Grouping objects

Introduction to collections

6.0

Main concepts to be covered

- Collections to group objects (e.g. **`ArrayList`**)
- Builds on the *abstraction* theme to simplify a problem into components
- Start making use of existing Java *library classes* to save time coding

The requirement to group objects

- Many applications involve *collections* of objects:
  - Personal organizers
  - Library catalogs
  - Student record systems
  - Music organizer
- Number of items to be stored is *dynamic*
  - Items added
  - Items deleted
  - Items retrieved

An organizer for music files

COLLECTION
- Contains a group of songs

ITEMS
- Songs are stored as its filename only
- No pre-defined limit on the number of songs

OPERATIONS
- Song files may be added
- Song files may be deleted
- How many song files are stored (i.e. size)
- Get a song filename from the group

Explore the *music-organizer-v1* project

Music collection example
MusicOrganizer

CLASS

* *MusicOrganizer* containing various song files

FIELDS

* Dynamic *ArrayList* storage for a varying number of song *String* filenames

METHODS

* *addFile*
* *removeFile*
* *getNumberOfFiles*
* *listFile*

Class libraries

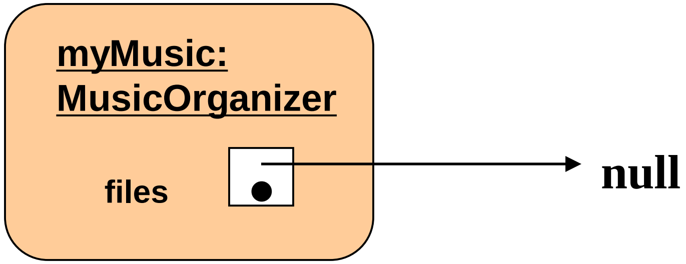
- Provides many useful classes (e.g. *String*)
- Don't have to write class from scratch
- Java calls its libraries *packages*
- Use library classes the same way as classes that you write (i.e *constructor/methods*)
- But do not appear in BlueJ class diagram
- Grouping objects is a recurring requirement that is handled in the **`java.util`** package (e.g. *ArrayList*)
- *ArrayList* library class will:
  - group the unsorted but ordered items
  - store item details
  - handle general access to the items

```java
import java.util.ArrayList;    // import statement
                               // first line of file
/**                            // before class definition
 * ...
 */
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 …
  - collection type: **ArrayList**
  - containing objects of type: **<String>**
- We say … "ArrayList of String"

```
private ArrayList<String> files;
```



*** Only 1 field named *files* is defined for the entire class**

Generic classes for items of any type

`ArrayList<parameter-type>`

- These collections are known and defined as *parameterized* or *generic* types
- ***parameter type*** between the angle brackets is the object type of the items in the list
  - **`ArrayList<Person>`**
  - **`ArrayList<TicketMachine>`**
- An ArrayList may store any object type, but ALL objects in the list will be the same type

Creating an ArrayList object in the constructor

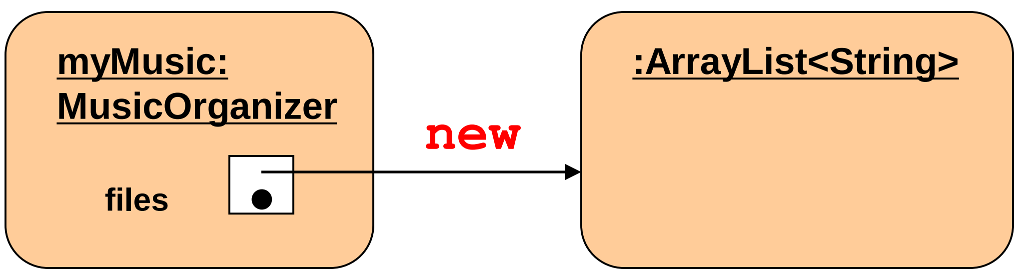
- In Java versions prior to version 7

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

- Java 7 introduced ‘diamond notation’

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

where the type parameter can be inferred from the variable it is being assigned to

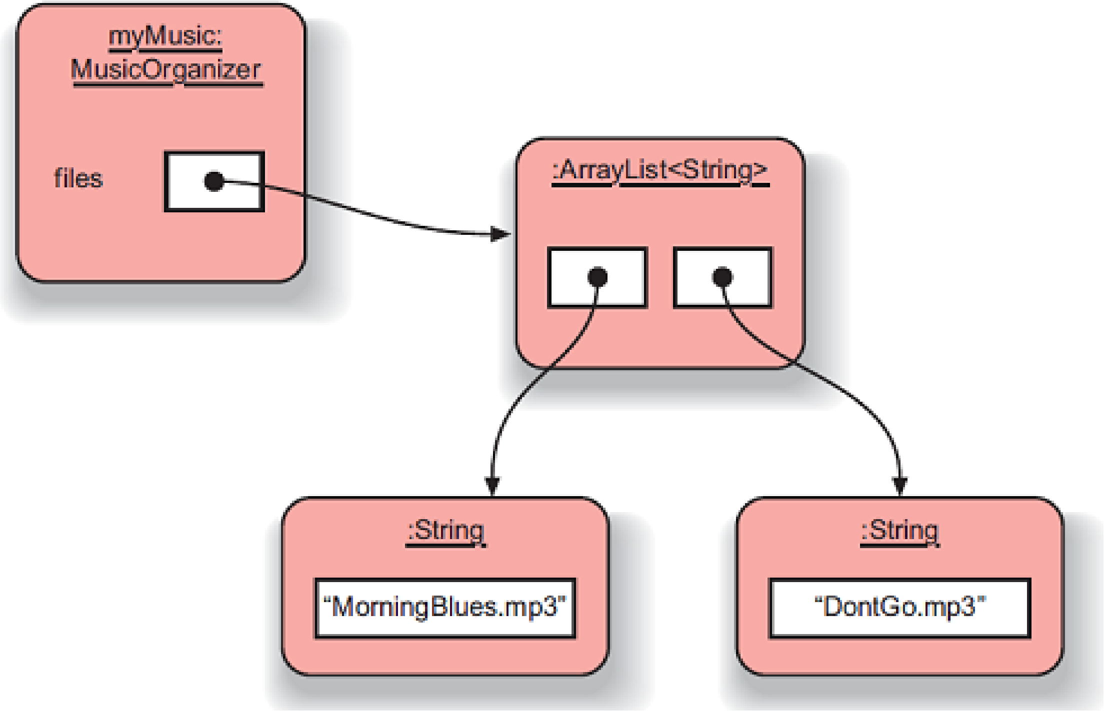


Key methods of class *ArrayList*

The ***`ArrayList`*** class implements list functionality with methods for the following operations:

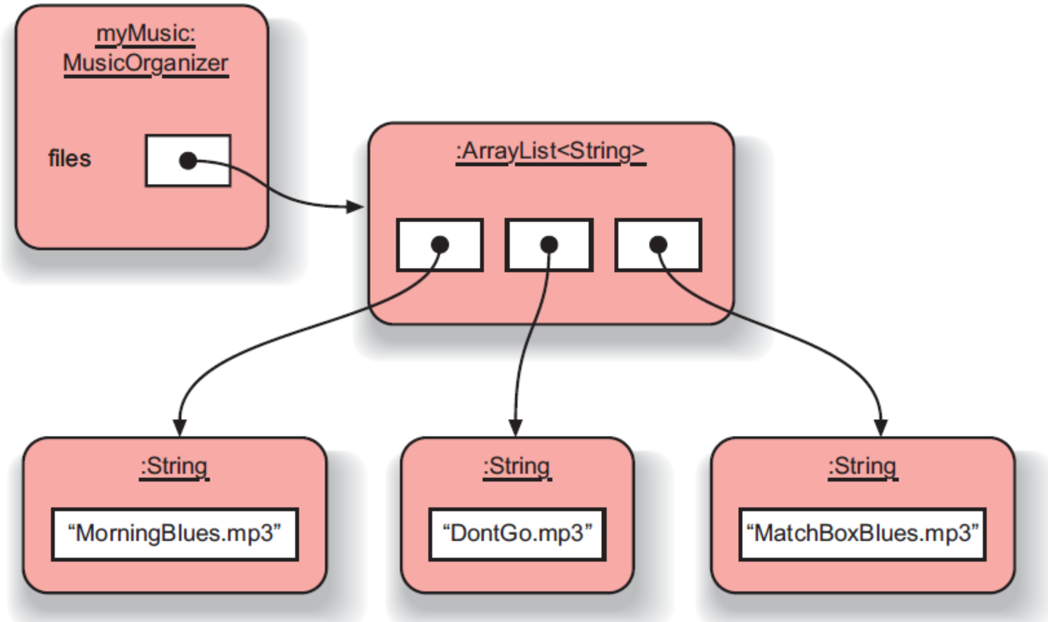
- **`add(item)`**
- **`add(index, item)`**
- **`remove(item)`**
- **`remove(index)`**
- **`get(index)`**
- **`size()`**
- **`isEmpty()`**

Object structures with *ArrayList* collections



- **Only a single field that stores an object of type *ArrayList<String>***
- **All work to access and manage the data is done in *ArrayList* object**
- **Benefits of abstraction by not knowing details of *how* work is done**
- **Helps us avoid duplication of information and behavior**

Adding a third file



- **Dynamic capacity with ability to increase and/or decrease as needed with its *add( )* and *remove( )* methods**
- **Keeps an internal count of the number of items with *size( )* method returning that count**
- **Maintains the items in the order inserted with each new item added to the end of the list**
- **As an item is removed, all items following after the removed item are shifted up and forward in order to fill the removed item's space**

Features of the collection

- It increases its capacity as necessary
- It keeps a private count of the number of items in the list
  - **size()** accessor
- It keeps the objects in order of adding, but is otherwise unsorted
- Details of how this is done are hidden
  - Does that matter?
  - Does not knowing 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

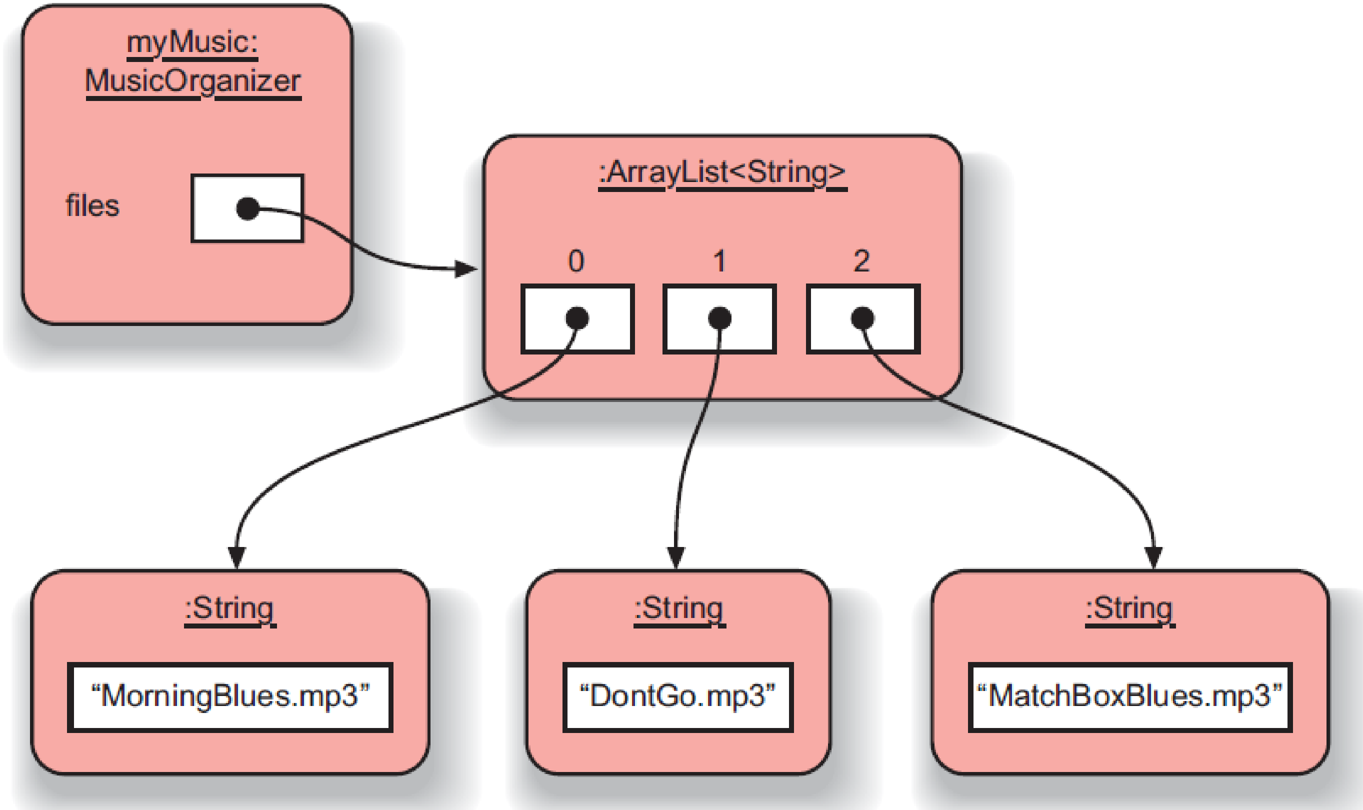
Using the collection

```
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*)

ArrayList Index numbering



- **Implicit numbering which starts with *index* 0 (same as String class)**
- **Last item in the collection has the index *size-1***
- **Thus, valid *index* values would be between [0 . . . size( )-1]**

Retrieving an object from the collection

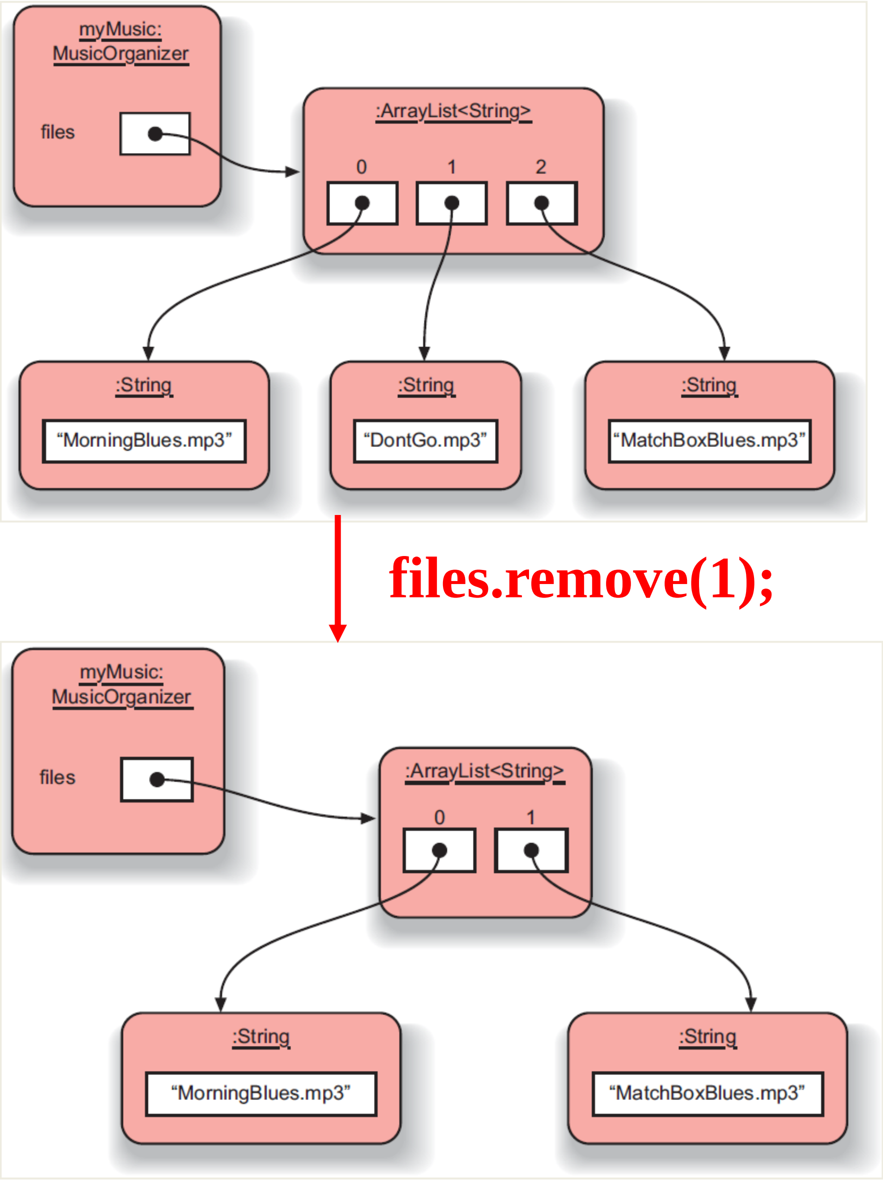
```
public void listFile(int index)
{
    if(index >= 0 && index < files.size()) {
        String filename = files.get(index);
        System.out.println(filename);
    }
    else {
        // This is not a valid index.
    }
}
```

Index validity checks between [0 ... size-1]

Retrieve and print the file name

Needed? (Error message?)

Removal may affect numbering



- **Removal process may change *index* values of other objects in the list**
- **Collection moves all subsequent items up by 1 position to fill the gap**
- **Indices of items in front of (preceding) the removed item are UNCHANGED**
- **Indices of items after (following) the removed item are decreased by 1**
- **Same "shifting" of items may also occur if adding new items into positions other than the end**

The general utility of indices

- Index values:
  - start at 0
  - are numbered sequentially
  - have no gaps in consecutive objects
- Using integers to index collections has a general utility:
  - next: `index + 1`
  - previous: `index – 1`
  - last: `list.size( ) – 1`
  - the first three: items at indices `0, 1, 2`
- We could use loops and iteration to access 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
- 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* or *iterative control structures* 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
}
```

Statement(s) to be repeated

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

<u>For each</u> *element* in *collection*, do the things in the *<u>loop body</u>*.

**** where `element` is indeed a variable declaration of type `ElementType` and the variable is known as the *<u>loop variable</u>***

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*

- ***for*** **keyword introduces loop with details between ( )**
- **loop variable *filename* is declared of type *String***
- **loop body repeated for each element in *files* ArrayList**
- **each time, variable *filename* holds one of the elements**
- **allows access to the object for that particular element**

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
- But the for-each loop does NOT provide the index position of the current element

Selective processing

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

```
public 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

**** using *if* statement to only print filenames matching the *searchString***

Critique of for-each

- Only use for any type of collection
- Accesses each element in sequence
- Same action for each element but may use selective filter using *if* statements
- Easy to write
- Termination happens naturally
- But, the collection cannot be changed
- There is no index provided during access
  - Not all collections are index-based
- Can NOT stop part way through loop
  - e.g. Find-the-first-that-matches
- Provides *definite iteration* of ENTIRE list
  - a.k.a. *bounded iteration*

for-each

PROS
- easy to use
- access to ALL items one-by-one
- ability to change the state of the item
- terminates automatically
- selective filter using *if-else* statements
- actions in body may be complicated with multiple lines
- use on ANY type of collection
- abstraction from details of how handling occurs

CONS
- no index provided
- can NOT stop during looping
- definite iteration of ALL items
- can NOT remove or add elements during loop
- use for collections only
- access must be to ALL items in sequence [0 to size-1]

Grouping objects

Indefinite iteration - the while loop

Main concepts to be covered

- The difference between iterations:
  - definite … size
  - indefinite (unbounded) … 0 - infinite
- 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
- There may be an absolute limit
  - i.e. check EVERY possible location
- Or, it may not be any at all
  - i.e. check 0 locations (you had them!)
- 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
  - The while loop supports flexibility
- We use a *boolean* condition to decide whether or not to keep iterating
- Maybe NO need to search to the end
- This is a *very* flexible approach
- Not tied to collections

While loop pseudo code

General form of a while loop

`while` keyword

boolean test (true or false)

```
while(loop condition) {
    loop body
}
```

Statements 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(true)

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

while(!(false))

```
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?

Infinite loop

for-each == while

```
public void listAllFiles()
{
    for(String filename : files) {
        System.out.println(filename);
    }
}
```

```
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

1. We have declared an index variable
2. The condition must be expressed correctly
3. We have to fetch each element
4. The index variable must be incremented explicitly

while loop search

PROS

- can stop at any time during looping
- indefinite iteration of SOME items using loop condition
- may change collection during loop
- use explicit index variable inside and outside of loop
- index variable records location of item at all times

CONS

- more effort to code
- requires index looping variable declaration
- maintain looping variable and manually increment
- correctly determine loop condition for termination
- must .get item using index to access the item
- NOT guaranteed to stop with possible infinite loop

for-each versus while

- **for-each**
  - easier to write
  - safer because it is guaranteed to stop
  - access is handled for you

Access ALL items without changing collection

- **while**
  - don't *have* to process entire collection
  - doesn't have to be used with a collection
  - take care to watch for an *infinite loop*

Access only SOME items, includes a record of the index location, and also could be used for non-collections

Searching a collection

- A re-occurring fundamental activity
- Applicable beyond collections
- Indefinite iteration because we don’t know exactly where to look
- We must code for both success (stops midway) and failure (after all searched) using an exhausted search
- Either MUST make the loop condition *false* to terminate the loop
- Even works if collection is empty

Finishing a search

So when do we finish a search?

- No more items to check:

```
index >= files.size()
```

OR

- 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.

Search condition

>= becomes <

FINISH search when:

- No more items OR Item is found

```
index >= files.size() ||  found
```

CONTINUE search ***while***:

- Still more items AND Item is *not* found

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

Search condition

>= becomes <

FINISH search when:

- No more items OR Item is found

```
index >= files.size() ||  found
```

CONTINUE search *while*:

- Still more items AND Item is *not* found

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

Search condition

OR becomes AND

FINISH search when:

- No more items **OR** Item is found

```
index >= files.size() ||  found
```

CONTINUE search ***while***:

- Still more items **AND** Item is *not* found

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

Search condition

OR becomes AND

FINISH search when:

- No more items **OR** Item is found

```
index >= files.size() || found
```

CONTINUE search *while*:

- Still more items **AND** Item is *not* found

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

Search condition

true becomes !true

FINISH search when:

- No more items OR Item is <u>found</u>

```
index >= files.size() ||  found
```

CONTINUE search ***while***:

- Still more items AND Item is <u>*not* found</u>

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

Search condition

true becomes !true

FINISH search when:

- No more items OR Item is found

```
index >= files.size() || found
```

CONTINUE search *while*:

- Still more items AND Item is *not* found

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

Searching a collection (using *searching*)

```
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 (using *found*)

```
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.
```

Method *findFirst*

```
public int findFirst(String searchString)
{
    int index = 0;
    boolean searching = true;

    while(searching && index < files.size())
    {
        String filename = files.get(index);
        if(filename.contains(searchString))
        {                                   // Match found
            searching = false;              // Stop searching
        }
        else                                // Not found here
        {                                   // Keep searching
            index++;                        // Move to next item
        }
    }

    if(searching)                           // NO match found
    {
        return -1;                          // Return out-of-bounds
    }                                       //   index for failures
    else
    {                                       // Return item index of
        return index;                       //   where it is found
    }
}
```

Indefinite iteration

- Does the search still work if the collection is empty (but not null)?
  - Yes! The loop's body would NOT be entered in that case.

- Important feature of *while*:
  - The body of the *while* could be executed *zero or more* times.

While with non-collections

```
// Print all even numbers from 2 to 30
```

local variable

START: index start

STOP: index end

increment

```
int index = 2;
while(index <= 30)
{
    System.out.println(index);
    index = index + 2;
}
```

NOTE: This while loop uses *definite iteration*, since it is clear from the start exactly how many times the loop will be repeated. But, we could NOT have used a *for-each* loop, because there is no collection of items.

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

String equality

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

tests identity

Do not use!!

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

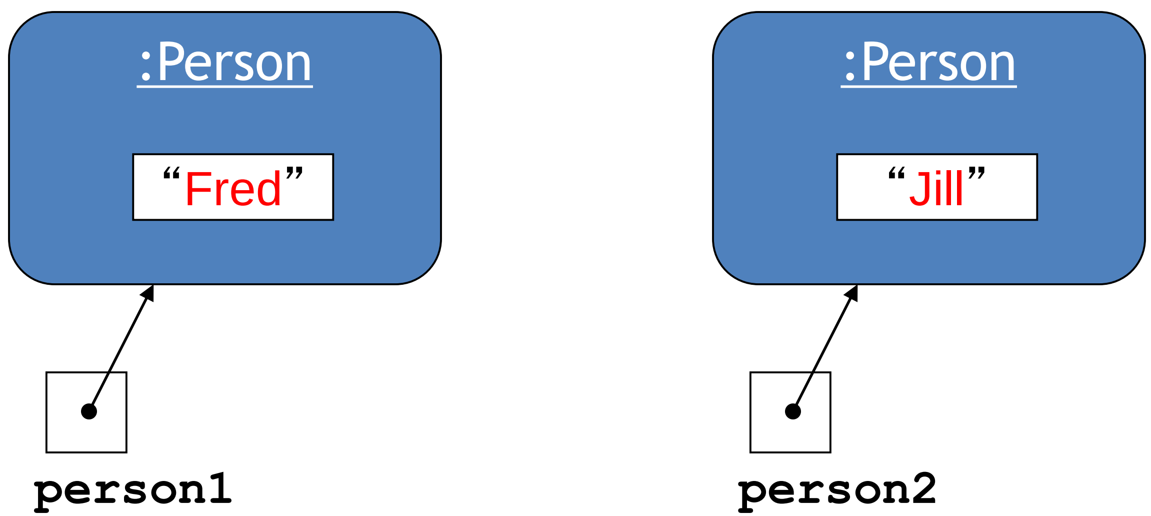
tests equality

Important:

Always use ***.equals*** to test String equality!

Identity vs equality 1

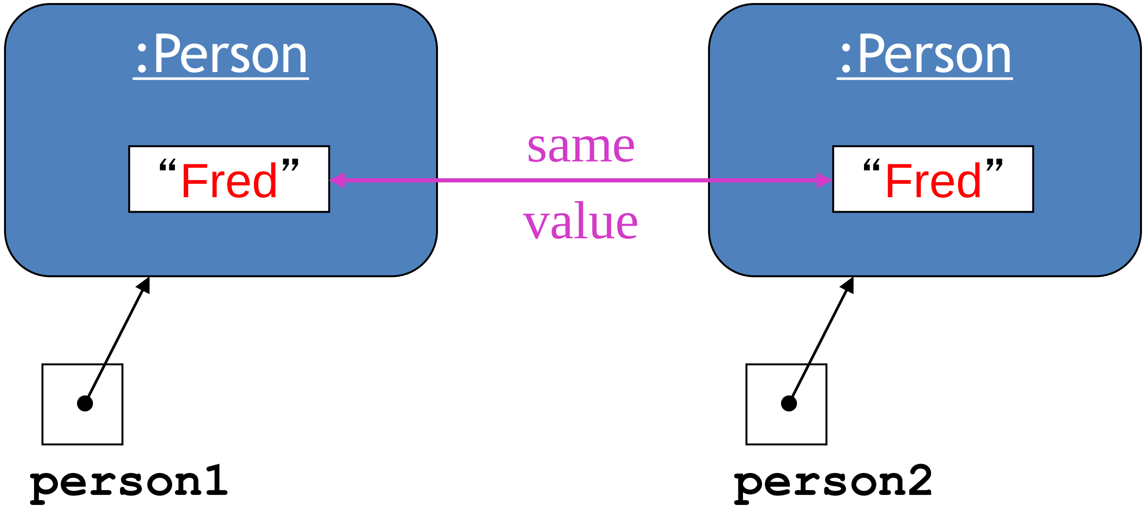
Other (non-String) objects:



`person1 == person2 ?` **`false`**

Identity vs equality 2

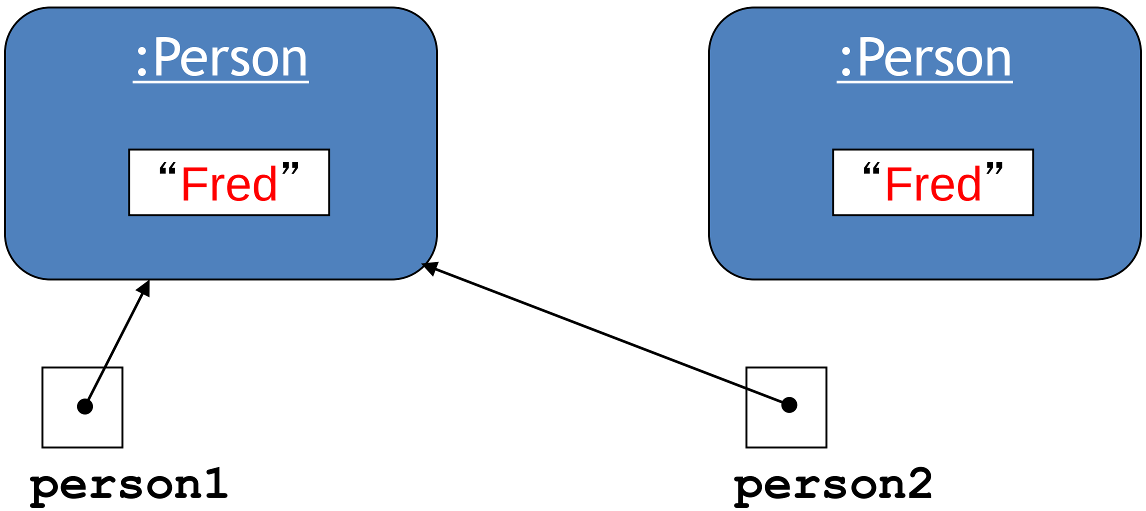
Other (non-String) objects:



person1 == person2 ? **false**

Identity vs equality 3

Other (non-String) objects:

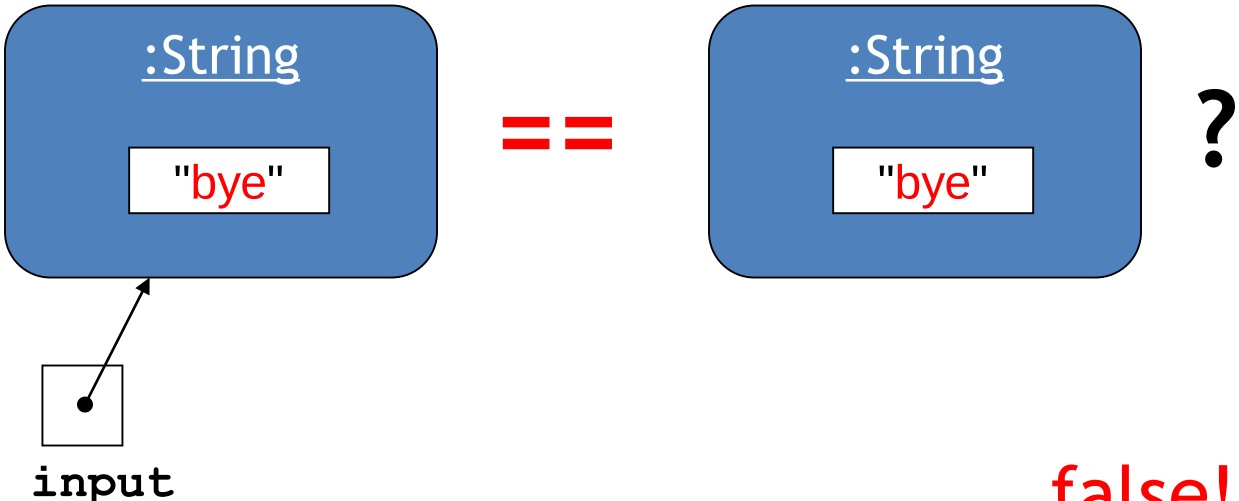


person1 == person2 ? **true**

Identity vs equality (Strings)

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

== tests identity

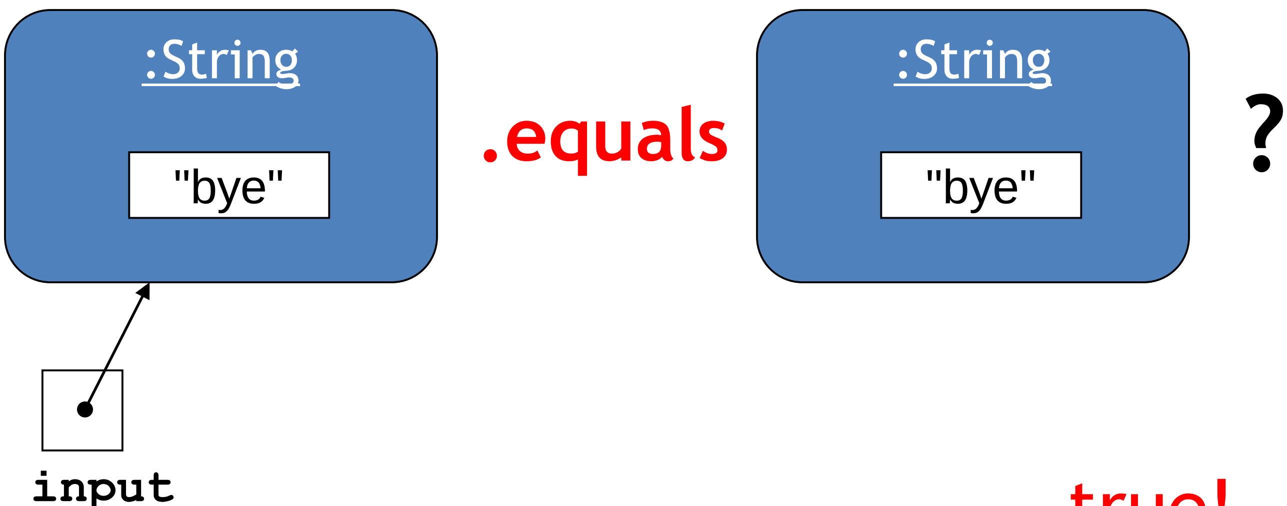


false!

Identity vs equality (Strings)

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

equals tests equality



true!

The problem with Strings

- The compiler <u>merges</u> 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

```
private ArrayList<String> tracks;
```

- No separate id for artist, title, etc…
- Make ***Track*** class with separate fields

```
private String artist;
private String title;
private String filename;
```

- Changes collection of music tracks

```
private ArrayList<Track> tracks;
```

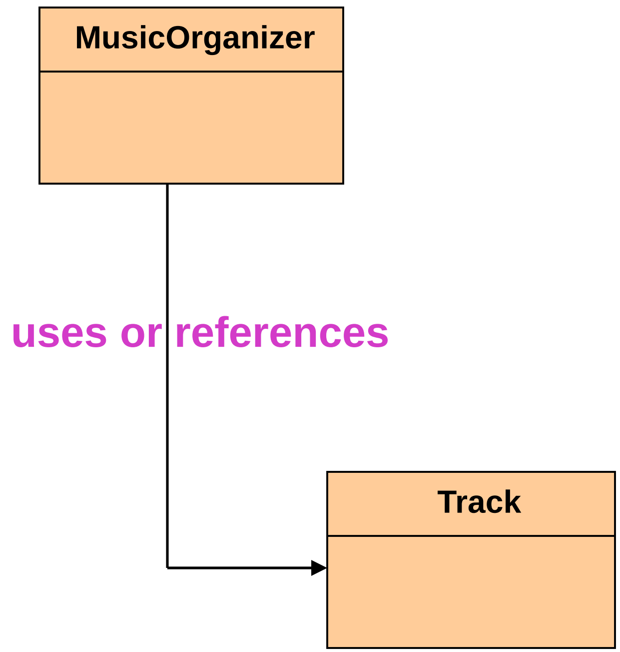
ArrayList of non-String objects

```
public class MusicOrganizer
{
    // ArrayList of Track objects
    private ArrayList<Track> tracks;
        :
        :
}



// non-String Track class definition
public class Track
{
    private String artist;
    private String title;
    private String filename;

        :
        :
}
```

Class diagram

```
public class MusicOrganizer
{
    private ArrayList<Track> tracks;

        :
        :
}
```

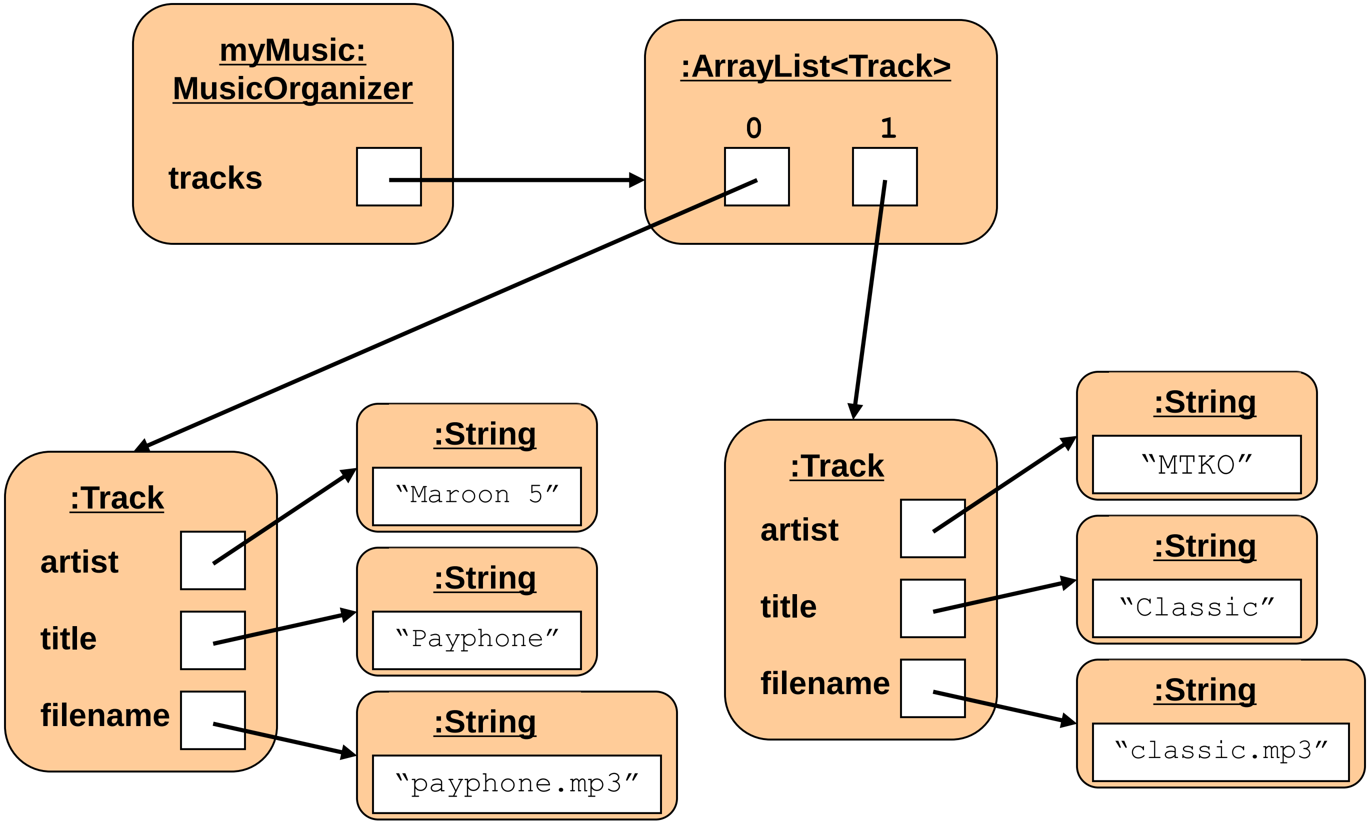
```
public class Track
{
    private String artist;
    private String title;
    private String filename;

        :
        :
}
```



Object diagram

Suppose the project consists of the following:

- 1 object instance of the ***MusicOrganizer*** class named ***myMusic***
- 2 instances of ***Track*** items in the ***tracks*** ArrayList field of ***myMusic***
  - ***new Track***("Maroon 5", "Payphone", "payphone.mp3")
  - ***new Track***("MTKO", "Classic", "classic.mp3")



Grouping objects

Iterator objects

`Iterator` type

- Third variation to iterate over a collection
- Uses a *while* loop and *Iterator* object
- But NO integer index variable
- Takes advantage of abstraction with use of library class (like *for-each*)
- *import java.util.Iterator;*
- *Iterator* class vs. *iterator( )* method

`Iterator` and `iterator()`

- Collections (e.g. ArrayList) 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
}
```

- **Declare variable *it* as type *Iterator* of *ElementType***
- **Use *iterator()* method of collection (e.g. ArrayList) and assign the returned *Iterator* object to variable *it***
- ***it* object *indexes* to the first element in the collection**
- ***it.hasNext()* checks to see if there is an object at the index**
- ***it.next()* will get the actual object and advance the index**

Iterator object example

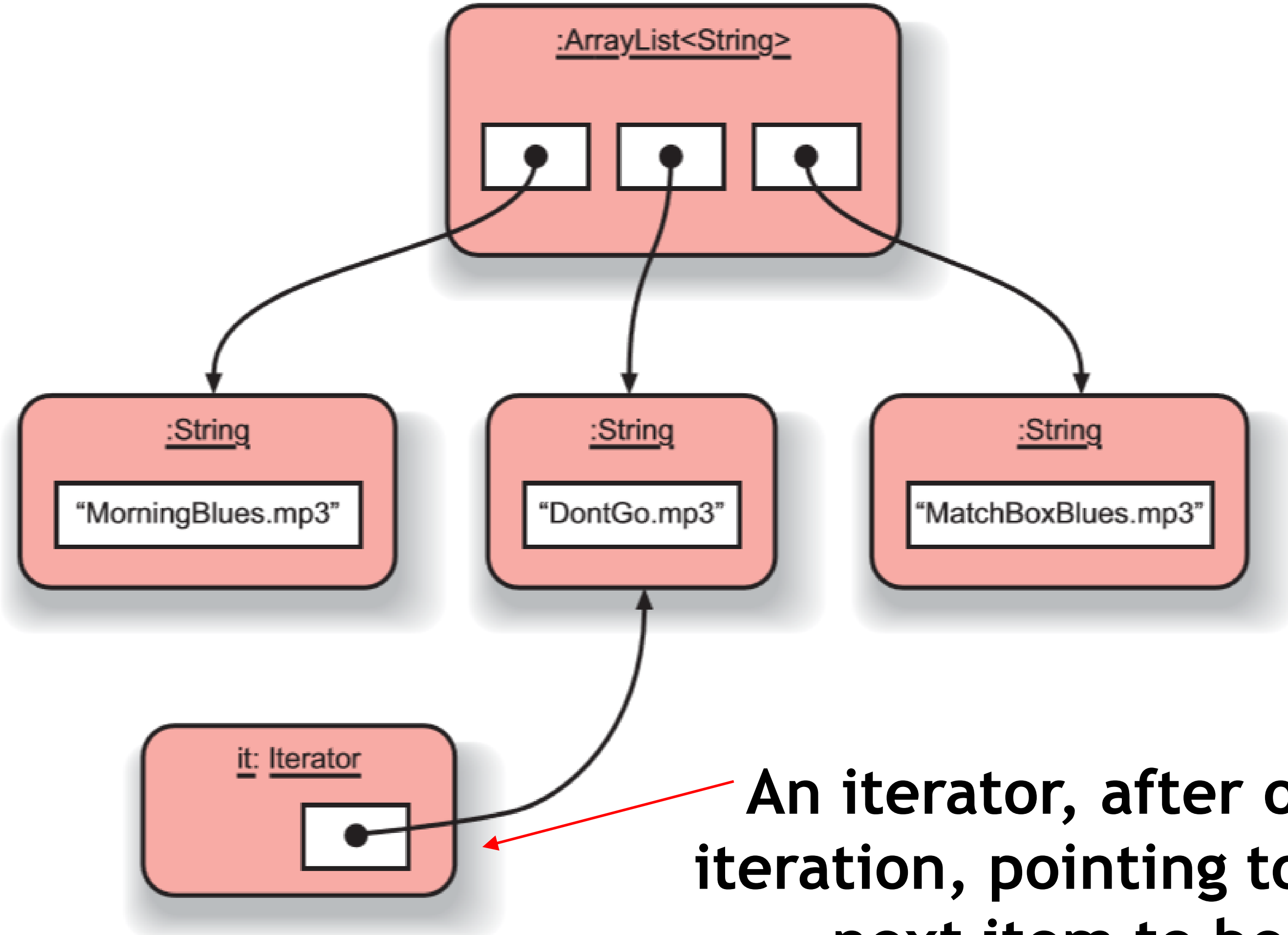
`java.util.Iterator`

returns an `Iterator` object

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

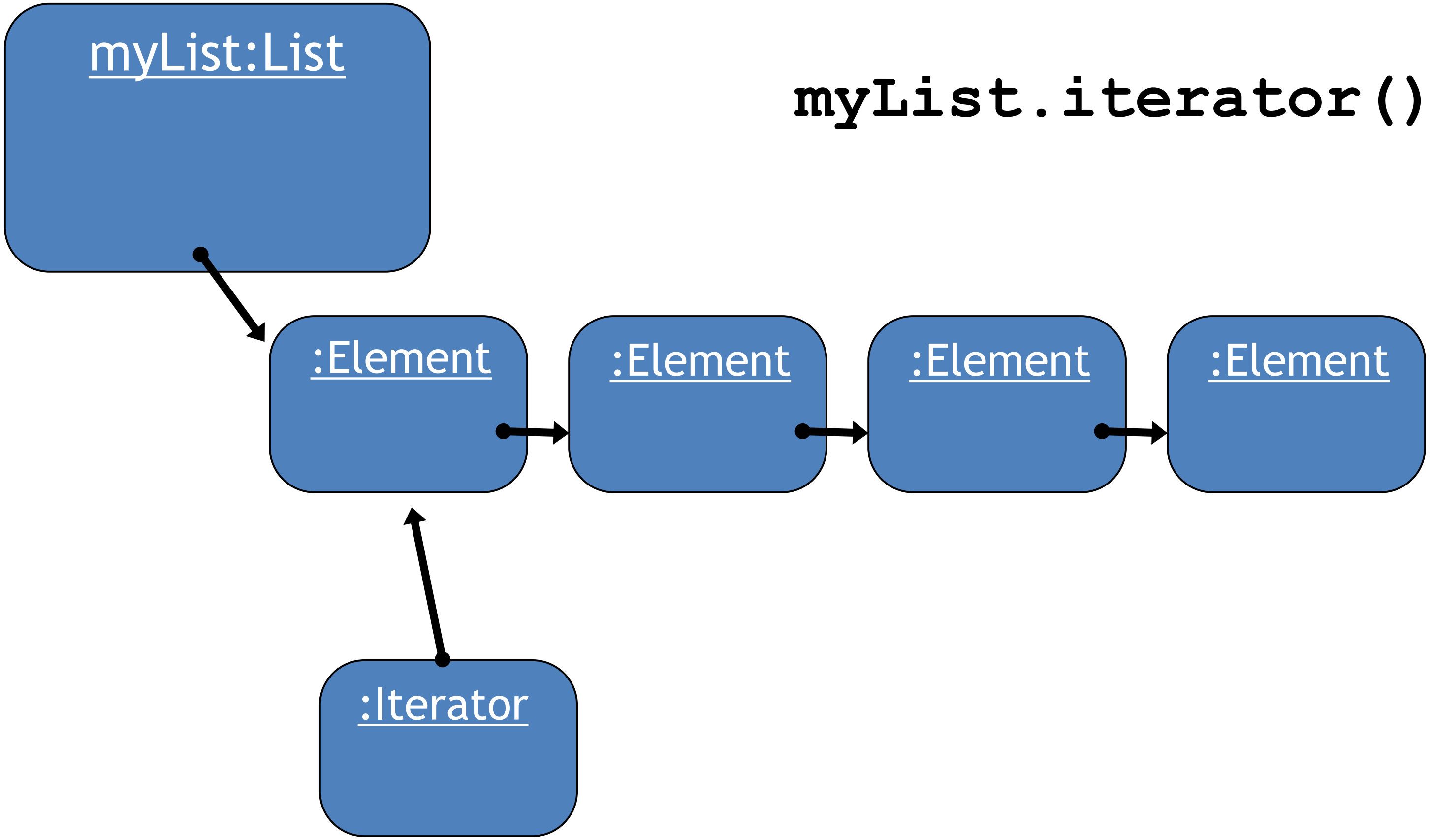
- **Prints ALL tracks in the collection (like while & for-each)**
- **Still use *while* … BUT do not need an index variable**
- ***Iterator* keeps track of current location, if there are any more items (*hasNext*) and which one to return (*next*)**
- ***Iterator.next* returns next item AND moves past that item (can NOT go back)**

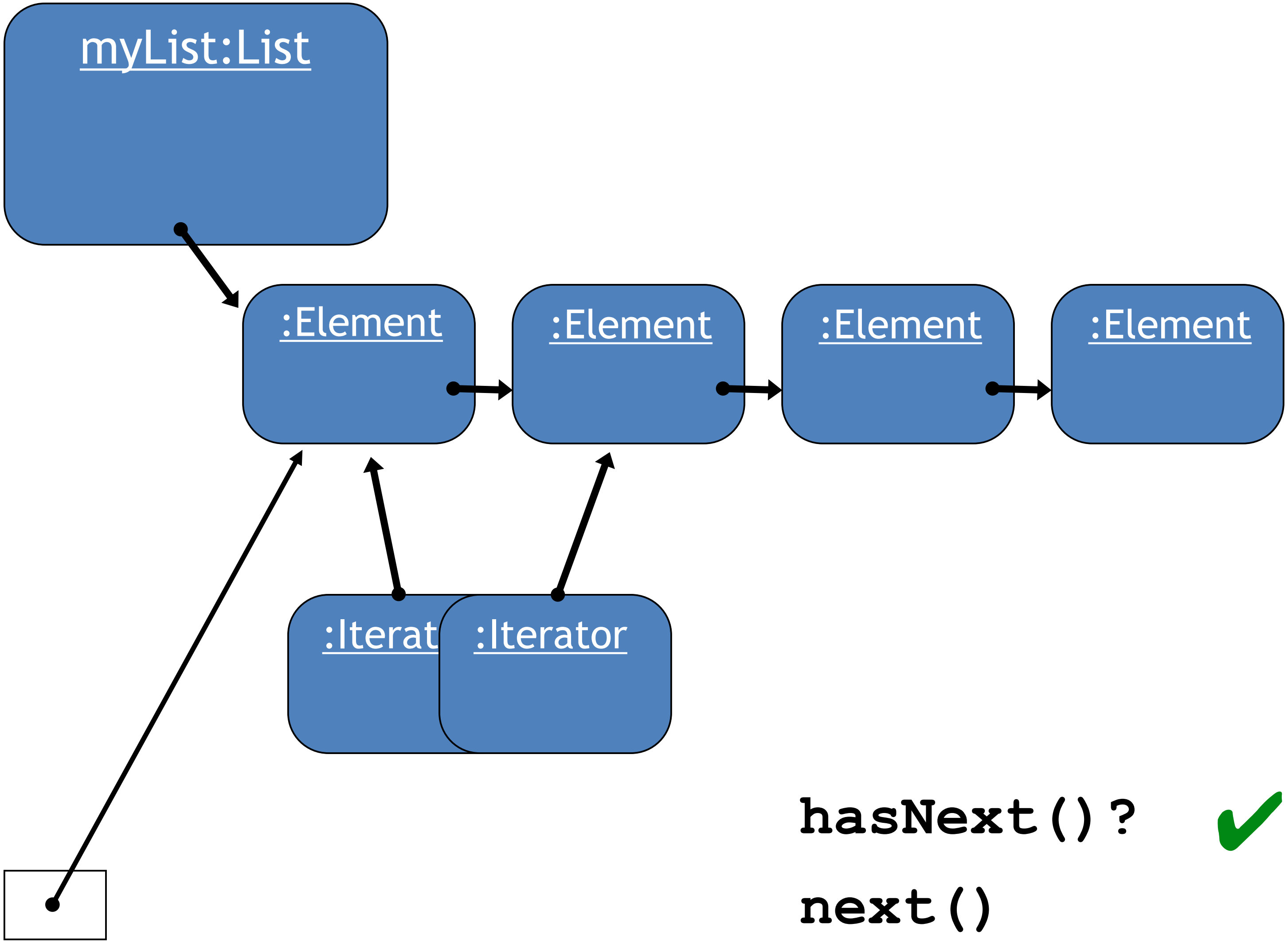
Iterator object



An iterator, after one iteration, pointing to the next item to be processed.

Iterator mechanics

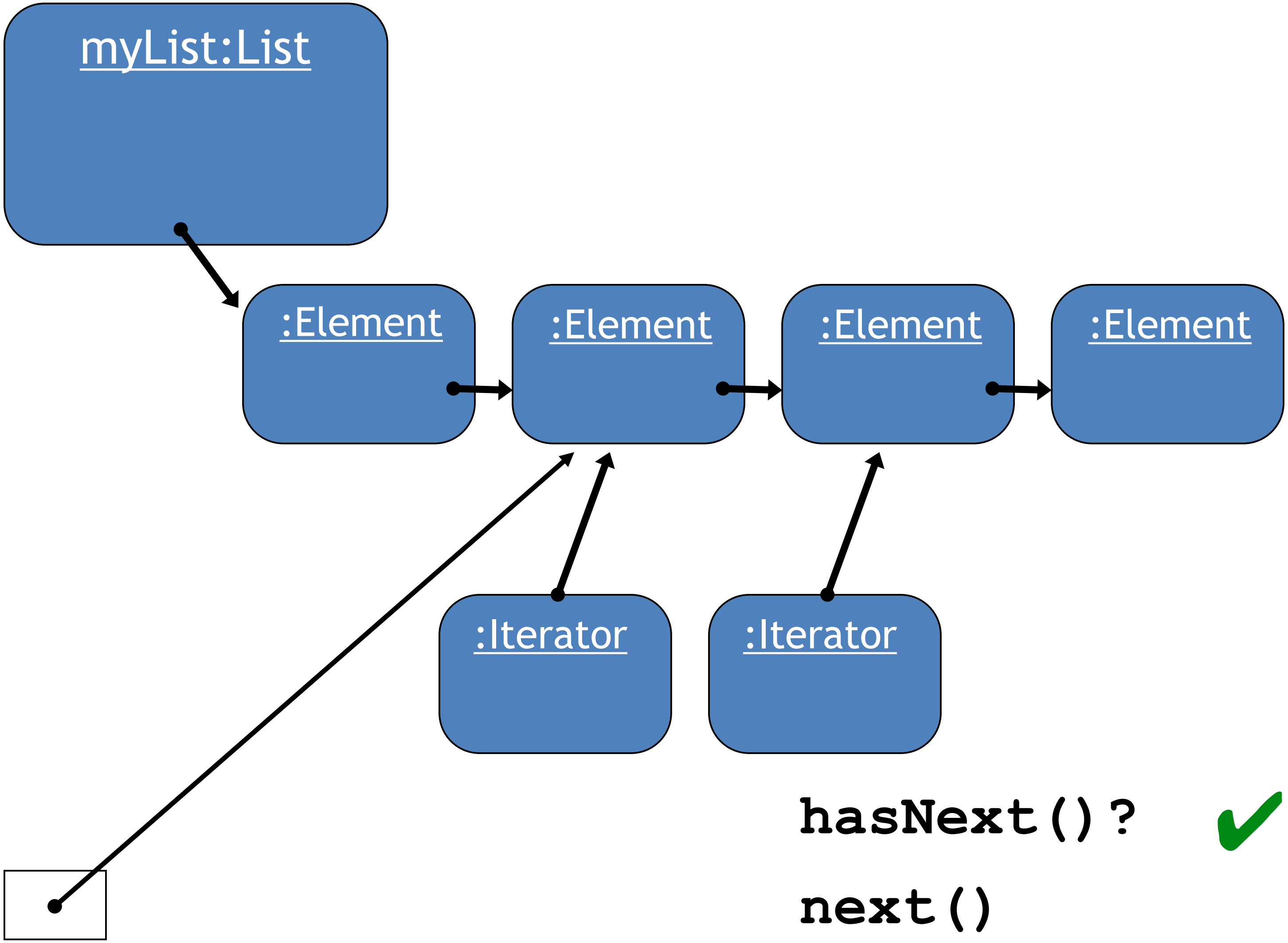

myList:List
myList.iterator()
:Element
:Element
:Element
:Element
:Iterator



`hasNext()?` ✔

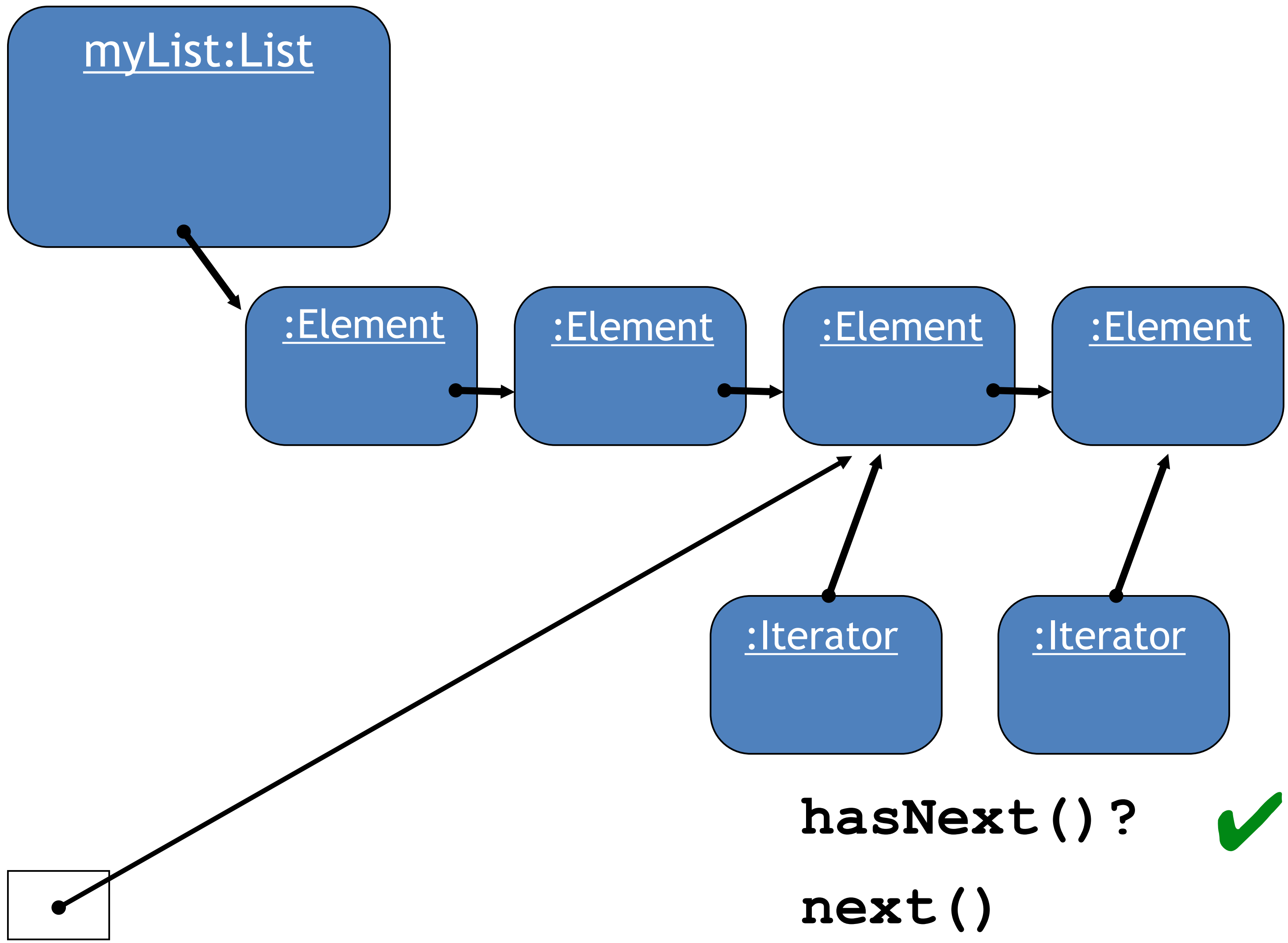
`next()`

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



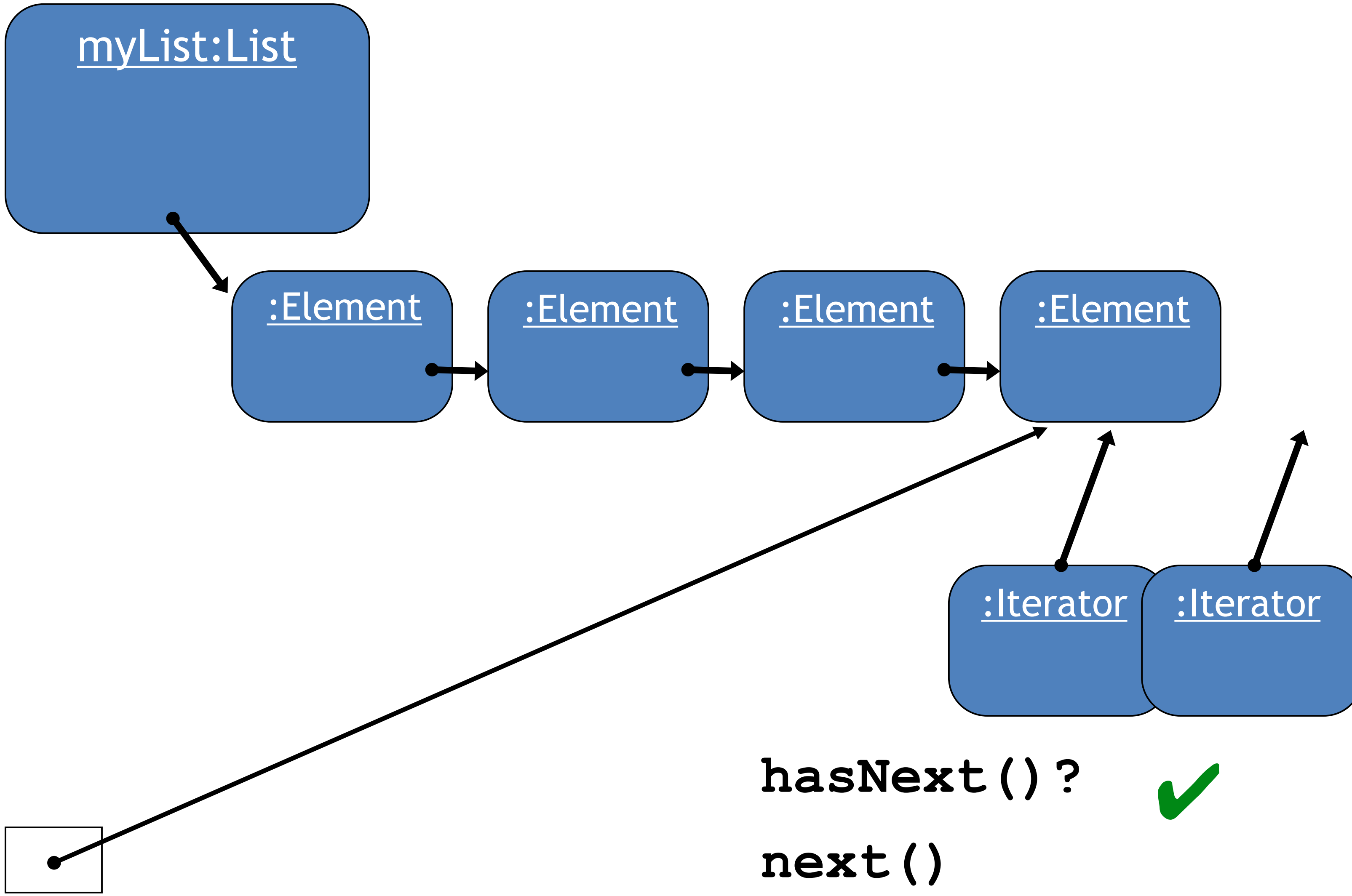
hasNext()? ✔

next()



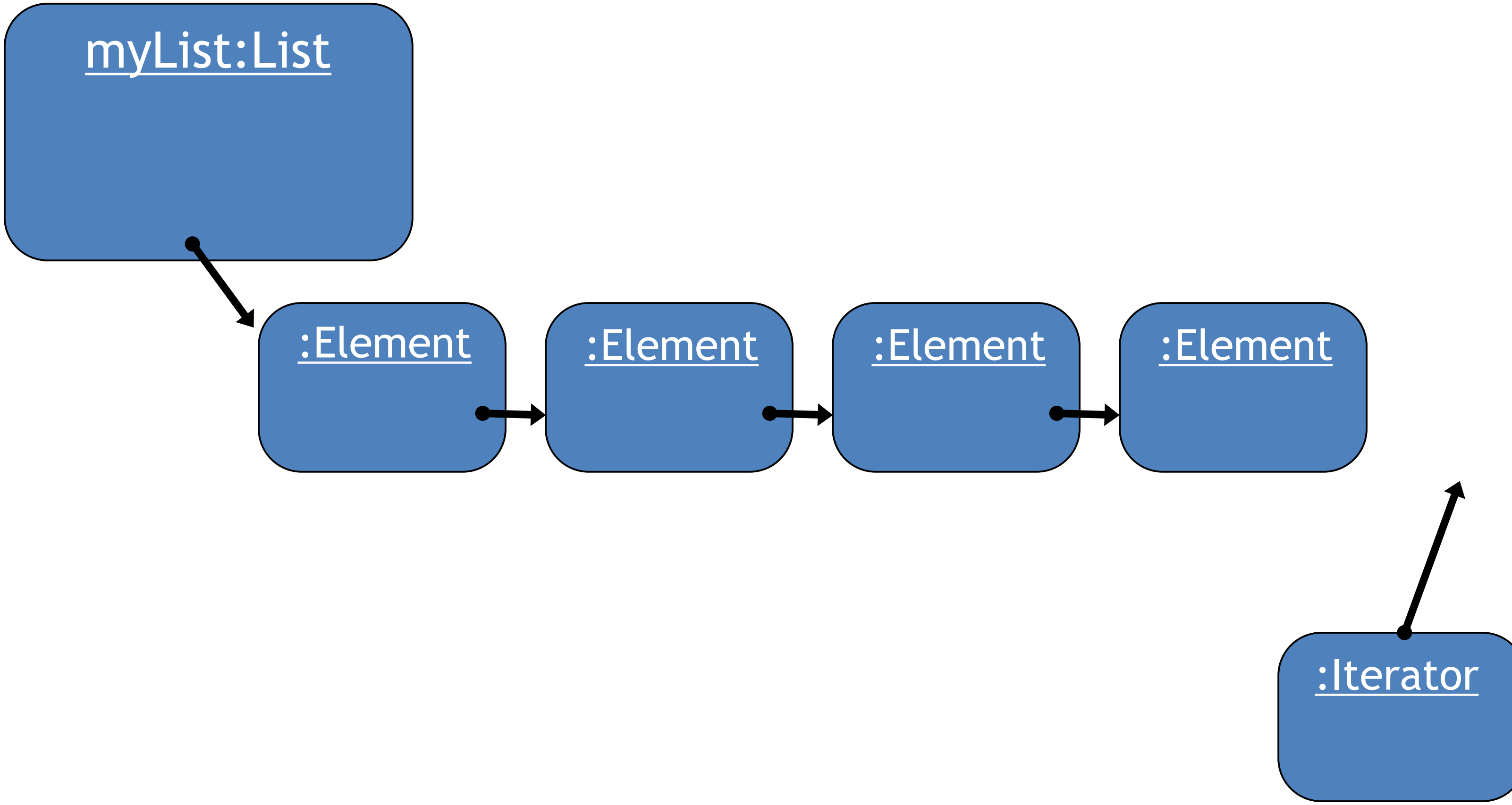
`hasNext()?` ✔

`next()`



`hasNext()?` ✔

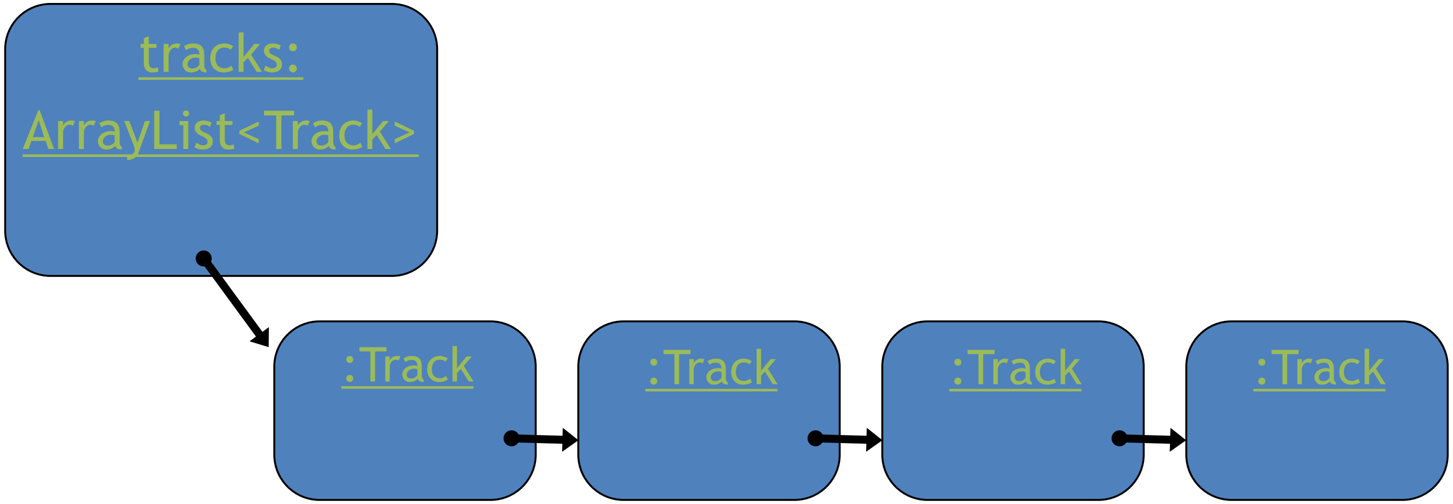
`next()`

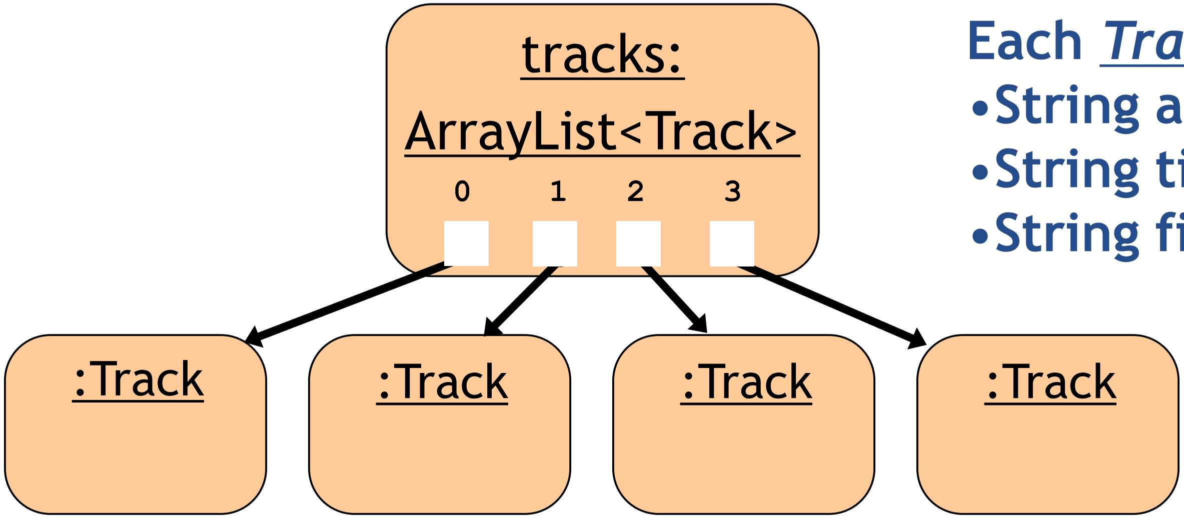


`hasNext()?` ✗

Track example

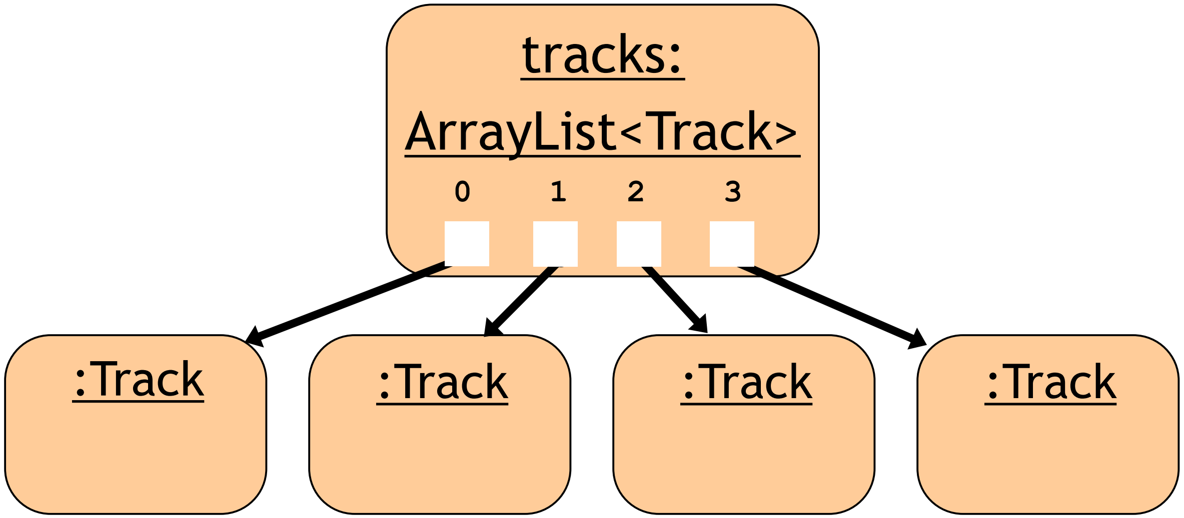
```
private ArrayList<Track> tracks;
```





Each *Track* has:
- **String artist**
- **String title**
- **String filename**

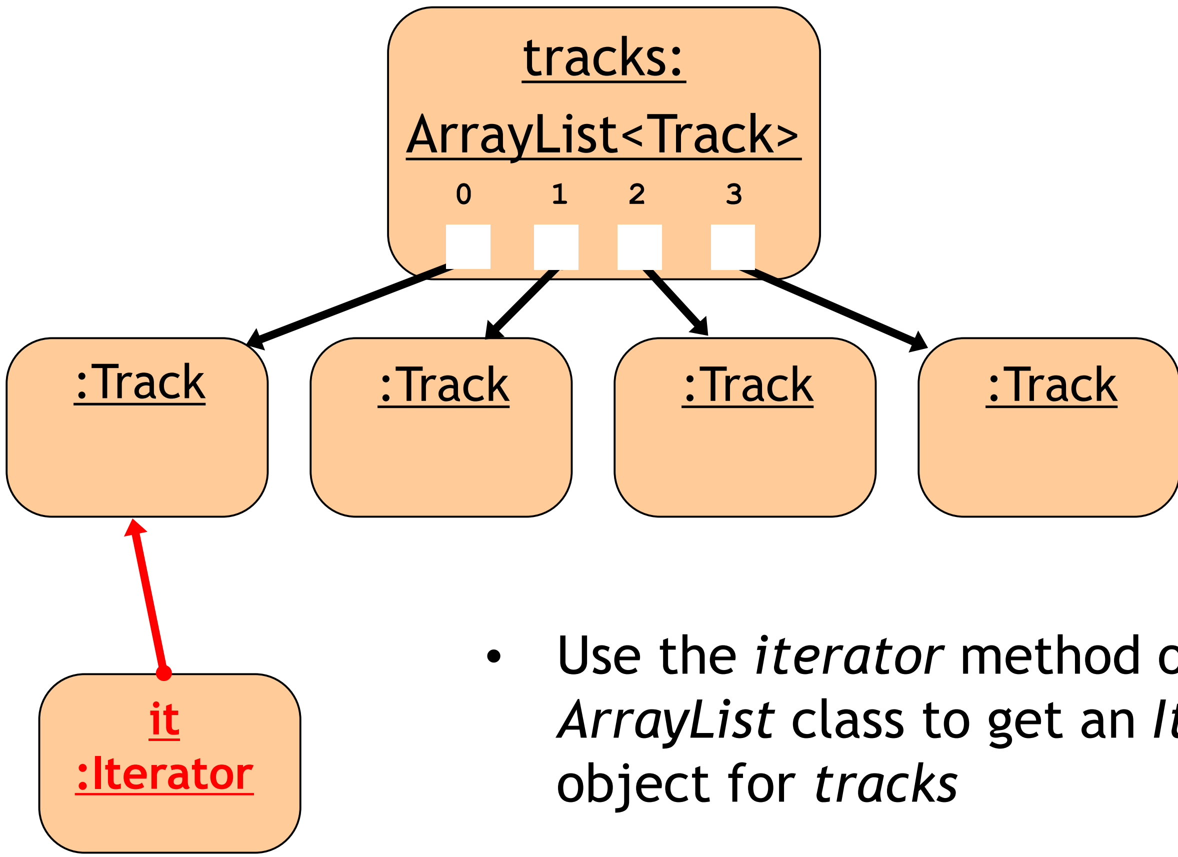
Track example




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

Track example

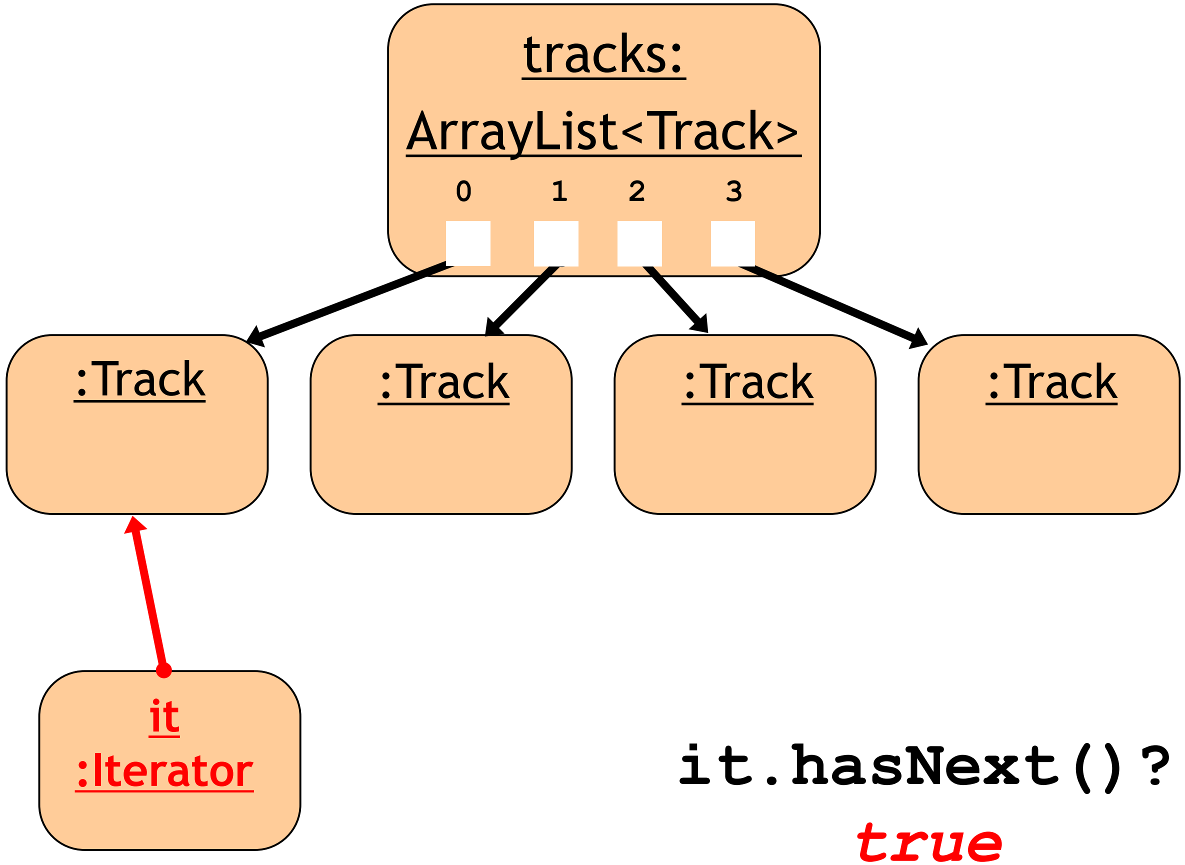
```
Iterator<Track> it = tracks.iterator();
```



- Use the *iterator* method of the *ArrayList* class to get an *Iterator* object for *tracks*
- Assigns the *Iterator* object to the local variable named *it*

Track example

```
while(it.hasNext())
```

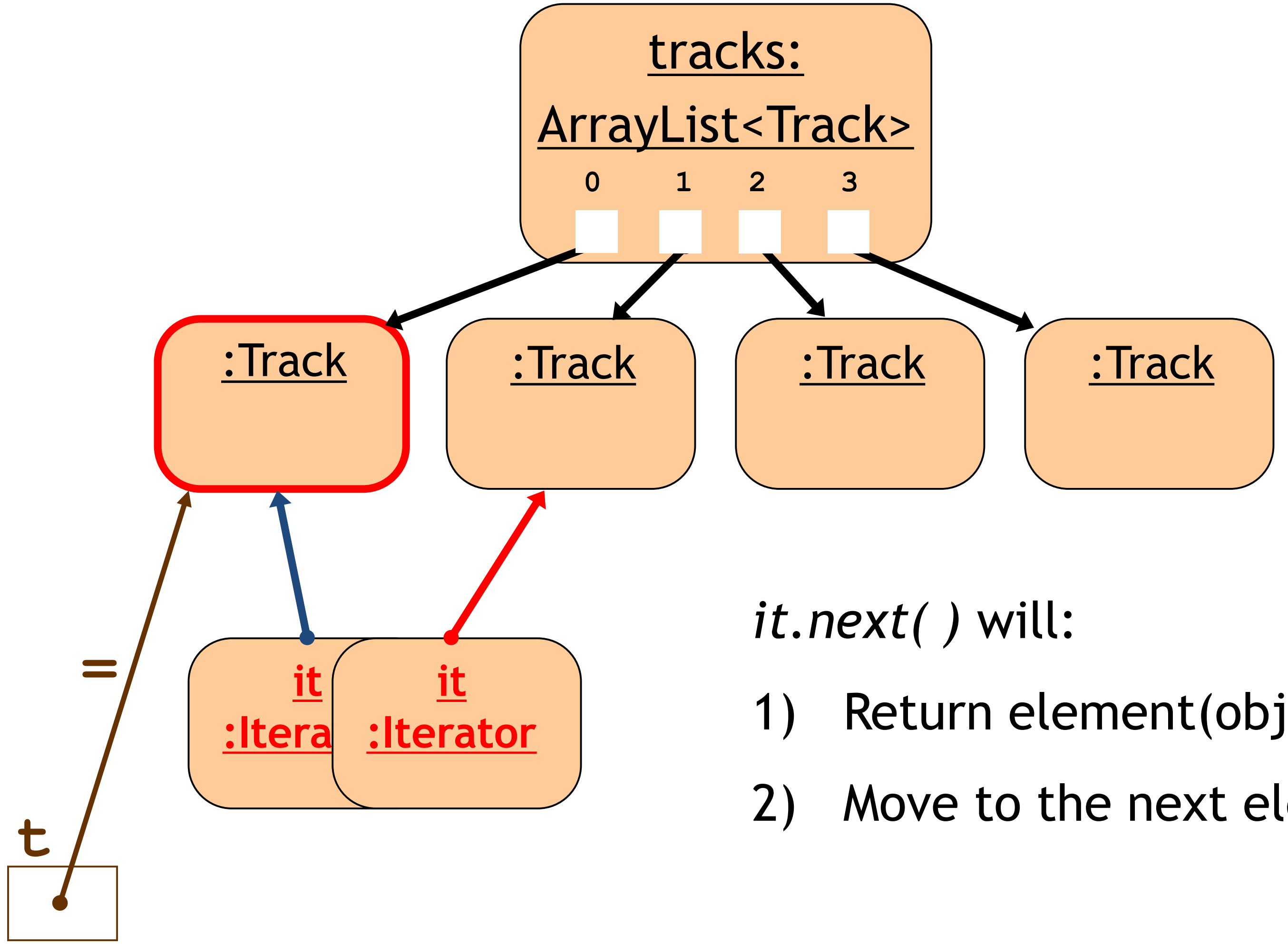



```
it.hasNext()?
```

`true`

Track example

```
Track t = it.next();
```

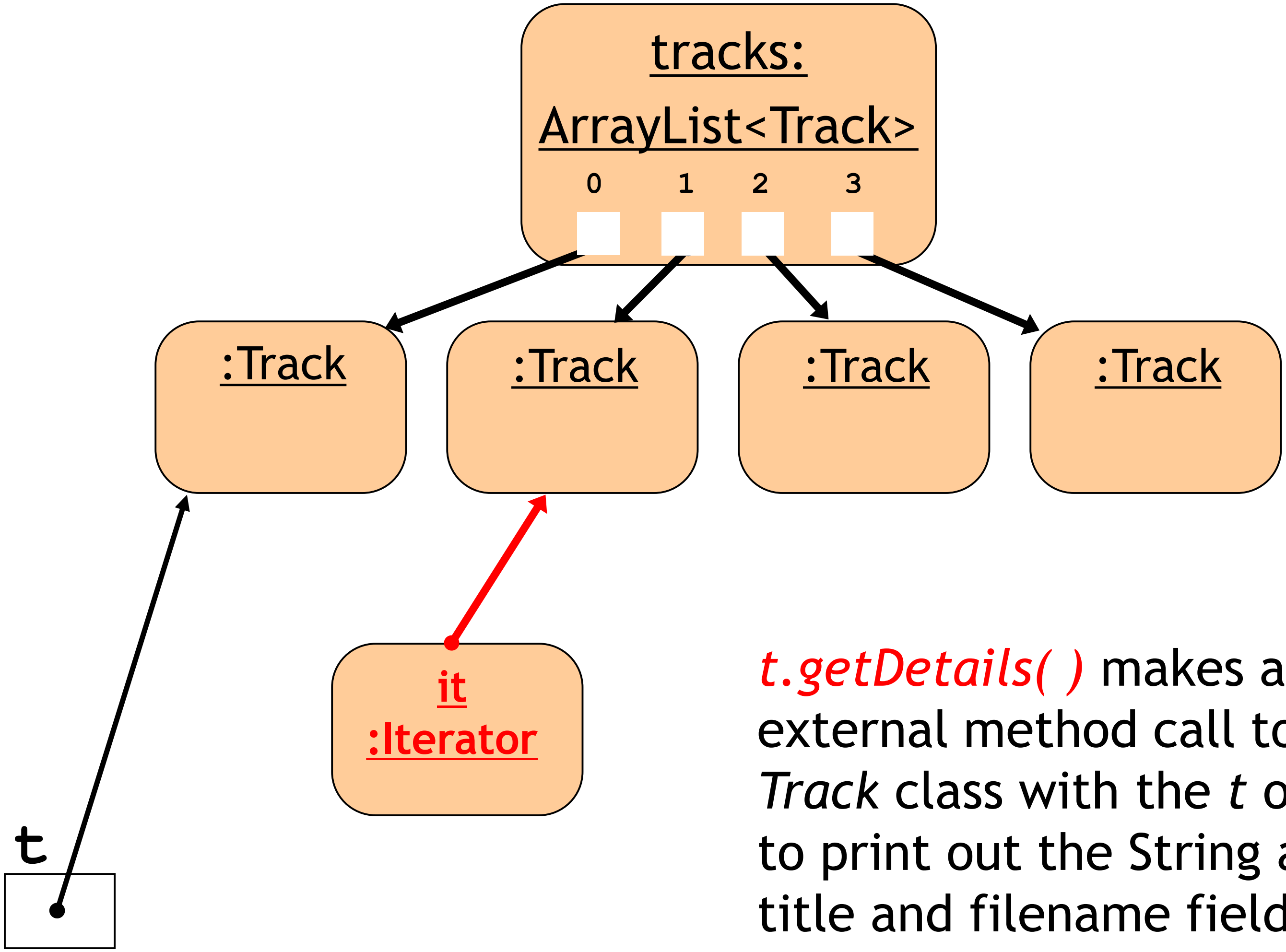


it.next( ) will:

1) Return element(object) at *it*
2) Move to the next element

Track example

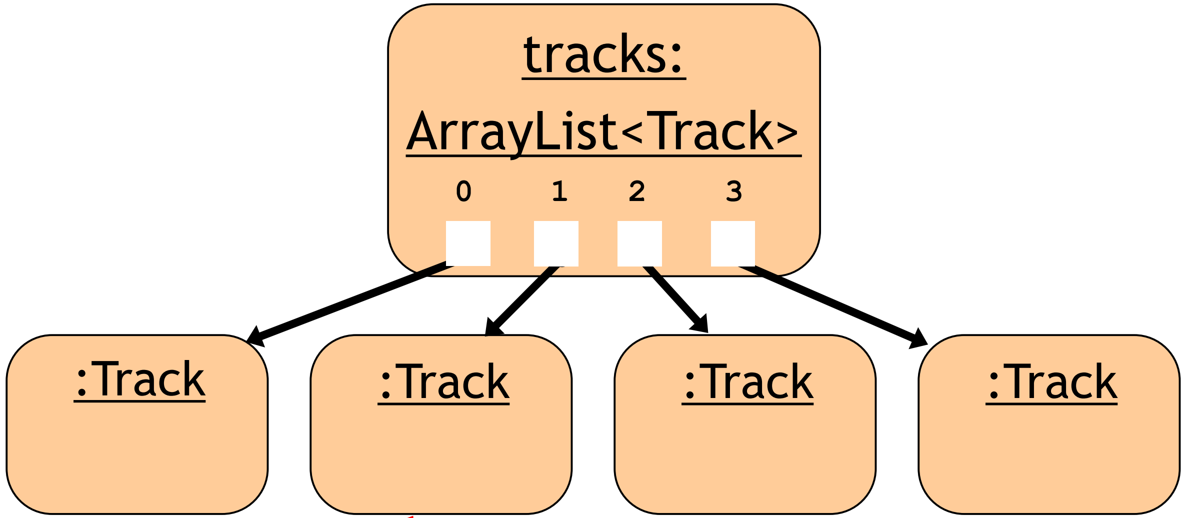
```
System.out.println(t.getDetails());
```

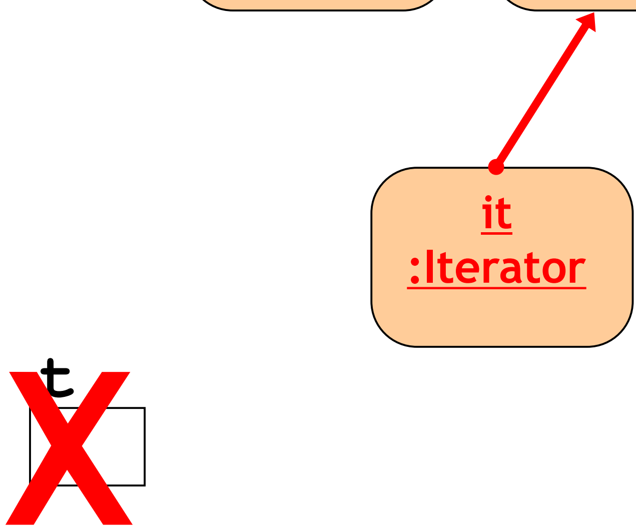


t.getDetails( ) makes an external method call to the *Track* class with the *t* object to print out the String artist, title and filename fields.

Track example

Exit 1st iteration of while body and repeat loop

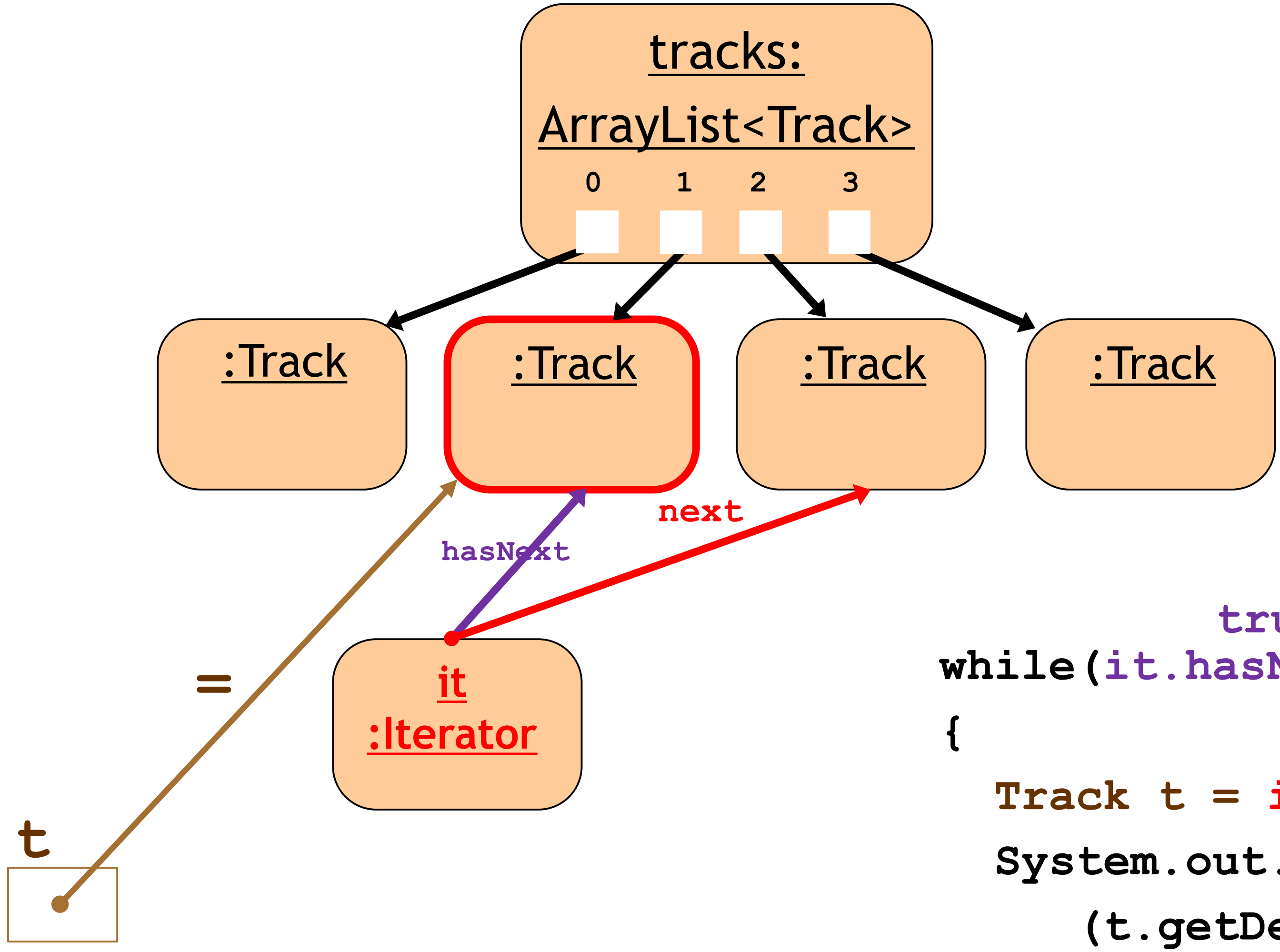





```
while(it.hasNext())
{
    Track t = it.next();
    System.out.println
                (t.getDetails());
}
```

Track example

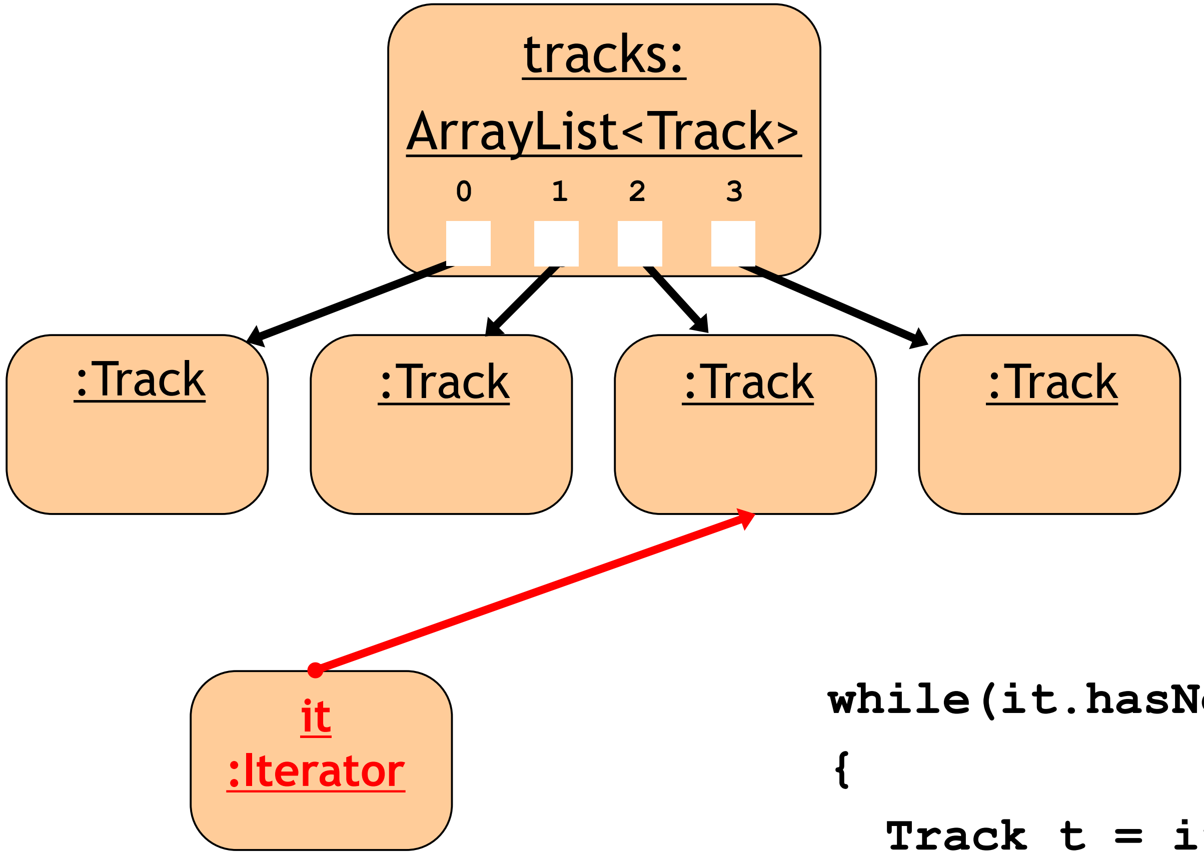
2nd iteration

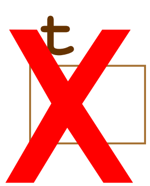



```
              true
while(it.hasNext())
{
   Track t = it.next();
   System.out.println
        (t.getDetails());
}
```

Track example

Exit 2nd iteration

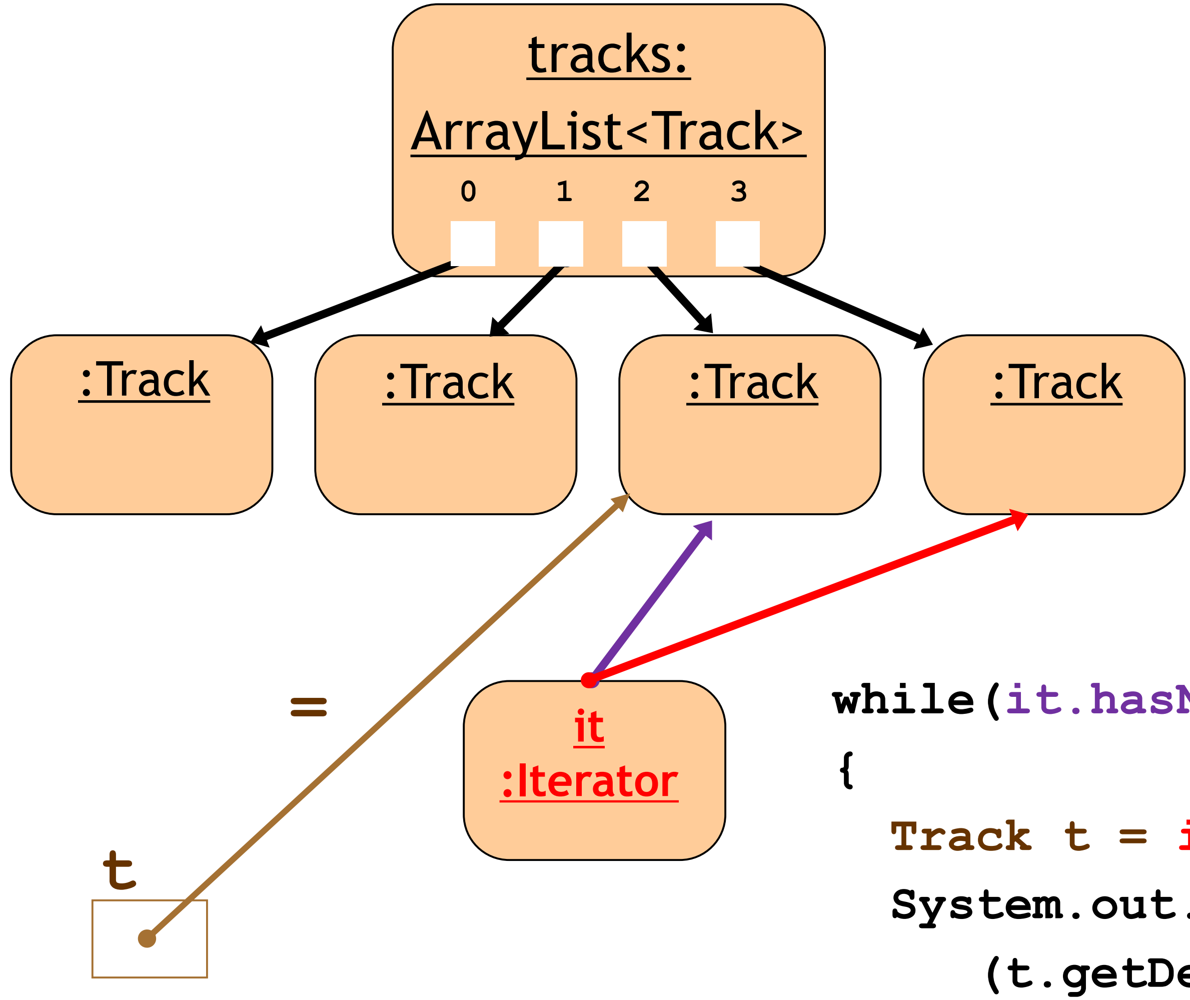





```
while(it.hasNext())
{
   Track t = it.next();
   System.out.println
      (t.getDetails());
}
```

Track example

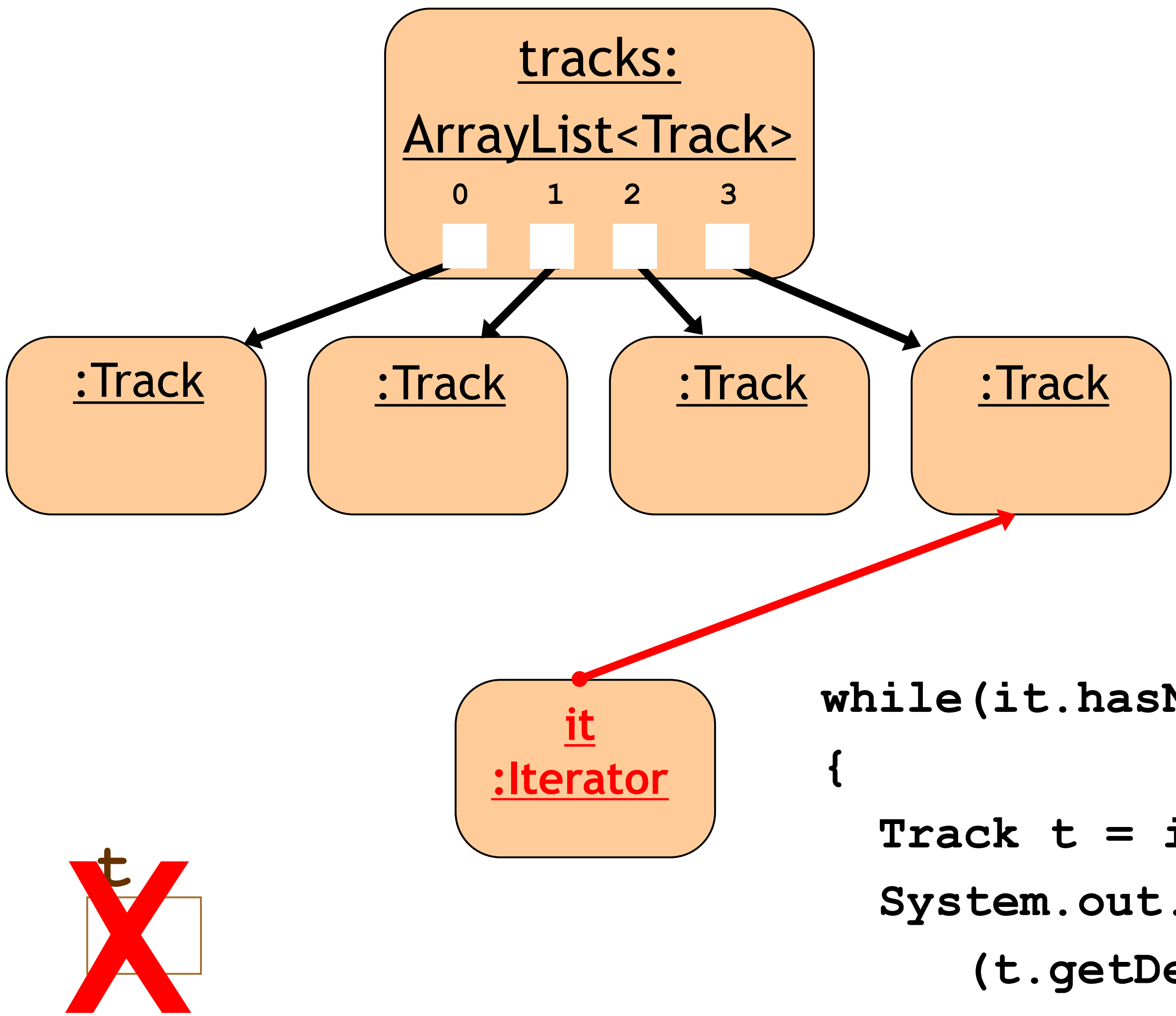
3rd iteration




```
while(it.hasNext())
{
   Track t = it.next();
   System.out.println
       (t.getDetails());
}
```

Track example

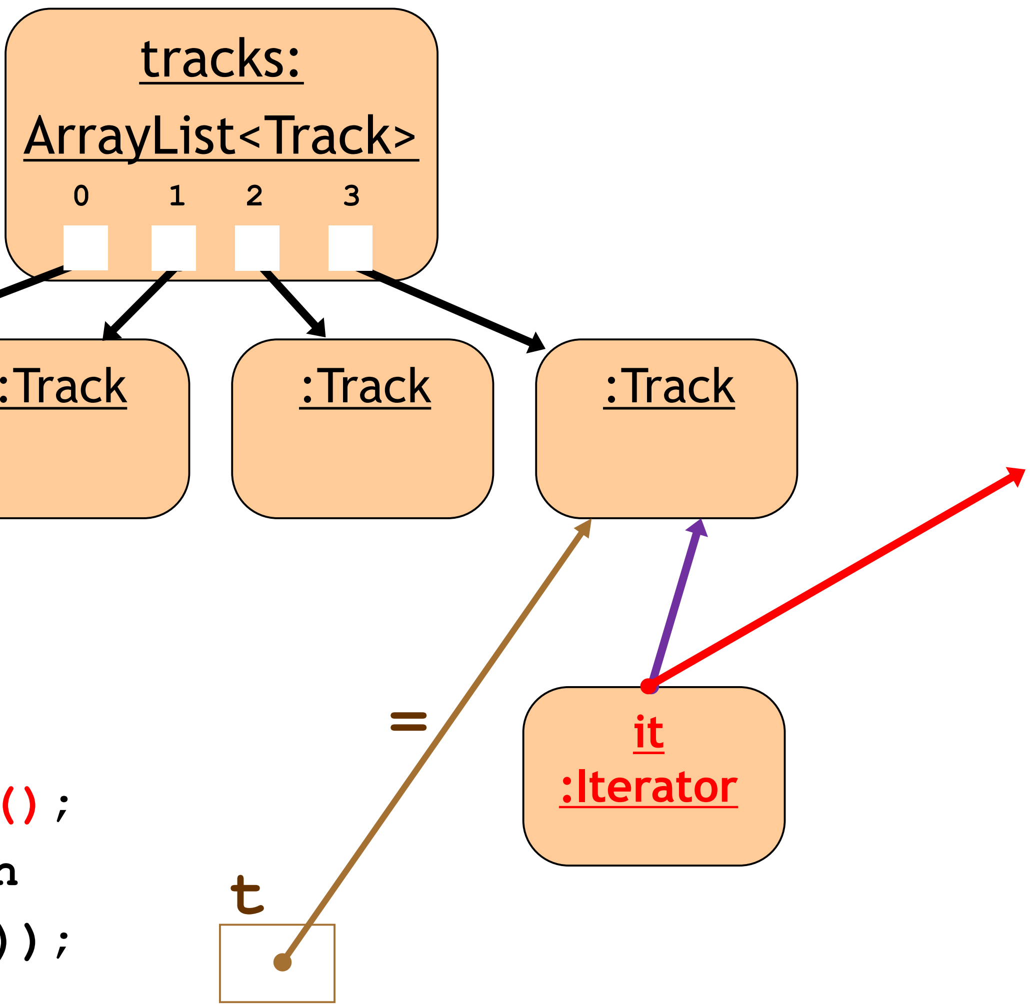
Exit 3rd iteration




```
while(it.hasNext())
{
   Track t = it.next();
   System.out.println
       (t.getDetails());
}
```

Track example

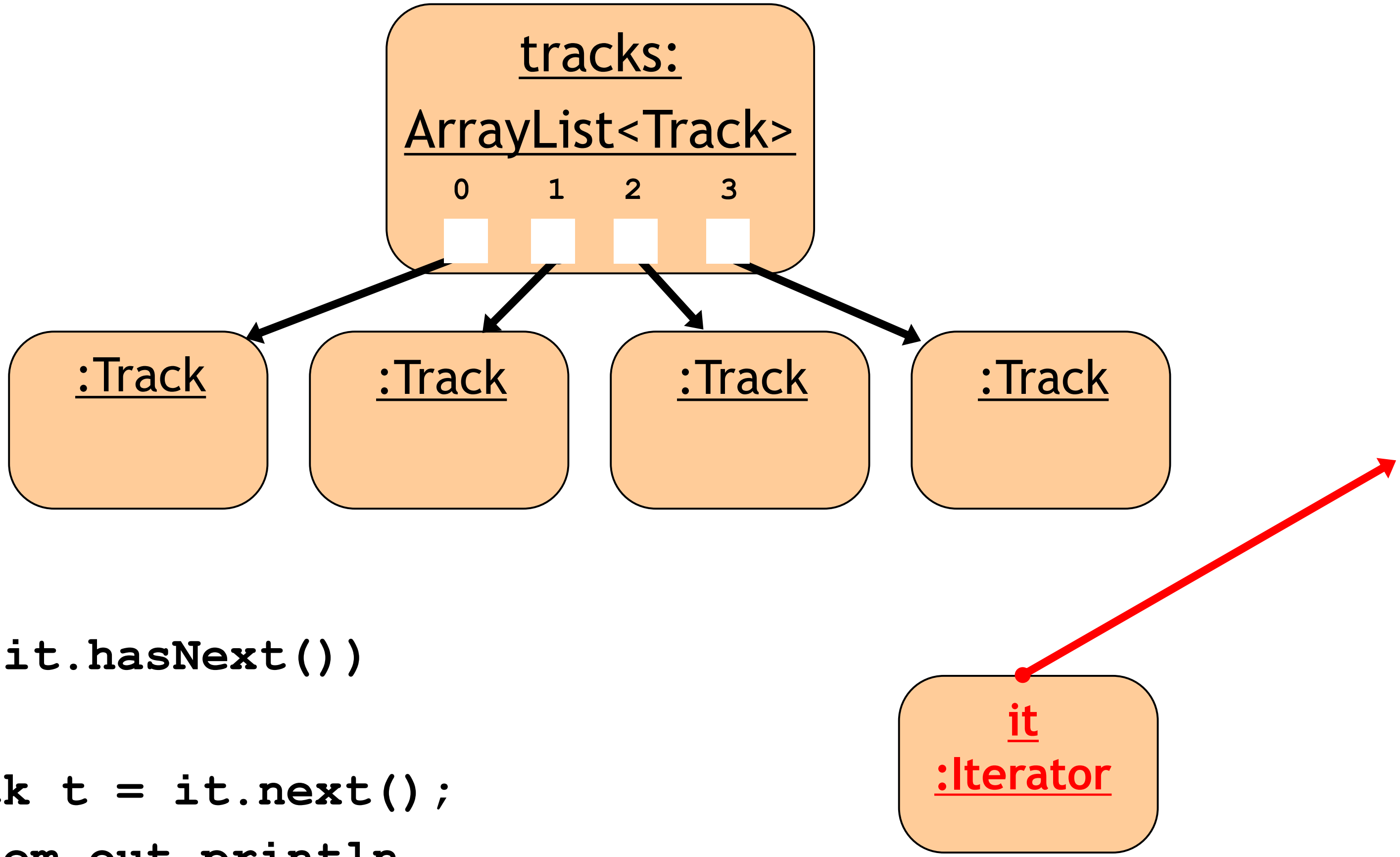
4th iteration



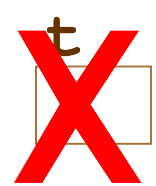

```
while(it.hasNext())
{
  Track t = it.next();
  System.out.println
      (t.getDetails());
}
```

Track example

Exit 4th iteration

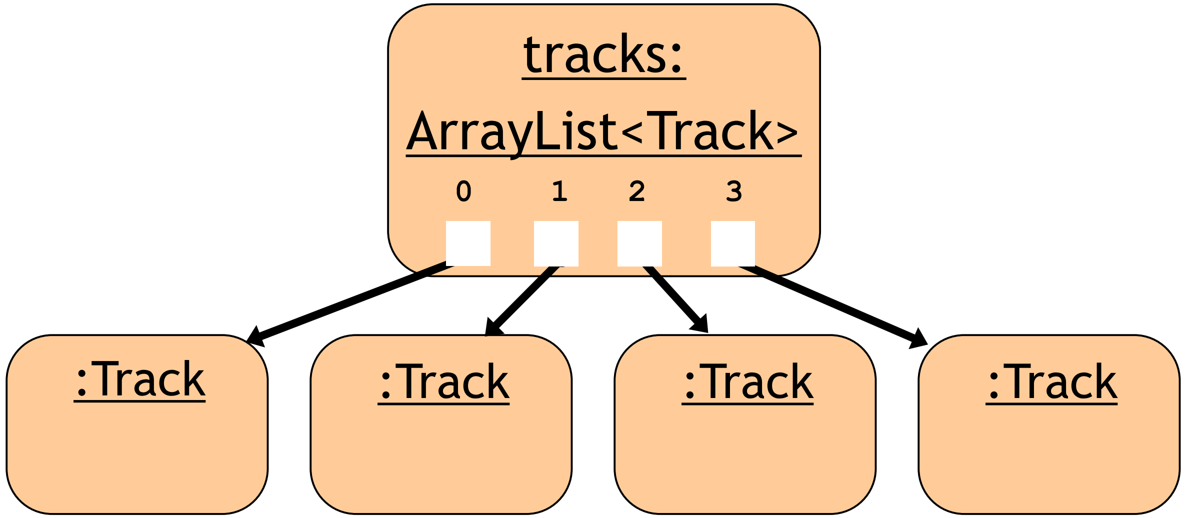


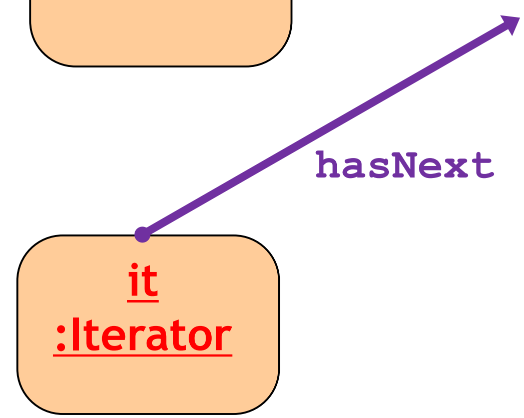

```
while(it.hasNext())
{
  Track t = it.next();
  System.out.println
      (t.getDetails());
}
```



Track example

5th iteration






```
           false
while(it.hasNext())
{
   Track t = it.next();
   System.out.println
       (t.getDetails());
}
```

NO more elements,
so the while loop STOPS!!

Index versus Iterator

- Ways to iterate over a collection:
  - for-each loop *(definite iteration)*
    - Process every element w/o removing an element
  - while loop *(indefinite iteration)*
    - Use if we might want to stop part way through
    - Use for repetition that doesn't involve a collection
  - `Iterator` object *(indefinite iteration)*
    - Use if we might want to stop part way through
    - Often used with collections where indexed access is not very efficient, or impossible
    - Available for all collections in the Java class library
    - Use to remove from a collection
- Iteration is important programming *pattern*

Removing elements

```
for each track in the collection
{
    if track.getArtist( ) is the out-of-favor artist:
        collection.remove(track)
}
```

- Impossible with a *for-each* loop
  - Trying to remove( ) during an iteration causes `ConcurrentModificationException`
- *while* loop possible, but NOT recommended
  - Easy to get indices wrong when removing
- Proper solution is use of *Iterator* with while

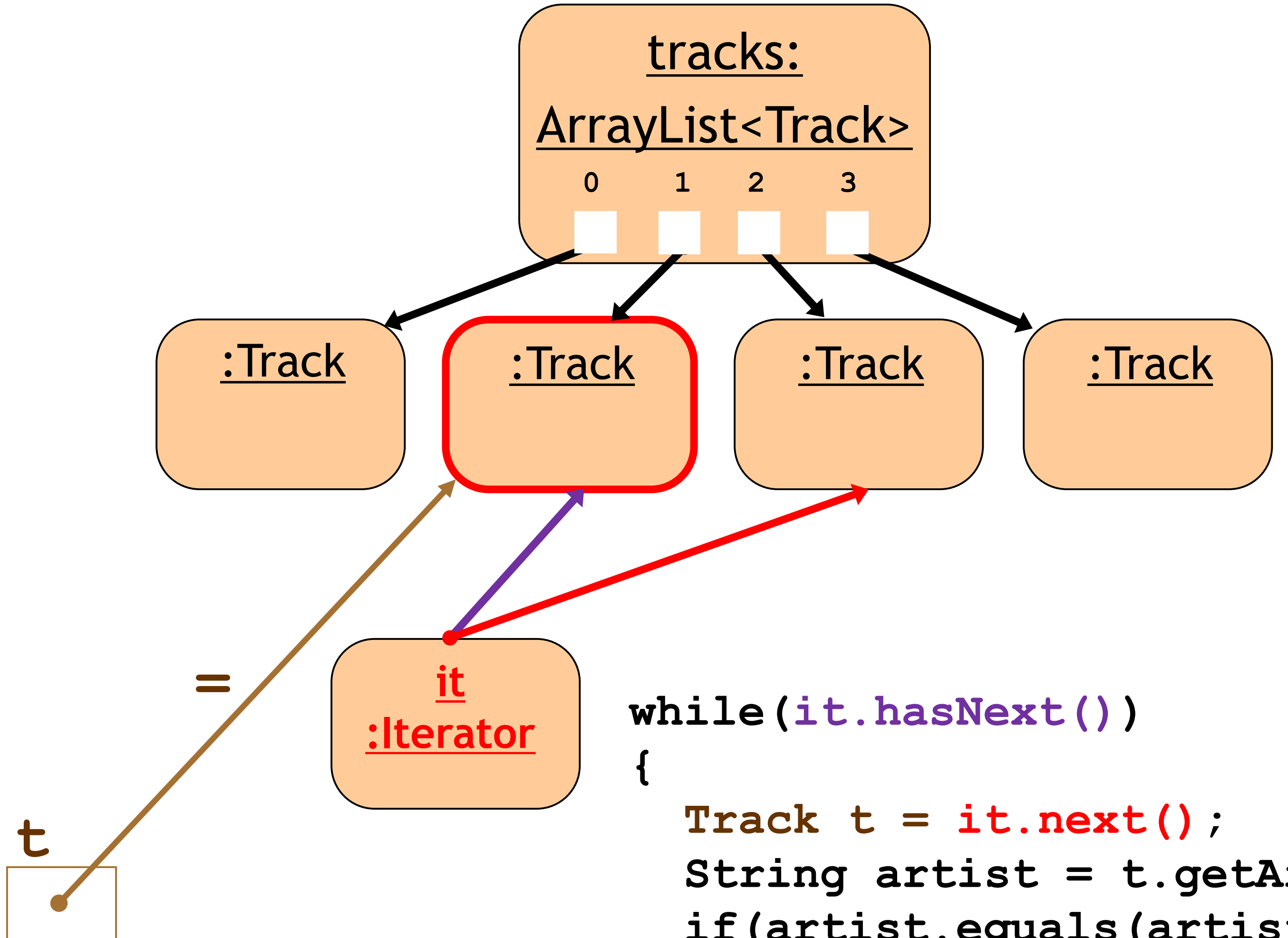
Removing from a collection

```
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.

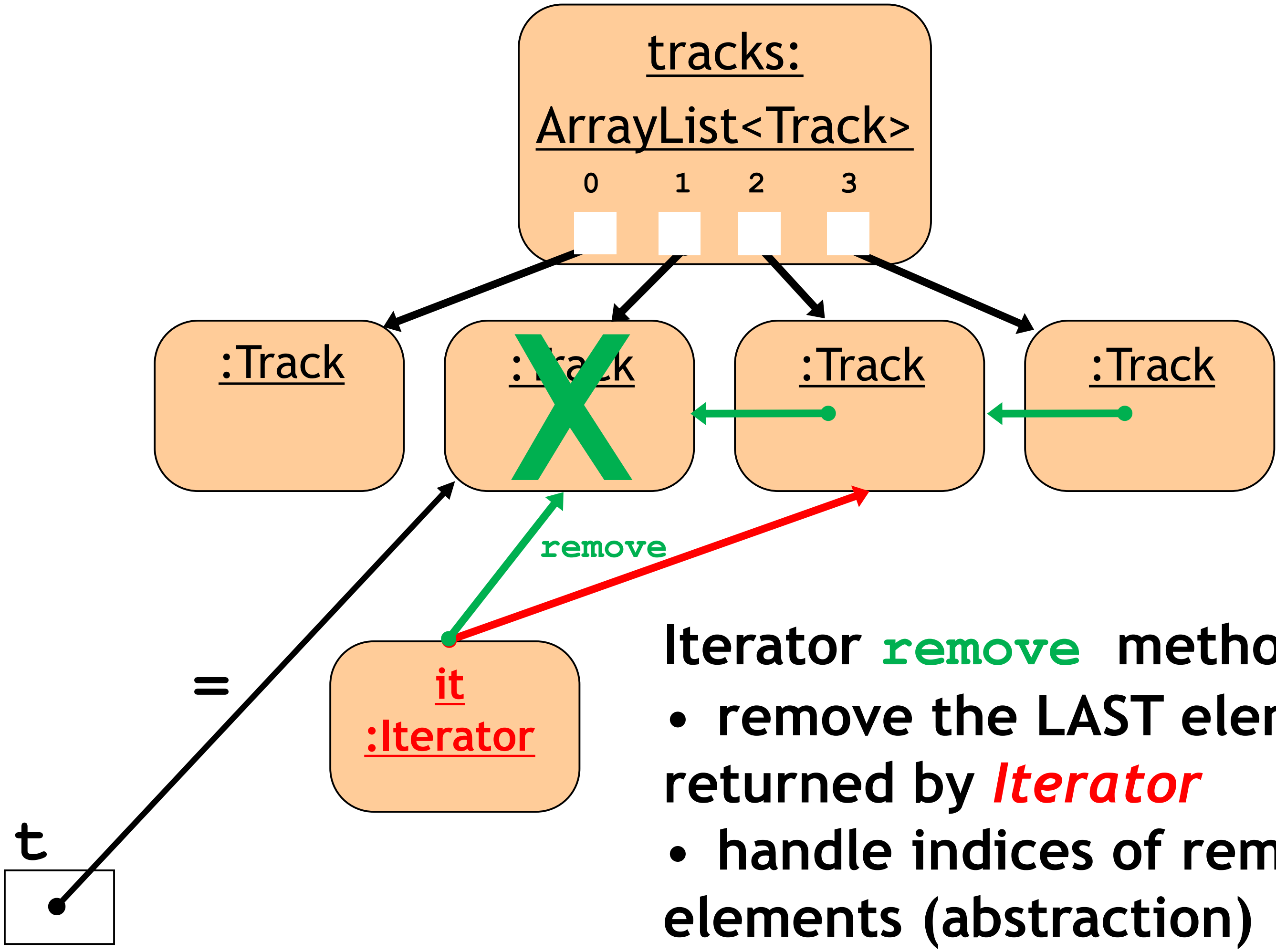
- **Does NOT use *tracks* collection variable in the loop body**
- **Must use *Iterator's remove( )* and NOT the *ArrayList's***
- ***Iterator's* can only *remove* the last retrieved using *next***
- **But it ALLOWS the element to be removed during loop**
- ***Iterator* abstracts removal and keeps iteration in sync**

Removing from a collection




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

Removing from a collection



Iterator `remove` method will:

- remove the LAST element that returned by *Iterator*
- handle indices of remaining elements (abstraction)
- keeps iteration properly in sync
- BUT limited to removing only last element

Removing from a collection without using an Iterator?

```
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

- Use an *ArrayList* to store an arbitrary number of object in a collection
- Loop statements allow a block of statements to be repeated
- *for-each* iterates over a whole collection
- *while* loop allows the repetition to be controlled by a *boolean* expression
- All collection classes provide ***`Iterator`*** objects that provide sequential access and modification to a whole collection

New COPY of an existing *ArrayList*

```
ArrayList<Track> copiedList = new ArrayList<Track>(tracks);
```

- Declare a variable with the same *ArrayList of* <*Element*> type as the original *ArrayList*
- Create a *new ArrayList* object (with the same element type as original) to store the copy in
  - Pass the original *ArrayList* as the parameter
- Point the variable to the new COPY of the original list with exact same contents

NOTE:
Only ONE instance of each object element – but TWO ArrayList objects which point to the same objects in exactly the same order!!

Random library class

```
import java.util.Random;
Random rand = new Random();
int index = rand.nextInt(size);
```

Generates a pseudo-random number by:

- Using the *Random* library class imported from the *java.util* package
- Creating an instance of class *Random* and assigning it to a local variable
- With that instance, call the method *nextInt* to get a number
    - Optional parameter – upper limit *size* passed

Collections library class

```
import java.util.Collections;
ArrayList<String> files = new ArrayList< >();
Collections.shuffle(files);
```

Shuffles the items in a collection by:

- Using the *Collections* library class imported from the *java.util* package
- Calls the method *shuffle* to randomly change the order of existing items in the collection without removing/adding items
    - Parameter – pass the entire collection

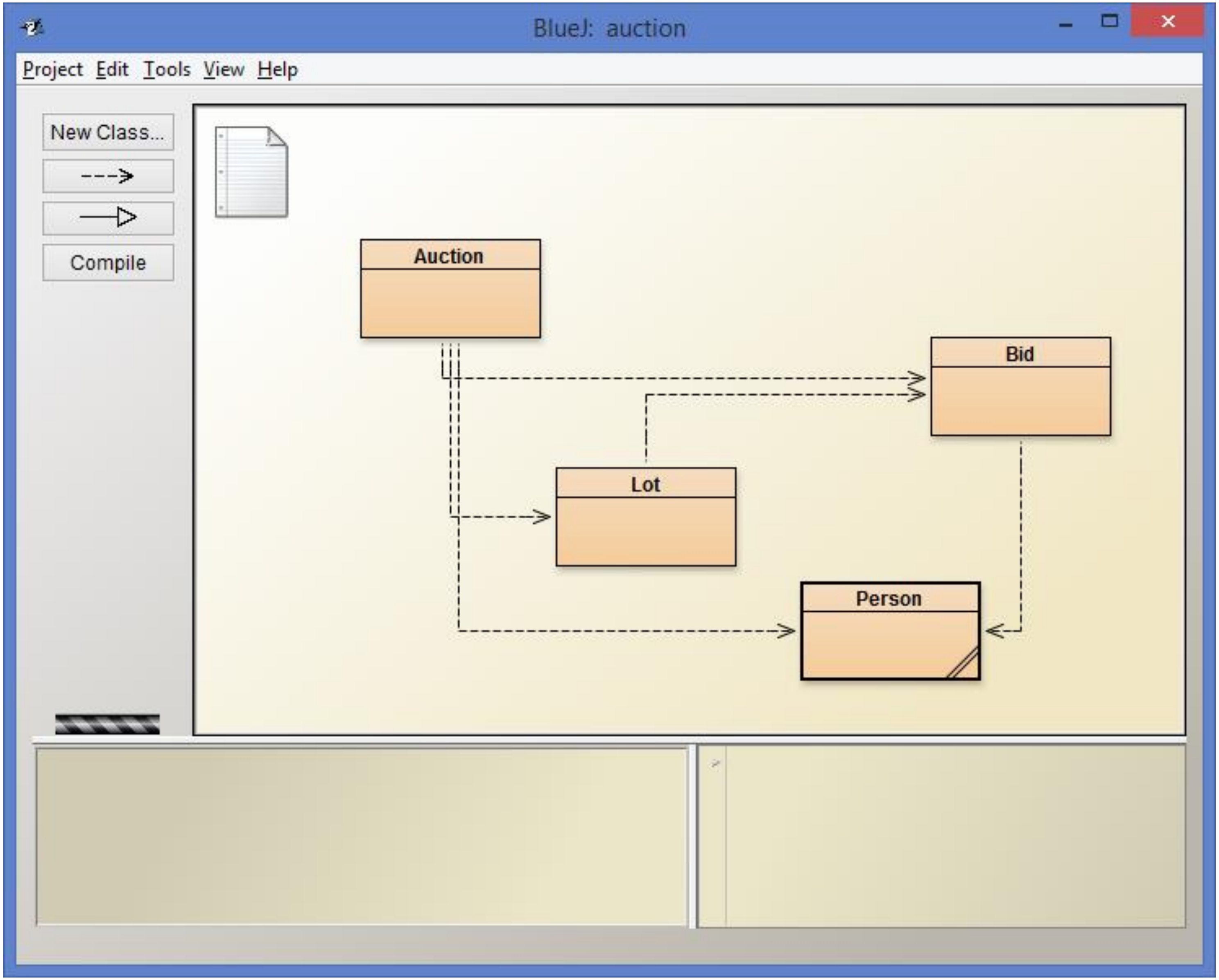
auction project example

Online *Auction* system with:

- Set of items (called *lots*) offered for sale
- Each *Lot* assigned a unique lot number
- A *Person* can try to buy the lot by bidding
- At close of auction, highest *Bid* wins lot
- Any lots with no bids remain sold at close
- Unsold lots may be offered in later auction

Classes: *Auction, Bid, Lot, Person*

auction project



The *auction* project

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

auction project

Online *Auction* system:

- Sell items via *enterLot* with String description
- *Auction* object creates *Lot* object for entered lot
  - lot *number* and *description* is assigned with no bidders
- Bidder *Person* can register with only their *name*
- Place bid with *bidFor* method of *Auction* object with lot number and how much to bid
  - Lot *number* passed so *Lot* objects internal to *Auction*
- *Auction* object transforms *bid* amount to object
- *Lot* records the highest bid object

null

- Used with object types
- Used to indicate ... 'no object'
- Used to indicate ‘no bid yet’ at the intialization of *highestBid* in the auction

```
highestBid = null;
```

- We can test if an object variable holds the ***null*** value:

```
if(highestBid == null) ...
```

- Attempt to de-reference *null* pointer:

```
NullPointerException
```

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:

```
Person bidder =
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`**

while versus *do-while*

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

How is a *do-while* loop different?

```
if collection has at least 1 element
{
  Iterator<ElementType> it = myCollection.iterator();
  do {
     call it.next() to get the next object
     do something with that object
        possibly it.remove()
  } while(it.hasNext());
}
```

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 most circumstance