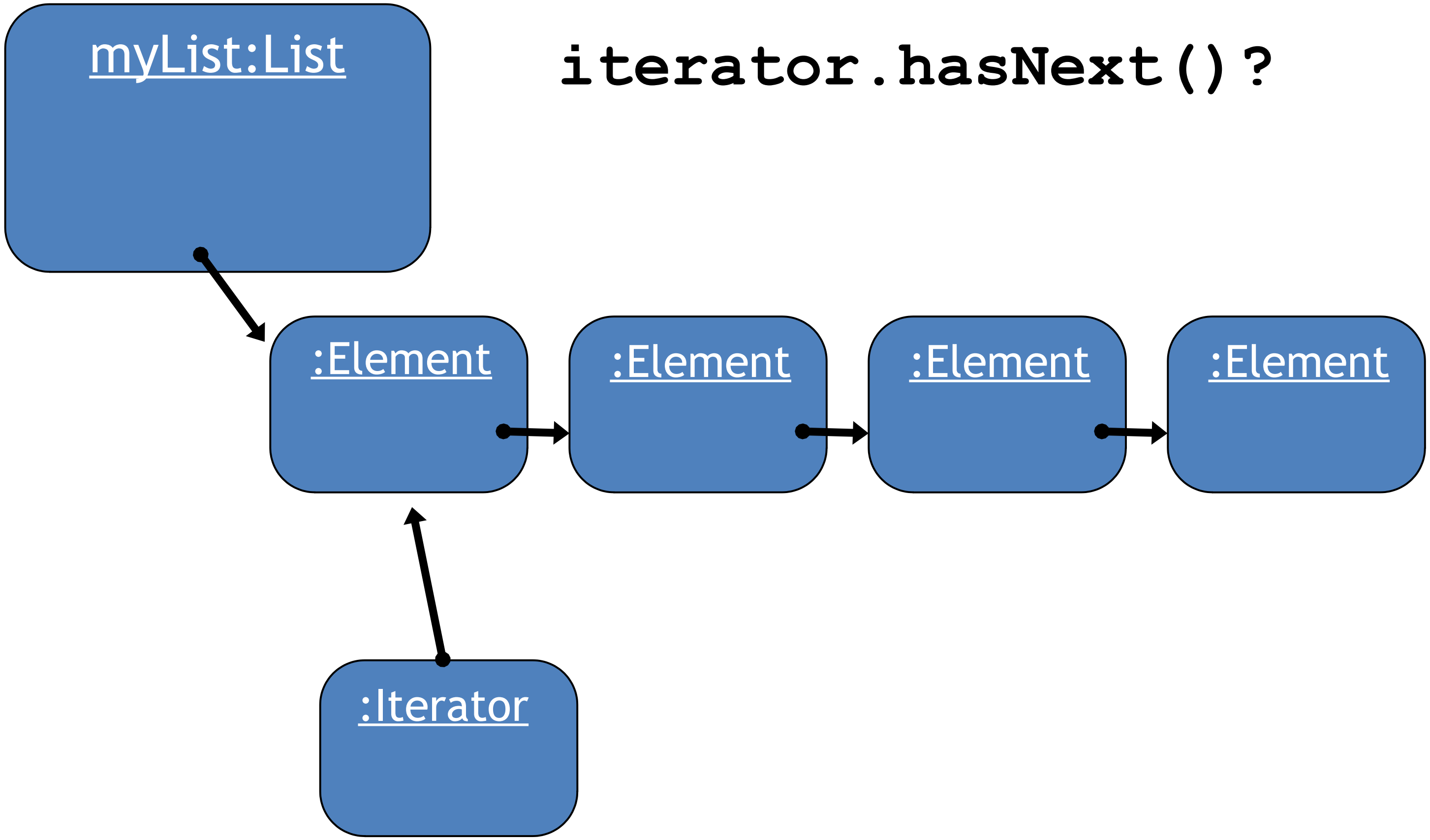



myList:List
:Element
:Element
:Element
:Element
:Iterator

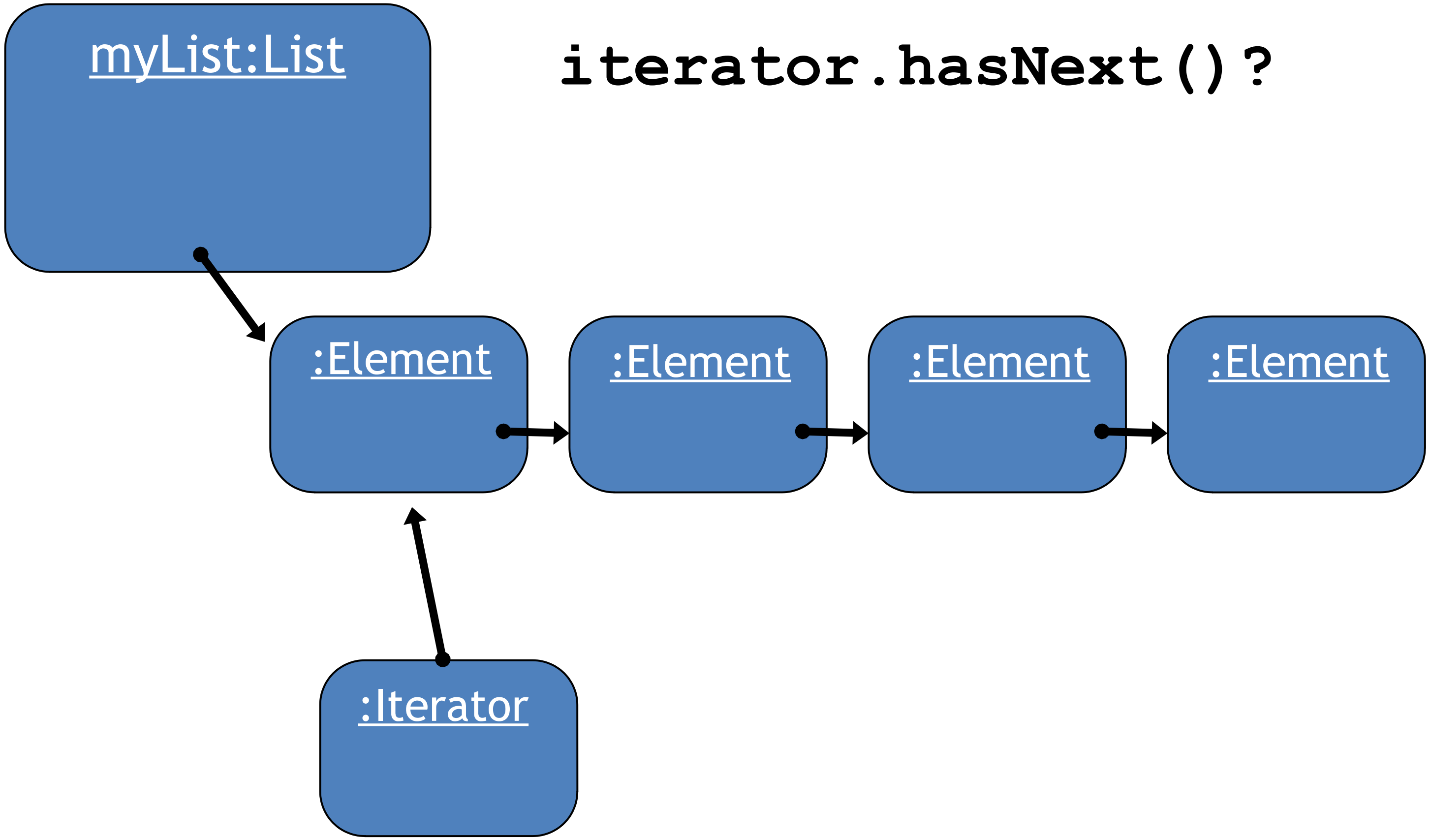
`iterator.hasNext()?`



`iterator.hasNext()?`

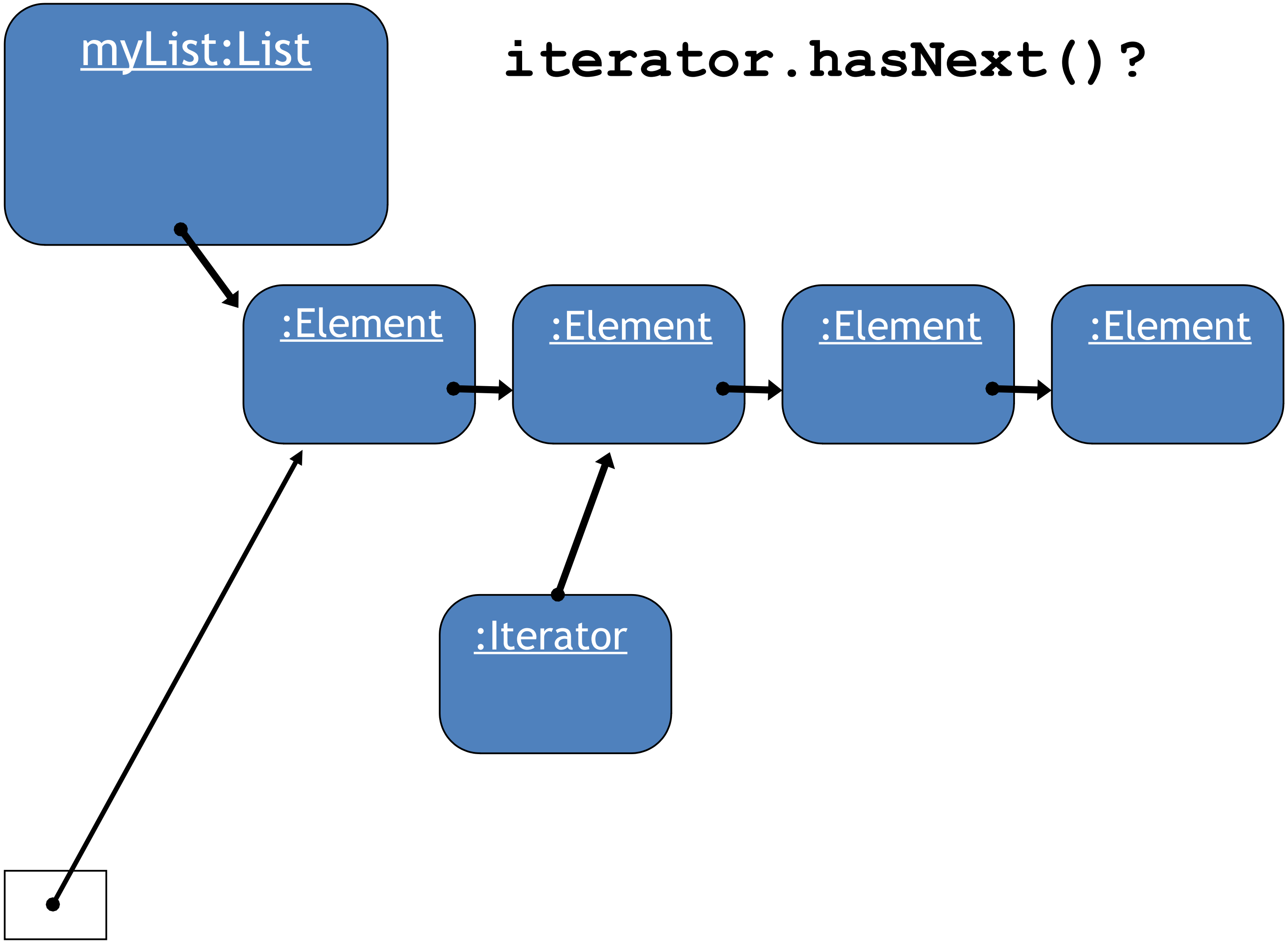


`iterator.hasNext()?`

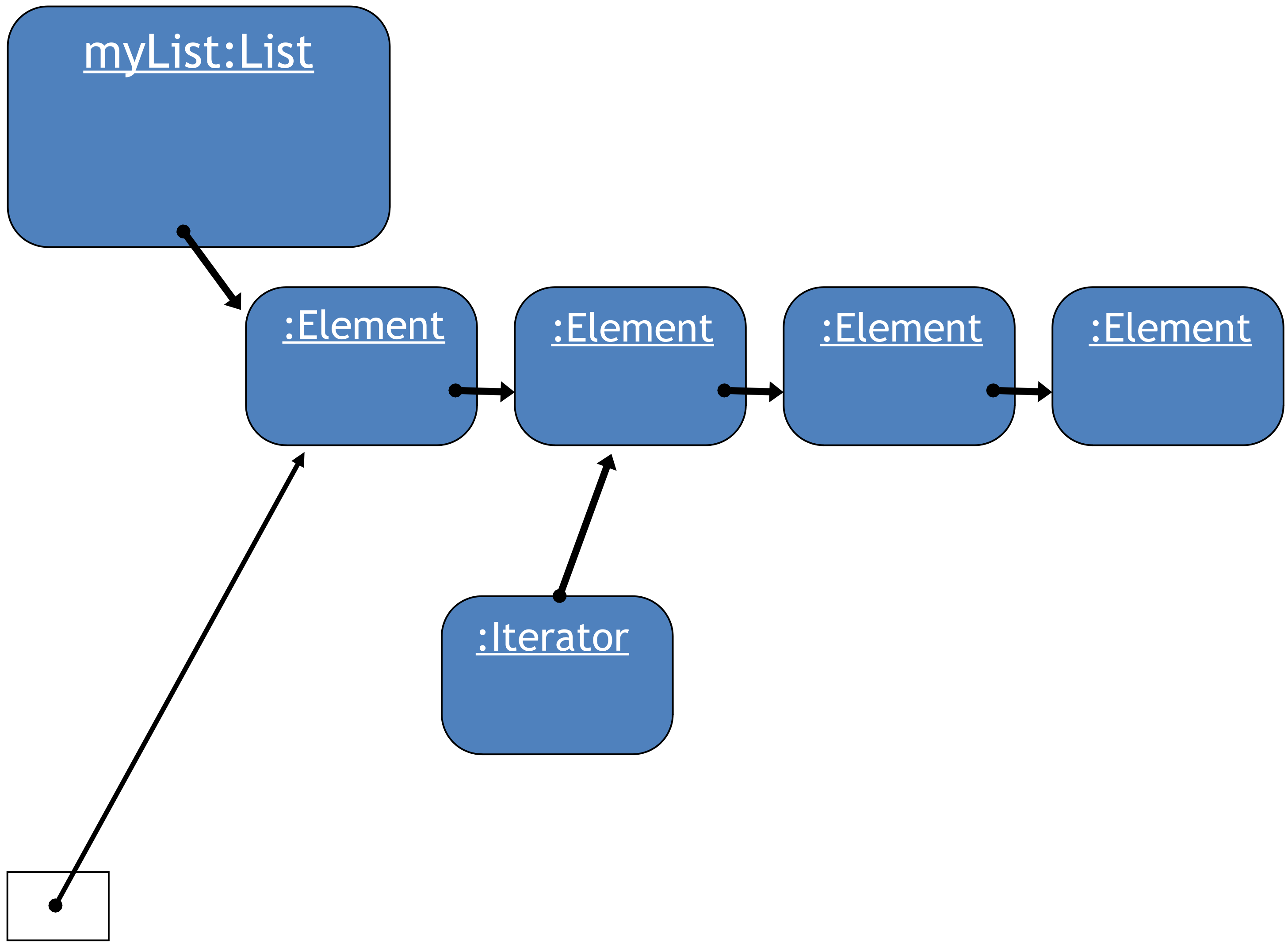



```
Element e = iterator.next();
```

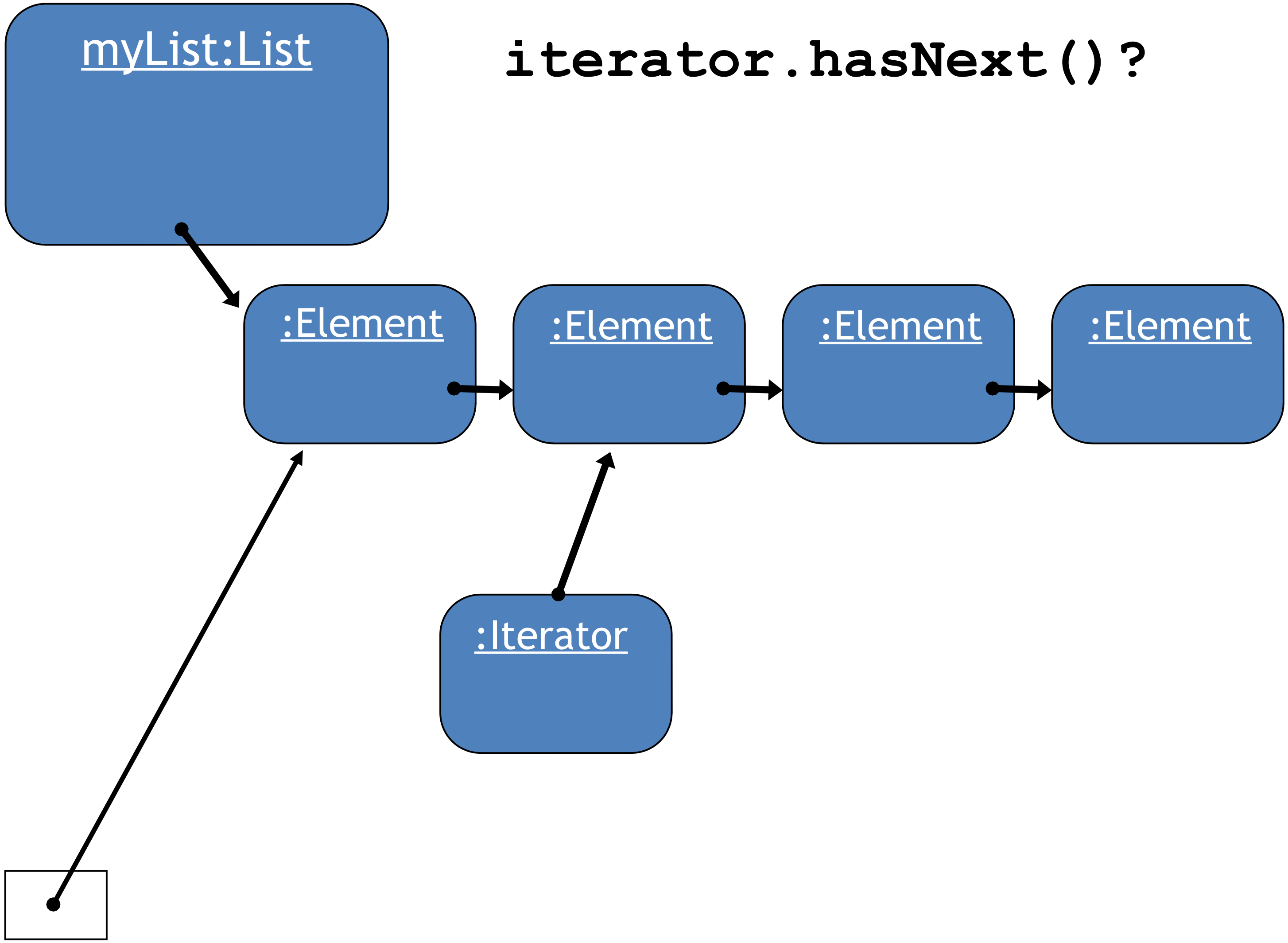
`iterator.hasNext()?`



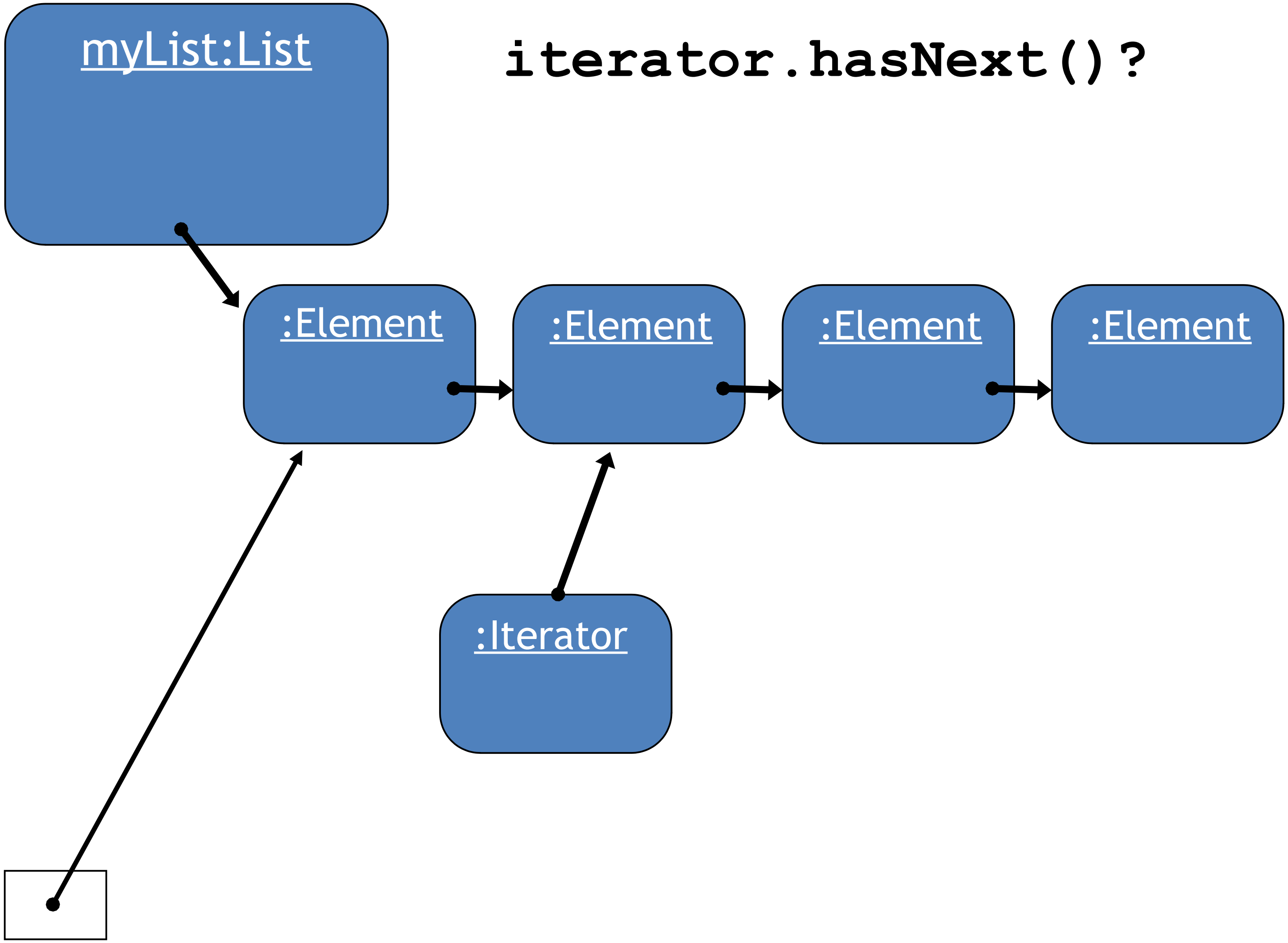
`Element e = iterator.next();`


myList:List
:Element
:Element
:Element
:Element
:Iterator

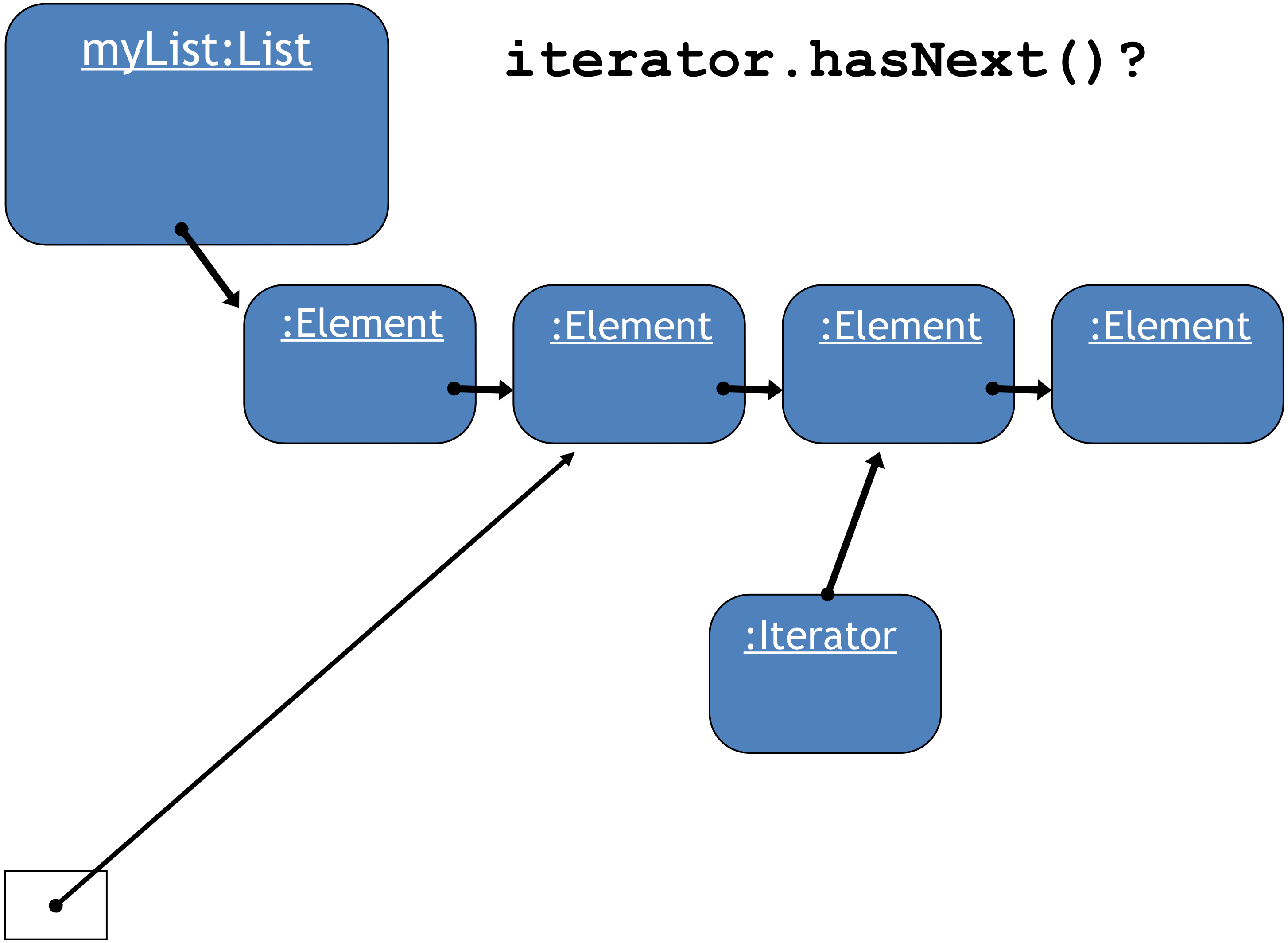
`iterator.hasNext()?`



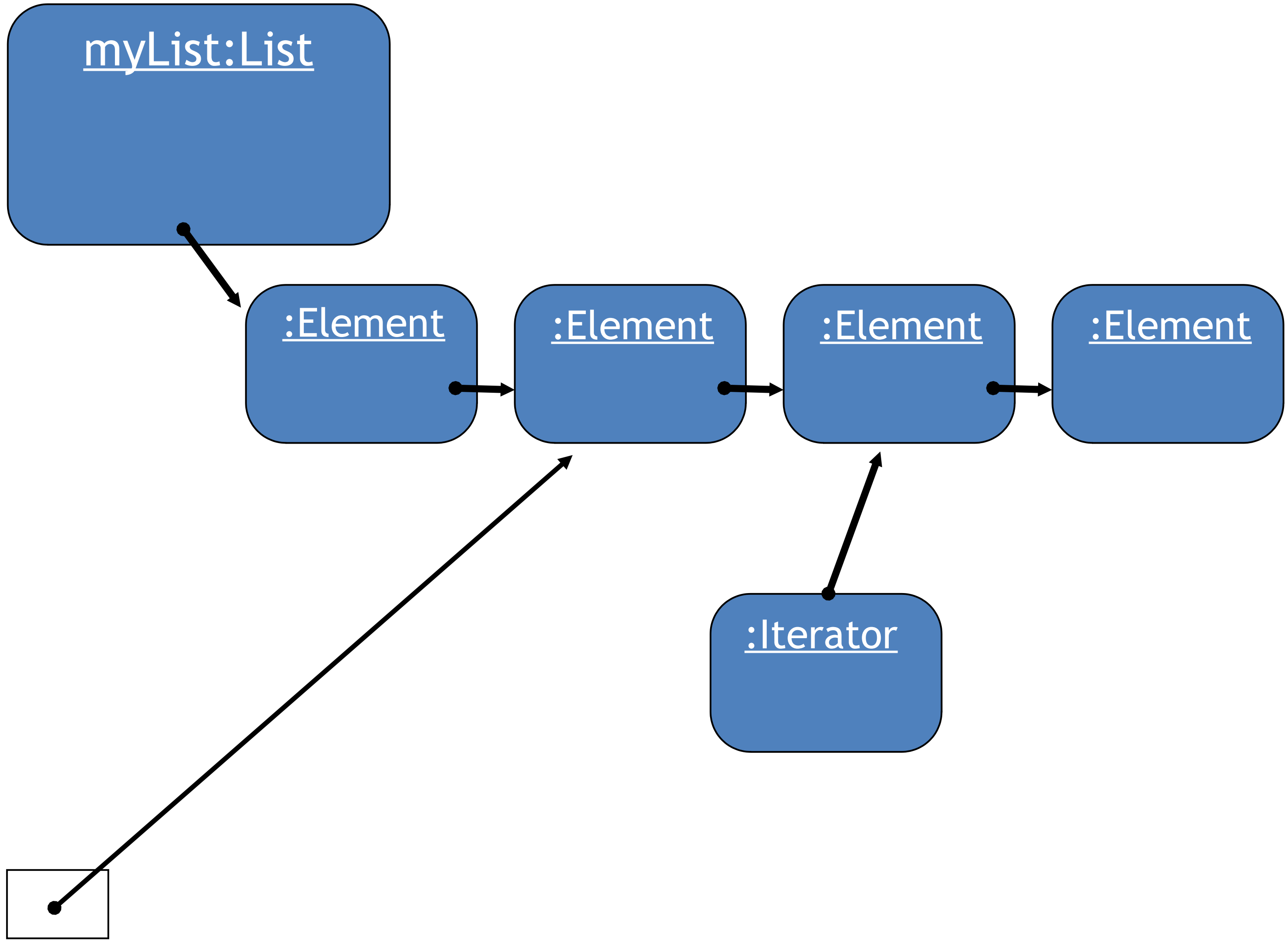
`iterator.hasNext()?`



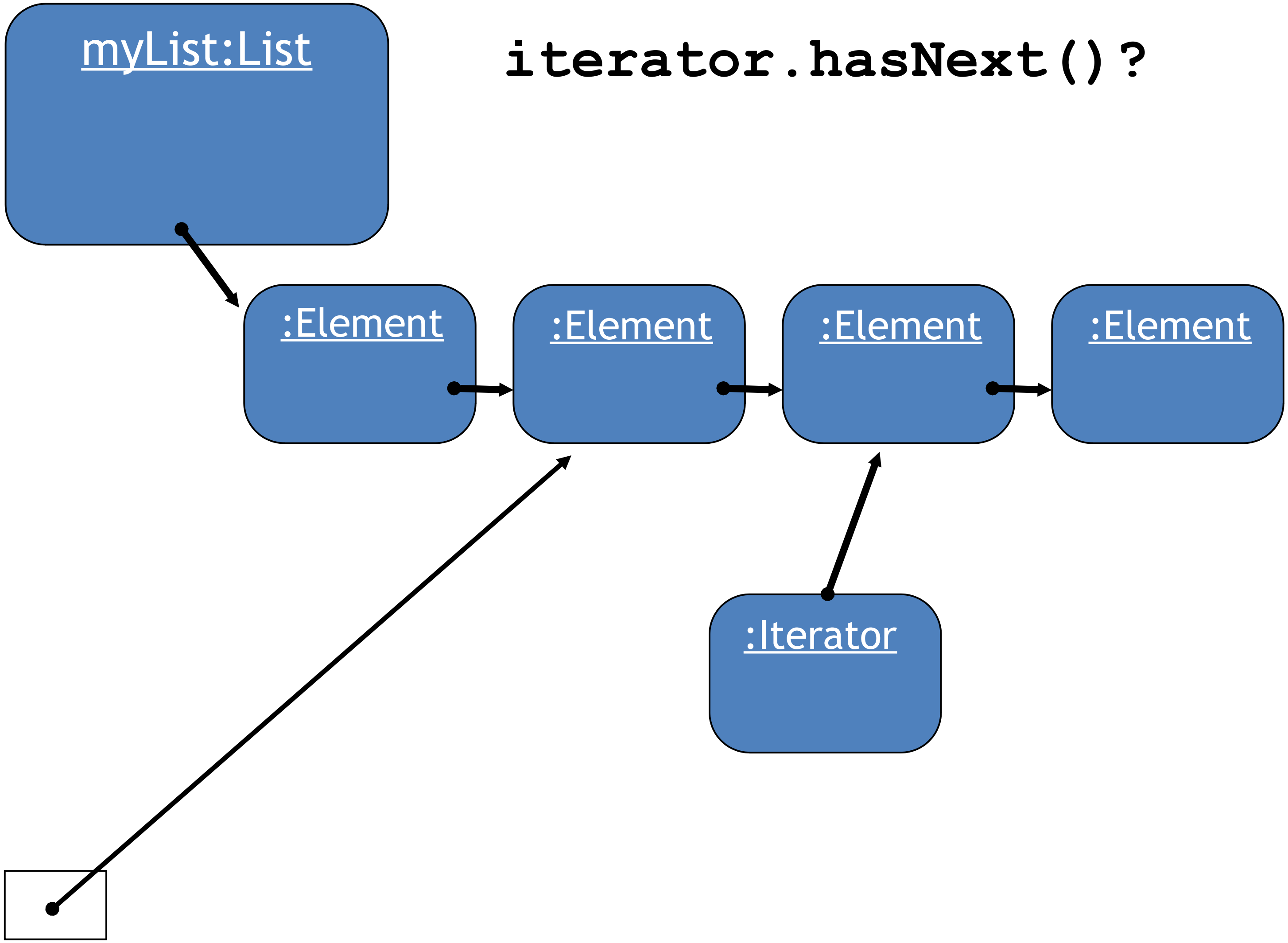
`iterator.hasNext()?`



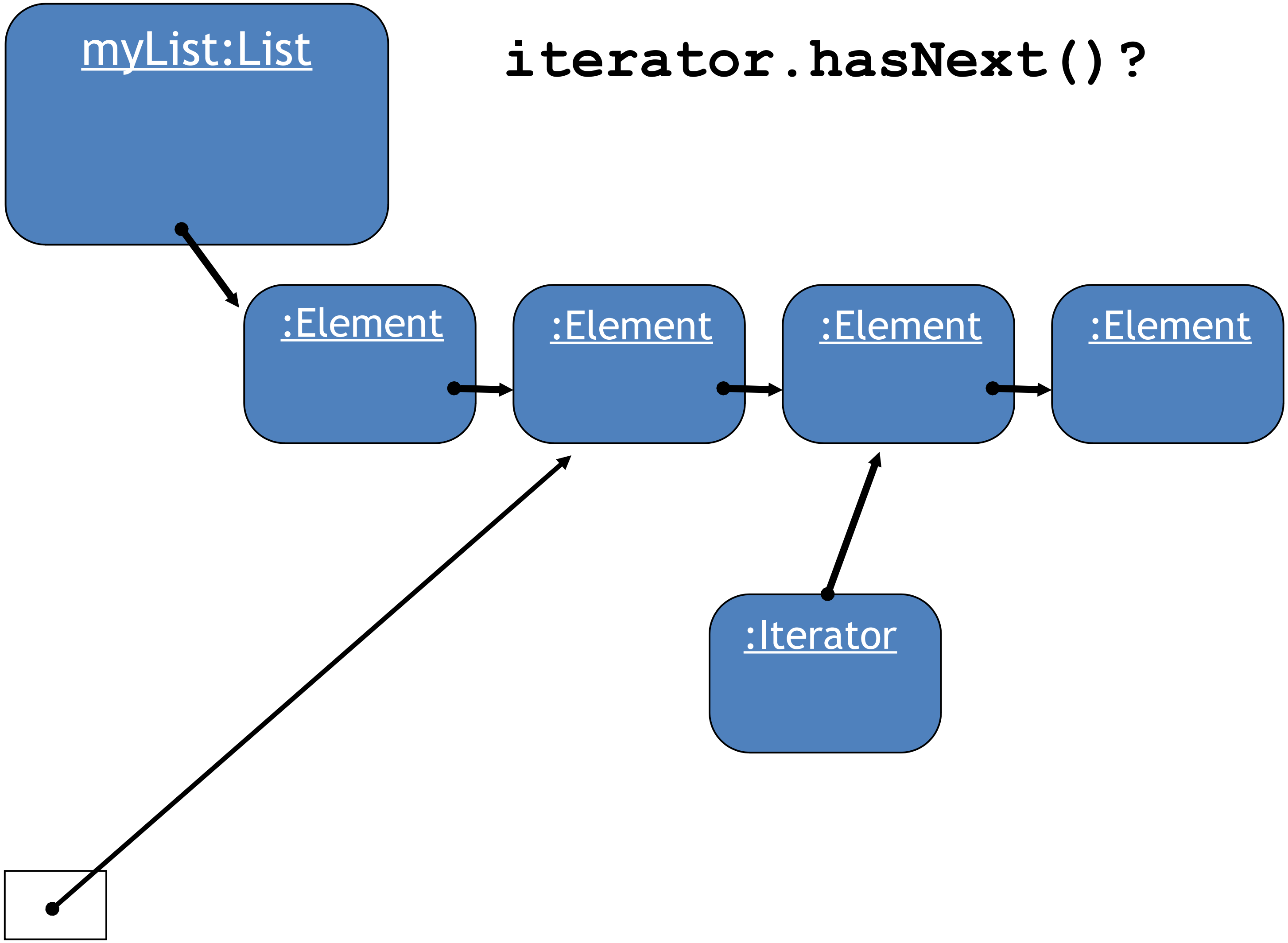
`e = iterator.next();`


myList:List
:Element
:Element
:Element
:Element
:Iterator

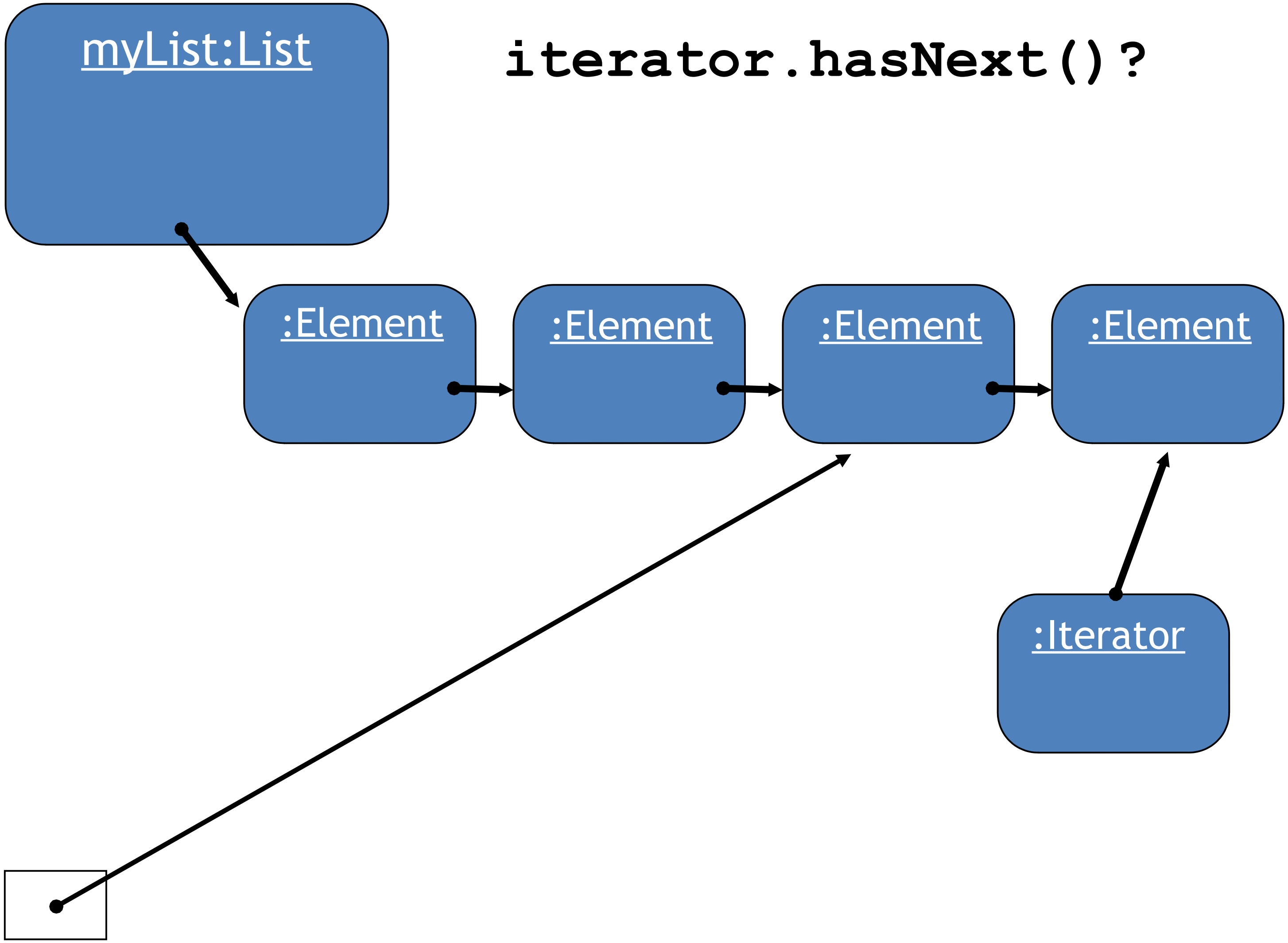
`iterator.hasNext()?`



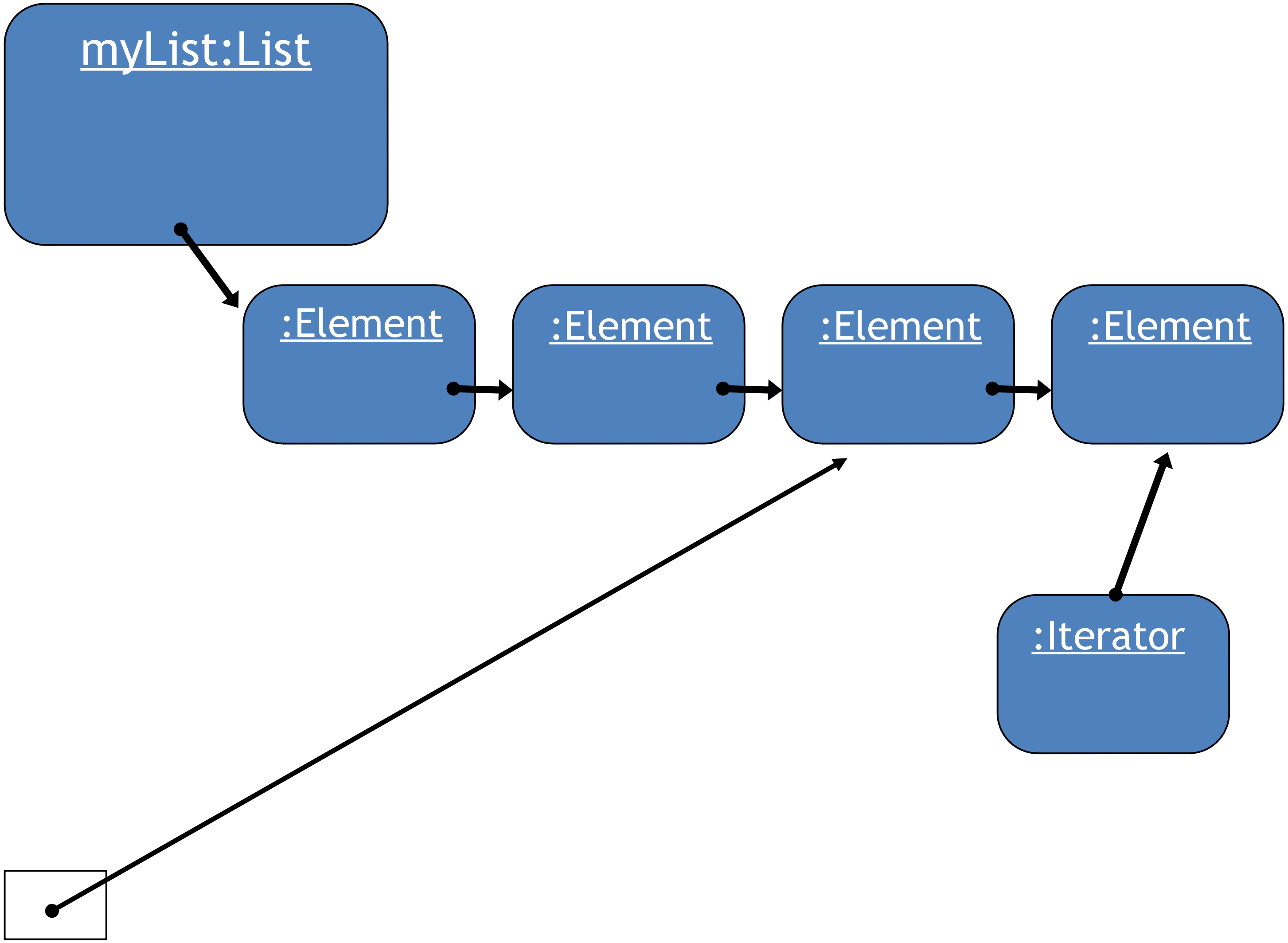
```
iterator.hasNext()?
```



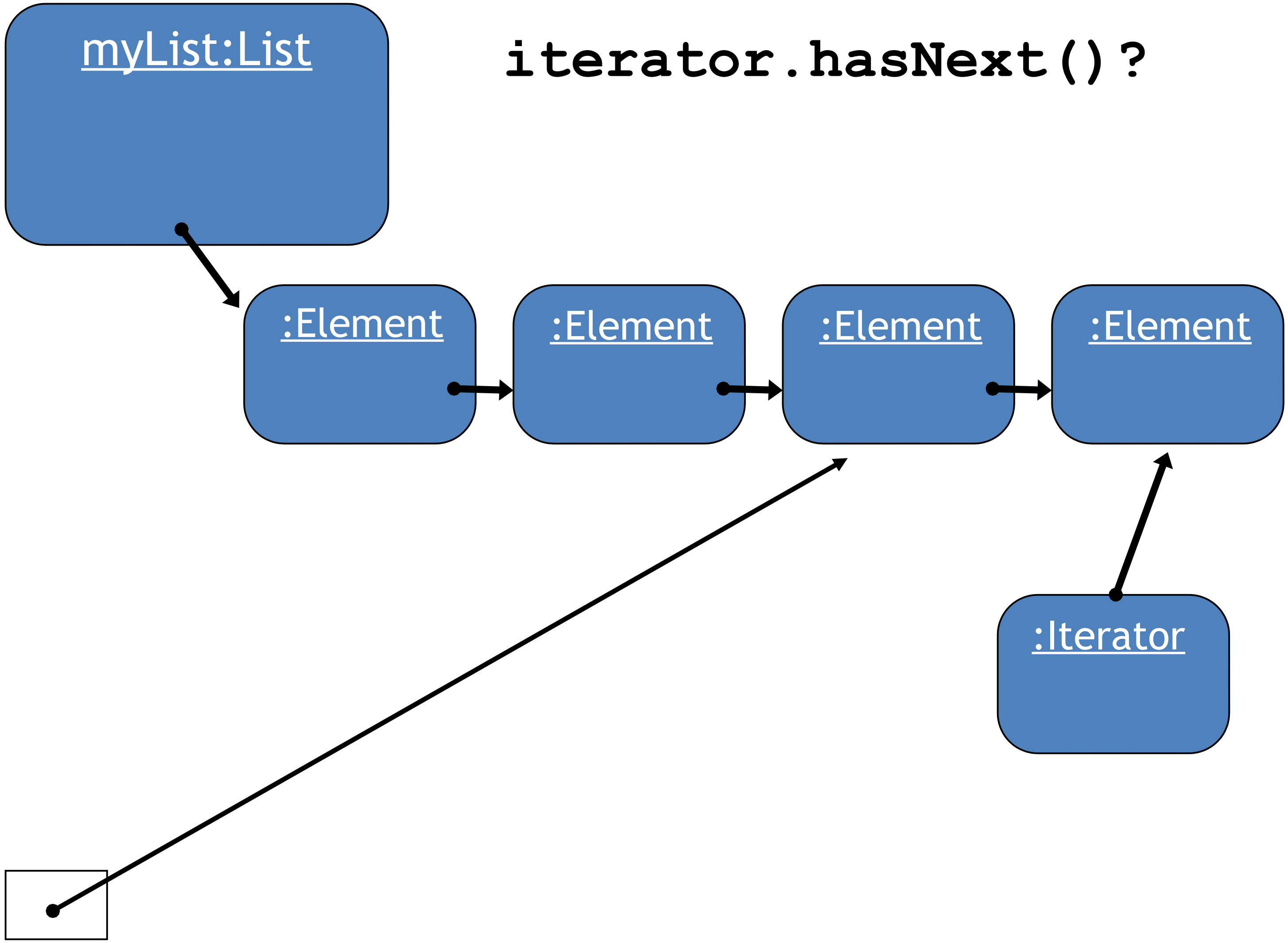
`iterator.hasNext()?`



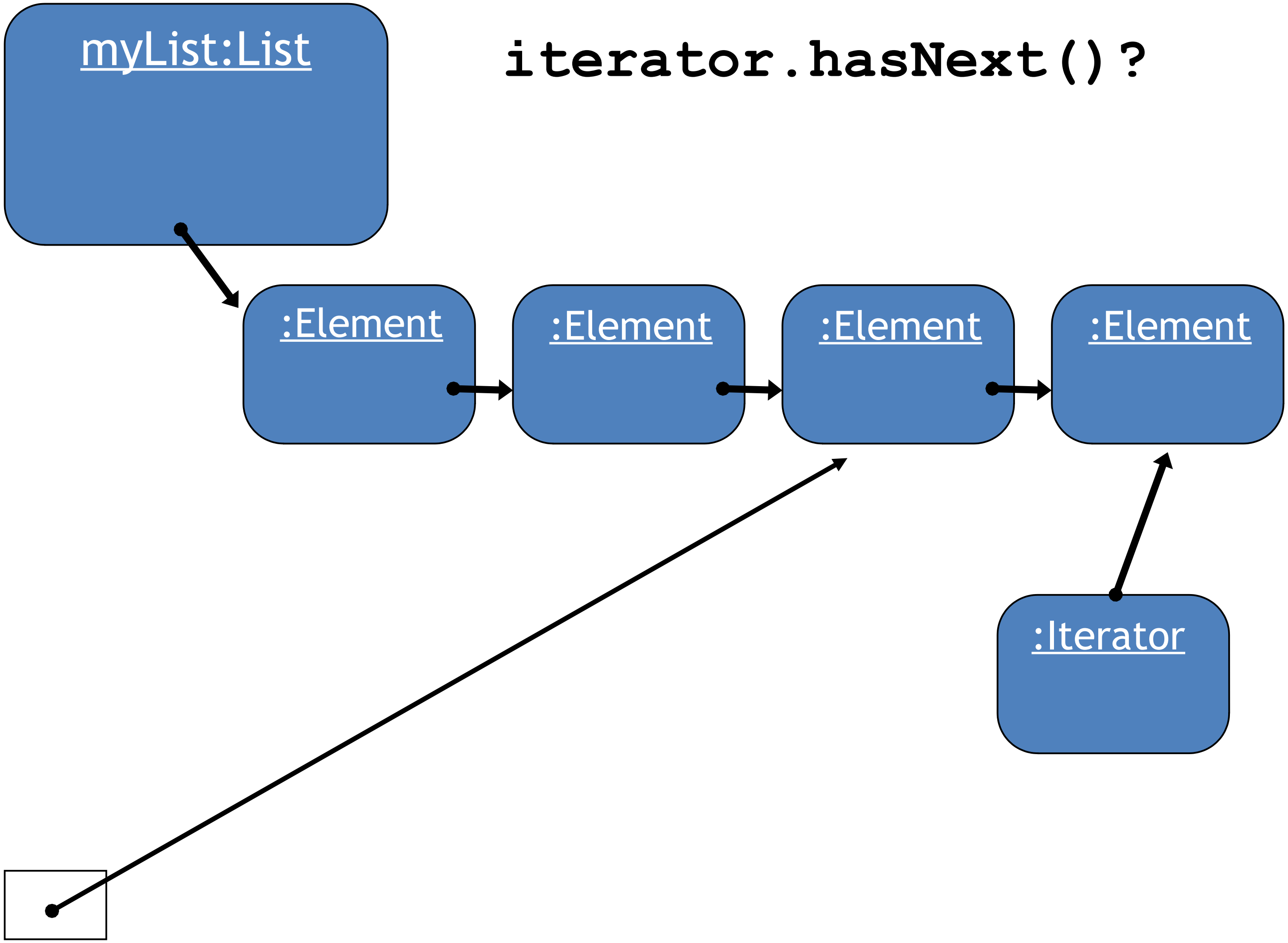
`e = iterator.next();`

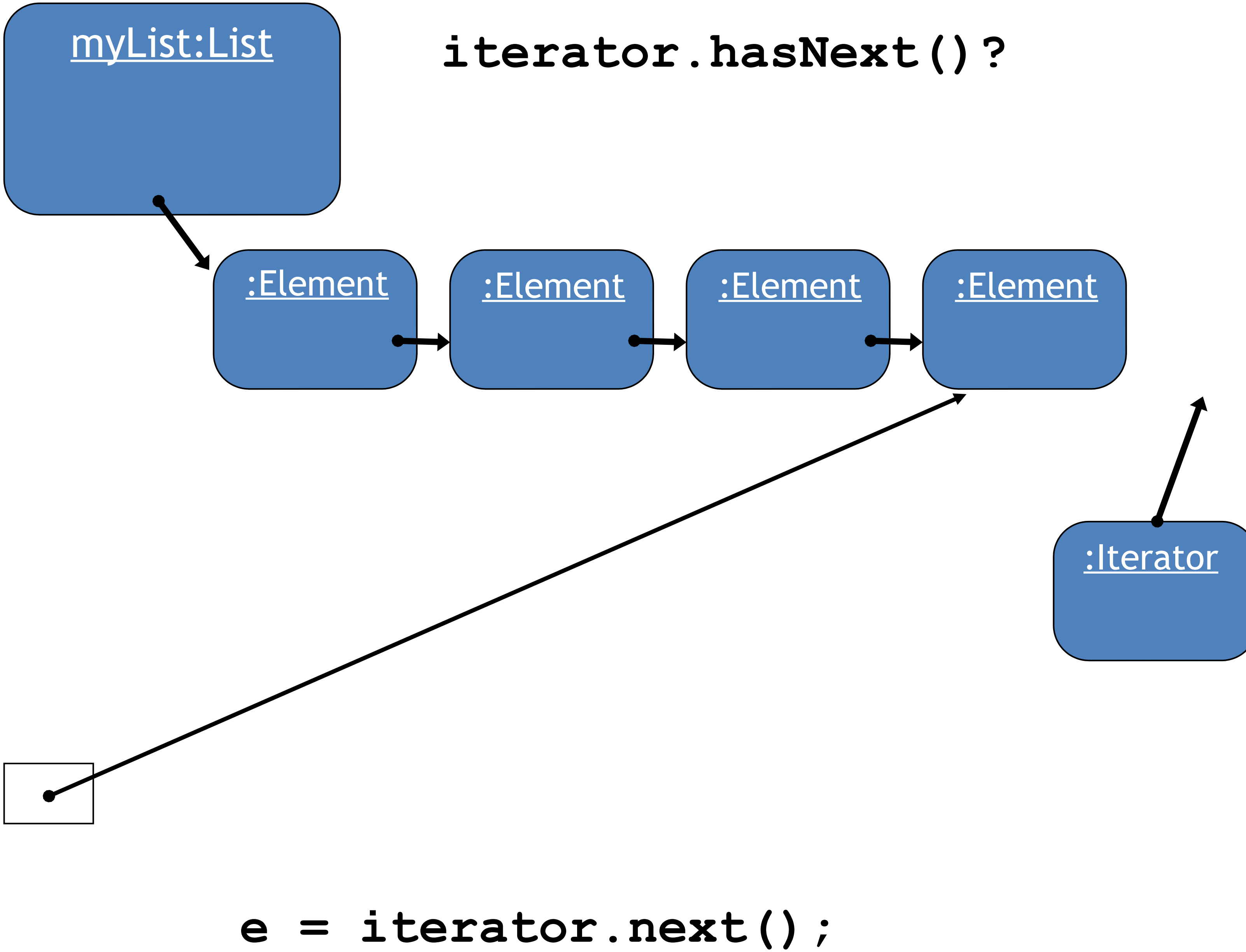

myList:List
:Element
:Element
:Element
:Element
:Iterator

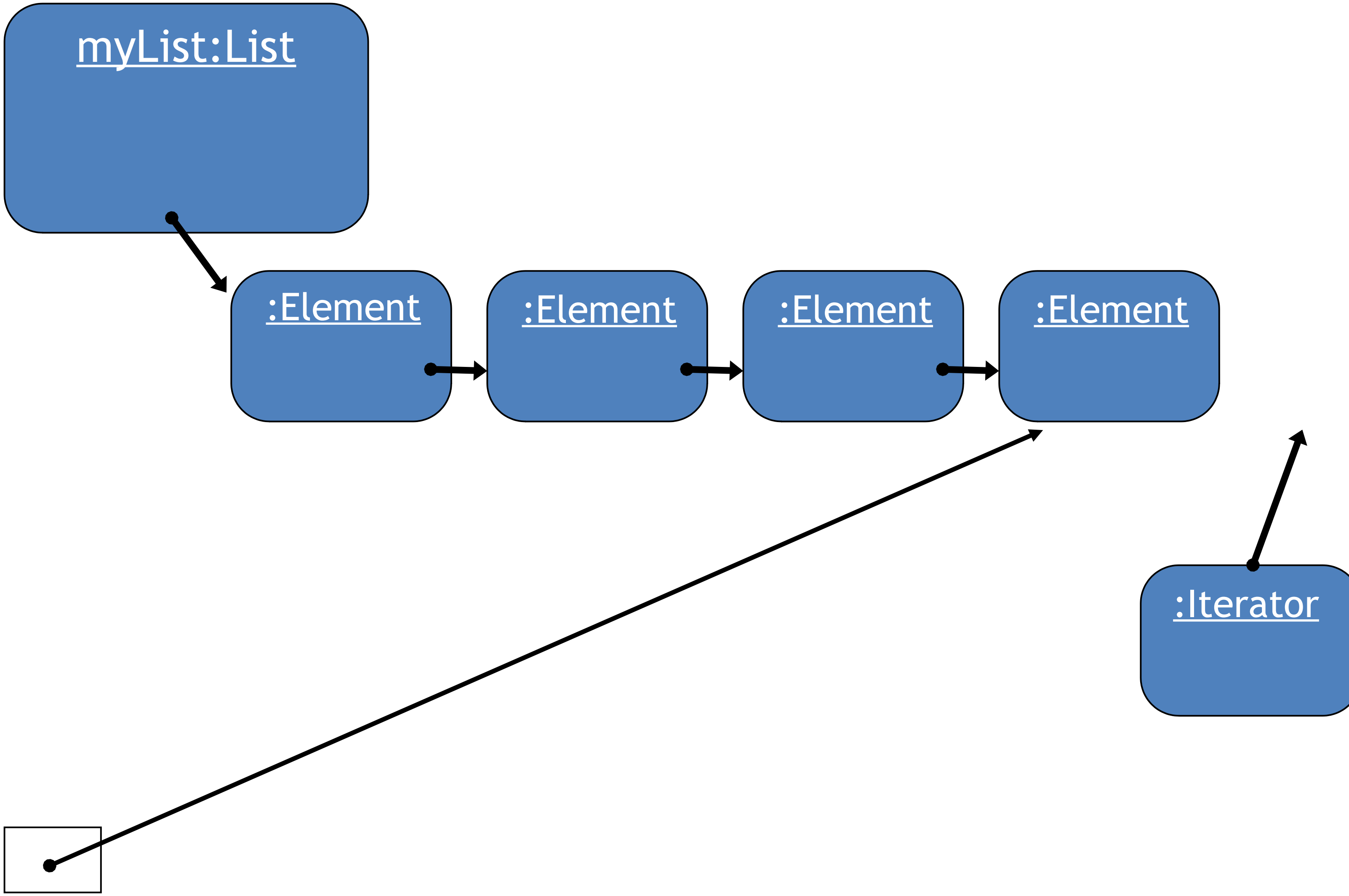
`iterator.hasNext()?`



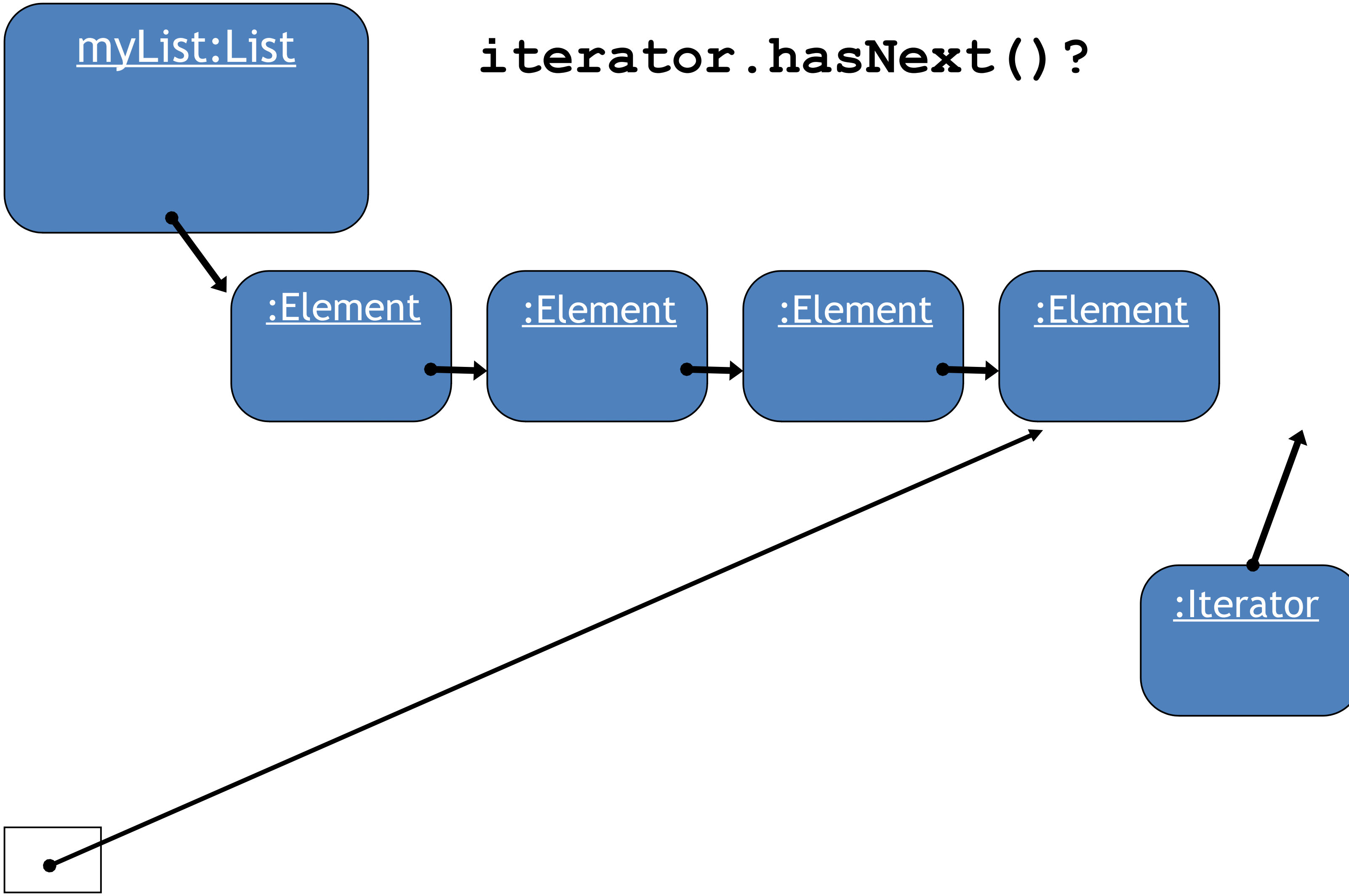
`iterator.hasNext()?`

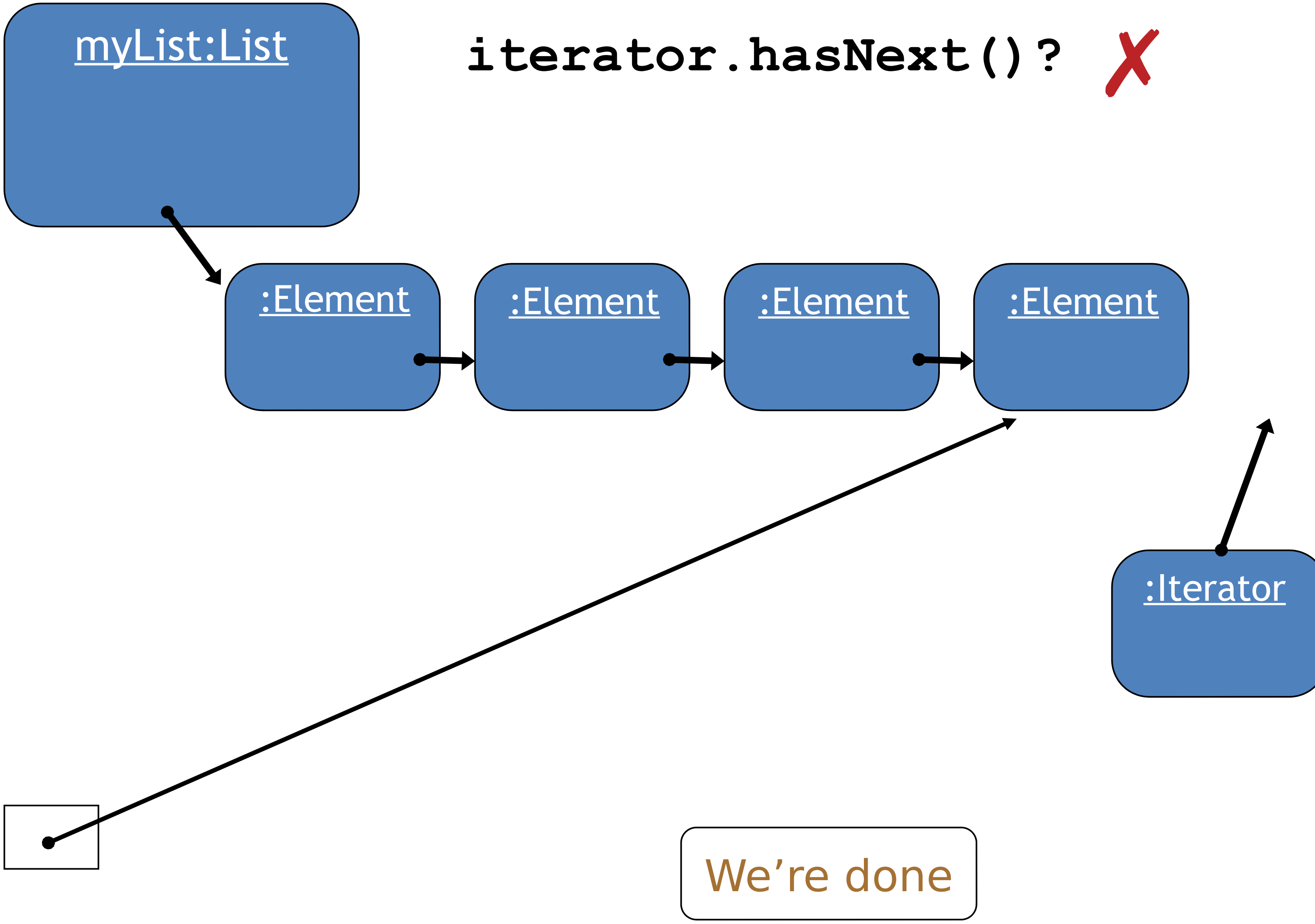



myList:List
iterator.hasNext()?
:Element
:Element
:Element
:Element
:Iterator
e = iterator.next();


myList:List
:Element
:Element
:Element
:Element
:Iterator

`iterator.hasNext()?`





myList:List
iterator.hasNext()? ✗
:Element
:Element
:Element
:Element
:Iterator
We're done

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!

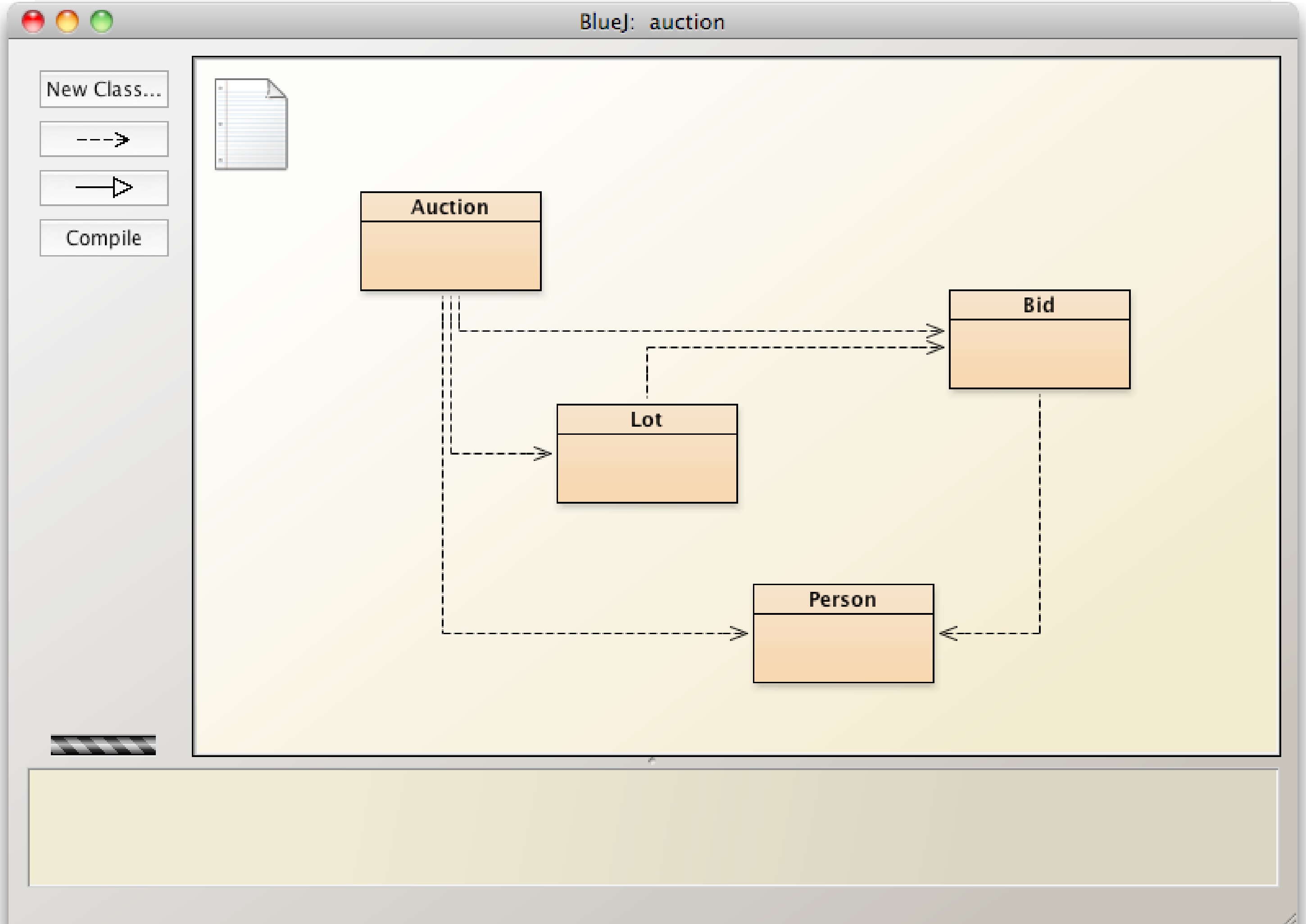
```
int index = 0;
while (index < tracks.size()) {
    Track t = tracks.get(index);
    String artist = t.getArtist();
    if (artist.equals(artistToRemove)) {
        tracks.remove(index);
    }
    index++;
}
```

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();
```

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

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

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.

Further library classes

Using library classes to implement more functionality

Main concepts to be covered

- Further library classes:
  - **`Set`** – avoiding duplicates
  - **`Map`** – creating associations

List, **Map** and **Set**

Declaration and instantiation

```
      List<String> l = new ArrayList<>();
Map<String, int[]> m = new HashMap<>();
         Set<Toto> s = new HashSet<>();
```

- **HashMap** is unrelated to **HashSet**, despite similar names.
- The second word reveals organizational relatedness.

Using sets

```
import java.util.HashSet;
import java.util.Set;
...
Set<String> mySet = new HashSet<>();

mySet.add("one");
mySet.add("two");
mySet.add("three");
mySet.add("two");

for (String element : mySet) {
   do something with element
}
```

Using sets

```
import java.util.HashSet;
import java.util.Set;
...
Set<String> mySet = new HashSet<>();

mySet.add("one");
mySet.add("two");
mySet.add("three");
mySet.add("two");

for (String element : mySet) {
   do something with element
}
```

duplicate
nothing added

Using sets

```
import java.util.HashSet;
import java.util.Set;
...
Set<String> mySet = new HashSet<>();

mySet.add("one");
mySet.add("two");
mySet.add("three");
mySet.add("two");
```

duplicate
nothing added

```
for (String element : mySet) {
   do something with element
}
```

in no particular
order

Using sets

```
import java.util.HashSet;
import java.util.Set;
...
Set<String> mySet = new HashSet<>();

mySet.add("one");
mySet.add("two");
mySet.add("three");
mySet.add("two");

for (String element : mySet) {
   do something with element
}
```

duplicate
nothing added

in no particular
order

Compare with code for an `ArrayList`!

Tokenising Strings

```
// Collects separate words in a string
public Set<String> getInput()  {
    System.out.print("> ");
    String inputLine =
          reader.nextLine().trim().toLowerCase();

    String[] wordArray = inputLine.split(" ");
    Set<String> words = new HashSet<>();

    for (String word : wordArray) {
        words.add(word);
    }
    return words;
}
```

Maps

- Maps are collections that contain *pairs* of objects.
- Parameterized with *two* types.
- Pairs consist of a *key* and a *value*.
- Lookup works by supplying a key, and retrieving a value.
- Example: a contacts list.

Using maps

- A map with strings as keys and values

:HashMap

"Charles Nguyen"	"(531) 9392 4587"
"Lisa Jones"	"(402) 4536 4674"
"William H. Smith"	"(998) 5488 0123"

Using maps

```
Map<String, String> contacts = new HashMap<>();

contacts.put("Charles Nguyen", "(531) 9392 4587");
contacts.put("Lisa Jones", "(402) 4536 4674");
contacts.put("William H. Smith", "(998) 5488 0123");



String number = contacts.get("Lisa Jones");
System.out.println(number);
```

Using maps

```
Map<String, String> contacts = new HashMap<>();

contacts.put("Charles Nguyen", "(531) 9392 4587");
contacts.put("Lisa Jones", "(402) 4536 4674");
contacts.put("William H. Smith", "(998) 5488 0123");
```

key

value

```
String number = contacts.get("Lisa Jones");
System.out.println(number);
```

Using maps

```
Map<String, String> contacts = new HashMap<>();

contacts.put("Charles Nguyen", "(531) 9392 4587");
contacts.put("Lisa Jones", "(402) 4536 4674");
contacts.put("William H. Smith", "(998) 5488 0123");
```

value key

```
String number = contacts.get("Lisa Jones");
System.out.println(number);
```

Collections and primitive types

- The generic collection classes can be used with all class types …
- … but what about *the primitive types*: **`int`**, **`boolean`**, etc.?
- Suppose we want an **`ArrayList`** of **`int`**?

Wrapper classes

- Primitive types are not objects types. Primitive-type values must be wrapped in objects to be stored in a collection!
- Wrapper classes exist for all primitive types:

simple type	*wrapper class*
int	Integer
float	Float
char	Character
…	…

Wrapper classes

wrap the value

```
int i = 18;
Integer iwrap = new Integer(i);
…
int value = iwrap.intValue();
```

unwrap it

In practice, *autoboxing* and *unboxing* mean we don't often have to do this explicitly

Autoboxing and unboxing

```
private List<Integer> markList;
...
public void storeMark(int mark) {
    markList.add(mark);
}
```

autoboxing

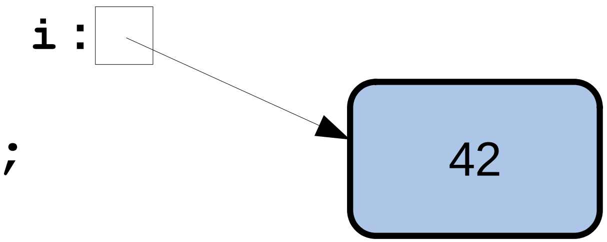
```
int firstMark = markList.remove(0);
```

unboxing

Wrapper classes are immutable

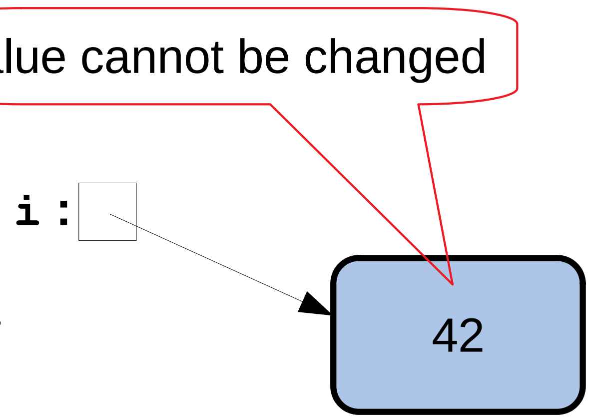
- Like **`String`**, **`Integer`** is immutable

```
Integer i = new Integer(42);
```



Wrapper classes are immutable

- Like **`String`**, **`Integer`** is immutable




```
Integer i = new Integer(42);
```

Wrapper classes are immutable

- Like **`String`**, **`Integer`** is immutable

```
Integer i = new Integer(42);
        i = new Integer(-13);
```