KIT101 Programming Fundamentals

PP Task 7.1 Arrays of Objects

Overview

Purpose:	Learn how to declare and work with arrays of objects.
Task:	Extend your custom data type program (PP Task 6.1 or CR Task 6.2) to use arrays.
Learning Outcomes:	2 3 4
Time:	Aim to complete this task before the start of Week 8.

Resources:

- Introductory Programming Notes:
  - 08 Making Decisions
  - 09 Repeating Actions with Loops
  - 10 Managing Collections with Arrays
  - 11 Creating Your Own Data Types
  - 13 Functional Decomposition
- Online Mini-lectures:
  - Arrays:
    - Introduction and key syntax
    - Tracing array code
    - Writing methods to work with arrays
  - Creating Your Own Classes

Note: This task should be attempted after you have **completed** 5.2PP *Collections of Strings* and 6.1PP *Objects as Records*, as you will adapt some functionality (code) from 5.2PP to extend your single-object 6.1PP program.

If you have completed 6.2CR *Objects with More Abilities* then use that as a starting point instead.

Submission Details

Upload the following to the MyLO submission folder for this task:

- Program **source code** implementing the data-oriented class and a program to manage an array of objects of that class (two .java files)
- A **screenshot** showing the execution of your program

Assessment Criteria

A 🏁 Completed submission will:

- Show that you are working with multiple values in an array, using methods for different parts of the program's functionality
- Show that the values are of your custom data type from task 6.1PP or 6.2CR
- Implement the required functionality for your chosen option, including bulk add of records when the program starts
- Follow the unit's coding style for layout, variable names (use of case), and commenting (including your name at the top)
- Compile and run, with the screenshot showing that it works, including data validation

Instructions

In this task you will extend your program from PP Task 6.1 (or CR Task 6.2 if you wish) to manage an array of objects of the class you created. The program can then add values to the array, output these values, and output some additional summary information.

Start with a **copy of your 6.1PP (or 6.2CR) program**. Also open your 5.2PP program as you will copy and adapt some of its code.

You will need to make the following changes in your main program:

- Create a method **to enter records in bulk from user input via the console**. This method will have parameters for a Scanner (to read user input), the array of objects, and the number of records to be added in bulk. It will use a loop to allow the user to enter details for that many entries. It can assume that the array is initially empty. This will be used as the first step in the program, after which the user will interact via a menu.
- Create/adapt a **printAll** method to print all of the objects in an array, one per line.
- Create/adapt a method to **add** a new object of your custom type to your array. This method will follow the naming convention of add*ClassName* used by the read*ClassName* method of 6.1PP.
- Create two methods to calculate summary information from your objects.
- Adjust main() to store an array of your objects, with a maximum capacity of 50.
    1. Read in the number of values the user wants to enter in bulk initially (store the number in `numContacts` [`numExpenses`, `numReadings`, etc., whichever suits your chosen option]), then…

    2. …call your **bulkAdd** method to handle that, and then…

    3. …show the user a menu.

Here is an *example*—with different menu options!—based on the idea of managing an address book for contacts (imagine that the class `Contact` is defined elsewhere):

```
Program: ContactsExample

Constants:
int CAPACITY, maximum number of objects, initialised to 50

Variables:
Scanner sc, to read user input, initialised when declared
Contact[] addressBook, contact list, initialised to hold CAPACITY elements
int numContacts, number of contacts in the collection
int choice, user's menu selection
String searchFor, a contact name to search for (just part of this example)

Steps:
1       Display "Contact Manager"
2       Prompt for "Enter initial number of entries (< " + CAPACITY + "): " and store in numContacts
3       Assign numContacts result of calling Math.min with numContacts, CAPACITY
            (That is, don't trust the user. Steps 2 & 3 may be combined)
4       Call bulkAdd with sc, addressBook, numContacts
            (bulkAdd will allow the user to enter numContacts entires, one after another)
5       Do
5-1     |   Display a menu "1. Add another contact
        |                   2. Display contacts
        |                   3. Find a contact
        |                   4. Quit"
5-2     |   Prompt user to "Enter option: "
5-3     |   Assign choice value of next int from sc (the Scanner)
5-4     |   Switch based on choice
5-4-1   |   |   In the case that choice is 1:
        |   |   |   Assign numContacts result of calling addContact with sc, addressBook, numContacts
5-4-2   |   |   In the case that choice is 2:
        |   |   |   Call printContacts with addressBook, numContacts
5-4-3   |   |   In the case that choice is 3:
        |   |   |   Prompt for "Which contact name? " and store in searchFor
        |   |   |   Display "Has contact: " + result of calling hasContact
        |   |   |       with addressBook, numContacts, searchFor
6       While choice is not 4
```

Tip: If you've completed CR Task 4.3 *User Input Functions* then consider copying *promptForInt* into your main program in order to display the menu and get the user's selection. A related tip: Use `\n` in a String to represent a newline character.

Notes on the following pages indicate the summary data you should calculate for your program (i.e., instead of a search function) based on the different options you may have chosen in PP Task 6.1.

Adding a new complex value to the array

In PP Task 5.2 you read a String from the user and called the **add** method to append it to the array: add's role was managing whether and where a simple value was added to the array, but not to interact with the user. In PP task 6.1 you implemented a **read*TypeName*** method to interact with the user and create (via its constructor) a more complex object. In the present task we suggest you copy (from your WordManager.java source file) and adapt add() to be similar to the following:

```
Method: int addContact(Scanner in, Contact[] addressBook, int numContacts)

Returns:
int, the numContacts of items in the address book; numContacts+1 if there was space to add

Parameters:
Scanner in, to read user input
Contact[] addressBook, the list of Contacts
int numContacts, the number of filled positions in the array

Steps:
1    If there is space in the addressBook (numContacts < addressBook.length) then
1-1  |   Assign addressBook[numContacts] the value of calling readContact with in
1-2  |   Increment numContacts by 1
     Else
1-3  |   Display "No more room in the address book"
2    Return numContacts
```

Note how this differs from your add() method in WordManager: instead of being given the value to add it accepts a Scanner to interact with the user. However, it then delegates that task to readContact(), which interacts with the user and creates the Contact object to store.

We would call **addContact** from main() as follows:

```
numContacts = addContact(sc, addressBook, numContacts);
```

which allows it to update the recorded *numContacts* of items if it actually added a value to the array.

Note: If your read*TypeName* method needs to use Scanner.nextLine() then, because it will always be reached after a call to nextInt() or similar, you will need an additional call to nextLine() to progress the Scanner past the end of the current line, ready to read the user's actual input. This work around looks like:

```
String someText; //will be a line of text

//Assume aScanner is some Scanner object, previously used for nextInt()
aScanner.nextLine(); //consume the \n after the user's last input
someText = aScanner.nextLine();
```

Tip: To test that your program correctly handles the array becoming full, temporarily reduce the array's `CAPACITY`.

Instructions for **each of the five options** are given below (remember you are to further develop whichever option you chose before):

1. Cost Calculator
2. Sensor Readings
3. High Scores
4. Casting Agent's Talent List
5. Make your own...

Option 1: Cost Calculator

Allow users to add Expense objects to the program, and use the following function to calculate the total of all of the expenses.

```
Method: int totalExpense(Expense[] expenses, int numExpenses)
```

```
Parameters:
Expense[] expenses, array of Expense objects
int numExpenses, current number of items in the array

Returns (result):
int, the sum of all expenses' costs

Variables:
int sum, the sum of expenses' costs, initialised to 0

Steps:
1     For int i is 0 to numExpenses - 1 (i.e., while i < numExpenses)
1-1   |   Assign sum the value of sum + the ith expense's cost
2     Return sum
```

Add a second method named **largestExpense** to find the largest expense.

Create a menu that allows the user to:

1. Add another expense
2. Display all expenses, one per line
3. Display total expense cost
4. Display largest expense
5. Quit

The program should loop until the user chooses to quit.

Option 2: Sensor Readings

Allow users to add Reading objects to the program, and use the following function to calculate the average of all of readings.

```
Method: double averageReading(Reading[] readings, int numReadings)

Parameters:
```

```
Reading[] readings, array of Reading objects
int numReadings, current number of items in the array

Returns (result):
double, the average of all readings' values

Variables:
double sum, the sum of readings' values, initialised to 0

Steps:
1     For int i is 0 to numReadings - 1 (i.e., while i < numReadings)
1-2   |   Assign sum the value of sum + the ith reading's value
2     If numReadings is larger than 0 then
2-1   |   Return sum / numReadings
      Else
2-2   |   Return sum (a default value if nothing to average)
```

Add a second method named **largestReading** to find the largest reading.

Create a menu that allows the user to:

1. Add another reading
2. Display all readings, one per line
3. Display average reading
4. Display largest reading
5. Quit

The program should loop until the user chooses to quit.

Option 3: High Scores

Allow users to add GameScore objects to the program, and use the following function to calculate the average score for a player.

```
Method: double averageUserScore(GameScore[] scores, int numScores, String name)

Parameters:
GameScore[] scores, array of GameScore objects
int numScores, current number of items in the array
String name, user name to filter scores

Returns (result):
double, the average score for user name

Variables:
double sum, the sum of name's scores, initialised to 0
int playerScores, number of game scores for name, initialised to 0

Steps:
1       For int i is 0 to numScores - 1 (i.e., i < numScores)
1-1     |   If the ith game score's userName equals name then
1-1-1   |   |   Assign sum the value of sum + the ith game scores's score
1-1-2   |   |   Increment playerScores by 1
2       If playerScores is larger than 0 then
2-1     |   Return sum / playerScores
        Else
2-2     |   Return sum (a default value if nothing to average)
```

Add a second method named **highestUserScore** to find the highest score for a particular user name.

Create a menu that allows the user to:

1. Add another score
2. Display all scores, one per line
3. Display average score for a user
4. Display highest score for a user
5. Quit

The program should loop until the user chooses to quit.

For 3 and 4, the user should be asked to enter the user name to check (place this code in main(), *not* inside averageUserScore or highestUserScore).

Option 4: Casting Agent's Talent List

Allow users to add Actors to the program, and use the following function to calculate the most valuable actor.

```
Method: Actor mostValuableActor(Actor[] actors, int numActors)

Parameters:
Actor[] actors, array of Actor objects
int numActors, current number of items in the array

Returns (result):
Actor, the actor with the highest revenue potential

Variables:
Actor highest, the most valuable actor

Steps:
1       Assign highest the revenue for the first element of actors
2       For int i is 1 to numActors - 1 (i.e., i < numActors)
2-1     |   If the ith actor's revenue is larger than highest's revenue then
2-1-1   |   |   Assign highest the revenue of the ith element of actors
3       Return highest
```

Add a second method named **averageActorRevenue** to find the average value of the recorded actors.

Create a menu that allows the user to:

1. Add another actor
2. Display all actors, one per line
3. Display most valuable actor
4. Display average actor revenue
5. Quit

The program should loop until the user chooses to quit.

Option 5: Make your own...

Use the above four examples to guide you to extend your own program. The functions should accept the array and possibly some other value and loop through the array to calculate some summary value (sum, count, average, min, max, etc.).

Add **two** methods to calculate values from the data you collected.