**Algorithms Updated 2024**

1. Introduction

An algorithm is a series of steps to solve a problem and must not be language-specific. The plan or steps can be coded in any language. That said, some standards are needed when representing algorithms, particularly in a teaching environment. These standards will be used in Information Technology examination papers (Paper 1 and Paper 2). Teachers must adhere to these standards in SBA Assessments, and candidates are expected to follow these standards when answering all papers and their PAT. Please read this document together with "**Class Diagrams Updated 2024**".

An algorithm can be represented in many ways, including flowcharts or pseudocode. This document focuses on pseudocode and its use to represent an algorithm.

**Note**: The prescribed pseudocode must be used instead of any Java or Delphi-specific methods. The Java or Delphi equivalent is provided to provide context and aid understanding.

2. Variables and Constants

2.1 Declaring Variables and Constants

In an algorithm, variables and constants **do not need** to be declared or typed since the variable type is language-dependent. For example, for the following programming code: String name (Java) and var name : string (Delphi) must **NOT** be included in pseudocode.

2.2 Naming Conventions

In pseudocode, **variables** (including other identifiers such as parameter names, object field names, and method names) should always start with a lowercase letter. If additional words are included, they should be capitalised, as in averageAge. **Constants**, which will always remain unchanged throughout a program, should be entirely capitalised, such as VAT. The undercore character separates words in constants, such as WARNING\_MESSAGE.

Where appropriate, use descriptive identifier names. Examples:

|  |  |
| --- | --- |
| **Variable Names** | **Data Stored** |
| sum, average, max | The sum, average and largest of a set of values. |
| i, j, k, loop, counter | Loops counters |
| size | Number of elements in an array |

|  |  |
| --- | --- |
| **Constant Names** | **Data Stored** |
| VAT, DISCOUNT, WARNING\_MESSAGE | The VAT is applied to all products, the discount offered on certain products, and the warning message (value of 2) icon in JOptionPane. |

|  |  |
| --- | --- |
| **Array Names** | **Data Stored** |
| strArr, intArr, numArray, friendArray | Arrays of strings, integers, numbers and Friend objects. It is recommended to include Arr or Array in the name. |

|  |  |
| --- | --- |
| **Method Names** | **Method Function** |
| sort, search, calculateAverage | Method names to sort, search and determine an average. |

2.3 Assignments and Arithmetic Operations

Although variables and constants are not declared in pseudocode, they must be named and assigned values. The standard symbol to indicate assignment, either at **initialisation** or as a result of an **arithmetic calculation**, is a left-pointing arrow (←).

**Note**: All pseudocode statements exclude a semicolon at the end of each statement.

2.3.1 Initialising Variables, Constants and Arrays

An integer variable: i ← 10 *//*i=10 *(Java)* i:=10*(Delhi)*

A string variable: name ← "Bob" *//Note " " for strings*

A character variable: old ← 'y' *//Note ' ' for characters*

A Boolean variable:flag ← false

A real variable: mass ← 25.63

A constant: VAT ← 15

An array literal: arr[ ] ← {5 ; 6 ; 4 ; 3}

An individual array element: graph[3] ← 15

**Note**: The first element of any array will always be position/index zero (0).

**Note**: For clarity in this document, additional explanations are included in comments and are prefixed with '//'.

2.3.2 Arithmetic Operations

The standard set of arithmetic operators is: + ; - ; \* ; / ; DIV; MOD; ^; sqrt

**Note**: Real division uses the **/** operator (/ in Delphi while Java overloads the / operator to perform both real and integer division depending on the value type it is dividing). **DIV** indicates integer division (div in Delphi and / in Java if both values on either side of the / sign are integers). **MOD** indicates the remainder after integer division (mod in Delphi and % in Java). Exponentiation is indicated with a **^** (power function in Delphi and Math.pow method in Java). Use the method **sqrt** to indicate the square root of a number.

2.3.2.1 Examples

age ← age + 10 *//Increasing age by 10*

multiply10 ← d \* 10

remainder ← d MOD 2 *//*d%2 *in Java*

sum ← sum + num *//Increasing sum by num*

x1 ← -b + (b^2 – 4\*a\*c)/(2\*a) *//*-b *would be coded as* -1\*b

hours ← minutes DIV 60

**Note**: The multiplication symbol '\*' must be written. For example, the mathematical term **2a** must be written as **2\*a**.

3. Input

Any data entered from the keyboard or through a GUI component must be prefixed with **input**.

Pseudocode for inputs does not require accompanying input messages.

**Note**: The GUI component name must not be mentioned, only the variable to which the data is assigned. For GUI selection components like radio buttons, there is no need to list options (e.g., "Male" or "Female"); only specify the storage variable (e.g., input gender).

3.1 Examples

input name

input password

input movieGenre *//User selecting a genre from a combo box*

4. Output

Any data displayed on the screen or in a GUI component must be prefixed with **display**.

4.1 Examples

display num

display "This text will appear as output"

display "He is " + age + " years old" *// + is used as a string concatenator*

display friendObject *//The object's toString will be called*

display friendObject.getName() *//Display the object's name field*

5. Objects

If an object is used in an algorithm, the object's methods must be referenced using dot notation. The constructor, accessor, mutator and toString methods do not need algorithms, as their function is well known. Any other methods should be previously described either in an algorithm or class diagram.

5.1 Instantiating an Object

Whilst we do not declare primitive types in an algorithm, objects must be instantiated using their constructor method. The object details should be previously provided.

**Note**: In the **class diagram**, the generic word 'Constructor' is used instead of the class name. Since an **algorithm** does not give the object type, the specific constructor name must be used to instantiate an object.

5.1.1 General format

objectName ← new ConstructorName (argument1, argument2,…)

5.1.2 Examples

friend ← new Friend("Jeremy",23)

5.2 Use of Objects in an Algorithm

All references to objects must be prefixed with the object's name. The object must be instantiated before being used unless its fields or methods are static. Static fields or methods which belong to the class will be called with the class name, e.g. Trees.getTotalArea().

5.2.1 Examples

numFriends ← friendArr.getSize() *//Public typed method*

newFriend ← friendArr.search("Tebogo") *//Public typed method*

name ← friend.getName() *//Public typed method*

display friend.toString() *//toString method can be left out*

friend.setAge(14) *//Public void method*

num ← Trees.getTotalArea() *//Public static typed method*

Trees.setTotalAreas(1400) *//Public static void method*

vat ← Invoice.VAT *//Public static constant field*

countPets ← Pets.total *//Public static field*

**Note**: Method calls are indicated by ().

5.3. String Manipulation

Functions that extract characters from a string or search for character(s) in a string pose difficulties due to programming language variations.

**Note**: The characters in a string are numbered from 0. For instance, in the string   
"Fried Tomatoes", the character 'F' is at index 0, 'r' is at index 1, 'i' is at index 2, and so on.

5.3.1 Length Example

The **length** method determines the number of characters in a string.

numChars ← length(stringVariable)

5.3.2 Extracting Characters

Write out a description of the method's function in words.

str ← "Fried tomatoes"

char1 ← character 2 in str *//The result will be '*i*'*

loop ← 3

char2 ← character loop in str *//The result will be '*e*'*

substr ← characters from 6 to 9 in str

*//The result would be "*toma*" from 6th character, including the 9th character*

*//In Java, the second argument will always be one more, e.g.* str.substring(6,**10**)

*//In Delphi, the second argument indicates the number of characters to be copied,*

*//e.g.* copy(str,6,**3**)

lastChar ← character length(str)-1 in str

*//The result is '*s*' - the last character in the string*

5.3.3 Searching for Character(s)

str ← "Fried tomatoes"

pos ← find position of 'i' in str *//The result is* 2

pos ← find position of "toe" in str *//The result is* 10

pos ← find position of "TOE" in str *//The result is* -1

Any other string manipulation methods must be similarly written.

5.4 DateTime Manipulation

5.4.1 Dates or DateTime Types

Portions of Dates or DateTime can be extracted using the NOW, DAY, MONTH and YEAR functions. The value NOW is used for the current date and DateTime. By convention, write these functions in capital letters. Since they are functions/methods, parentheses should be used when calling them, for example, YEAR(dateOfBirth).

5.4.2 Time or DateTime Types

Portions of time can be extracted with the HOUR, MINUTE and SECOND functions.

6. Arrays

Arrays need to be declared in an algorithm so the number of elements is known. Array elements are referenced using square brackets [ ]. Arrays are assumed to start at 0 and end 1 less than the declared size.

6.1 Declare Arrays

Contrary to the rule that algorithm variables do not have a type, a generic array type needs to be given in the declaration. Use the following data types in class diagrams: integer, real, string, character, and boolean (see **Class Diagram Updated 2024** **section** **2.2**). In some cases, the number of elements in the array may not be relevant to the algorithm.

6.1.1 General Format for Declaring Arrays

fieldName [ ] : type

fieldName [ ] : type [number of elements]

6.1.2 Examples

intArr [ ] : integer [10] *//The array will be numbered from 0 to 9*

markArr [ ] : real [20]  *//The array will be numbered from 0 to 19*

petName [ ] : string [30] *//The array will store up to 30 names of pets*

friendArr [ ] : Friend [50] *//The array will store up to 50 Friend objects*

someArray [ ] : character *//The number of elements is not relevant*

6.1.3 Array Literals

Arrays can be declared and instantiated in one assignment statement. The left arrow replaces the colon as the array elements are given values. No array type is included.

weekDaysArr [ ] ← {"Monday","Tuesday","Wednesday", Thursday","Friday"}

*//Declare and initialise an array where "*Monday*" is stored in*

*//weekDaysArr[0], "*Tuesday*" in weekDaysArr[1], and so on.*

*//The array will be numbered from 0 to 4*

h*o*urlyRatesArr [ ] ← {100,200,300,400}

*//Declare and initialise an array*

*//The array will be numbered from 0 to 3*

Thearrays weekDaysArr and hourlyRatesArr are array literals. See **Class Diagram Updated 2024 section** **4.4.2**.

data[ ] ← split line around # delimiter *//Splits the string in the line variable*

*//around the "#" delimiter and stores*

*//the data items in the* data *array.*

The data array is declared and assigned values by splitting a string using the '#' token.

7. Text Files

7.1 Reading from a Text File

The following algorithm reads multiple data items from a text file and instantiates an array of **Employee** objects.

open file "Employees.txt" for reading

size ← 0

while the file has data

begin

line ← read line from file

name, surname, dobStr ← extract data from line

dob ← convert dobStr in format yyyy/MM/dd to a Date object

empArr[size] ← new Employee(name, surname,dob)

size ← size + 1 *//*size++ *is also acceptable*

end while

close file

7.2 Writing to a Text File

In the following example, the employee names in the array **empArr** are written to a text file called "Employees.txt".

open file "Employees.txt" for writing

for i ← 0 to size – 1, inc by 1 *//Arrays are numbered from 0 to size-1*

begin

write empArr[i].getName() to file

end for

close file *//All "open file" commands must have a corresponding "close file"*

**Note**: To append to a text file, replace **for writing** with **for appending** to the open file line.  
open file "Employees.txt" for appending

8. Indenting Blocks of Code

All blocks of code are indented for readability, starting from the left margin, as in a program. The programming structures, conditions, loops, and methods start in an indented section. Such "inner blocks" are delimited with a vertically aligned **begin** and **end**. Indent any code between a **begin** and **end**. The end is followed by the structure name, such as end if, end while, end for, end case, and end method.

8.1 Examples

|  |  |
| --- | --- |
| **Single Structure Format** | **Single Structure Example** |
| structure  begin  line of code  line of code  line of code  end <structure> | if age < 21  begin  line of code  line of code  line of code  end if |

|  |  |
| --- | --- |
| **Nested Structure Format** | **Nested Structure Example** |
| structure1  begin  structure2  begin  line of code  line of code  line of code  end <structure2>  end <structure1> | for k ← 0 to size-1, inc by 1  begin  if arr[k].getGenre() = "Action"  begin  line of code  line of code  line of code  end if  end for |

9. Conditions with Relational and Logical Operators

The standard set of **relational operators** is: > ; < ; >= ; <= ; = ; ≠ (not equals).

The standard set of **logical operators** is NOT, AND, OR, which are written in upper case.

**Note**: The order of operations for logical operators is NOT first, then AND and lastly OR.

9.1 Examples

9.1.1 Relational Operators Examples

grade < 10 *//Conditions are utilised in conditional statements, e.g. if grade<10,*

sum > 25 *//and in loops, e.g. while grade< 10*

ave >= 50

age = 21 *//Use a single = sign instead of two equal signs (as seen in Java)*

gender = "Female" *//Use the = operator when checking whether strings are equal,*

*//instead of the equals method (as seen in Java)*

k ≠ 30

time ≠ 05:30:00 *//Any acceptable time format can be used*

9.1.2 Logical and Relational Operators Examples

NOT(name = "XYZ") *//Logical operator is in capitals*

name = "XYZ" AND y <= 3.56

year ≠ 2000 OR month < 4

date <= NOW()

name > "Fred"

DAY(dob) = 14 AND MONTH(dob) = 2

dob > 25/5/1990 AND NOT(name = "XYZ")

*//Any acceptable date format can be used*

dob > 25/5/1990 OR stored = true AND average <80 *//AND executed first*

(dob > 25/5/1990 OR stored = true) AND average <80 *//OR executed first*

*//OR is evaluated before AND because of parenthesis (brackets)*

When comparing string, Date, Time and DateTime types, use the relational operators   
>, <, >=, <=, = and ≠.

**Note**: Use parenthesis to change the order of operations for multiple conditions.

9.1.3 Conditional Examples

Relational and logical operators would generally be used in conditional or looping statements such as:

if x < 10

if k ≠ 30

while z < 15 AND y ≠ 10

while x < 5 OR k > 11

if sqrt(num) = round(sqrt(num)) *//Test if a number is a perfect square*

if num MOD 2 = 0 *//Test if a number is even.* if(num%2==0) *in Java*

if MONTH(dob) = 2

if YEAR(dob) < YEAR(25/5/1990)

if YEAR(dob) < YEAR(NOW())

10. Conditions and Loops

The main structures are:

**Conditions**:

* if …..then…..else statement
* switch / case structure

**Loops**:

* for …. loop
* while ….. loop
* do while ….loops / repeat….until loop

**Note**: It is important to specify the increment of the loop counter in a for….loop.

10.1 Examples

10.1.1 If Statements

if k < 10

begin

line of code

line of code

line of code

end if

**Pseudocode Java Equivalent**

if d > 15 *if (d > 15) //Condition does not need brackets*

begin *{ //Replace with* begin

line of code *instruction1; //Remove semi-colons*

line of code *instruction2;*

end *} //Replace with* end

else *else*

begin *{ //Replace with* begin

line of code *instruction3;*

line of code *instruction4;*

end if *} //Replace with* end<structure>

if d >= 15

begin

line of code

line of code

end

else

begin *//Include a* begin..end *for every* if

if x ≠ 12

begin

line of code

line of code

end if

end if

10.1.2 Switch/Case Statements

case age of

18 : line of code

20 : line of code

default : line of code

end case

case temperature of

36 :

begin *//If the option has multiple lines of code, a begin/end is needed*

line of code

line of code

end

40 : line of code

end case

*(This structure is the equivalent of a switch/case in Java)*

10.1.3 For Loops

for k ← 1 to 10, inc by 1

begin

line of code

line of code

line of code

end for

*//Looping through a string:*

for i ← 0 to length(stringVariable)-1, inc by 1  *//The characters in a string*

begin *//are numbered from 0,*

letter ← character i in stringVariable *//so loop from 0 to length-1.*

display letter

end for

*//Looping through elements in an array of objects:*

for j ← 0 to size-1, inc by 1 *//The variable* size *stores the number of objects.*

begin *//In the array, the first element has an index of 0,*

sum ← sum + arr[j].getPrice() *//second element has an index of 1, …,*

end for *//and the last element has an index of* ***size-1****.*

10.1.4 While Loops

while x < 10

begin

line of code

line of code

line of code

end while

*(This structure is equivalent in Java and Delphi)*

10.1.5 Do While Loops

A do …. while loop does not have a begin and end. The while statement indicates the end of the structure.

do

line of code

line of code

line of code

while x < 10

*(This structure is equivalent to a* repeat…until *in Delphi)*

11. Methods

A method's algorithm should include the method header with parameters and must correspond to the method call if one is provided.

11.1 Method Header Format:

**Note**: Method headers in pseudocode follow the same format as class diagram method headers, except they include the word **method** after the access modifier (i.e. public, private or protected).

A method header consists of the following parts:

* Access modifier - use the words public, private or protected
* Followed by the word **method**.
* Next, the **method name** is always followed by a pair of parenthesis whether there are parameters or not, e.g. public method calculateAverage( )
* The **formal parameters** inside the parenthesis of the method header must always be typed, e.g. method calculateAverage(intArr [ ] : integer)
* **Typed methods** (in Java and referred to as functions in Delphi) return a value and must include a corresponding return statement in the algorithm. The return type must be added at the end of the method header, preceded by a colon. For example,

method calculateAverage(intArr [ ] : integer) : real

* **Void methods** (in Java and referred to as procedures in Delphi) do not return a value. No return type should be added at the end of the method header. For example,

method createInitials(fullName : string)

When calling a method, it is not necessary to specify the **types** of the **arguments (actual parameters)**.

11.2 Method Header Examples

public method sortNames()

*//This method accepts no parameters*

public method addToArray(inPos : integer, inFriend : Friend)

*//This method accepts an integer and a Friend object as parameters*

private method createInitials(fullName : string)

*//This method accepts one string parameter*

public method calculateAverage(intArr [ ] : integer) : real

*//A typed method which accepts an array parameter and returns a real number*

11.3 Complete Method Examples

11.3.1 Void Method (Procedure)

The following method assumes that the variable size is one more than the last array element. For example, if the array elements are numbered from **0** to **6**, size will be **7**.

public method sortNames() *//The method returns no data*

begin

for i ← 0 to size – 2, inc by 1 *//Loop ends at* arr[5]

begin

for j ← i+1 to size - 1, inc by 1 *//Loop ends at* arr[6]

begin

if arr[i] > arr[j]

begin

temp ← arr[i]

arr[i] ← arr[j]

arr[j] ← temp

end if

end for (j) *// (j) added for clarity*

end for (i) *// (i) added for clarity*

end method *//The void method does not include a return statement*

11.3.2 Typed Method (Function)

private method findLargest(arr [ ] : integer) : integer

*//Method returns an integer\**

begin

hi ← arr[0] *//Always initialise the highest to the first item*

for i ← 1 to size – 1, inc by 1 *//The for loop starts at 1*

begin *//since arr[0] has been processed*

if arr[i] > hi

begin

hi ← arr[i]

end if

end for

return hi*//The typed method must include a return statement*

end method

11.4 Calling a Method

sortNames() *//Calling a void method/procedure*

addToArray (3, friend) *//Calling a void method/procedure*

max ← findLargest(arr [ ] : real)

*//Calling a typed method/function*

b ← calculateAverage(intArr [ ] : integer)

*//Calling a typed method/function*

When calling a method in another class, prefix the method call with the object's name if the method is non-static.

name ← friend.getName() *//Will use the friend object’s data in the method*

newFriend ← friendArr.search("Ed")*//Will use the friendArr array of objects in search*

friendArr.addToArray (3, friend) *//Calling a void method/procedure*

Calling static methods in another class will be called with the class name.

num ← Trees.getTotalArea() *//Public static typed method*

12. Inheritance

An algorithm in a subclass may refer to fields or methods in the superclass.

12.1 Constructor Method

A constructor method is neither typed nor void but must be named using the class name. All constructor methods will have the public access modifier.

12.1.1 Example of Constructor Method in Superclass

Assuming a superclass called SomeSuperClass has two private fields:

- f1 : integer

- f2 : Date

The pseudocode of its constructor method to assign values to the fields:

public method SomeSuperClass(inF1 : integer, inF2: string)

begin

f1 ← inF1

f2 ← convert inF2 in format yyyy/MM/dd to Date format

end method

12.1.2 Example of Constructor Method in Subclass

Assuming the subclass SomeSubClass inherits from SomeSuperClass with an additional real array field.

- f3 [] : real

examine the pseudocode of its constructor method:

public method SomeSubClass(inF1 : integer, inF2: Date,

inF3 [ ] : real)

begin

super(inF1, inF2)

f3 ← inF3

end method

**Note**: It is a common convention to name parameters with the corresponding field name prefixed with **in**.

12.2 toString Method in a Subclass

Call the superclass **toString** method in a subclass **toString** method, then add anything needed from the subclass.

public method toString(): string

begin

temp ← super.toString()

temp ← temp + "Extra string value"

return temp

end method

12.3 Method Overriding

Two methods are overridden if they have the same method name and the same parameters in inherited classes. The **toString** method in the subclass will automatically **override** the **toString** method in the superclass. Any other subclass method that overrides a superclass method must prefix the superclass method call with the word **super**.

The method defined in the superclass:

public method calculateCost() : real

begin

cost ← numStaff \* 300

return cost

end method

The overridden method in the subclass adds 75 to the value returned by the superclass method.

public method calculateCost() : real

begin

cost ← **super**.calculateCost() + 75

return cost

end method

It is unnecessary to indicate in an algorithm that one method overrides another (which is often shown in NetBeans).

12.4 Determine an Object Type and Casting

In pseudocode, use the words **"is a"** to determine an object's type (whether it is an object of the superclass or an object of the child class). In Java, the keyword instanceof is used, and in Delphi, the keyword is.

Place the desired type in parenthesis to convert/cast the object to the subclass type.

12.4.1 Example 1

The following example casts an object to call **someMethod** in the subclass.

if objectName **is a** SubClass

begin

temp ← (SubClass) objectName

display temp.someMethod()

end if

12.4.2 Example 2

The following example counts the number of objects that are not instances of the subclass:

count ← 0

if NOT(arr[i] **is a** SubClass)

begin

count++

end if

12.4.3 Example 3

In this example, a user wants to invite their friends to a party if they have visited the user at least 15 times. The superclass, called **Friend**, has two non-static fields to store the name and the number of times the friend has visited (numVisits) the user. The public static constant field MIN\_VISITS is set to 15 to record the minimum times a friend should visit the user. The subclass **BestFriend** inherits from **Friend** with an additional field to store the friend's email address needed to send the party invite.

|  |
| --- |
| **Friend** |
| - name : string  - numVisits : integer  + MIN\_VISITS = 15 : integer |
| + Constructor(inN : string,  inNV : integer)  + getName() : string  + getNumVisits() : integer  + toString() : string |

|  |
| --- |
| **FriendManager** |
| - fArr [ ] : **Friend** [200]  - size : integer |
| + Constructor()  + partyInvite()  + toString() : string |

|  |
| --- |
| **BestFriend** |
| - email : string; |
| + Constructor(inN : string,  inNV : integer,  inE : string)  + emailPartyInvite();  + toString() : string |

The **FriendManager** class contains a field to store an array of 200 Friend objects called fArr, and the size field records the number of elements in the array. The **FriendManager** class has an array of **Friend** objects. The **FriendManager** is the composed class, and the **Friend** is the component class. See **Class Diagrams Updated 2024 section 6.2**.

In the **partyInvite** method in the **FriendManager** class, a **Friend** array object is converted to a **BestFriend** object if the friend has visited at least 15 times (NUM\_VISITS) then the friend is emailed a party invite. If the **Friend** is a **BestFriend** but has not visited 15 times, they are converted to a **Friend** object. If the **Friend** is a **BestFriend** and has visited at least 15 times, they are emailed a party invite.

Using the class diagrams, the algorithm to invite a friend to the party if they are a **BestFriend** and have visited at least 15 times is as follows:

public method partyInvite()

begin

for i ← 0 to size – 1, inc by 1

begin

*//Check if the array element is a BestFriend*

if fArr[i] is a BestFriend

begin

*// Check to see if they have visited 15 times or more*

if fArr[i].getNumVisits() >= Friend.MIN\_VISITS

begin

*//Send an invite*

fArr[i].emailPartyInvite()

end

else

begin

*//Change BestFriend to a Friend as they didn’t visit often enough*

fArr[i] ← new Friend(fArr[i].getName(),

fArr[i].getNumVisits())

end

end

else

*//Otherwise the array element is a non-BestFriend (i.e, a friend)*

begin

*//Test to see if the Friend has visited 15 times or more*

if fArr[i]. getNumVisits() >= Friend.MIN\_VISITS

begin

*//Change to a BestFriend*

fArr[i] ← new BestFriend(fArr[i].getName(),

fArr[i].getNumVisits(),

"email@gmail.com")

*//Send an invite to the new BestFriend*

fArr[i].emailPartyInvite()

end if

end if

end for

end method

**Note**:

The above algorithm first tests if the fArr[i] object is an instance of the subclass **BestFriend** in the if part of the outer if statement and assumes that fArr[i] is **not** an instance of the subclass **BestFriend** in the else part (i.e., a regular friend).

Testing if the array element **fArr[i]** is a **Friend** type in the if part of the statement will NOT be correct. Since all objects in the **fArr** array are declared as instances of the **Friend** superclass from the **Friend** array declaration fArr [ ] : Friend [200] in the **FriendManager** class. This check will always succeed, and the else part will never be executed.

The pseudocode to first test for a regular friend (not a best friend) without returning all the array elements is as follows:

|  |  |
| --- | --- |
| **INCORRECT PSEUDOCODE** | **CORRECT PSEUDOCODE** |
| if fArr[i] is a Friend  *//Check if array element is a Friend*  else  *//Check if the array element is a BestFriend* | if **NOT**(fArr[i] is a **BestFriend**)  *//Check if array element is a non-BestFriend*  else  *//Check if the array element is a BestFriend* |

13 Further Examples

13.1 Example 1

The following algorithm adds real array values below 60 and determines the average of the added values. The value 60 is a private static field called **minValue** in a class called **MyNumbers**.

size ← 10

sum ← 0

count ← 0

for k ← 0 to size – 1, inc by 1

begin

if realArray[k] < MyNumbers.getMinValue()

*//The accessor method of the private field minValue is used*

begin

sum ← sum + realarray[k]

count ++

end if

end for

average ← sum / count

display average

13.2 Example 2

The following typed method/function counts the number of identical adjacent letters in a word.

public method countRepeatingLetters(word : string) : integer

begin

count ← 0  *//Normally, the upper limit of the for loop would be 'length(str) - 1'.*

*//However, since the last letter will not be checked against*

*//an adjacent letter, the upper limit is length(str) – 2:*

for i ← 0 to length(str) – 2 in str, inc by 1

begin

if character i in str = character i+1 in str

begin

count++

end if

end for

return count

end method