School of Computer Science

Lesson 5: Grouping objects. Flexible-size collections

Introduction to Programming

Academic year 2023-2024

Concepts

- **Flexible-size collections**
- **Loops**
- **Iterators**

Introduction

- There are different ways in which we can group objects into **collections**.
- **Collection**. *It is a **data structure** to **store** several elements and **access** them.*
- **Collection objects** are **objects** that can **store** an **arbitrary number of objects**.

Flexible-size collections (I)

- Sometimes we need to **group objects into collections**.
  - PDAs store dates, meetings, tasks, birthdays, etc.
  - Libraries keep records about their books.
  - Universities keep records about their students.
  - Auto repair shops keep records about the cars they work on.

Flexible-size collections (II)

- The number of elements stored within a collection changes along time.
- How could we store an arbitrary number of objects?
  - Defining a single class with plenty of attributes?
  - Using something which does not require prior knowledge about the number of elements to store?
  - Using something which allows us to set an upper limit for the amount of elements to store?

Class libraries

- OOL are usually shipped with **class libraries**.
- These libraries provide hundreds or thousands of classes which are useful for developers (over 4000 in Java 8).
- Java provides different libraries organized as **packages**.
  - The `ArrayList` class is defined in the `java.util` package and it is a collection class.

Example: A personal organizer

- We have to model a personal organizer with the following features:
  - It should allow us to store notes.
  - There is no limit for the number of notes.
  - Notes are to be shown individually.
  - We want to be able to know the total number of notes at a given moment.

Example: A personal organizer

```
import java.util.ArrayList;

public class PersonalOrganizer
{
 private ArrayList<String> notes;

 public PersonalOrganizer()
  {
    notes = new ArrayList<String>();
  }
}
```

Import sentence

(Declaration) Each note is a String

We have to create an object of type `ArrayList<String>`

Example: A personal organizer

```
import java.util.ArrayList;
```

- To obtain access to the class.
- Now, the class `ArrayList` from the package `java.util` is usable in our own class.
- That kind of statement must precede the class definition.
- Imported classes are used as our own ones.

Example: A personal organizer

```
private ArrayList<String> notes;
```

- When using collections we have to specify two types:
  - The collection type (`ArrayList`)
  - The data type for the elements to store in the collection (`String`)
    - Must be an object data type
- You read it: "*ArrayList of String*".

Example: A personal organizer

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

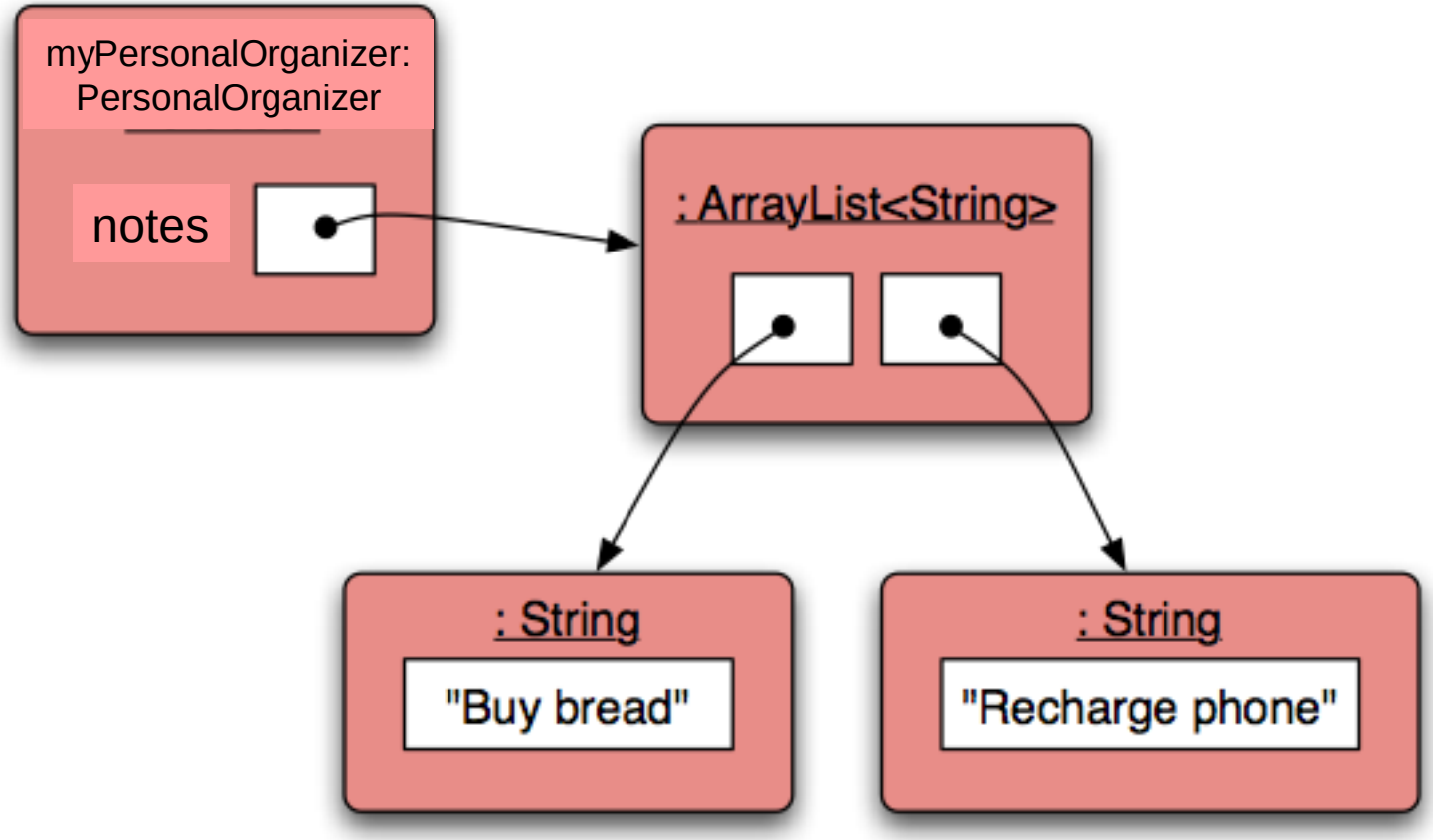
- We create an object of the type `ArrayList<String>` and we store the reference to that object in the attribute `notes`.
- We have to specify the full type.
- There are no parameters (in this case).

Example: A personal organizer

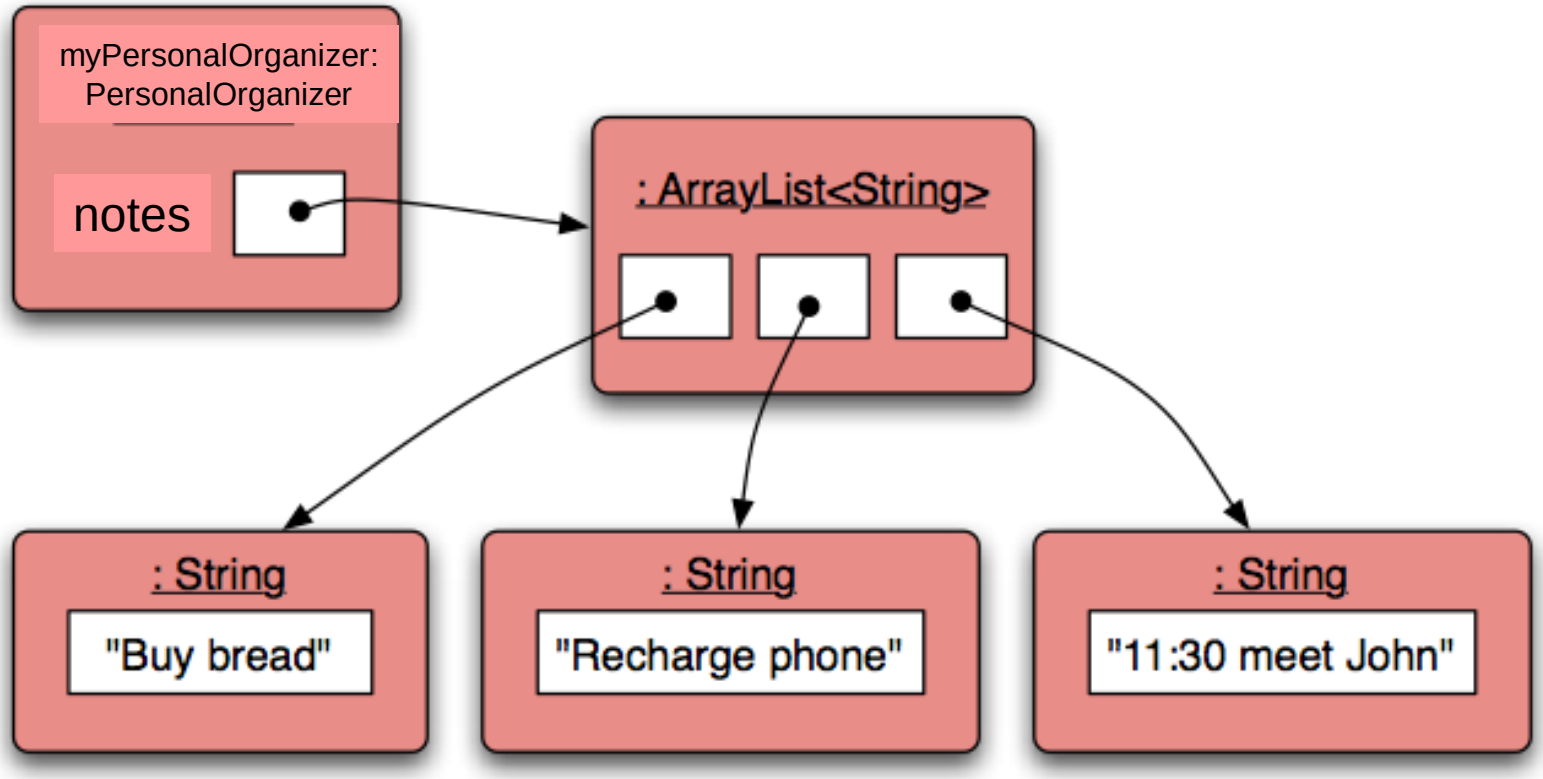
- Classes like `ArrayList` which are parameterized with a second type are known as **generic classes**.
- The `ArrayList` class provides many methods but, to get started, we will use only `add()`, `size()` and `get()`

```
// Store a new note in the organizer
public void saveNote (String note) {
    notes.add(note);
}
```

Object diagrams with collections



Saving a third note



Some features of `ArrayList`

- It is able to **increase its capacity** as needed.
- It keeps a **private counter containing the number of stored elements**. The `size()` method returns that value.
- It keeps the **elements ordered** as added, so they can be obtained in the same order.
- All the **inner details** about its operation **are hidden**.

Using the collection

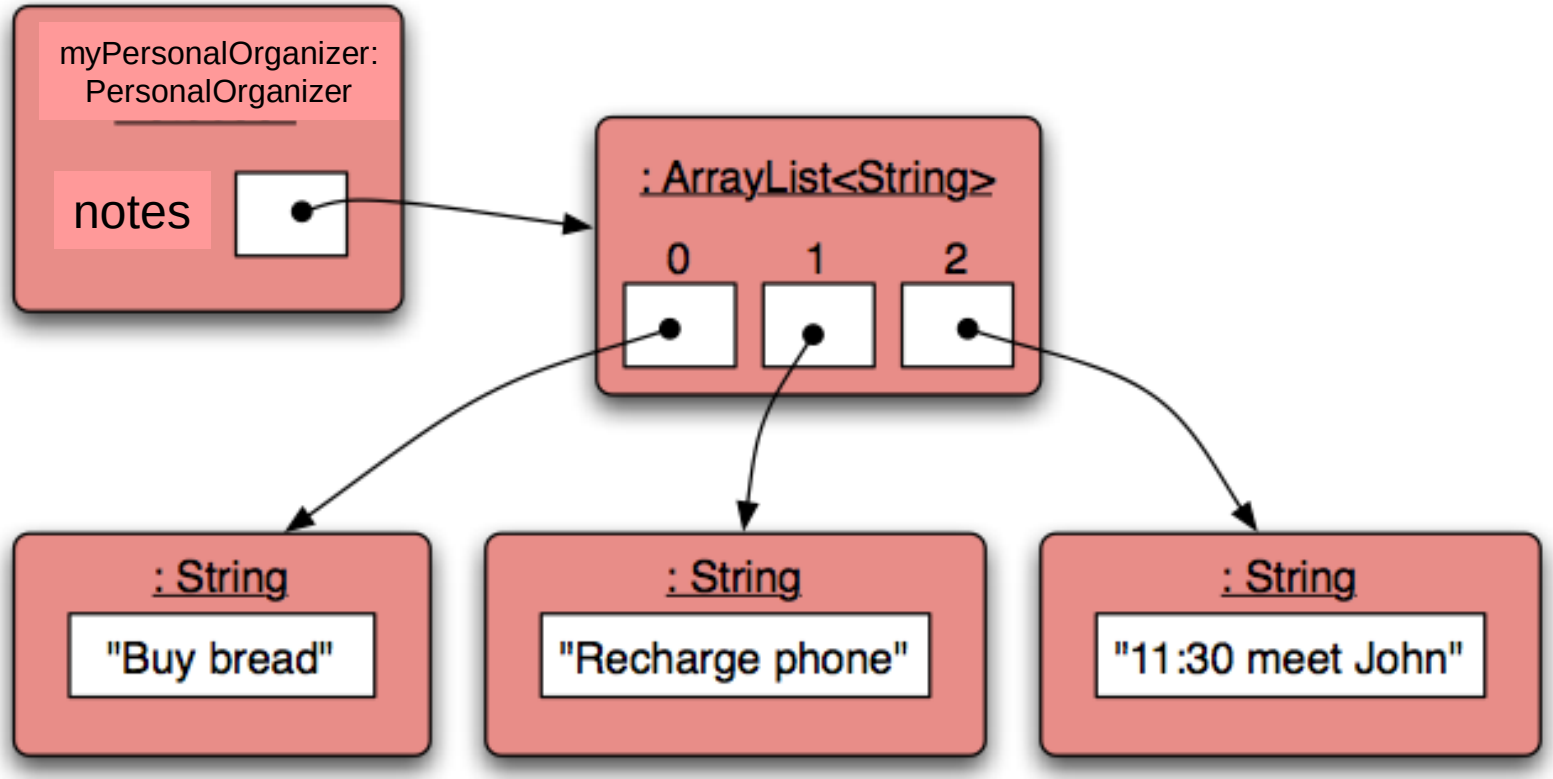
```
// We store a new note in the organizer
public void saveNote(String note){
    notes.add(note);
}
```

Adding a new note

```
// We return the number of notes within the
// organizer
public int numberOfNotes(){
   return notes.size();
}
```

Returns the number of notes (*delegation*)

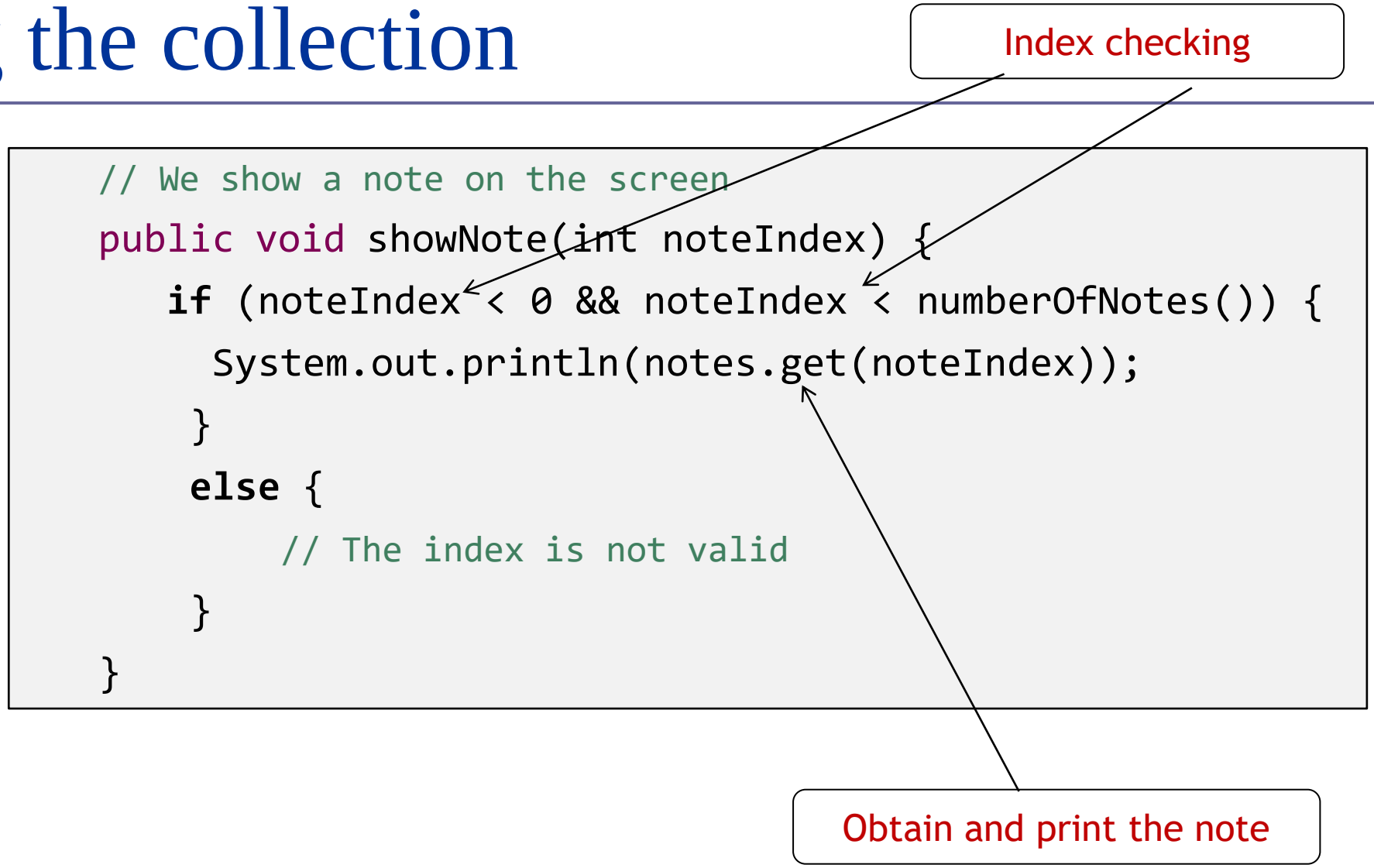
Numbering within collections



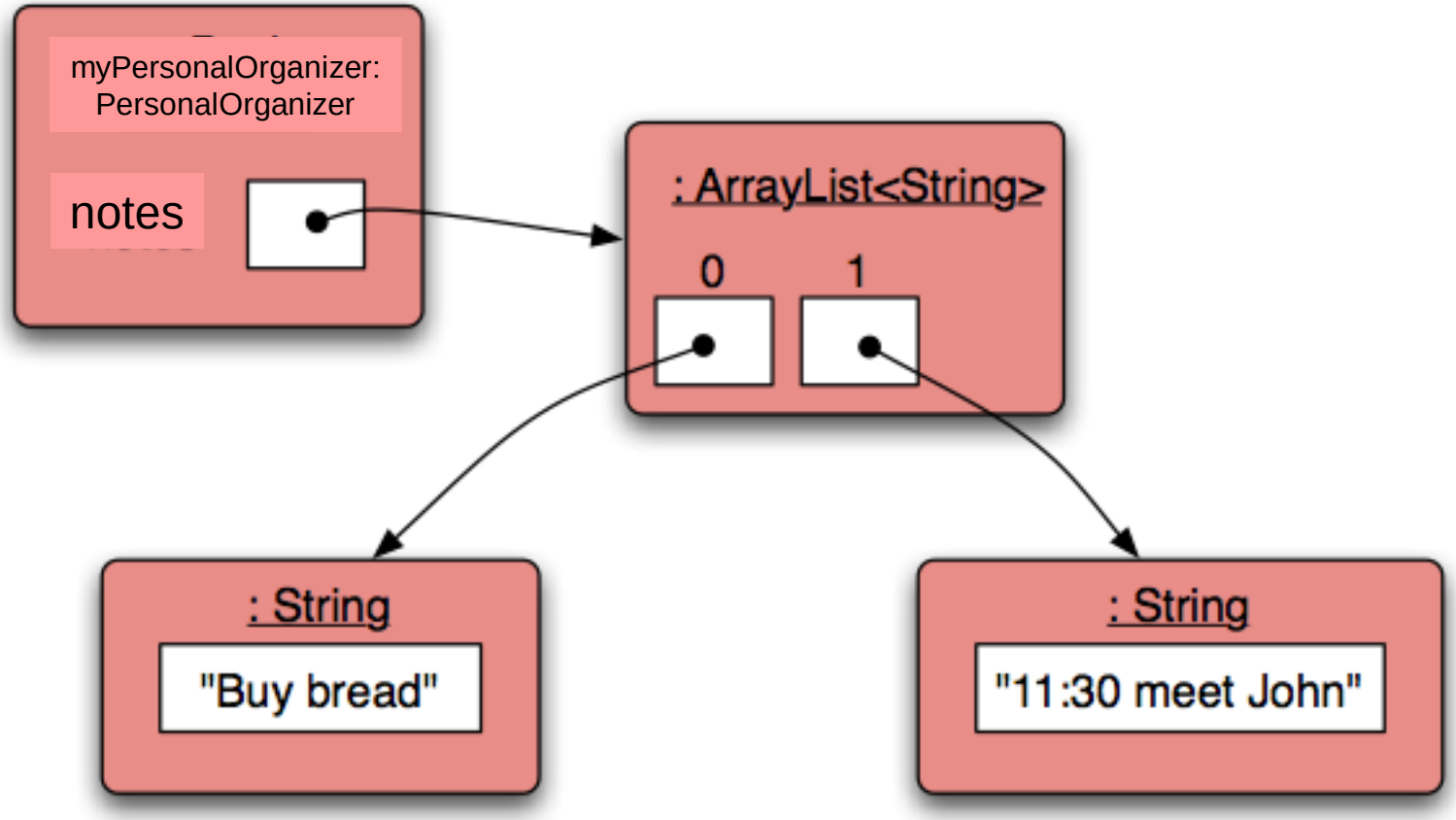
Numbering within collections

- Elements stored within a collection have an implicit numbering **starting at zero**.
- The position for a given object within a collection is known as its **index**.
  - The first element has index 0.
  - The second element has index 1.
  - …

Using the collection



Removing an item from a collection



Removing an item from a collection

- The `ArrayList` class provides the `public void remove(int index)` method, which receives as a parameter the index of the item to be removed.
- The "problem" is that, when removing an item, the rest of indices are modified.
  - By removing the item with index 0 the whole collection is shifted to the left; i.e. their indices decrease by one.
- Besides that, we can add new items into any position, not only at the end of the collection.
  - `public void add(int index, E element)`
  - This also modifies the indices of the following elements.

Using the collection

```
// We remove a note from the organizer
public void removeNote(int noteIndex) {
    if (noteIndex < 0 && noteIndex < numberOfNotes()) {
        notes.remove(noteIndex);
     }
     else {
         // The index is not valid
     }
}
```

Index checking

Remove the note at that position

Exercises

- Write a method call to remove the third object stored in the **notes** collection.
- Let’s suppose an object is stored within a collection with index 6. Which would be the index for that object after removing the objects in positions 0 and 9 (in that order)?

Generic classes (I)

```
ArrayList<String>
```

- We are using the `ArrayList` class, which requires a second type as a parameter.
- This kind of classes are known as **generic classes**.
  - They do not define a single type but potentially many (`ArrayList` of `String` objects, `ArrayList` of `Person` objects, `ArrayList` of `Table` objects,…).

Generic classes (II)

```
private ArrayList<Person> studentsCouncil;
private ArrayList<TicketMachine> stationMachines;
```

- `ArrayList<Person>` and `ArrayList<TicketMachine>` are two different types.
- We cannot assign references from one to the other even when they are derived from the same class.
- **Exercise**. Write the declaration for a private attribute with name `library` based on `ArrayList` and storing items of the `Book` type.

Review (I)

- Collections are used to store an arbitrary number of objects.
- Class libraries provide classes to define collections.
- In Java, class libraries are organized into *packages*.
- We have used the `ArrayList` class from the package `java.util`.

Review (II)

- We can add and remove items from a collection.
- Each item has an index.
- Index values change when items are removed (or new items added).
- The most important methods of the class `ArrayList` are `add`, `get`, `remove` and `size`.
- `ArrayList` is a parameterized class, defining a generic type.

Processing a whole collection

- Sometimes we need to list all the elements from a collection.
- **Example**. What would be the signature for the method `showNotes`? Would it need parameters?
- What would these sentences do?

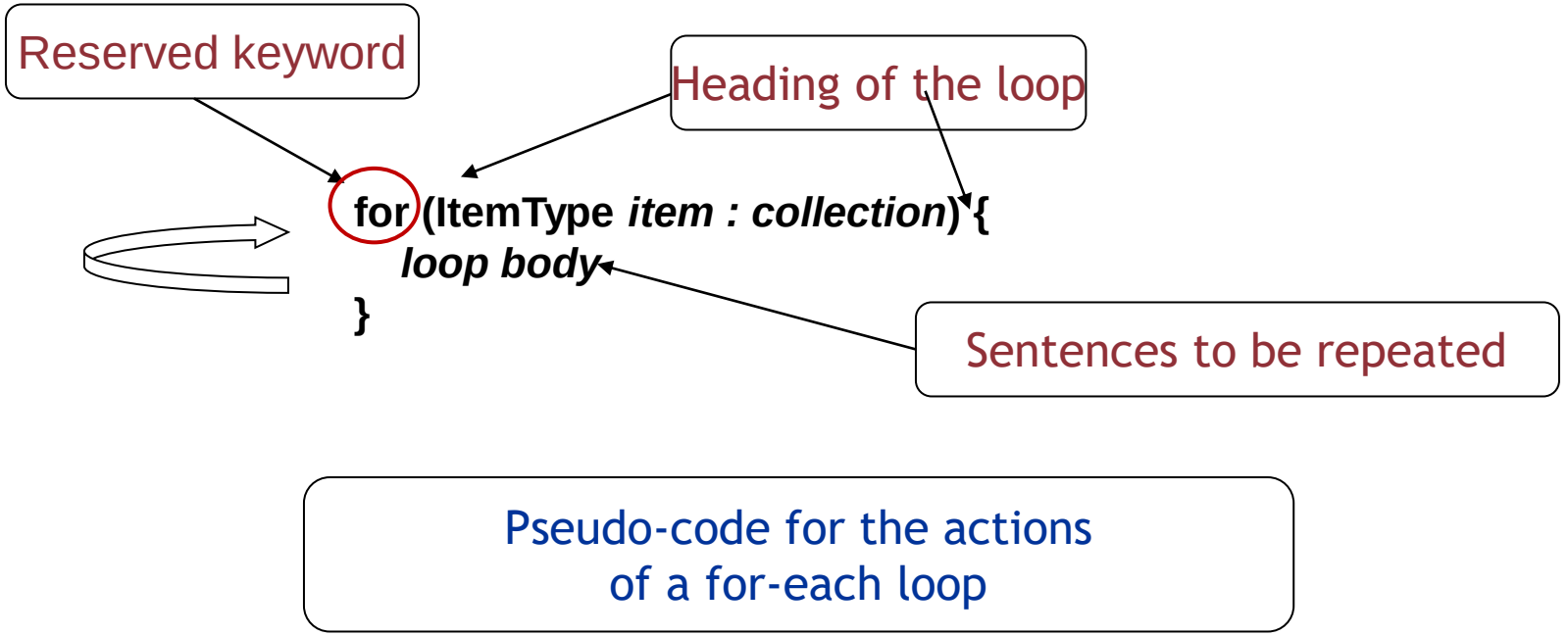
```
System.out.println(notes.get(0));
System.out.println(notes.get(1));
System.out.println(notes.get(2));
...
```

- How many sentences would we need to show all the notes?

The **for-each** loop

- Loops are used to execute a block of sentences many times without writing them repeatedly.
- When do you need a loop?
  - If you need to repeat some actions over and over.
  - If you need to check the number of repetitions.
  - Many times, when working with collections, you need to perform the same action on each item.

Pseudo-code for the **for-each** loop



For each item **within** the collection
execute the sentences in the loop body

Example of using for-each

```
/**
 * Show all notes in the organizer
 */
public void showNotes()
{
    for(String note : notes) {
        System.out.println(note);
    }
}
```

Local variable. It's used to store each of the items from the collection.

For each *note* **in** the *notes* collection, print *note*

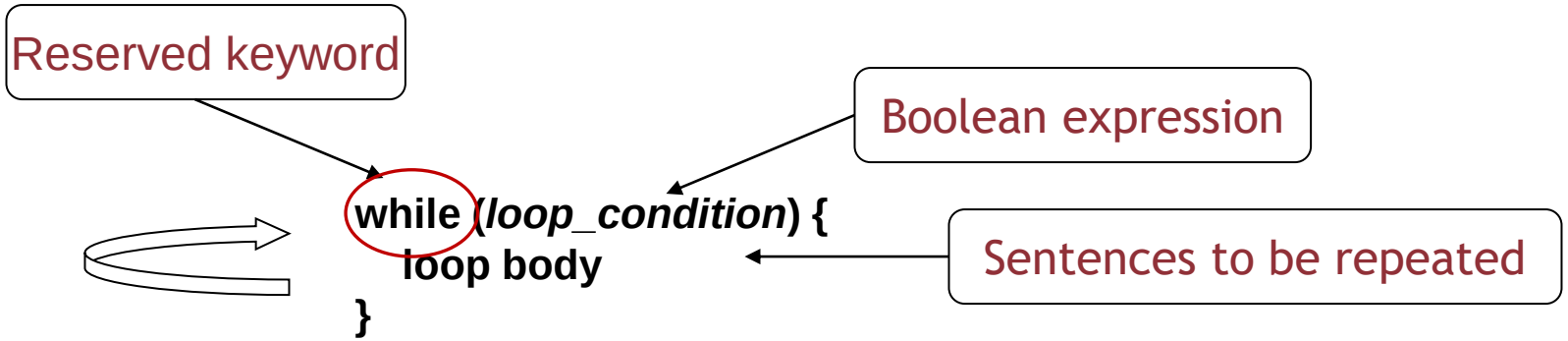
Each item in the collection is assigned iteratively to the local variable `note` and, for each of these assignments, the body is executed once.

Exercise

- Define the class `Phone` and use an `ArrayList` to store phone numbers.
- Write the method `addPhone` to add a new contact (just the phone number).
- Write the methods `showPhone` and `removePhone` provided an index. You should show a error message if the index is not valid.
- Write the method `printPhones` to show all phones.

While loop in pseudo-code

- Similar to **for-each** but more flexible



Pseudo-code for the actions of a while loop

While loop_condition is true
    execute the actions in the body

Example of using the while loop

```
/**
 * Show all of the notes in the organizer.
 */
public void showNotes()
{
    int index = 0;
    while(index < notes.size()) {
        System.out.println(notes.get(index));
        index++;
    }
}
```

Local variable. It's used to go through all of the indices in the collection

Increase `index` by one

While the index value is less than the collection's size
    Print the note and then
    Increase index

Exercise

- Modify the method `printPhones` replacing the for-each loop with a while loop.
- Modify the method `printPhones` to show only those phones with an even index.

for-each versus **while**

- **for-each**:
  - Easy to write.
  - There are no infinite loops by definition.
- **while**:
  - We can partially process the collection.
  - It can be used NOT only with collections.
  - It requires caution to avoid infinite loops.

Example

```
/**
 * Show all of in the organizer.
 */
public void showNotes() {
    int index = 0;

    while(index < notes.size()) {
        System.out.println(notes.get(index));
    }
}
```

Infinite loop!

Exercise

```
int index = 0;

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

What's that code doing?

Searching within a collection

- Searching within a collection can finish with one of two possible results
  - Success
    - After some iterations, the searched-for element were found
  - Fail
    - After having checked all necessary elements, the searched-for element were not found

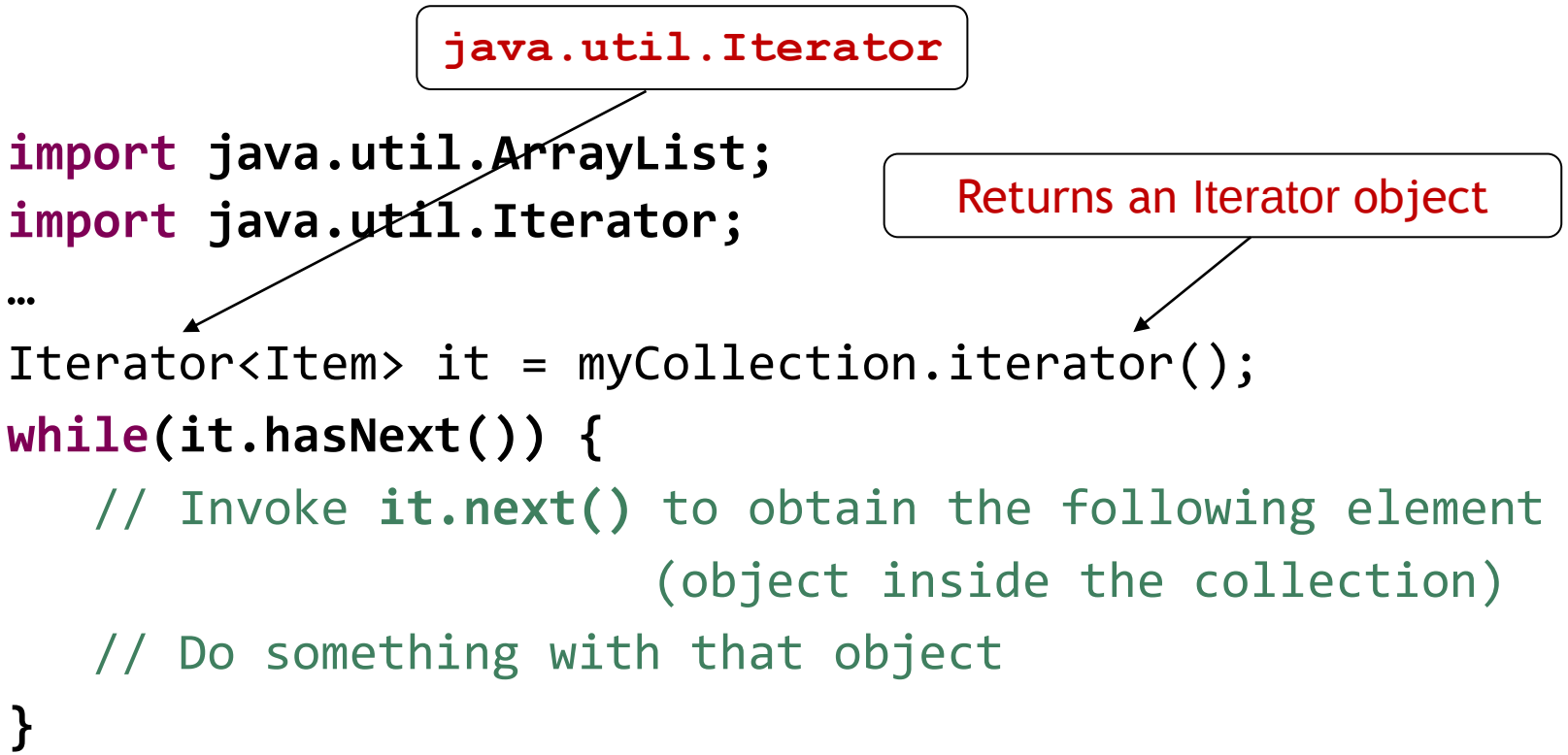
Example. Searching within a collection

```
int index = 0;
String word = "date";
boolean found = false;
while(index < notes.size() && !found) {
    String note = notes.get(index);
    if(note.contains(word)) {
        found = true;
    }
    else {
         index++;
    }
}
// The loop stops when a note containing the desired
// word is found or the end of the collection is
// reached.
```

Iterators

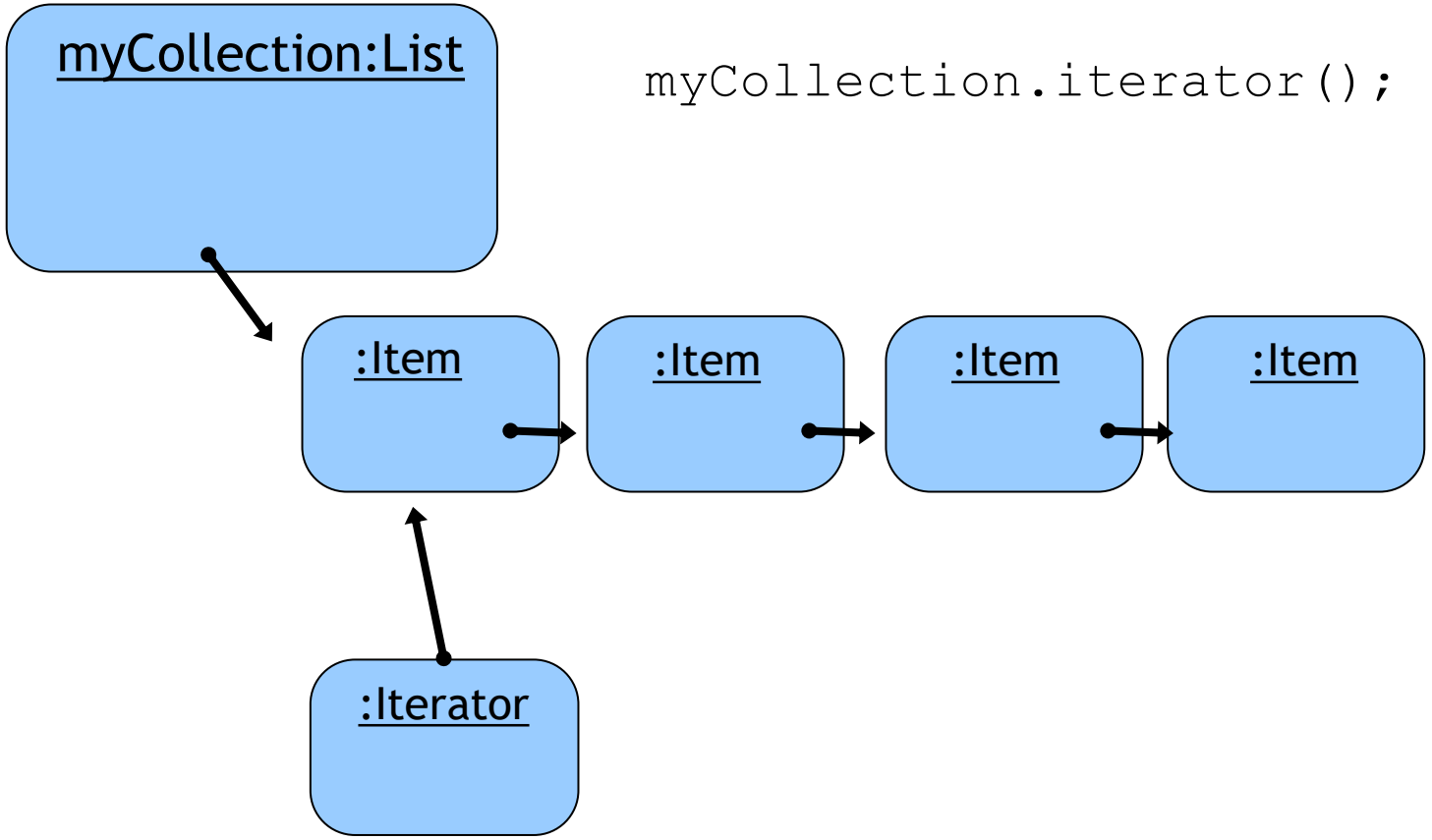
- They provide an additional way to go through a collection.
- An **iterator** is an object providing useful functionality for iteratively working on the items within a collection.
- The `iterator` method from the `ArrayList` class returns an object of the `Iterator` class.
- The `Iterator` class is also defined in the `java.util` package.

Using an Iterator object

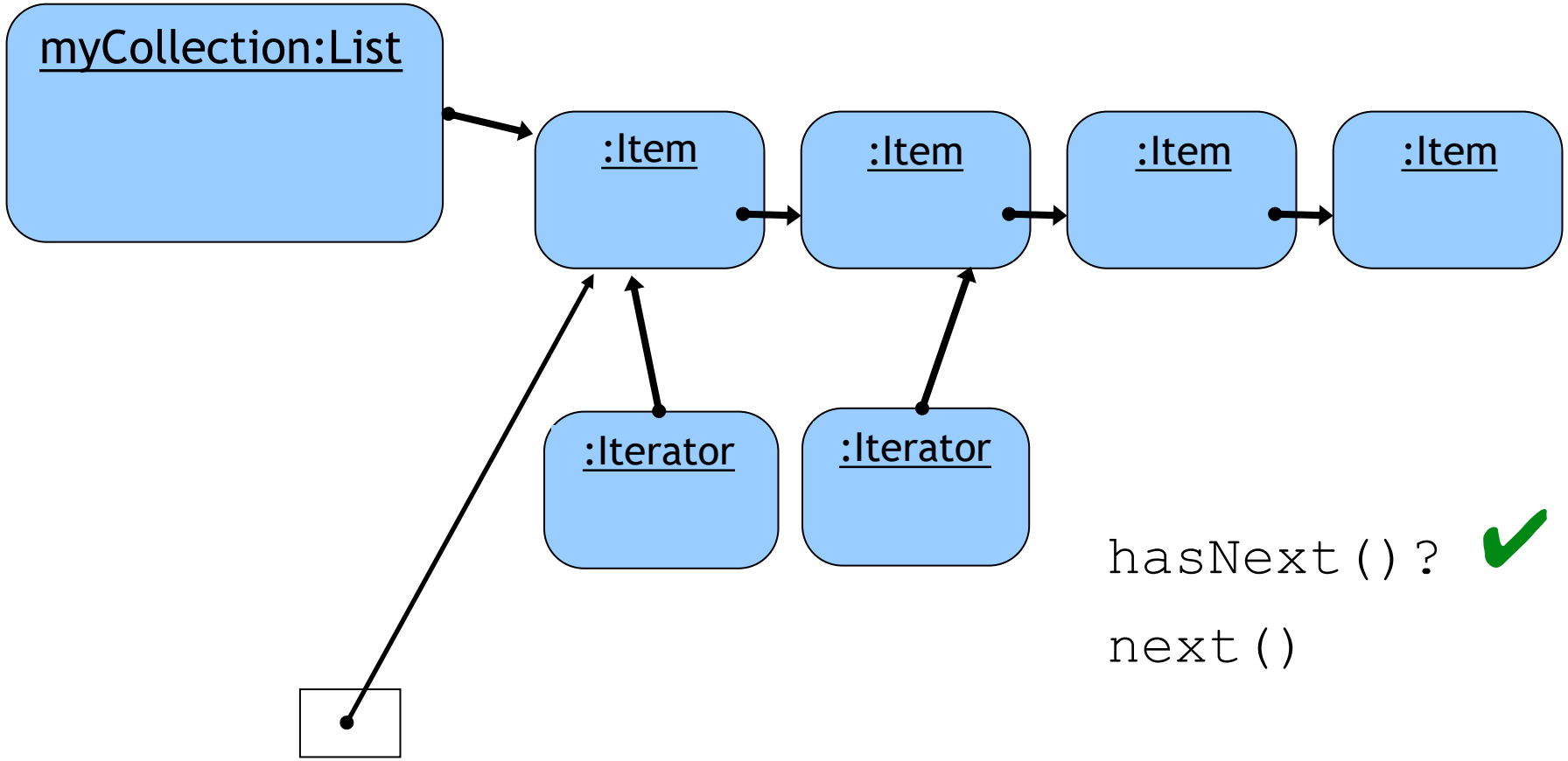



```
import java.util.ArrayList;
import java.util.Iterator;
…
Iterator<Item> it = myCollection.iterator();
while(it.hasNext()) {
   // Invoke it.next() to obtain the following element
                        (object inside the collection)
   // Do something with that object
}
```

Iterators

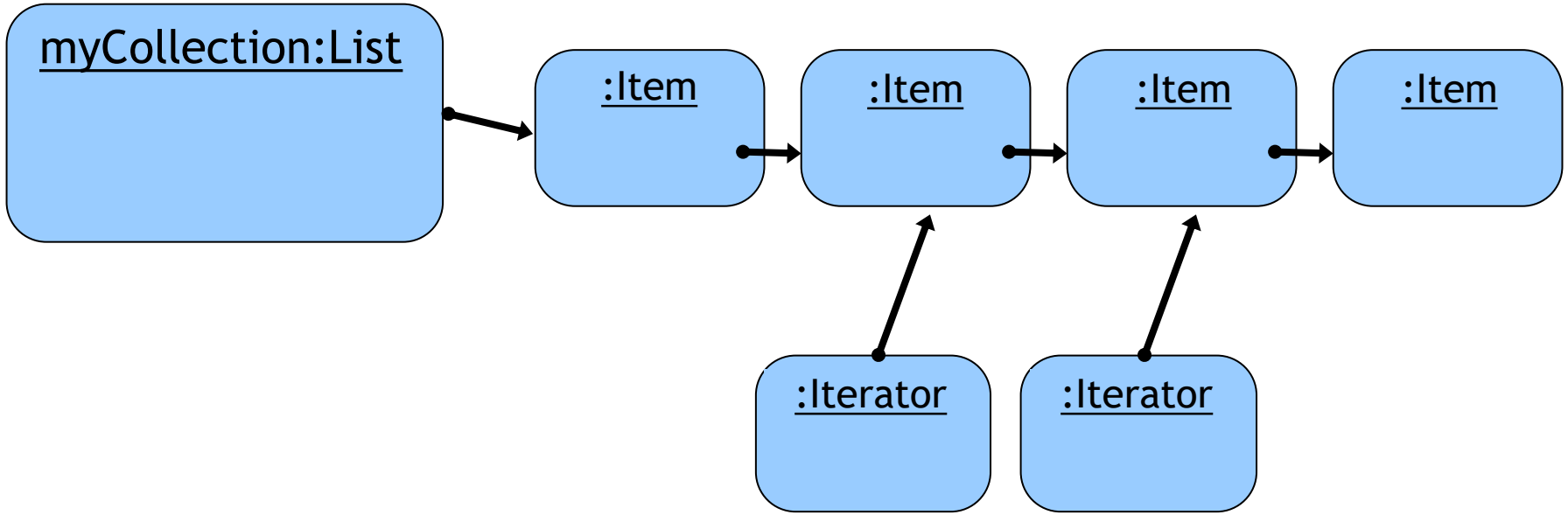


Iterators

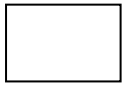



```
Item e = it.next();
```

Iterators

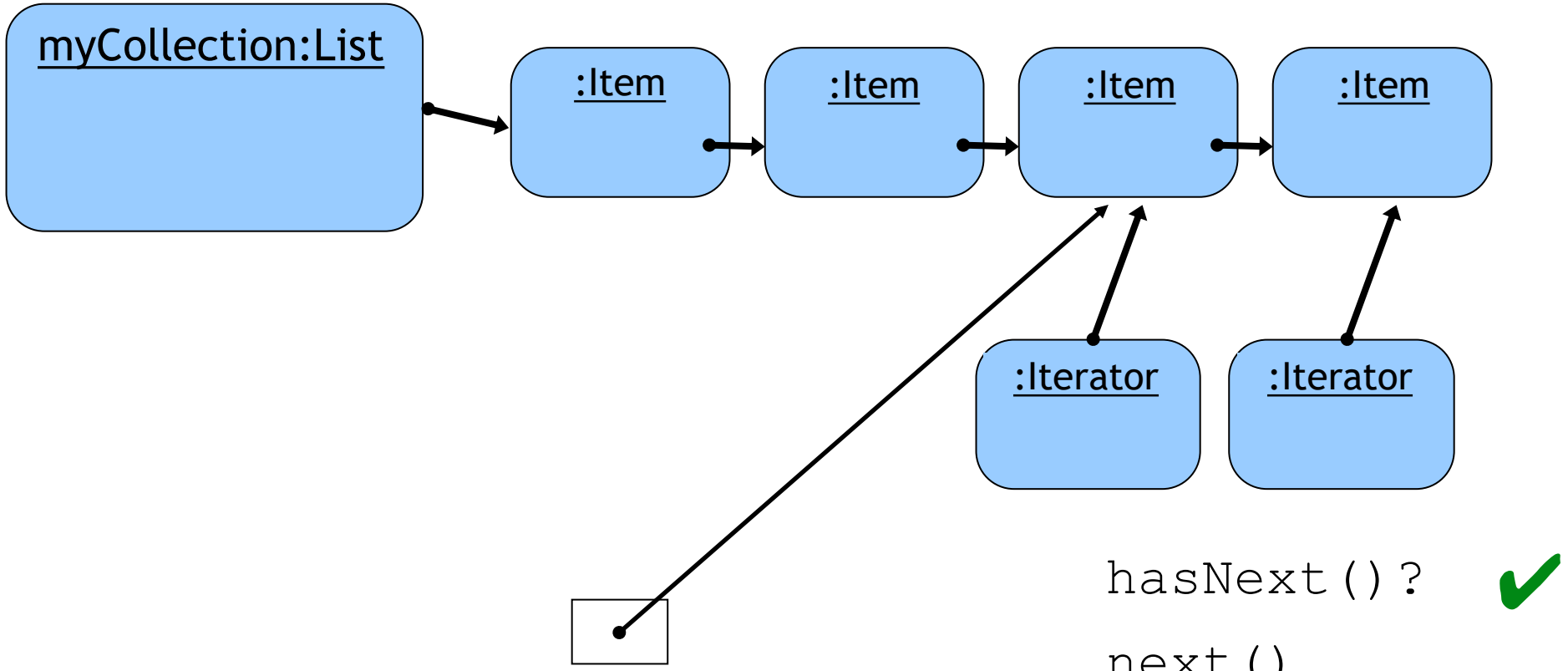



```
hasNext()?
next()
```



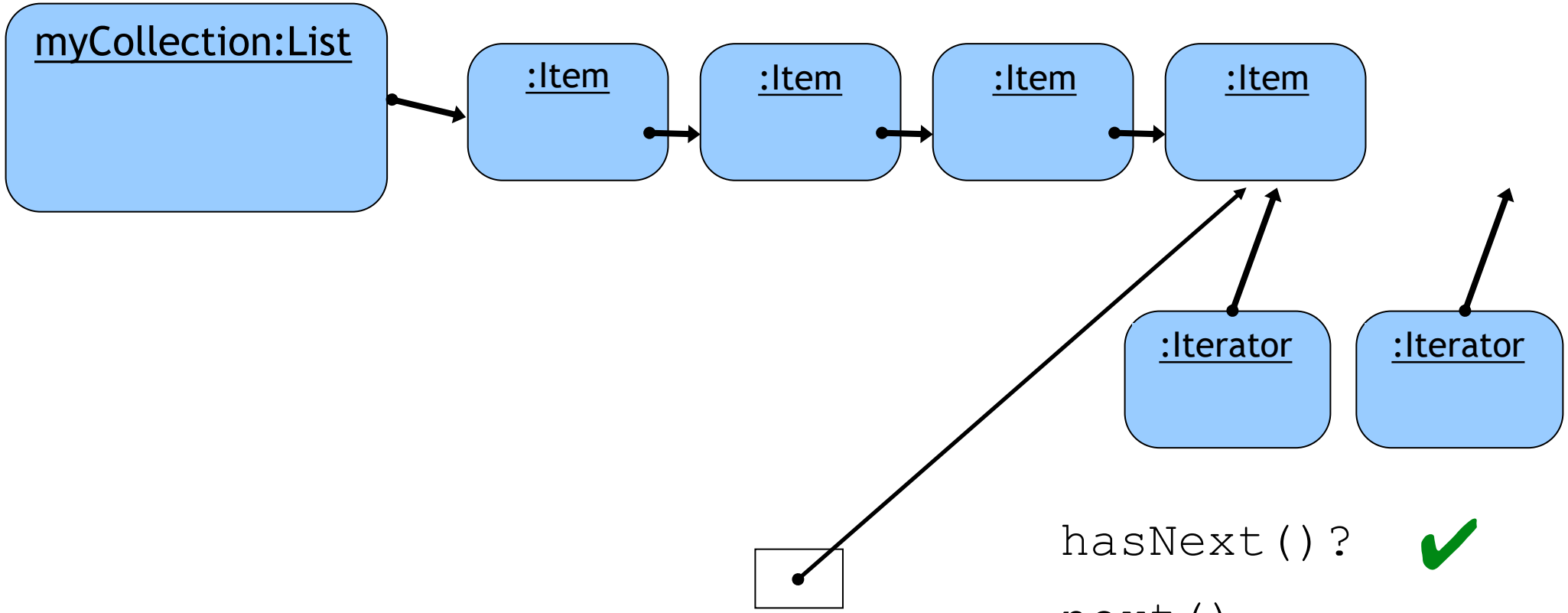
```
Item e = it.next();
```

Iterators



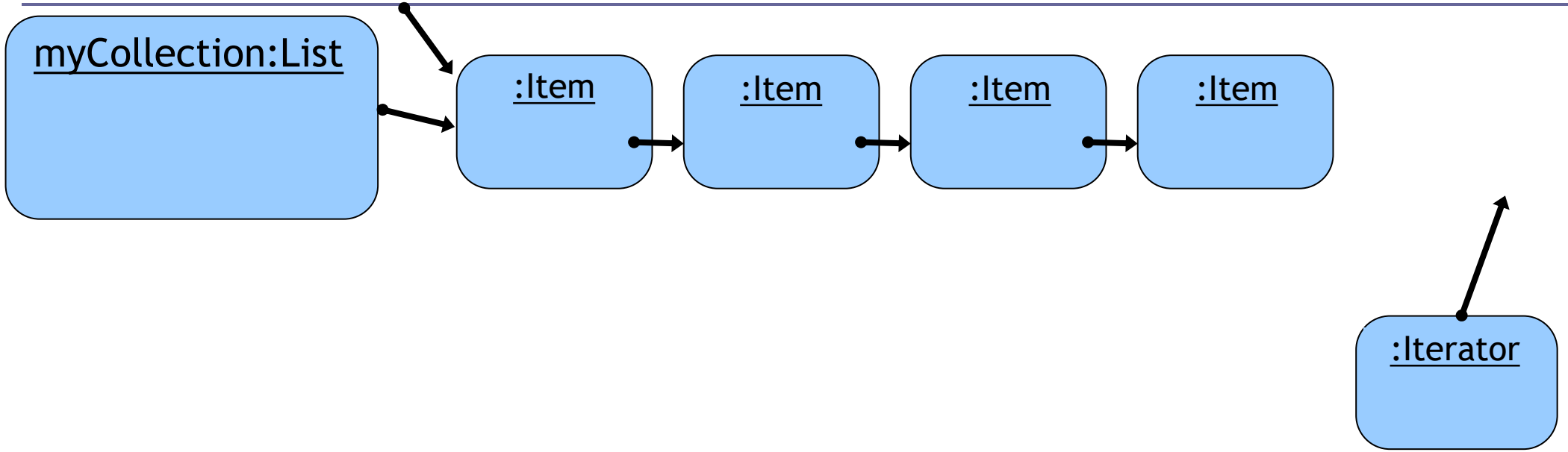

```
Item e = it.next();
```

Iterators




```
Item e = it.next();
```

Iterators



`hasNext()?` ✗

Using an Iterator object

```
import java.util.ArrayList;
import java.util.Iterator;

Iterator<Item> it = myCollection.iterator();
while(it.hasNext()) {
 // Invoke it.next() to obtain the following element
 // Do something with that object
}
```

java.util.Iterator

Returns an Iterator object

```
public void showNotes()
{
    Iterator<String> it = notes.iterator();
    while(it.hasNext()) {
        System.out.println(it.next());
    }
}
```

Different ways of going through a collection

- **for-each loop**
  - Use it when you want to process all of the items.
- **while loop**
  - Use it when you want to stop before the end of the collection or just process certain items.
  - Use it to repeat a block of sentences (no collection needed).
- **Iterator object**
  - Use it when you want to stop before the end of the collection or just process certain items (similar to while).
  - Commonly used with collections (apart from ArrayList) where indices are not useful.

- All collection classes in Java provide iterators. Iteration is an important **pattern** in programming.

Removing elements

- Elements from within a collection cannot be removed while iterating with a for-each loop
- But **can be done** while iterating **using iterators**

```
public void removeNotes(String
noteToRemove) {
    Iterator<String> it = notes.iterator();
    while(it.hasNext()) {
        String note = it.next());
        if (note.equals(noteToRemove))
           it.remove();
    }
}
```

Review

- The while loop allows a block to be executed many times until the condition is false.
- The for-each loop processes all of the items within a collection.
- The while loop allows us to control the iteration by means of a boolean expression.
- All collection classes provide `Iterator` objects to allow sequential access to the items within the collection.

Anonymous objects (I)

Class with two attributes `name` and `phone`

```
private ArrayList<Friend> friends;

    public Contacts()
    {
       friends = new ArrayList<Friend>();
    }
```

We can add a new contact in two different ways:

```
friends.add (new Friend("Sergio", "12121212"));
```

```
Friend newFriend = new Friend(("Sergio", "12121212"));

friends.add(newFriend);
```

Anonymous objects (II)

```
friends.add (new Friend("Sergio", "12121212"));
```

We are doing two things:

- Creating a new `Friend` object
- Passing that new object to the `add` method of `ArrayList`

```
Friend newFriend = new Friend(("Sergio", "12121212"));

friends.add(newFriend);
```

If `newFriend` is not used again in the method, the first version avoids declaring a variable with such a limited use. **It is better to create an anonymous object**.

Chaining method calls

```
public class Plane {
    Person pilot;
    char identifier;
    int fuel;

   // rest of methods omitted
}
```

```
Plane plane = new Plane();
Person pilot = plane.getPilot();
String name = pilot.getName();
```

Instead of declaring `pilot` and `name` variables just to invoke the `getPilot()` and `getName()` methods, a more compact implementation **using anonymous objects** is possible:

```
System.out.println(plane.getPilot().getName());
```

plane.getPilot() returns an anonymous **Person** object and then, **getName()** is invoked over that object