Chapter 4 – The *ArrayList* Class

Contents

[1 Wrapper Classes 2](#_Toc95990313)

[2 Introduction to the *ArrayList* Class 3](#_Toc95990314)

[3 *ArrayList* Basics 4](#_Toc95990315)

[3.1 Basic *ArrayList* Methods 4](#_Toc95990316)

[3.2 Exercises 7](#_Toc95990317)

[3.3 Creating and Populating an *ArrayList* 9](#_Toc95990318)

[3.4 Sorting an *ArrayList* 10](#_Toc95990319)

[4 The *Object* Class’s *equals* Method 10](#_Toc95990320)

[4.1 Exercises 13](#_Toc95990321)

[5 Methods that rely on *equals* 14](#_Toc95990322)

[6 Example: 1-to-Many 15](#_Toc95990323)

[6.1 Exercises 21](#_Toc95990324)

[7 Lists of Lists 21](#_Toc95990325)

[7.1 Exercises 22](#_Toc95990326)

[8 Legacy *ArrayList* 23](#_Toc95990327)

[9 Array vs. *ArrayList* 23](#_Toc95990328)

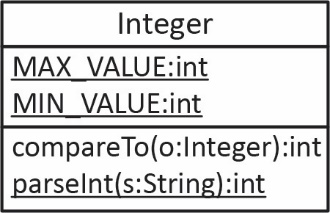
In this chapter we introduce the *ArrayList[[1]](#footnote-1)* class from the Java API. An *ArrayList* is similar to an array, but has many more features. It is especially useful when implementing a 1-many relationship. Before we can discuss the *ArrayList* class, we must first introduce *wrapper classes.*

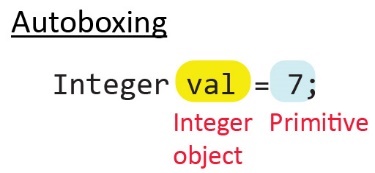
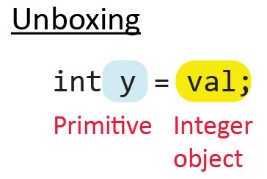
# Wrapper Classes

All primitive data types have a corresponding *wrapper class* which is simply a class that represents a primitive as an object. All the wrapper classes are shown in the table below

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Primitive Type** | **Wrapper Class** |  | **Primitive Type** | **Wrapper Class** |
| char | Character |  | long | Long |
| byte | Byte |  | float | Float |
| short | Short |  | double | Double |
| int | Integer |  | boolean | Boolean |

Why do we use wrapper classes? One reason is that there are places in the Java language where a primitive type is not allowed, but the corresponding wrapper class instance is.

The wrapper class for *int* is *Integer*. A few of the members are shown in the class diagram on the right. There is a constructor for the class; however, it is deprecated[[2]](#footnote-2). Instead, Java uses a technique called *autoboxing*. *Autoboxing* refers to Java’s ability to turn a primitive into an object whose class is the corresponding wrapper class. In other words, it “boxes” the primitive (wraps it up in an object). For example:

*Unboxing* (as shown in the figure above, on the right) refers to Java’s ability to turn a wrapper object into its corresponding primitive type. For example:

We probably will not need to explicitly box or unbox in this class; however, it is useful to know that this is what is occurring in the some of the *ArrayList* methods we consider in the next section. We consider the *compareTo* method in a later chapter. Its use with strings was considered in Chapter 1, Appendix 1 and functions similarly with the wrapper classes. In addition, the numeric-type wrapper classes can be compared with the equality and relational operators (==, <, <=, *etc.*). For example:

Integer x = 7;

Integer y = 4;

**if**(x>y) {

System.***out***.println("X is larger");

}

**int** diff = x.compareTo(y); // 1

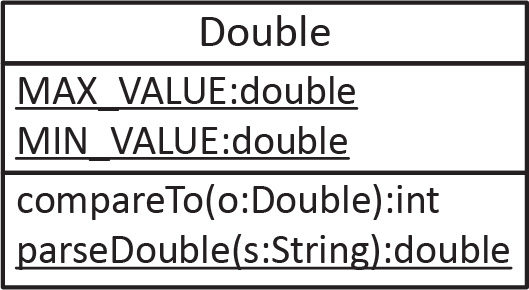
diff = y.compareTo(x); // -1

x = y;

diff = y.compareTo(x); // 0

The static variables, *MAX\_VALUE* and *MIN\_VALUE* are occasionally useful and represent the largest and smallest numbers, respectively, that can be represented as an *int.* For example, if you were searching an *int* array for the smallest value, you might initialize the minimum this way:

**int** min = Integer.***MAX\_VALUE***;

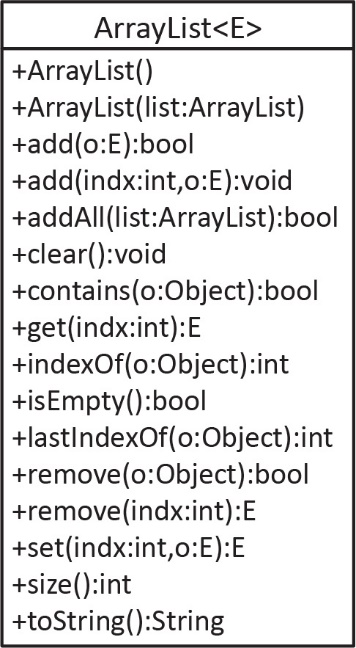
Of course, you are familiar with the static *parseInt* method.

The wrapper class for *double* is *Double*. A few of the members are shown in the class diagram on the right. Boxing and unboxing occur in the same way as with *Integer.*

Double val = 5.5; // Boxing

**double** y = val; // Unboxing

# Introduction to the *ArrayList* Class

*ArrayList* is a class in the Java API and is similar to an array in that it is used to store objects in an indexed list, just like an array; however, it also has useful methods to manipulate the list. Some of the major methods are shown in the diagram on the right. We consider most of these methods in the material that follows. A complete reference for all the members is found in the API[[3]](#footnote-3).

*ArrayList* is defined in the *java.util* package and so to use it, you must import it:

import java.util.ArrayList;

The *ArrayList* class is a *generic class* which is a concept in a broader topic called *generics[[4]](#footnote-4)*. A brief introduction to generics is considered here. The “E” in *ArrayList<E>* in the class diagram above, is a *generic type parameter*. What this means is that you must specify what type of objects the array list will hold when you create an *ArrayList*. For example, to create an *ArrayList* of integers, doubles, and *Accounts*, respectively:

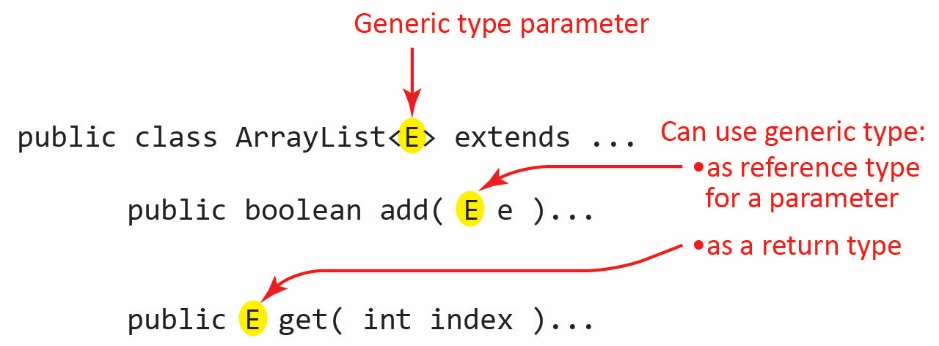
ArrayList<Integer> ints = **new** ArrayList<>();

ArrayList<Double> doubs = **new** ArrayList<>();

ArrayList<Account> accounts = **new** ArrayList<>();

The generic type argument used to create an *ArrayList* must be a class; it cannot be a primitive. Thus, if we want to store *ints* then we must use *ArrayList<Integer>.* As we will see, the boxing and unboxing is automatic.

If you look at the source code for the *ArrayList* class, it will look similar to this:



Essentially, you can think of it this way: when you declare:

ArrayList<Integer> ints = **new** ArrayList<>();

“Integer” is substituted everywhere there is an “E” in the class. Thus, the *add* method accepts an *Integer* which due to autoboxing can be an *int.* Similarly, the *get* method returns an *Integer* which due to unboxing is converted to an *int*.

# *ArrayList* Basics

The code for the example in this section is in the *example\_arraylist\_integer* package.

## Basic *ArrayList* Methods

Below we provide examples of most of the methods in the class diagram above. Here we consider an *ArrayList* of *Iteger*; however, everything would be the same with any other wrapper class, and *String* class. Most will be the same with an *ArrayList* of a custom class. We discuss this in a later [section](#_The_ArrayList_Class’s).

1. We can create an *ArrayList* to hold integers with this statement:

ArrayList<Integer> ints = **new** ArrayList<>();

Note:

* We do not have to specify how many items the *ArrayList* can hold as we do with an array. It will hold as many items as the memory on your computer will allow.

1. The *ArrayList* class has an *add(obj)* method to add objects to the end of the list (after the last one that was added, just as we did when using an array to implement 1-to-many). For example:

ints.add(47);

inserts 47 into the first position in the list:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | … |
| 47 |  |  |  |  |  |  |  |  |  |  |  |

Internally, an *ArrayList* uses an array to hold the objects. Thus, 47 is stored at index=0. If we continue to add ints:

ints.add(91);

ints.add(16);

They will be stored as shown below:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | … |
| 47 | 91 | 16 |  |  |  |  |  |  |  |  |  |

1. The *ArrayList* class has a *size* method that returns the number of objects in the list. For example:

**int** size = ints.size();) // 3

Note:

* + *size=3* in the example.
  + Just as when we consider the array to implement 1-many, there are no “holes” in an *ArrayList*. In other words, there are always elements in positions 0 through *size()-1.*

1. The *ArrayList* class has a *get(i)* method to obtain a reference to the object at index, *i*. The index of elements is the same as an *Array*, it is zero-based. For example:

**int** x = ints.get(1);

System.***out***.println( x ); // 91

Note:

* If the index is less than 0, or greater than *size()-1*, then a runtime error will occur.

1. You can iterate over an *ArrayList* using an enhanced *for* loop just as you would an *Array* or with an indexed loop*.* For example:

|  |  |  |
| --- | --- | --- |
| **Enhanced *for* loop** |  | **Indexed loop** |
| **for**(**int** i : ints) {  System.***out***.print(i + ", ");  } |  | **for**(**int** i=0; i<ints.size(); i++) {  System.***out***.print(ints.get(i) + ", ");  } |

1. The *ArrayList* class has a *add(index, obj)* method that adds *obj* at *index* moving the other items over one to the right (if necessary).For example, the current *ArrayList* has:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | … |
| 47 | 91 | 16 |  |  |  |  |  |  |  |  |  |

And when we execute:

ints.add(1,33);

the result is:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | … |
| 47 | 33 | 91 | 16 |  |  |  |  |  |  |  |  |

1. The *ArrayList* class has a *contains(obj)* method that returns *true* if it contains *obj* and *false* otherwise*.* For example:

System.***out***.println(ints.contains(91)); // true

System.***out***.println(ints.contains(4)); // false

1. The *ArrayList* class has an *indexOf(obj)* method that returns the index where *obj* is located, or -1 if not found. For example:

System.***out***.println(ints.indexOf(91)); // 2

System.***out***.println(ints.indexOf(5)); // -1

1. The *ArrayList* class has a *remove(index:int)* method that removes the *obj* at *index* from the list moving items to the right over one to the left (if necessary). It also returns the removed item (but of course we don’t have to catch the return). For example, the current *ArrayList* has

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | … |
| 47 | 33 | 91 | 16 |  |  |  |  |  |  |  |  |

And when we execute:

**int** x = ints.remove(1);

System.***out***.print(x); // 33

the result is:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | … |
| 47 | 91 | 16 |  |  |  |  |  |  |  |  |  |

The index must be between 0 and *size()-1*, inclusive, otherwise, a runtime error will result.

1. The *ArrayList* class has an overloaded remove method, *remove(obj)* method that removes *obj* from the list if it is found, returning *true* in this case, or *false* otherwise.For example, the current *ArrayList* has:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | … |
| 47 | 91 | 16 |  |  |  |  |  |  |  |  |  |

And when we execute:

**boolean** isRemoved = ints.remove((Integer)91);

System.***out***.print(isRemoved); // true

the result is:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | … |
| 47 | 16 |  |  |  |  |  |  |  |  |  |  |

Note:

* There are two remove methods: *remove(index)* and *remove(object)*. In the example above, where we use *remove(object)*, we must represent the integer we are seeking to remove, 91 as an *Integer* object. If we represented it as *int, e.g. remove(91)*, it would call the *remove(index)* method and try to remove from index=91 *remove(index:int).*
* To correctly use *contains(obj), indexOf(obj), lastIndexOf(obj),* or *remove(obj),* the generic type of the *ArrayList* (*e.g. Integer, Double, String, etc.*) must override the *equals* method. Most classes in the Java API do this; however, when we write custom classes to store in an *ArrayList*, we will need to override *equals*. We consider this in a later [section](#_The_Object_Class’s).

1. The *ArrayList class* allows duplicate elements. The only reason we mention this is that later in the semester, we will learn a somewhat similar class, *Set*, that does not allow duplicate elements. For example:

ints.add(47);

results in:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | … |
| 47 | 16 | 47 |  |  |  |  |  |  |  |  |  |

1. The *ArrayList* class has a *set(index:int, obj)* method that replaces the item at *index* with *obj.* It also returns the replaced value (which we do not have to catch).For example:

**int** z = ints.set(2,5);

System.***out***.println(z); // 47

results in:

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | … |  |
| 47 | 16 | 5 |  |  |  |  |  |  |  |  |  |  |

1. The *ArrayList* class has an *addAll(list:ArrayList)**[[5]](#footnote-5)* method that adds all the elements in *list* to this *ArrayList*.For example:

// Create a second arraylist and add some values

ArrayList<Integer> ints2 = **new** ArrayList<>();

ints2.add(51); ints2.add(9); ints2.add(7);

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | … |
| ints2 | 51 | 9 | 7 |  |  |  |  |  |  |  |  |  |

// Add the values in second list to first list

ints.addAll(ints2);

results in:

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | … |
| ints | 47 | 16 | 5 | 51 | 9 | 7 |  |  |  |  |  |  |

1. The *ArrayList* class has a *toString* method that displays all the values. This can be useful for testing and debugging. For example:

System.***out***.println(ints);

Produces: [47, 16, 5, 51, 9, 7]

1. The *ArrayList* class has a constructor that accepts another *ArrayList*5*.* For example:

ArrayList<Integer> ints3 = **new** ArrayList<>(ints);

Creates a new *ArrayList*, *int3* intialized with the values in *ints*:

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | … |
| ints3 | 47 | 16 | 5 | 51 | 9 | 7 |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | … |
| ints3 | 47 | 16 | 5 | 51 | 9 | 7 |  |  |  |  |  |  |

1. The *ArrayList* class defines the *isEmpty* method that returns *true* if the list is empty (size=0) and *false* otherwise.

System.***out***.println(ints.isEmpty()); // false

System.***out***.println(ints.size()); // 6

1. The *ArrayList* class defines the *clear* method to remove all the items from the list and sets the *size* to 0. For example:

ints.clear();

System.***out***.println(ints.isEmpty()); // true

System.***out***.println(ints.size()); // 0

1. As stated above, some of the methods we considered so far have a return value, which can be useful in some circumstances.

|  |  |
| --- | --- |
| **Method** | **Return** |
| remove(o:Object):bool | *true* if the remove was successful, *false* otherwise |
| remove(indx:int):E | The element that was removed |
| set(indx:int,o:E):E | The element that was replaced |

## Exercises

1. (Solution in *exercise\_arraylist\_string* package) Suppose you have this code in main:

ArrayList<String> vals = **new** ArrayList<>();

vals.add("abc"); vals.add("def"); vals.add("ghi"); vals.add("jkl"); vals.add("mno");

String m = "zzz";

Write a single line of code to do the following. If any method returns something, you must declare a variable to catch the return.

* Get the number of items in the list
* Add *m* to the end of the list
* Add *m* to the 4th position in the list
* Get the first item in the list
* Get the 3rd item in the list
* Get the last item in the list
* Replace the value of the 3rd item in the list with *m*
* Remove the first item in the list
* Remove the 5th item in the list
* Remove the last item in the list
* Remove *m*
* See if *m* is in the list
* Get the index where *m* is located in the list
* Remove all items in the list

Write a few lines of code to do the following.

* Print all the items in the list in a single line with a “-“ in between each.
* Print every other item in the list in a single line with a “-“ in between each.

1. (Solution in *exercise\_stringerator* package) Consider the following *Stringerator* class which simply holds a list, *words*, of strings, which are passed in through the constructor.

**public** **class** Stringerator {

**private** ArrayList<String> words = **new** ArrayList<>();

**public** Stringerator(String[] newWords) {

**for**(String word : newWords) {

words.add(word);

}

}

@Override

**public** String toString() {

**return** words.toString();

}

}

Add the methods below to this class.

1. *countWordsThatEqual(word:String):int* – accepts a word and returns how many occurrences of the word occur in *words.* For example:

String[] words = {"cat", "dog", "ant", "dog"};

Stringerator s = **new** Stringerator(words);

System.***out***.println(s.countWordsThatEqual("cat")); // 1

System.***out***.println(s.countWordsThatEqual("dog")); // 2

System.***out***.println(s.countWordsThatEqual("zebra")); // 0

Hint: loop through *words* and compare each one to *word*.

1. *moveFirstToEnd()* – moves the first word to the end

Before move: [E, A, B, C, D]

After move : [A, B, C, D, E]

1. *swap(i:int,j:int* – accepts two integers and swaps the words at those locations. If either of the two indices is invalid, it should do nothing.

Before Swap : [A, D, C, B, E]

After swap(1,3) : [A, B, C, D, E]

Suggestion: This is the perfect place for a tiny helper method to check if the indices are valid. For example:

**private** **boolean** areIndicesValid(**int** i, **int** j) {

// Return true if both i and j are valid.

}

Why is this a good place for a helper method? The variable, i, requires 2 checks to determine if it is valid, and j requires 2 also. Thus, a total of 4 checks. It is better to hide these details in their own method and by using a descriptive name for the method, out code for *swap* is simpler to understand:

**public** **void** swap(**int** i, **int** j) {

**if**(areIndicesValid(i,j)) {

// Your code goes here

}

}

1. *getMirrorImage():ArrayList* – returns a new arraylist of words that contains the original words followed by a mirror image of the words

String[] words = {"cat", "dog", "ant"};

Stringerator s = **new** Stringerator(words);

System.***out***.println("Original words: " + s); // [cat, dog, ant]

ArrayList<String> mirror = s.getMirrorImage();

System.***out***.println("Mirror: " + mirror); // [cat, dog, ant, ant, dog, cat]

Hint:

* You need to create a new arraylist, *mirror*
* You need to put the words in *words* into *mirror*. There are 3 ways to do this
* Loop over *words* backwards (descending loop) and add each one to *mirror*

1. *getLocationsOf(newWords:ArrayList<String>):ArrayList<Integer> –* accepts an arraylist of words and returns an arraylist of the locations of those words in *words* using -1 if an input word is not found

Words: [A, B, C, D, E]

Words to search for: [C, Z, E, X, A, F]

Locations: [2, -1, 4, -1, 0, -1]

Hint:

* You need to create the return arraylist.
* Loop through *newWords* and use *indexOf* on *words* for each new word.

## Creating and Populating an *ArrayList*

The *Arrays[[6]](#footnote-6)* class has a useful static method, *asList* that accepts an array and returns an *ArrayList.* This is useful for creating and populating an *ArrayList,* particularly when testing. For example:

Integer[] temp = {2,5,7,3,9,6};

ArrayList<Integer> vals = **new** ArrayList<>(Arrays.*asList*(temp));

Or, with more concise notation:

ArrayList<Integer> vals2 = **new** ArrayList<>(Arrays.*asList*(2,5,7,3,9,6));

## Sorting an *ArrayList*

We can sort an *ArrayList* whose generic type is a wrapper class, or *String,* or many other classes in the Java API using the static *sort* method in the *Collections[[7]](#footnote-7)* class. For example, if we have a list, *ints* with these values:

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | … |
| ints3 | 47 | 16 | 5 | 51 | 9 | 7 |  |  |  |  |  |  |

Then:

Collections.*sort*(ints);

Results in:

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | … |
| ints3 | 5 | 7 | 9 | 16 | 47 | 51 |  |  |  |  |  |  |

Note:

* This will only work for classes that implement the *Comparable* interface, which is something we study in the next chapter. The wrapper classes, and *String* implement *Comparable* so we can correctly sort any lists of these.
* If you want to sort a list, but preserve the order of the source list, then you must make a copy and sort that. For example:

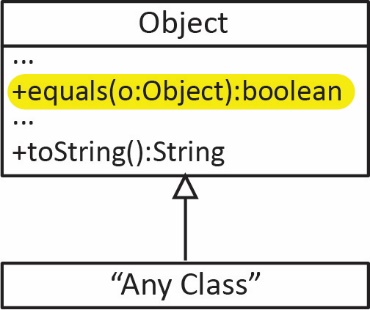
// Create new list from original

ArrayList<Integer> sortedInts = **new** ArrayList<>(ints);

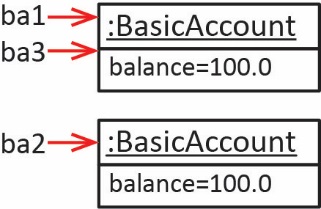
// Sort the new list

Collections.*sort*(sortedInts);

# The *Object* Class’s *equals* Method

As discussed in Chapter 3, Section 5, every class inherits certain methods from the *Object* class. As shown in the diagram on the right, one of those methods is *equals.* The *Object* class defines an *equals* method as shown on the right. Thus, every class inherits the *equals* method. The implementation of *equals* in the *Object* class returns *true* if two objects occupy the same location in memory and *false* otherwise.

For example, consider the code below (code in *example\_equals\_basic\_account* package) and the object diagram on the right. Since *ba1* and *ba3* point to the same object in memory, they are equal*.* Since, *ba1* and *ba2* point to the different objects in memory, so they are not equal*.*

BasicAccount ba1 = **new** BasicAccount(100.0);

BasicAccount ba2 = **new** BasicAccount(100.0);

BasicAccount ba3 = ba1;

System.***out***.println(ba1.equals(ba3)); // true

System.***out***.println(ba1.equals(ba2)); // false

Thus, the implementation of *equals* in the *Object* class is exactly the same as the “==” boolean operator:

System.***out***.println(ba1==ba3); // true

System.***out***.println(ba1==ba2); // false

Many classes in the Java API override *equals* to define what it means for two objects to be “equal” in a particular context. For example, the *String* class overrides *equals* to return *true* if the contents of two strings are the same. For example:

String x = "Cat";

String y = "Hat";

String z = "Cat";

System.***out***.println(x.equals(y)); // false

System.***out***.println(x.equals(z)); // true

It is frequently useful to override the *equals* method to supply your own, custom definition of *equals*. For example, you may want to have a situation where two different *Person* objects are considered equal if they have the same SSN (regardless of their name, or any other details). I call this *logical equality.* In other words, implementing *equals* so that two distinct objects in memory are considered equal. Why this is useful will be illustrated in the next [section](#_The_ArrayList_Class’s). For now, the short answer is that the *ArrayList* class has some methods that depend on *equals* being overridden.

The signature of the *equals* method is:

**public** **boolean** equals(Object o)

Generally, we want the *equals* method to compare two objects of the same type*.* However, notice that *equals* accepts an *Object*. Thus, when we override it should cast *o* to the class that is overriding *equals* (usually).

Suppose we have a *Person* class as shown below and we want to we override *equals* to return *true* when two *Person* objects have the same *ssn,* regardless of their *names.* The code for this and subsequent variations are found in the *example\_equals\_person* package.

|  |  |
| --- | --- |
| **Class** | **Sample Code** |
| **public** **class** Person {  **protected** **int** ssn;  **protected** String name;    **public** Person(String name, **int** ssn) {  **this**.ssn = ssn;  **this**.name = name;  }    @Override  **public** **boolean** equals(Object o) {  **if**(o **instanceof** Person) {  Person p = (Person)o;  **if**(**this**.ssn == p.ssn) {  **return** **true**;  }  **else** {  **return** **false**;  }  }  **else** {  **return** **false**;  }  }  } | Person p1 = **new** Person("Shay", 123);  Person p2 = **new** Person("Shay", 456);  Person p3 = **new** Person("Julie", 123);  System.***out***.println(p1.equals(p2)); // false  System.***out***.println(p1.equals(p3)); // true  Notice that *p1* and *p3* are different objects in memory. However, they are considered equal because they have the same SSN, regardless of the fact that they have different *names*. |

Note that the *equals* method above can be written much more succinctly

**public** **boolean** equals(Object o) {

**if**(o **instanceof** Person) {

Person p = (Person)o;

**return** **this**.ssn == p.ssn;

}

**return** **false**;

}

Alternatively, if we want two *Person* objects to be considered equal if they have the same *ssn* and *name,* we could override equals as shown below. Notice that when we are comparing *name*, which is a *String*, that we use the *String* class’s *equals* method.

**public** **boolean** equals(Object o) {

**if**(o **instanceof** Person) {

Person p = (Person)o;

**return** (**this**.ssn == p.ssn) &&

(**this**.name.equals(p.name));

}

**return** **false**;

}

(The code for this variation is not in the code download) Suppose a *Person* class has *firstName* and *lastName* properties and that two *Person* objects should be considered equal if both their first and last names are the same:

**public** **boolean** equals(Object o) {

**if**(o **instanceof** Person) {

Person p = (Person)o;

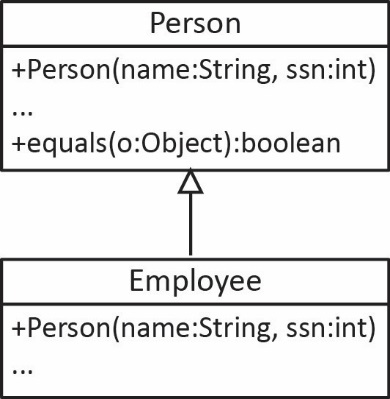
**return** (**this**.lastName.equals(p.lastName)) &&

(**this**.firstName.equals(p.firstName));

}

**return** **false**;

}

Example – Suppose we have the *Person* class above, which overrides *equals* so that two *Person* objects are considered equal if they have the same *ssn.* Suppose we also have a subclass of *Person* named *Employee*. If *Employee* does not override *equals* then it inherits the *Person* classes’ *equals* method. Thus, a *Person* and an *Employee* could be equal.

Person p1 = **new** Person("Shay", 123);

Employee e1 = **new** Employee("Jeri", 123);

Employee e2 = **new** Employee("Jeri", 789);

Employee e3 = **new** Employee("Suze", 789);

System.***out***.println(p1.equals(e1)); // true

System.***out***.println(e1.equals(e2)); // false

System.***out***.println(e2.equals(e3)); // true

## Exercises

1. (Solution in *exercise\_martian\_equals*) Suppose you have a *Martian* class with *wavelength* and *elevation* properties which are both *int*. Write an *equals* method for this class so that two *Martian* objects are considered equal if both their *wavelength* and *elevation* properties are equal.
2. (Solution in *exercise\_department\_equals*) Suppose you have a *Department* class with *code* and *number* properties which are both *String*. For example, a *code* might be, “Engineering”, and *number* might be, “023”. Write an *equals* method for this class so that two *Department* objects are considered equal if both their *code* and *number* properties are equal.
3. (Solution in *exercise\_rectangle\_equals*) Consider the *Rectangle* class below. Add an overridden *equals* method so that two *Rectangles* are considered equal if their areas are within 0.1 of each other.

**public** **class** Rectangle {

**private** **double** length;

**private** **double** width;

**public** Rectangle(**double** length, **double** width) {

**this**.length = length;

**this**.width = width;

}

**public** **double** area() {

**return** length\*width;

}

@Override

**public** String toString() {

String msg = String.*format*("len=%,.2f, wid=%,.2f, area=%.2f",

length, width, area());

**return** msg;

}

}

For example, the following test code:

**public** **class** RectangleTest {

**public** **static** **void** main(String[] args) {

*testEquals*();

}

**public** **static** **void** testEquals() {

Rectangle r1 = **new** Rectangle(2.0,4.0);

Rectangle r2 = **new** Rectangle(2.0,5.0);

Rectangle r3 = **new** Rectangle(1.95,4.06);

String msg = String.*format*("r1.area()=%.2f, r2.area()=%.2f,

r3.area()=%.2f", r1.area(), r2.area(), r3.area());

System.***out***.println(msg);

System.***out***.println("r1.equals(r2)=" + r1.equals(r2));

System.***out***.println("r1.equals(r3)=" + r1.equals(r3));

}

}

Would produce these results:

r1.area()=8.00, r2.area()=10.00, r3.area()=7.92

r1.equals(r2)=false

r1.equals(r3)=true

# Methods that rely on *equals*

As stated earlier, an *ArrayList* can hold any type of object, in particular, objects from a custom class:

ArrayList<Person> people = **new** ArrayList<>();

ArrayList<CheckingAccount> accounts = **new** ArrayList<>();

It is important to remember that an *ArrayList* (like an Array) stores references to objects. Thus, when you use *get* you are getting a reference to the object. If you then use that reference to call a method that changes the state of the object, then the *ArrayList* contains a reference to this changed object (of course!).

If you want to use the methods below on an *ArrayList* of a custom class, then the class must override *equals.*

**public** **boolean** contains​(Object o)

**public** **int** indexOf​(Object o)

**public** **int** lastIndexOf​(Object o)

**public** **boolean** remove​(Object o)

For example, consider Java’s implemention of *indexOf* below. Note that *elementData* is the array inside the *ArrayList* that holds the elements. Here, we see that the method simply loops over the elements in the array and checks to see if any one of them is *equal* to the argument, *o*. Note also that the *contains* method simply delegates to *indexOf*.

|  |  |
| --- | --- |
| ***ArrayList indexOf* Method** | ***ArrayList contains* Method** |
| **public** **int** indexOf(Object o) {  ...  **for** (**int** i = 0; i < size; i++) {  **if** (o.equals(elementData[i]))  **return** i;  }  **return** -1;  } | **public** **boolean** contains(Object o) {  **return** indexOf(o) >= 0;  } |

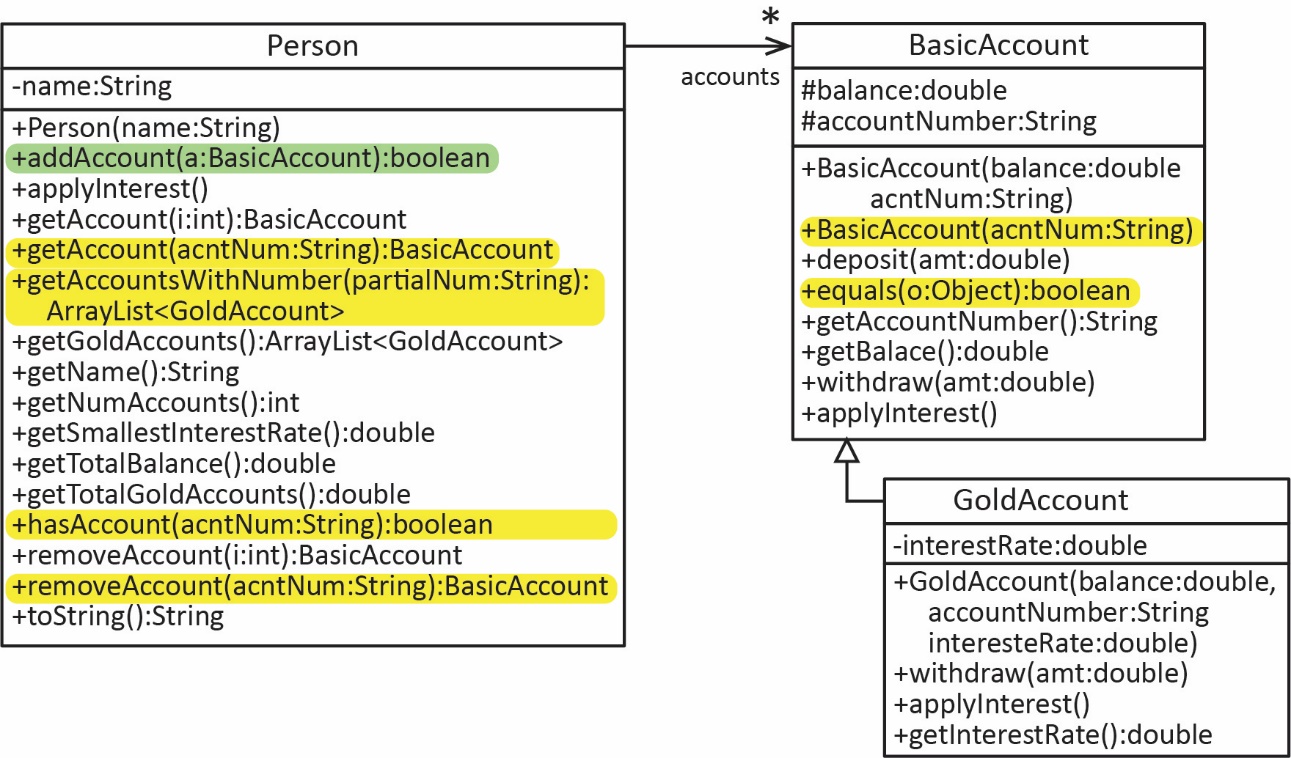
(Code is in *example\_equals\_arraylist\_dogs* package) For example, suppose we have a situation where two *Dog* objects should be considered equal if they have the same name. As shown below on the left, we override *equals*. On the right, we illustrate the methods above.

|  |  |
| --- | --- |
| ***Dog* Class** | **Example** |
| **public** **class** Dog {  **private** String name;    **public** Dog(String name) {  **this**.name = name;  }    **public** **boolean** equals(Object o) {  **if**(o **instanceof** Dog) {  Dog d = (Dog)o;  **return** name.equals(d.name);  }  **return** **false**;  }  } | ArrayList<Dog> dogs = **new** ArrayList<>(  Arrays.*asList*(  **new** Dog("Juno"),  **new** Dog("Leo"),  **new** Dog("Chaps"),  **new** Dog("Ace"),  **new** Dog("Chaps")  ));  System.***out***.println(  dogs.contains(**new** Dog("Ace"))); // true  System.***out***.println(  dogs.indexOf(**new** Dog("Chaps"))); // 2  System.***out***.println(  dogs.indexOf(**new** Dog("Zoro"))); // -1  System.***out***.println(  dogs.lastIndexOf(**new** Dog("Chaps"))); // 4  System.***out***.println(  dogs.remove(**new** Dog("Chaps"))); // true  System.***out***.println(  dogs.remove(**new** Dog("Zoro"))); // false |

# Example: 1-to-Many

The code for this example is in the *example\_person\_accounts\_arraylist* package.

Consider the major example considered in Ch 3, Section 8, and continued through the end of that chapter. Here, consider that same example except that we change the array that holds the *BasicAccounts* to an *ArrayList*. We also add a few new methods (highlighted in yellow in the class diagram below). We also change the requirements of the *addAccount* method. See the code found on the Schedule. Below, we will consider all the methods in the *Person* class, and the two highlighted methods in *BasicAccount.*



1. Before we look at the modifications to the *Person* class, we choose to add a new requirement for accounts: two *BasicAccount* objects are considered the same (equal) if they have the same *accountNumber*. Thus, we override the *equals* method in *BasicAccount* to reflect this:

**public** **boolean** equals(Object o) {

**if**(!(o **instanceof** BasicAccount)) {

**return** **false**;

}

BasicAccount otherAccount = (BasicAccount)o;

**return** **this**.accountNumber.equals(otherAccount.accountNumber);

}

1. Now, we begin to modify the *Person* class. First, consider the instance variables from the array implementation:

**private** String name;

**private** BasicAccount[] accounts = **new** BasicAccount[10];

**private** **int** numAccounts = 0;

For the *ArrayList* implementation, we need:

**private** String name;

**private** ArrayList<BasicAccount> accounts = **new** ArrayList<>();

Notice that we no longer need the *numAccounts* instance variable. The reason is simple: the *ArrayList* itself has a *size* method that always tells exactly how many objects it holds. Thus, *getNumAccounts* changes:

|  |  |
| --- | --- |
| **Array implementation** | ***ArrayList* implementation** |
| **public** **int** getNumAccounts() {  **return** numAccounts;  } | **public** **int** getNumAccounts() {  **return** accounts.size();  } |

1. We add two new requirements for the *addAccount* method:

* An account is added only if it doesn’t already exist (*i.e.* there isn’t an existing account with the same *accountNumber*).
* Return *true* if the account is added successfully, and *false* otherwise.

Thus, we modify the *addAccount* to first check to see if the account we are attempting to add already exists.

**public** **boolean** addAccount(BasicAccount a) {

**if**(accounts.contains(a)) {

**return** **false**;

}

accounts.add(a);

**return** **true**;

}

Remember that the *contains* method works, here, because we have overridden *equals* in *BasicAccount*.

1. Next, we consider the *applyInterest* method which changes only slightly:

|  |  |
| --- | --- |
| **Array implementation** | ***ArrayList* implementation** |
| **public** **void** applyInterest() {  **for**(**int** i=0; i<numAccounts; i++)  accounts[i].applyInterest();  } | **public** **void** applyInterest() {  **for**(BasicAccount a : accounts) {  a.applyInterest();  }  }  // Alternate version using indexed loop  **public** **void** applyInterestAlternate() {  **for**(**int** i=0; i<accounts.size(); i++) {  accounts.get(i).applyInterest();  }  } |

Notice in the first *ArrayList* implementation, we no longer require an indexed for loop. The *for-each* loop iterates over only the accounts that have been added. However, we can use an indexed for loop as shown in: *applyInterestAlternate.*

1. Next, we consider the *getAccount* method which changes only slightly:

|  |  |
| --- | --- |
| **Array implementation** | ***ArrayList* implementation** |
| **public** BasicAccount getAccount(**int** i) {  **if**(i>=0 && i<numAccounts) {  **return** accounts[i];  }  **return** **null**;  } | **public** BasicAccount getAccount(**int** i) {  **if**(i>=0 && i<accounts.size()) {  **return** accounts.get(i);  }  **return** **null**;  } |

1. Next, we add a new, overloaded *getAccount* that accepts an account number and returns the account that has an account number that matches the argument, if it exists, and *null* otherwise.

**public** BasicAccount getAccount(String accountNumber) {

1. A brute-force solution is:

**public** BasicAccount getAccount2(String accountNumber) {

**for**(BasicAccount acnt : accounts) {

**if**(acnt.getAccountNumber().equals(accountNumber)) {

**return** acnt;

}

}

**return** **null**;

}

1. A better approach that makes use of the *BasicAccount*’s *equals* method, indirectly through the use of *indexOf* is:

**public** BasicAccount getAccount(String accountNumber) {

BasicAccount acntKey = **new** BasicAccount(accountNumber, 0.0); // Create dummy account

**int** pos = accounts.indexOf(acntKey); // Find location of dummy account in list

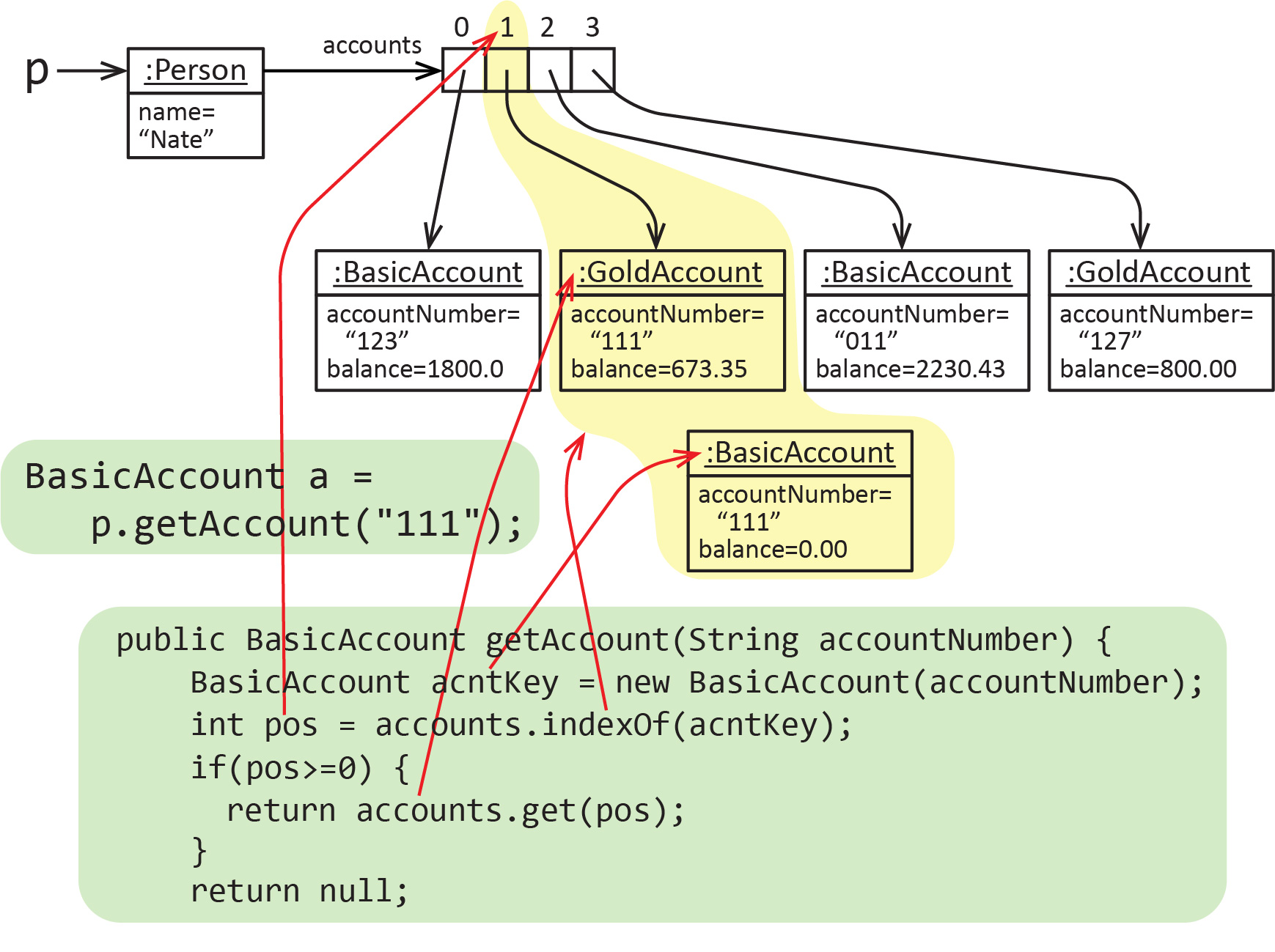
**if**(pos>=0) { // If the location >=0, then found

**return** accounts.get(pos); // Return the real (matching) account

}

**return** **null**;

}

This approach creates a *dummy* account, *acntKey* using the supplied account number and an arbitrary initial balance of 0.0. Then, the dummy account is passed to *indexOf* to find the position of the dummy in the *accounts* list. Provided the return position is 0 or greater, the account at that location is returned.

Why is this approach better? If you replace a loop with a method call (*indexOf*), the code is simpler and less subject to error.

To make this approach slightly simpler, we introduce a new constructor in the *BasicAccount* class which only accepts an account number:

**protected** BasicAccount(String accountNumber) {

**this**(accountNumber, 0.0);

}

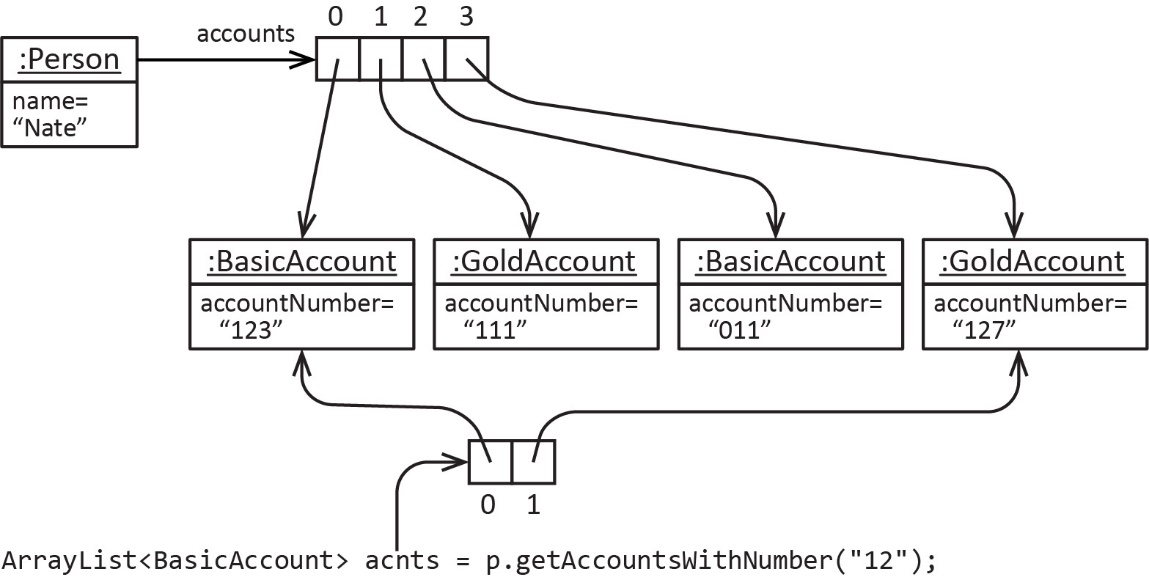
Thus, the first line of the method can be replaced with:

BasicAccount acntKey = **new** BasicAccount(accountNumber); // Create dummy account

1. Next, we introduce a new method that accepts the beginning of an account number (a partial account number) and returns an *ArrayList* of all accounts that begin with that partial account number.

**public** ArrayList<BasicAccount> getAccountsWithNumber(String partialNum) {

In other words, the method returns accounts whose first characters exactly match *partialNum*. For example, suppose a *Person* has 4 accounts as shown below. Then, calling the method with “12” returns a list with the first and last accounts.



In this case, we will have to use *brute force*, that is, loop through all the accounts and see which one’s match.

**public** ArrayList<BasicAccount> getAccountsWithNumber(String partialNum) {

ArrayList<BasicAccount> acntMatches = **new** ArrayList<>();

**int** len = partialNum.length();

**for**(BasicAccount a : accounts) {

**if**(a.getAccountNumber().substring(0,len).equals(partialNum)) {

acntMatches.add(a);

}

}

**return** acntMatches;

}

1. The *getGoldAccounts* method is considerably simplified with the introduction of the *ArrayList*.
2. With the array approach, we had to use a helper method (*getNumGoldAccount* – not shown) to count the number of *GoldAccounts* so that we could set the appropriate size for the return array:

**public** GoldAccount[] getGoldAccounts() {

GoldAccount[] gAcnts = **new** GoldAccount[getNumGoldAccounts()];

**int** j=0;

**for**(**int** i=0; i<numAccounts; i++) {

BasicAccount a = accounts[i];

**if**(a **instanceof** GoldAccount) {

gAcnts[j++] = (GoldAccount)a;

}

}

**return** gAcnts;

}

1. With the *ArrayList* approach, we don’t need to count the *GoldAccounts* because an *ArrayList* can hold any number of objects:

**public** ArrayList<GoldAccount> getGoldAccounts() {

ArrayList<GoldAccount> gAcnts = **new** ArrayList<>();

**for**(BasicAccount a : accounts) {

**if**(a **instanceof** GoldAccount) {

gAcnts.add((GoldAccount)a);

}

}

**return** gAcnts;

}

1. The *getTotalBalance, getSmallestInterestRate,* and *getTotalGoldAccounts* methods require small changes:

**public** **double** getTotalBalance() {

**double** sum=0.0;

**for**(BasicAccount a : accounts) {

sum += a.getBalance();

}

**return** sum;

}

**public** **double** getSmallestInterestRate() {

**double** smallestIntRate = Double.***MAX\_VALUE***;

**for**(BasicAccount a : accounts) {

**if**(a **instanceof** GoldAccount) {

GoldAccount ga = (GoldAccount)a;

**if**(ga.getInterestRate()<smallestIntRate) {

smallestIntRate = ga.getInterestRate();

}

}

}

**return** smallestIntRate;

}

**public** **double** getTotalGoldAccounts() {

**double** sum = 0.0;

**for**(BasicAccount a : accounts) {

**if**(a **instanceof** GoldAccount) {

sum += a.getBalance();

}

}

**return** sum;

}

1. We add a new method, *hasAccount* to see if an account exists for a supplied account number. The approach is similar to *getAccount(acntNum:String)* where we used a dummy account as the search key for the *contains* method:

**public** **boolean** hasAccount(String accountNumber) {

BasicAccount acntKey = **new** BasicAccount(accountNumber);

**if**(accounts.contains(acntKey)) {

**return** **true**;

}

**return** **false**;

}

Alternately, we could use the *indexOf* method:

**public** **boolean** hasAccountAlternate(String accountNumber) {

BasicAccount acntKey = **new** BasicAccount(accountNumber);

**int** pos = accounts.indexOf(acntKey);

**if**(pos>=0) {

**return** **true**;

}

**return** **false**;

}

1. Next, we modify the *removeAccount(i:int)* method to reflect that we are using an *ArrayList.* Notice, of course, that we no longer need to shift the accounts to the right of the one being deleted, over one position to the left, because, the *ArrayList* does this for us.

**public** BasicAccount removeAccount(**int** i) {

**if**(i>=0 && i<accounts.size()) {

BasicAccount retAccount = accounts.get(i);

accounts.remove(i);

**return** retAccount;

}

**return** **null**;

}

Note, that we could do this in one line of code the *ArrayList’s remove(i:int)* method returns the item that was removed.

**public** BasicAccount removeAccountAlternate(**int** i) {

**if**(i>=0 && i<accounts.size()) {

**return** accounts.remove(i);

}

**return** **null**;

}

1. We introduce a new method, *removeAccount(acntNum:String)* that removes and returns an account based on an account number. Notice that we use the “dummy” account approach again as we did with the overloaded *getAccount* method that accepts an account number:

**public** BasicAccount removeAccount(String accountNumber) {

BasicAccount acntKey = **new** BasicAccount(accountNumber);

**int** pos = accounts.indexOf(acntKey);

**if**(pos>=0) {

**return** accounts.remove(pos); // uses index to remove

}

**return** **null**;

}

Note, that we could use the alternate version below which uses *remove(obj:Object)*. However, it does not return the object that was removed (it returns *true* if the item was removed and *false* otherwise).

**public** BasicAccount removeAccountAlternate(String accountNumber) {

BasicAccount acntKey = **new** BasicAccount(accountNumber);

**int** pos = accounts.indexOf(acntKey);

**if**(pos>=0) {

BasicAccount retAccount = accounts.get(pos);

accounts.remove(acntKey); // uses Object to remove

**return** retAccount;

}

**return** **null**;

}

## Exercises

1. (Solution in *practice\_probem\_corporation\_salesreport\_arraylist*) Consider the *CorporationReports* class from a Ch. 3 Exercise. There, we saw that *CorporationReports* had many *SalesReports* (and subclass *DetailedSalesReport*). Do the following in the *CorporationReports* class:
2. Change the *reports* instance variable from an array to an *ArrayList*.
3. Remove the *numReports* instance variable.
4. Change the *getDetailedReports* method so that it returns an *ArrayList<DetailedReport>* instead of an array. Modify the method appropriately so that it supports this change and adapts to the *reports* instance variable (*ArrayList*)
5. Modify all methods to adapt to the *reports* instance variable (*ArrayList*)*.* Hint: the example above is a bit more complicated because of the fact that we override *equals* and enforce a “no duplicates” policy. For this reports problem, we are not overriding *equals*, and so some of the methods are simpler than the corresponding ones above.
6. Modify the test code.

# Lists of Lists

The code for the example in this section is in the *example\_list\_of\_lists\_of\_scores* package.

The *generic type argument* for an *ArrayList* can be any valid *reference type*. For example, suppose we have a number of lists and we would like to group them together. If the lists are all the same type, then we can create a list of lists. For example, suppose we have test scores for several sections of a class, where each section’s scores are in its own list.

ArrayList<Integer> sec1Scores = **new** ArrayList<>(Arrays.*asList*(78,89,82,94,73));

ArrayList<Integer> sec2Scores = **new** ArrayList<>(Arrays.*asList*(98,94,86,91,93,85 ));

ArrayList<Integer> sec3Scores = **new** ArrayList<>(Arrays.*asList*(63,78,74,68));

Suppose we want to create an *ArrayList, sections* to store these lists, then we would specify the *generic type argument* as *ArrayList<Integer>* as that is the type of each element in the list.

ArrayList<ArrayList<Integer>> sections = **new** ArrayList<>();

Next, we add the three sections:

sections.add(secScores1); sections.add(secScores2); sections.add(secScores3);

We can access the second list:

ArrayList<Integer> sec = sections.get(1);

We can iterate over the list of lists using an enhanced for loop:

**for**(ArrayList<Integer> section : sections) {

**for**(**int** score : section) {

System.***out***.print(score + " ");

}

System.***out***.print("\n");

}

Or, we can iterate over this list of lists using an indexed loop:

**for**(**int** i=0; i<sections.size(); i++) {

ArrayList<Integer> section = sections.get(i);

**for**(**int** j=0; j<section.size(); j++) {

**int** score = section.get(j);

System.***out***.print(score + " ");

}

System.***out***.print("\n");

}

## Exercises

1. (Solution in *exercise\_wordlists* package) Consider the following class:

**public** **class** WordLists {

ArrayList<ArrayList<String>> lists = **new** ArrayList<>();

**public** WordLists() {}

}

Add the following methods to this class:

|  |  |
| --- | --- |
| **Method** | **Description** |
| addList | Accepts an ArrayListof strings and adds it to lists*.* |
| getList | Accepts an integer index and if it is valid, returns the list at that index |
| countOccurrences | Accepts a string, *word* and returns the number of times wordoccurs in total over all the lists. No partial matches, words in the lists must exactly match wordto be counted. |
| getAllWordsSorted | Returns an ArrayListof all the words in all the lists, sorted. |
| getTotalNumWords | Returns the total count of all the words in all the lists |

1. (Solution in *exercise\_list\_of\_lists\_of\_blobs* package) Suppose you have a *Blob* class that contains an integer *code* which is supplied when it is created:

**public** **class** Blob {

**int** code;

**public** Blob(**int** code) {

**this**.code = code;

}

@Override

**public** String toString() {

**return** "Blob code=" + code;

}

}

Consider this snippet of code:

ArrayList<Blob> blobs1 = **new** ArrayList<>(Arrays.*asList*(

**new** Blob(2), **new** Blob(8), **new** Blob(6)));

ArrayList<Blob> blobs2 = **new** ArrayList<>(Arrays.*asList*(

**new** Blob(9), **new** Blob(4)));

ArrayList<Blob> blobs3 = **new** ArrayList<>(Arrays.*asList*(

**new** Blob(2), **new** Blob(8), **new** Blob(2), **new** Blob(3)));

ArrayList<ArrayList<Blob>> blobs = **new** ArrayList<>();

blobs.add(blobs1);

blobs.add(blobs2);

blobs.add(blobs3);

Write a static method, *concatenateBlobList* that accepts a list of lists of *Blobs* similar to the one shown above. This method should return a list of *Blobs* that contains all the blobs in all the lists.

1. (Solution in *exercise\_list\_of\_lists\_of\_dogs* package)Study the code below carefully. Fill in the blanks so that this code works properly.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ dogs1 = **new** ArrayList<>(Arrays.*asList*(**new** Dog(), **new** Dog()));

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ dogs2 = **new** ArrayList<>(Arrays.*asList*(**new** Dog(), **new** Dog(),

**new** Dog()));

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ dogLists = **new** ArrayList<>();

dogLists.add(dogs1);

dogLists.add(dogs2);

**for**( \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ dogs : dogLists ) {

**for**(\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ d : dogs)

System.***out***.println(d);

}

# Legacy *ArrayList*

Generics was introduced in Java 1.5 (2004). Prior to that, the *ArrayList* class held *Object* instances. For backwards compatibility, Java allows the non-generic versions of all generic classes (and interfaces) in the API to be used (however, you will get a compile warning). For example, the *ArrayList* below is defined without generics.

ArrayList dogs = **new** ArrayList();

We can add a *Dog* because a *Dog* is an *Object:*

dogs.add(**new** Dog("Spot"));

Since the non-generic *ArrayList* holds *Object* instances, a cast is required when retrieving items (unless you want to retrieve an element as an *Object*):

Dog dog = (Dog)dogs.get(0);

Even though *dogs* is ostensibly for holding *Dog* objects, we can add any type of object:

dogs.add( **new** Computer("fast") );

So now the *ArrayList* holds a *Dog* at index 0, and a *Computer* at index 1. Suppose we *get* the *Object* at index 1 and (incorrectly) cast it as a *Dog*:

Dog d = (Dog)dogs.get(1);

The code compiles, but it will generate a runtime error when the line is executed and throws a *ClassCastException*. Thus, one of the benefits of generics as it provides *type safety*, meaning we detect errors at compile time.

# Array vs. *ArrayList*

At this point in the course, I usually hear from a student, “why did we study arrays, when an *ArrayList* is simpler and more powerful?”. Some comments in response:

1. An array is the basis for numerous data structures in computing, including the *ArrayList* itself, so it must be mastered in an introductory programming course. As an analogy, before using a nail gun, one must master the use of a hammer.
2. When speed and memory are critical, then an array will provide better performance.
3. If you have a fixed number of items, an array can be more efficient, and some consider it to be simpler.
4. An *ArrayList* can’t store primitives. Thus, when primitives are added to an *ArrayList* they are wrapped (boxed) with the appropriate wrapper class, *i.e.* as an object. This increases memory. For example, an *int* requires 4 bytes, an *Integer* requires 16 bytes.
5. An array can be 2-d, 3-d, or higher. These are useful for representing the visual environment of a game, an image, terrain maps, matrices, *etc.*
6. *Collections.sort* converts the list to be sorted to an array before sorting.
7. However, in my experience, I use an *ArrayList* instead of an array 95% of the time.

1. <https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html> [↑](#footnote-ref-1)
2. *Deprecated* means that Oracle recommends that a class (or method, constructor, or field) not be used as it might not be supported in future releases. [↑](#footnote-ref-2)
3. <https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html> [↑](#footnote-ref-3)
4. <https://docs.oracle.com/javase/tutorial/java/generics/index.html> [↑](#footnote-ref-4)
5. Technically, it accepts any type of *Collection,* a supertype of *ArrayList*. We consider this in a later chapter. [↑](#footnote-ref-5)
6. <https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html> [↑](#footnote-ref-6)
7. <https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html> [↑](#footnote-ref-7)