Grouping objects

Introduction to collections

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 system.
- The number of items to be stored varies.
  - Items added.
  - Items deleted.

An organizer for music files

- Track files may be added.
- There is no pre-defined limit to the number of files.
- It will tell how many file names are stored in the collection.
- It will list individual file names.
- Explore the *music-organizer-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 classes for doing this.

```
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>();
    }

    ...
}
```

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”.

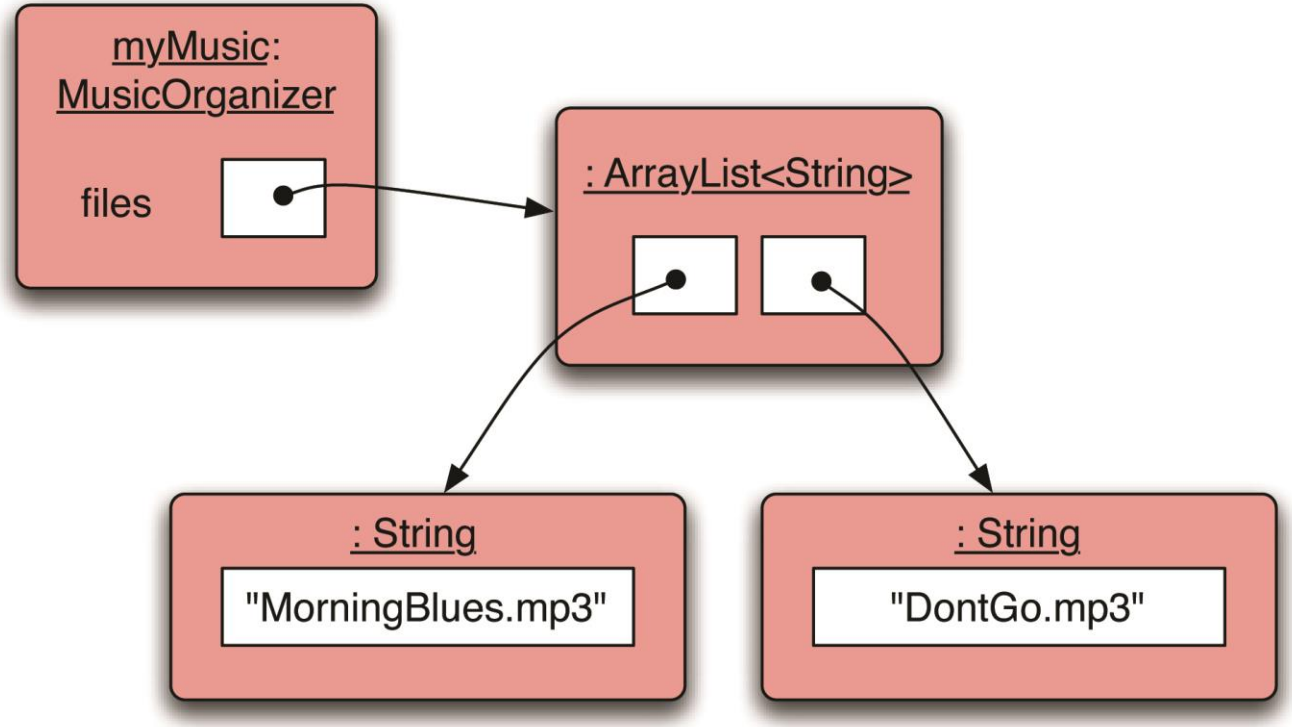
Generic classes

- 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.

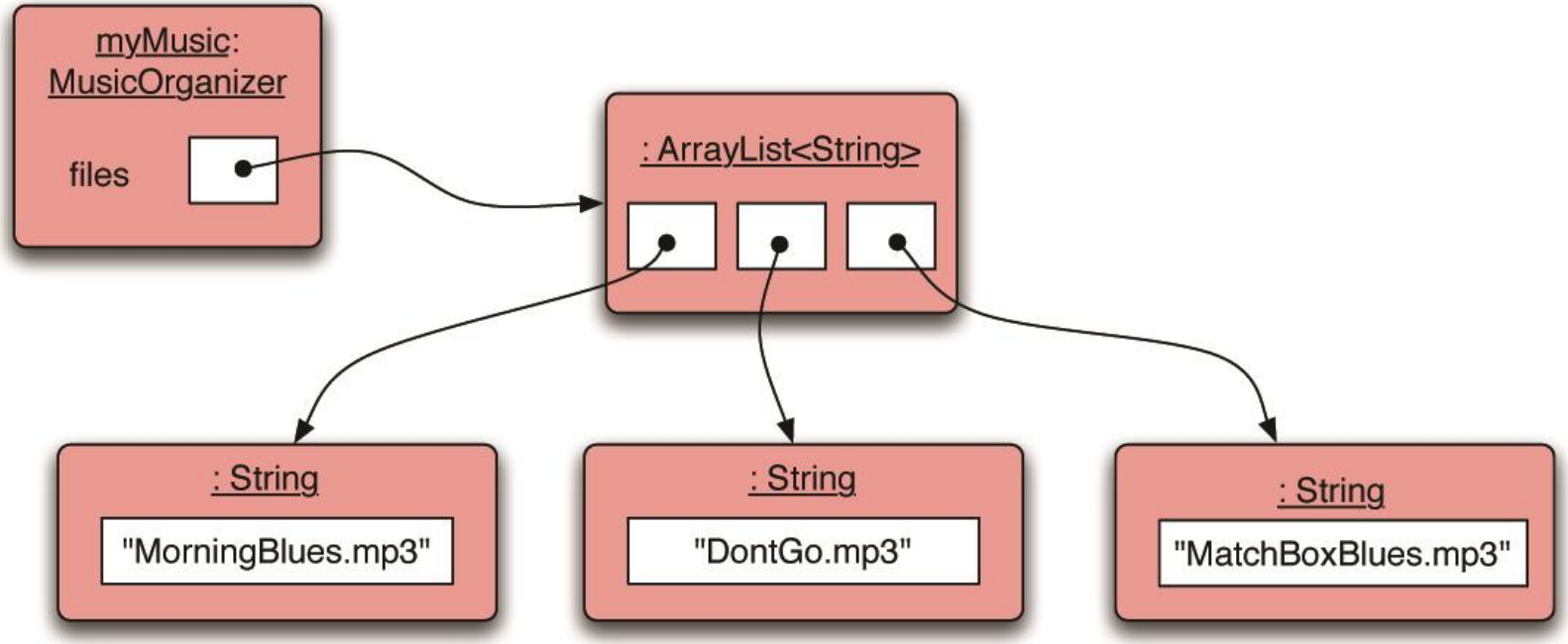
Creating an ArrayList object

- 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.

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.
  - Does that matter? Does not knowing how prevent us from using it?

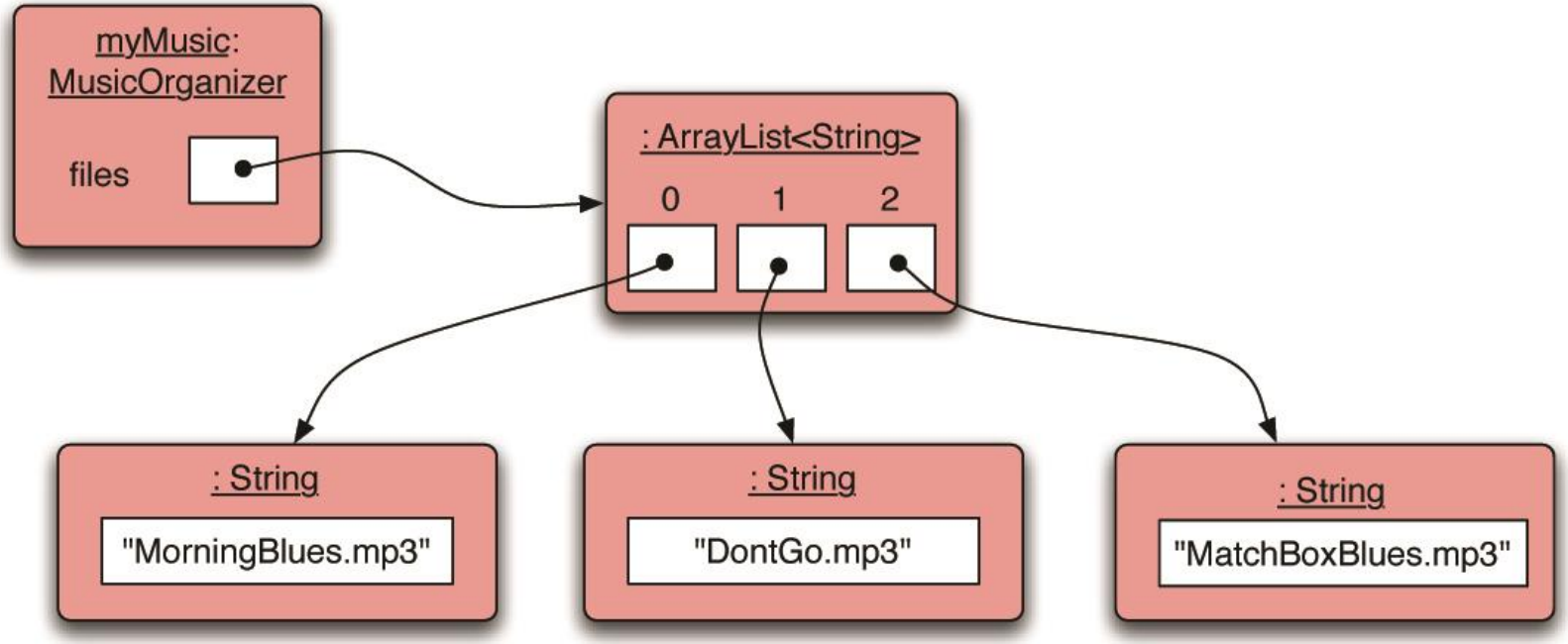
Using the collection

```
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*)

Index numbering



Retrieving an object

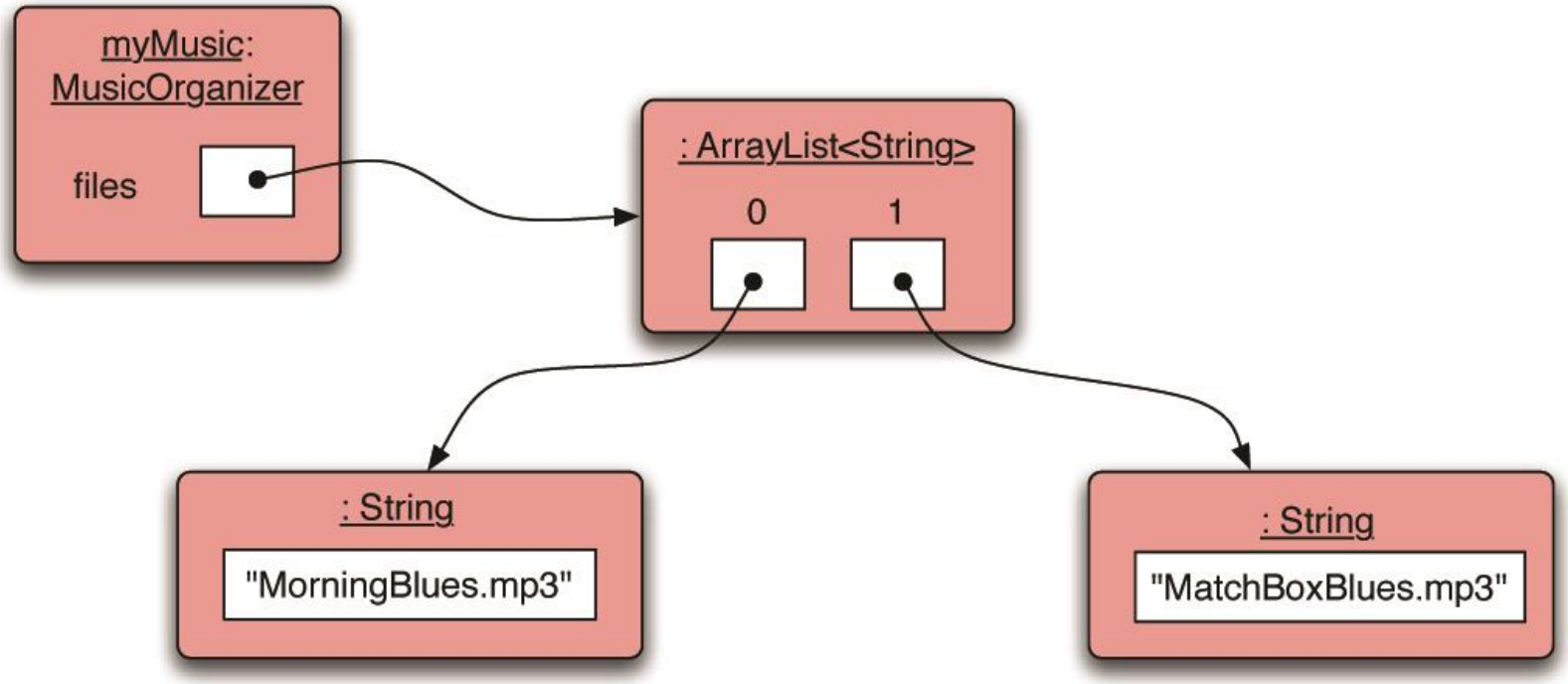
```
public void listFile(int index)
{
    if(index >= 0 &&
            index < files.size()) {
        String filename = files.get(index);
        System.out.println(filename);
    }
    else {
        // This is not a valid index.
    }
}
```

Index validity checks

Retrieve and print the file name

Needed? (Error message?)

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 popular errors...

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");
    }
}
```

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");
    }
}
```

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");
    }
}
```

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");
    }
}
```

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");
    }
}
```

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");
    }
}
```

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);
}
```

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);
}
```

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);
}
```

Grouping objects

Collections and the for-each loop

Main concepts to be covered

- Collections
- Loops: the for-each loop

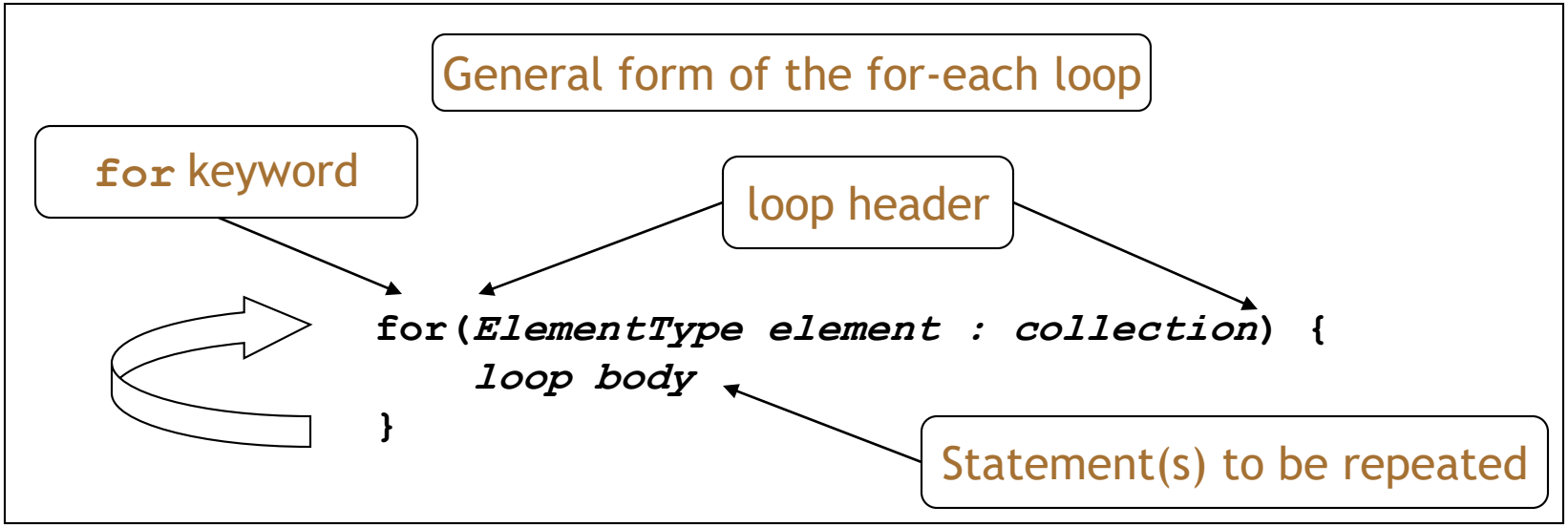
Iteration

- 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

- We often want to repeat some actions over and over.
- Loops provide us with a way to control how many times we repeat those actions.
- With collections, we often want to repeat things once for every object in a particular collection.

For-each loop pseudo code



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

For each *element* in *collection*, do the things in the *loop body*.

A Java example

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

for each *filename* in *files*, print out *filename*

Review

- Loop statements allow a block of statements to be repeated.
- The for-each loop allows iteration over a whole collection.

Selective processing

- Statements can be nested, giving greater selectivity:

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

Critique of for-each

- Easy to write.
- Termination happens naturally.
- The collection cannot be changed.
- There is no index provided.
  - Not all collections are index-based.
- We can't stop part way through;
  - e.g. find-the-first-that-matches.
- 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.
- The while loop

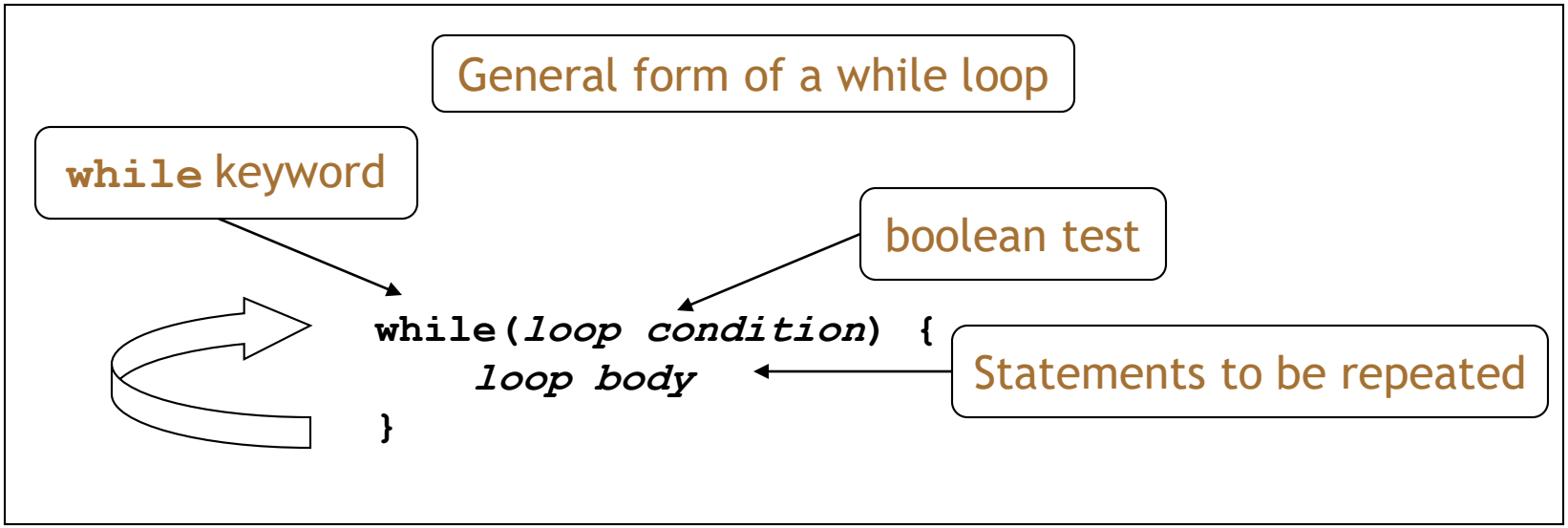
Search tasks are indefinite

- We cannot predict, *in advance*, how many places we will have to look.
- Although, there may well be an absolute limit – i.e., checking every possible location.
- '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 each object in a collection.
- Sometimes we require more variation than this.
- We use a boolean condition to decide whether or not to keep going.
- A while loop provides this control.

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?

A Java example

```
/**
 * 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 be an *infinite loop*.

Searching a collection

- A fundamental activity.
- Applicable beyond collections.
- Necessarily indefinite.
- We must code for both success and failure – exhausted search.
- Both must make the loop's condition *false*.
- The collection might be empty.

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

- With a while loop 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 found = false;
while(index < files.size() && !found) {
    String file = files.get(index);
    if(file.contains(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 will be executed *zero or more* times.

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;
}
```

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

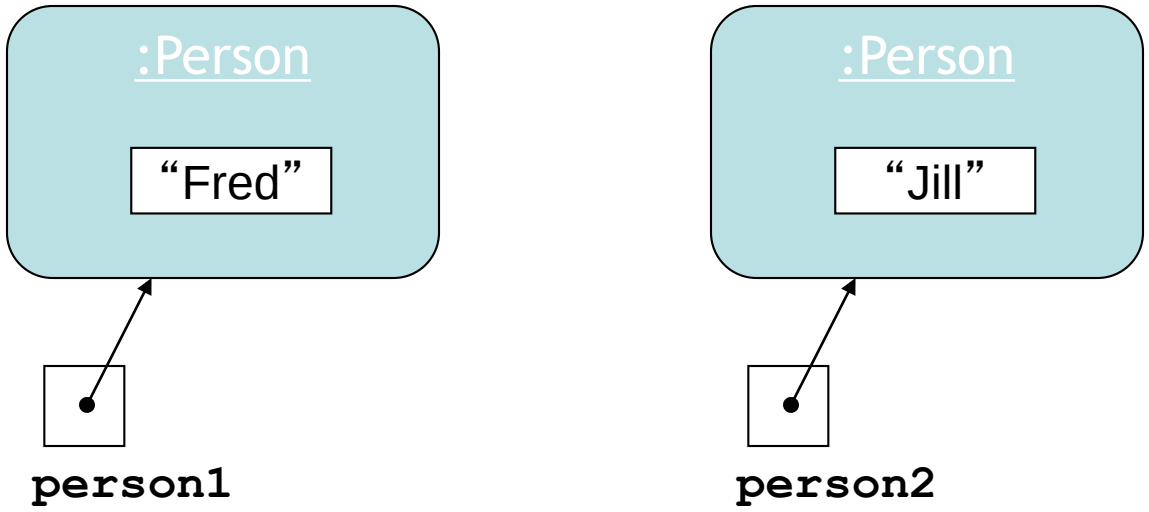
```
if(input.equals("bye")) {
    ...
}
```

tests equality

Always use `.equals` for text equality.

Identity vs equality 1

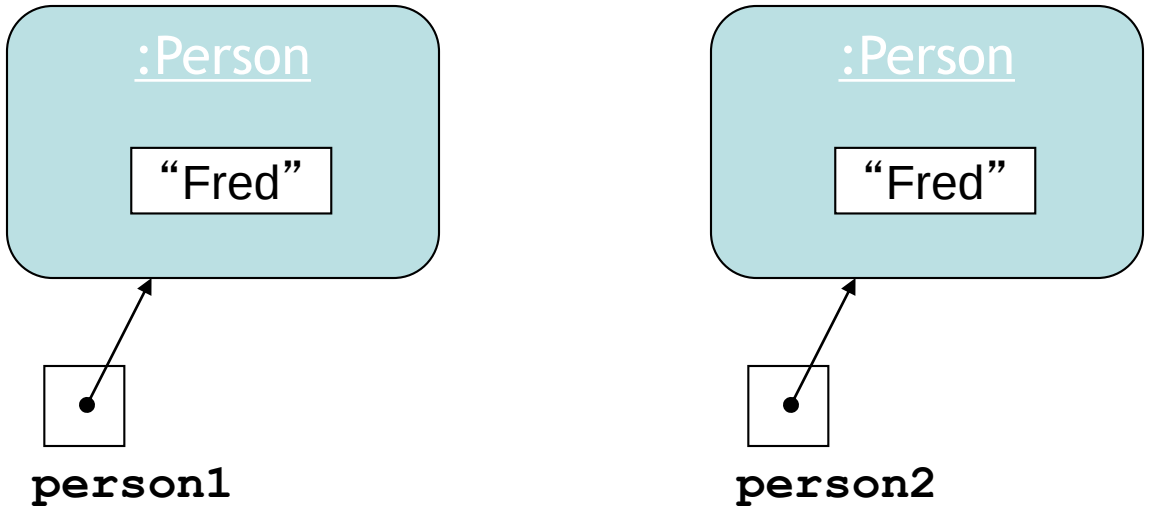
Other (non-String) objects:




```
person1 == person2 ?
```

Identity vs equality 2

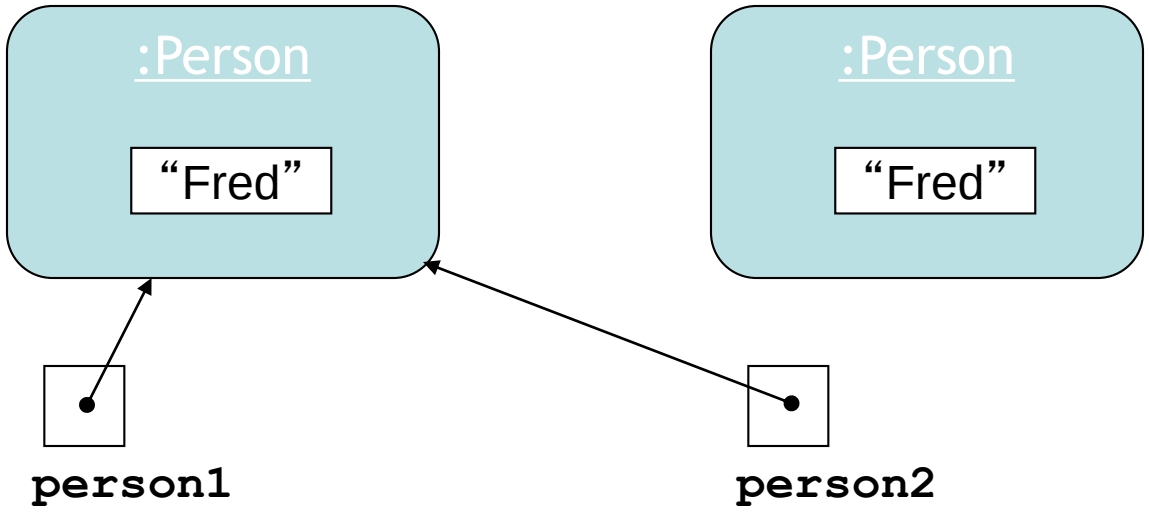
Other (non-String) objects:




```
person1 == person2 ?
```

Identity vs equality 3

Other (non-String) objects:

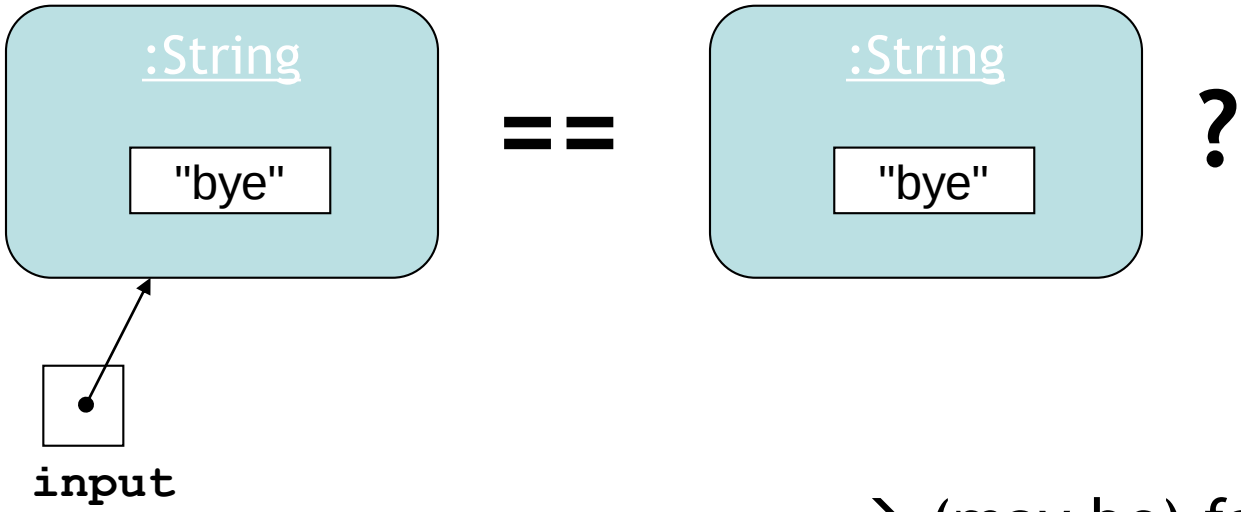



```
person1 == person2 ?
```

Identity vs equality (Strings)

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

== tests identity

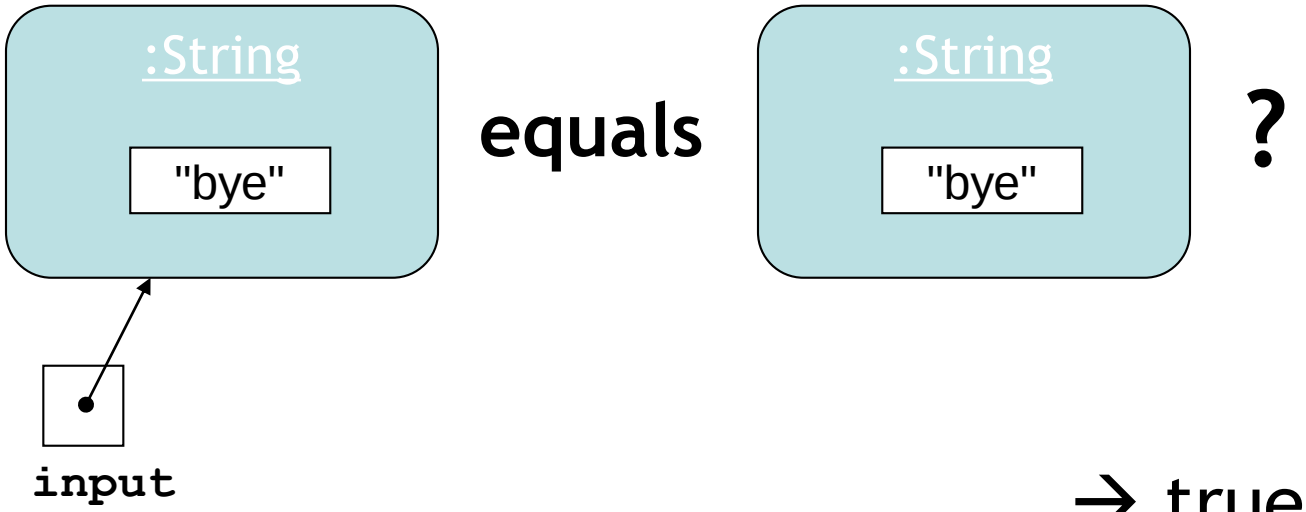


→ (may be) false!

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.

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

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()`**

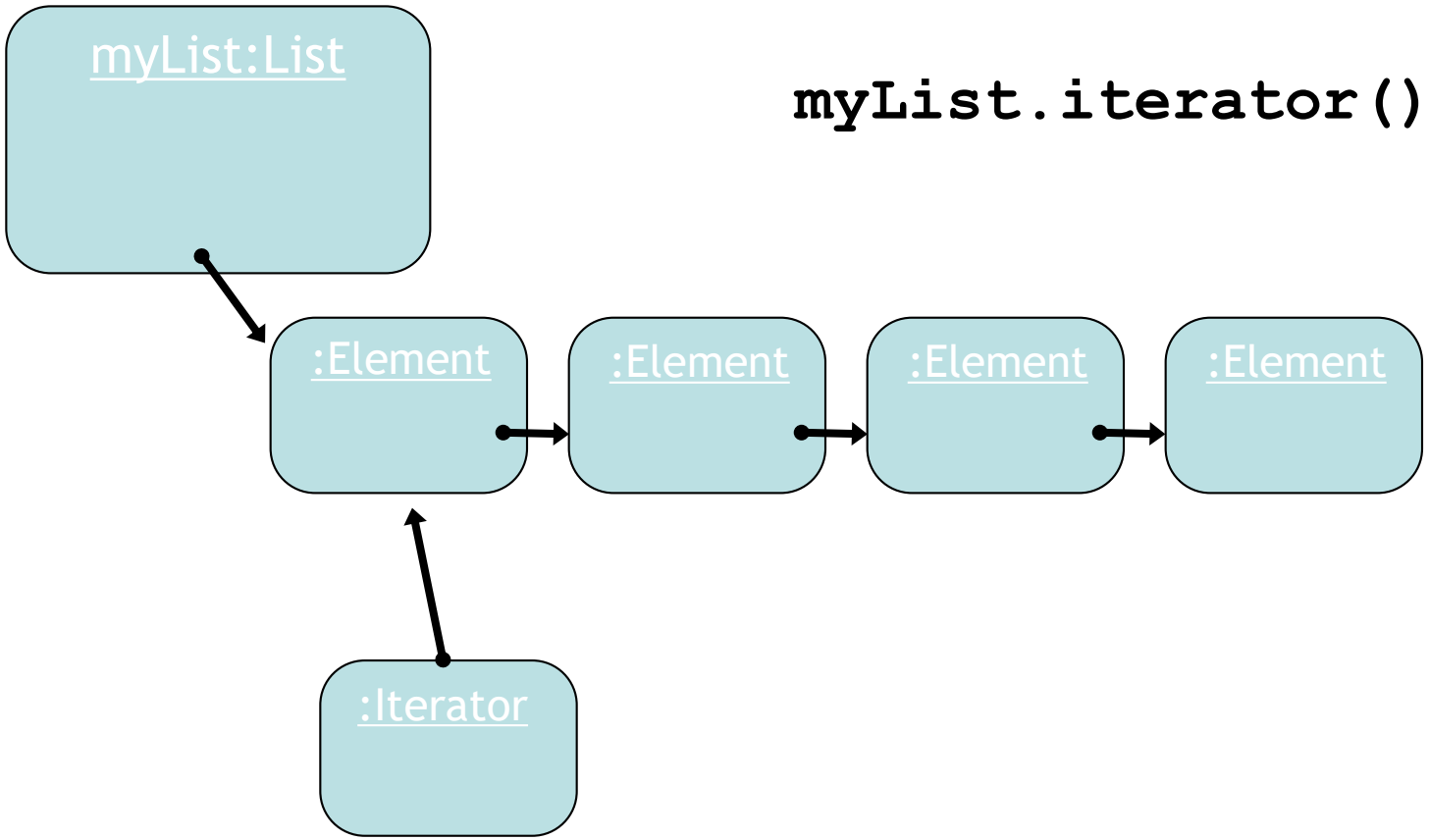
Using an Iterator object

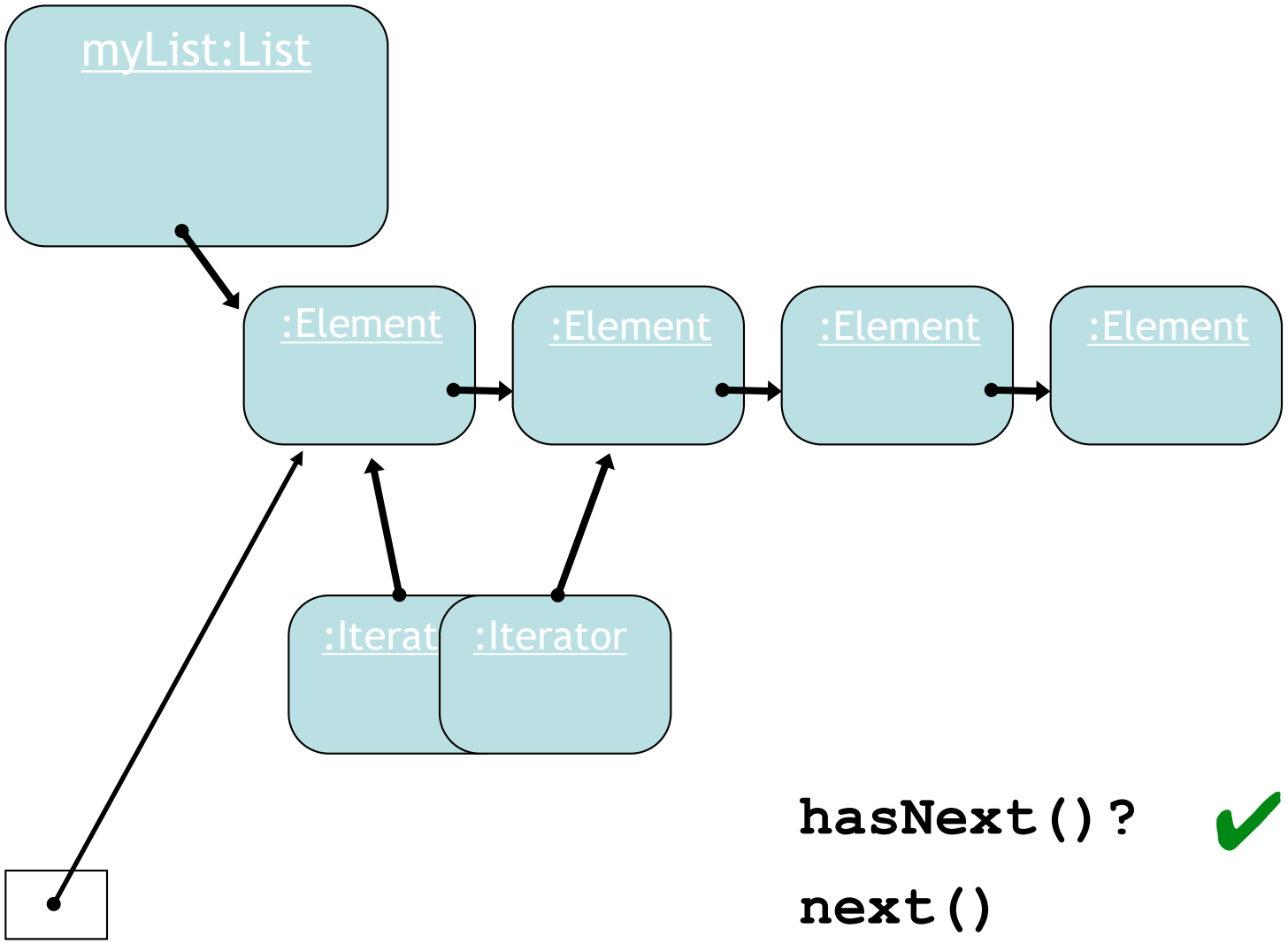
java.util.Iterator

returns an Iterator object

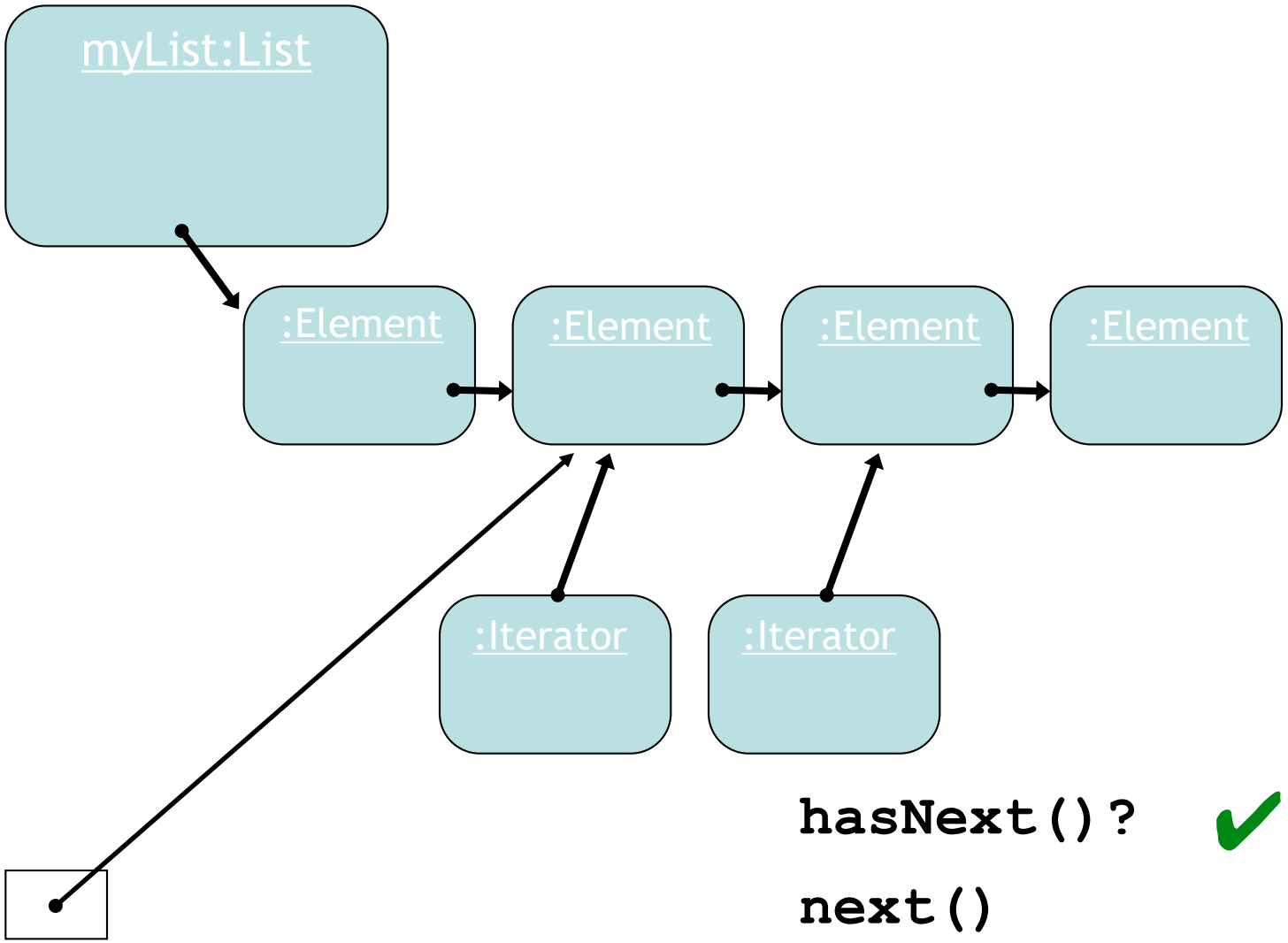
```
Iterator<ElementType> it = myCollection.iterator();
while(it.hasNext()) {
    call it.next() to get the next object
    do something with that object
}

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

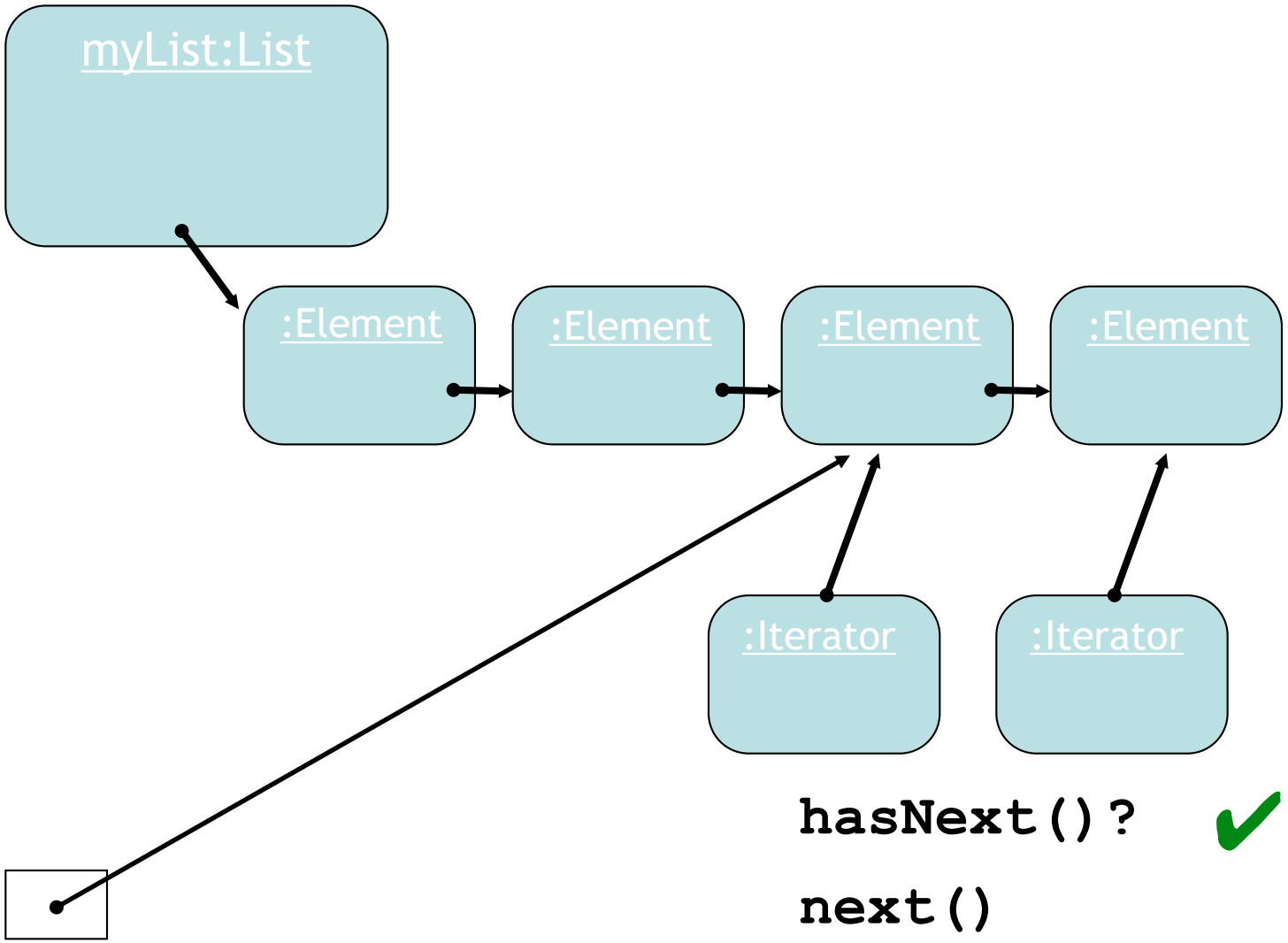
Iterator mechanics

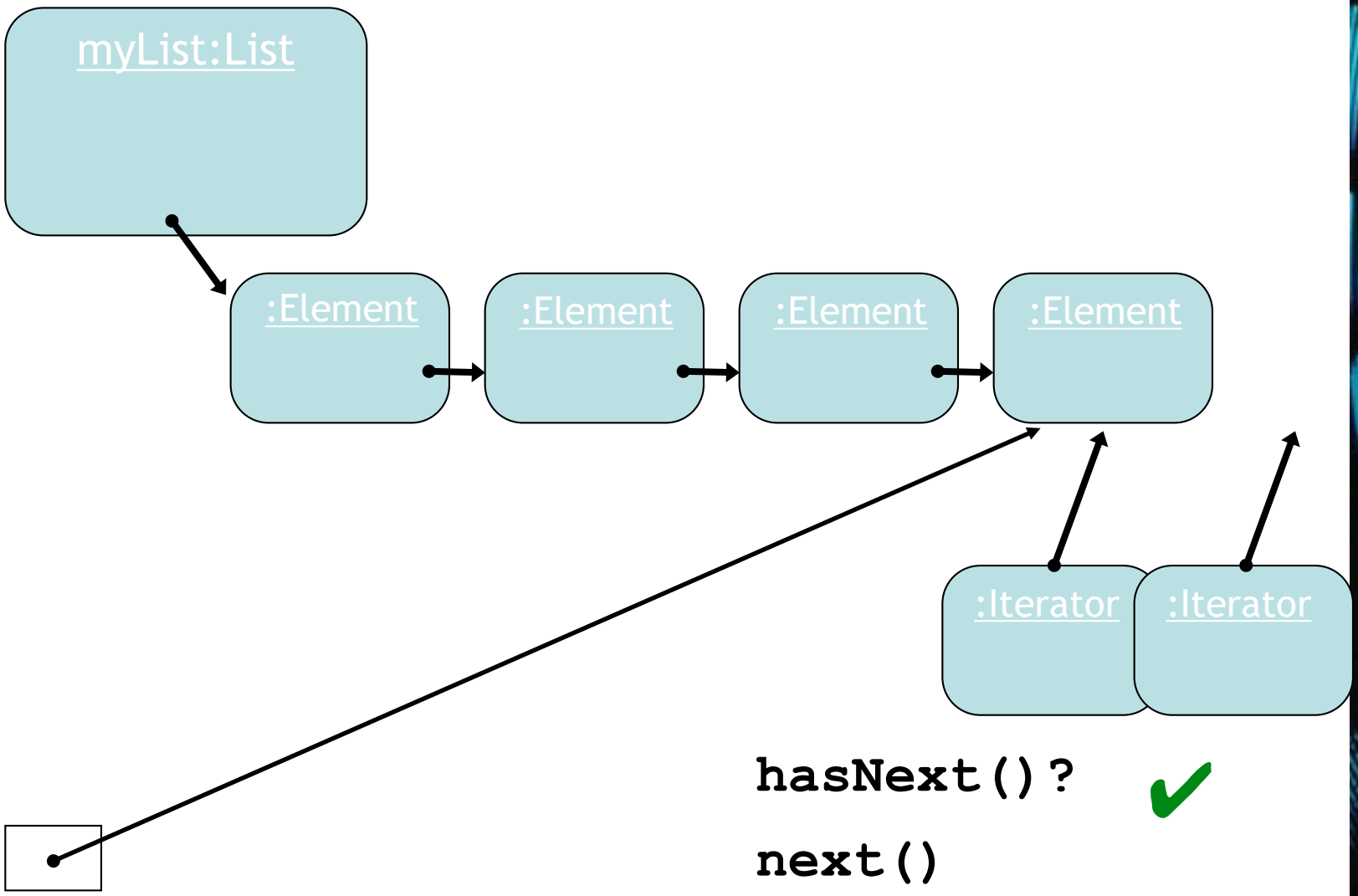

myList:List
myList.iterator()
:Element
:Element
:Element
:Element
:Iterator

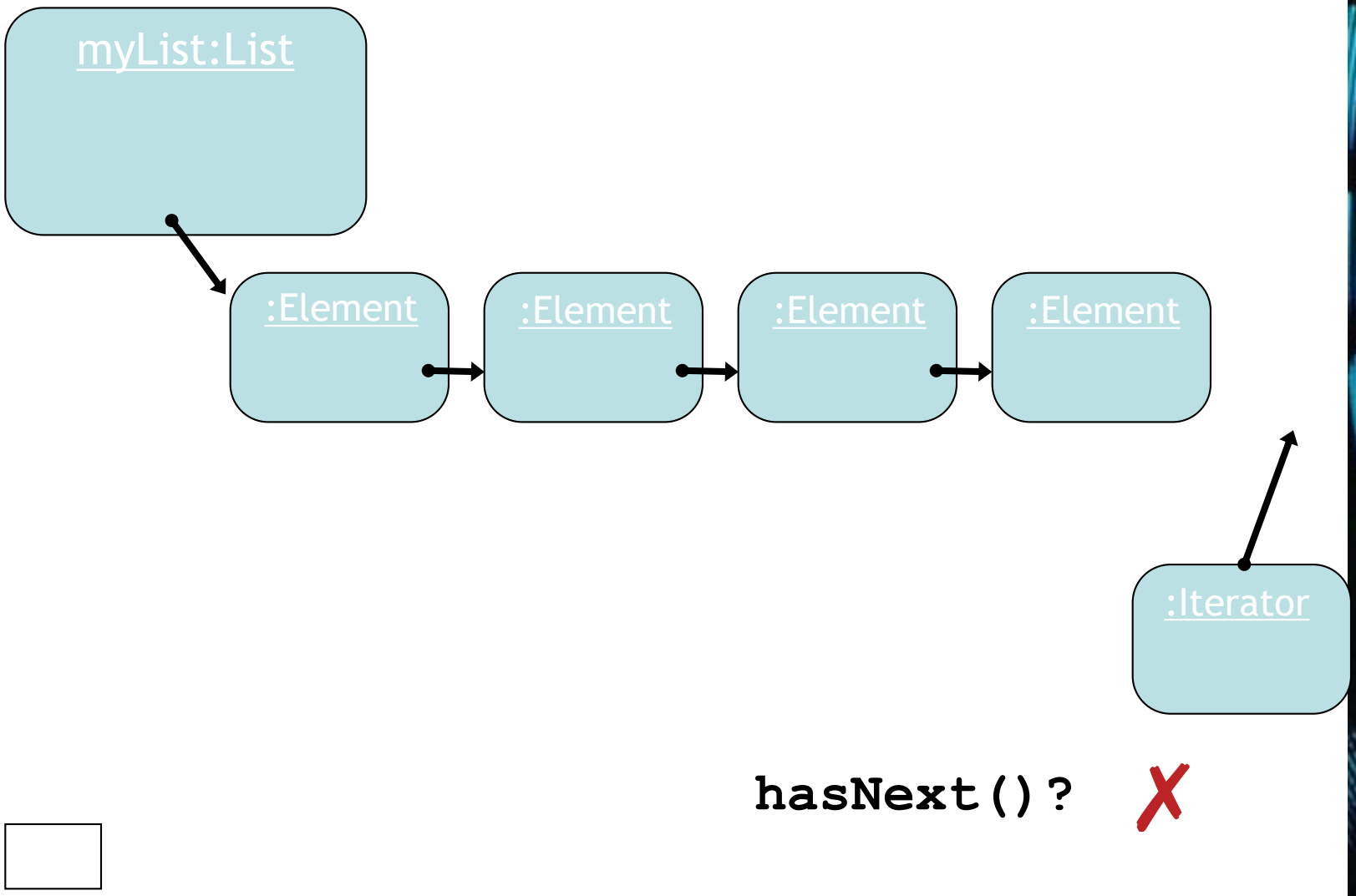



```
Element e = iterator.next();
```


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()

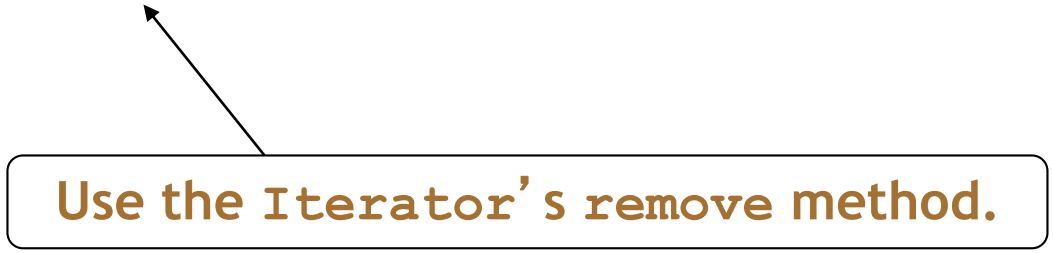

myList:List
:Element
:Element
:Element
:Element
:Iterator
hasNext()?

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();
    }
}
```



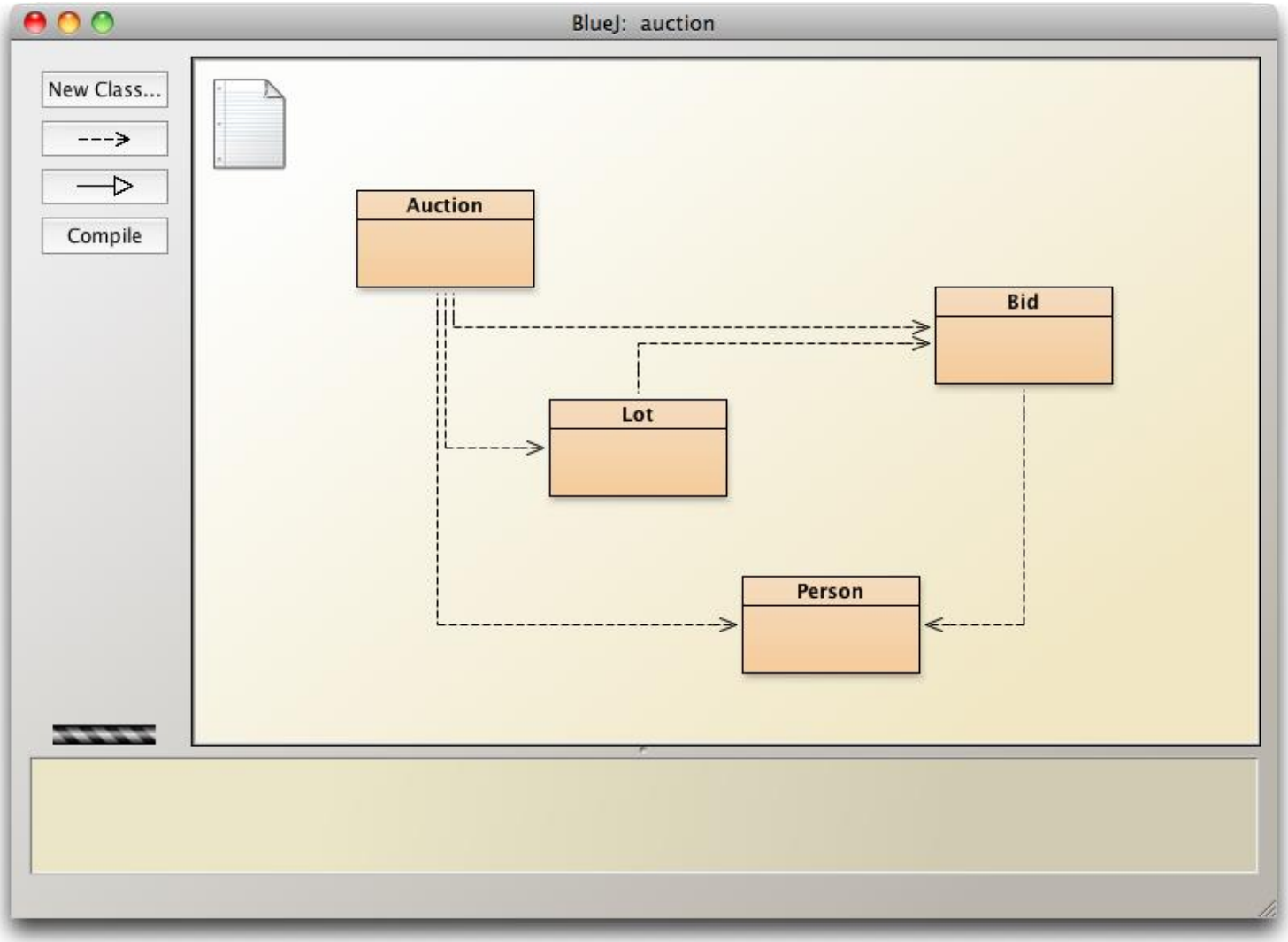
Review

- 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();
```

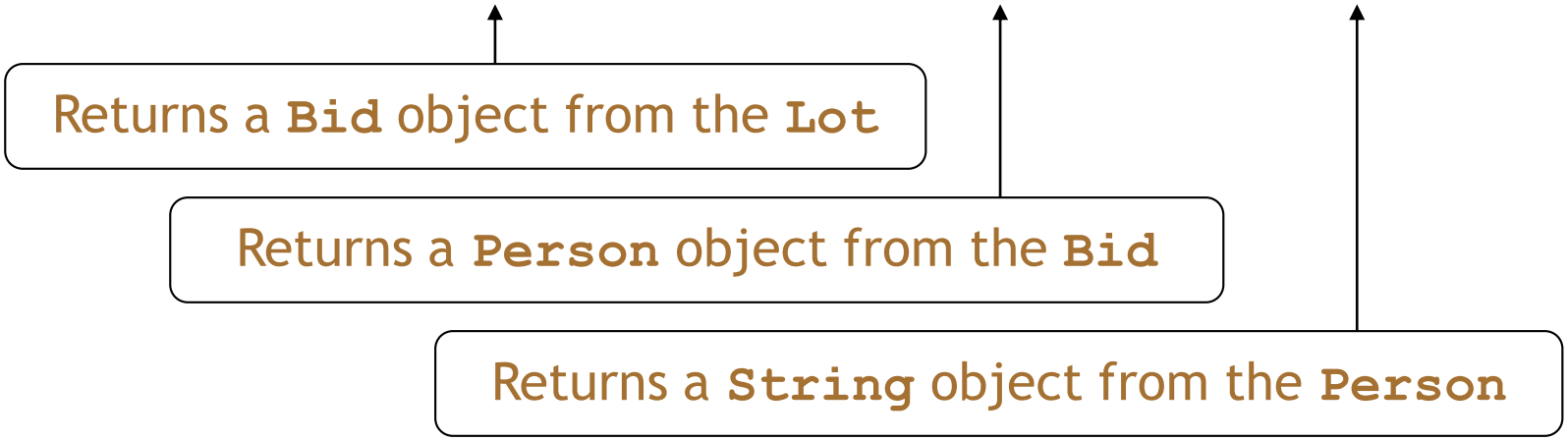
- We can use the anonymous object concept and *chain* method calls:

```
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();
```



Grouping objects

arrays

Fixed-size collections

- Sometimes the maximum collection size can be pre-determined.
- A special fixed-size collection type is available: an *array*.
- Unlike the flexible **`List`** collections, arrays can store object references or primitive-type values.
- Arrays use a special syntax.

The *weblog-analyzer* project

- Web server records details of each access.
- Supports analysis tasks:
  - Most popular pages.
  - Busiest periods.
  - How much data is being delivered.
  - Broken references.
- Analyze accesses by hour.

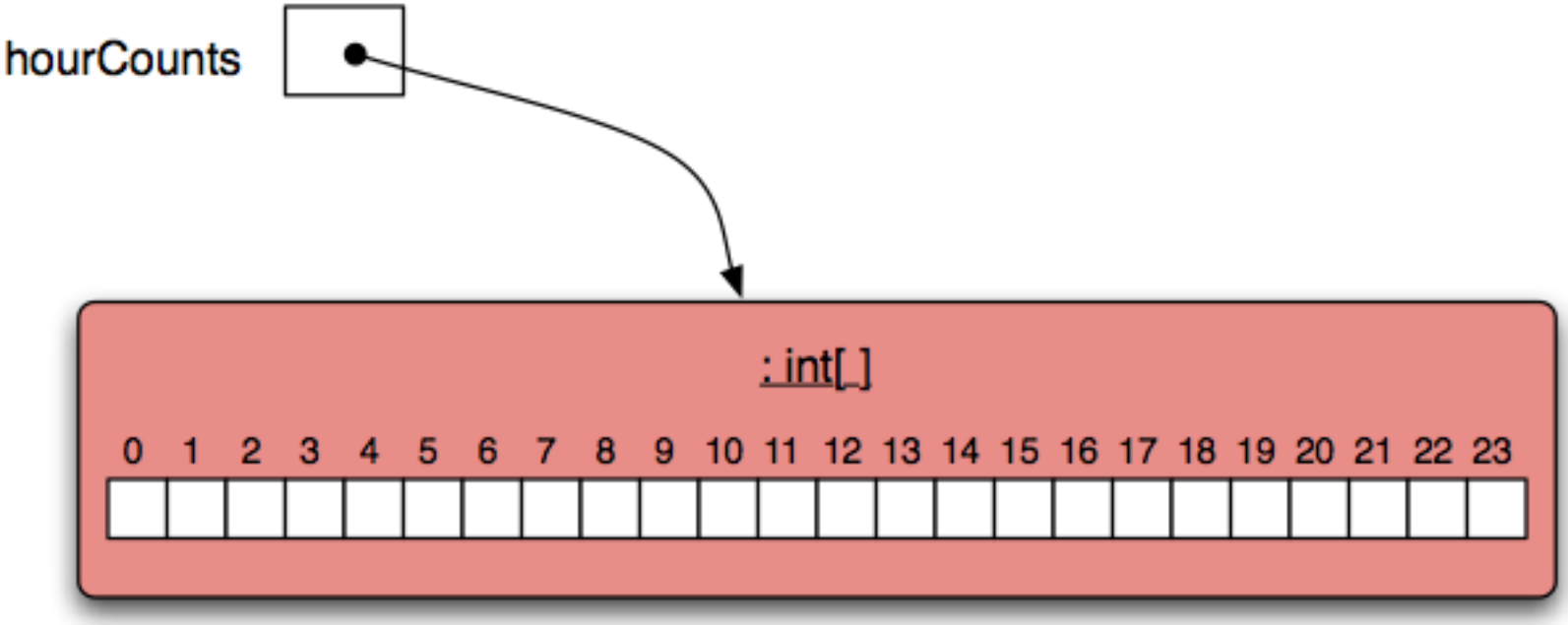
Creating an array object

```
public class LogAnalyzer
{
    private int[] hourCounts;
    private LogfileReader reader;

    public LogAnalyzer()
    {
        hourCounts = new int[24];
        reader = new LogfileReader();
    }
    ...
}
```

Array *variable declaration* — does *not* contain size

Array *object creation* — specifies size

The `hourCounts` array



Using an array

- Square-bracket notation is used to access an array element: `hourCounts[...]`
- Elements are used like ordinary variables.
- The target of an assignment:

```
hourCounts[hour] = ...;
```

- In an expression:

```
hourCounts[hour]++;
adjusted = hourCounts[hour] - 3;
```

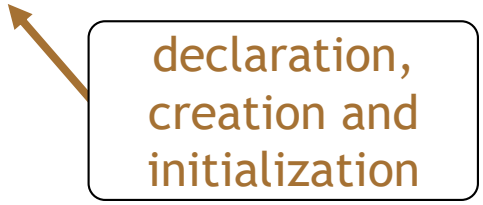
Standard array use

```
private int[] hourCounts;
private String[] names;

...

hourCounts = new int[24];

...

hourcounts[i] = 0;
hourcounts[i]++;
System.out.println(hourcounts[i]);
```

Array literals

- The size is inferred from the data.

```
private int[] numbers = { 3, 15, 4, 5 };
```



- Array literals in this form can only be used in declarations.
- Related uses require **`new`**:

```
numbers = new int[] {
    3, 15, 4, 5
};
```

Array length

```
private int[] numbers = { 3, 15, 4, 5 };

int n = numbers.length;
```

- NB: **`length`** is a field rather than a method!
- It cannot be changed – 'fixed size'.

The for loop

- There are two variations of the for loop, *for-each* and *for*.
- The for loop is often used to iterate a fixed number of times.
- Often used with a variable that changes a fixed amount on each iteration.

For loop pseudo-code

General form of the for loop

```
for(initialization; condition; post-body action) {
    statements to be repeated
}
```

Equivalent in while-loop form

```
initialization;
while(condition) {
    statements to be repeated
    post-body action
}
```

A Java example

for loop version

```
for(int hour = 0; hour < hourCounts.length; hour++) {
    System.out.println(hour + ": " + hourCounts[hour]);
}
```

while loop version

```
int hour = 0;
while(hour < hourCounts.length) {
    System.out.println(hour + ": " + hourCounts[hour]);
    hour++;
}
```

Practice

- Given an array of numbers, print out all the numbers in the array, using a for loop.

```
int[] numbers = { 4, 1, 22, 9, 14, 3, 9};

for ...
```

Practice

- Fill an array with the Fibonacci sequence.

0 1 1 2 3 5 8 13 21 34 ...

```
int[] fib = new int[100];

fib[0] = 0;
fib[1] = 1;

for ...
```

for loop with bigger step

```
// Print multiples of 3 that are below 40.
for(int num = 3; num < 40; num = num + 3) {
    System.out.println(num);
}
```

Review

- Arrays are appropriate where a fixed-size collection is required.
- Arrays use a special syntax.
- For loops are used when an index variable is required.
- For loops offer an alternative to while loops when the number of repetitions is known.
- Used with a regular step size.