Programming Design and Implementation

Lecture 4: Repetition

Updated: 18th March, 2020

Mark Upston
Discipline of Computing
School of Electrical Engineering, Computing and Mathematical Sciences (EECMS)

Copyright © 2020, Curtin University
CRICOS Provide Code: 00301J

Outline

WHILE Loops

Loop Issues

DO-WHILE Loops

FOR Loops

Nesting Loops

“Zero” Marks

- A student who does any of the following in a submitted, assessable answer will receive heavy penalties, up to and including **zero** marks for that question:
  - Uses **`continue`**
  - Uses **`break`** in any other place than a **`switch`** statement
  - Uses **`goto`**
  - Has more than one **`return`** statement in a method
  - Has a **`return`** statement in a method anywhere but the last statement of the method
  - Uses **`System.exit()`** anywhere but the last statement of the **`main()`** method
  - Uses global variables for anything other than class fields
  - Uses a ternary operator
- Note: similar efforts in pseudo code will also receive zero marks

Repetition AKA Looping

- In programming terms, a loop is a section of code that is repeated a number of times (0 to many)
- Three types of loops are available:

  The difference is how the repetition is controlled

  - **`WHILE`**:
    - Execute zero or more times
  - **`DO-WHILE`**:
    - Execute one or more times
  - **`FOR`**:
    - Execute a fixed number of times
- You choose the appropriate loop based on the above

WHILE Loop

Pseudo Code:

```
WHILE boolExpression DO
    Body of loop
ENDWHILE
ASSERTION: boolExpression is false
```

Java:

```
while(boolExpression)
{
    statements;
} // boolExpression is false
```

- Repetition is controlled by a logical expression at the top of the loop
  - If the boolean expression is **true** the body of the loop is executed

WHILE Loop (2)

- The expression is repeatedly checked before the first statement is executed (again)
    - If the boolean expression is false then program execution jumps to the first statement after the body of the loop
    - If the boolean expression is false the very first time it is encountered then statements in the loop will never execute
- Generic properties:
    - The fact that it is a `WHILE` loop is clearly stated
    - The boolean expression is clear
    - Which statements are contained within the loop are clear

WHILE Loop - Menu Example

Pseudo Code:

```
close := FALSE
WHILE NOT close DO
   OUTPUT "Enter Choice"
   INPUT choice
   CASE choice OF
      a OR A
         doAStuff <- none
      b OR B
         doBStuff <- none
      e OR E
         close := TRUE
      DEFUALT
         OUTPUT "Invalid Choice"
   ENDCASE
ENDWHILE
ASSERTION: close is true
```

Java:

```
char choice;
close = false;
while(!close)
{
  System.out.println("Enter Choice");
  choice = charInput();
  switch(choice)
  {
    case 'a': case 'A':
      doAStuff();
    break;
    case 'b': case 'B':
      doBStuff();
    break;
    case 'e': case 'E':
      close = true;
    break;
    default:
      System.out.
          println("Invalid Choice");
  }
}
```

Infinite Loop

- Must ensure that the logic of a while loop will allow it to finish
- A loop which cannot finish is called an Infinite Loop
- Three major causes:
  - Logical expression can never be **`false`**
  - None of the statements in the loop will cause the logical expression change to **`false`**
  - The insertion of a semi-colon in the wrong place has caused the body of the loop to fall outside the loop
- Good use of assertion statements means that:
  - Infinite Loops will almost never occur within your algorithm
  - Infinite Loops will usually only occur in your code because of typographical errors

Logical Error

Logical Expression can never be **false**

```
x := 0
WHILE x NOT EQUAL TO 11 DO
  OUTPUT x
  INCREMENT x BY 2
ENDWHILE
ASSERTION: x is equal to 11
```

Should be:

```
x := 0
WHILE x < 11 DO
  OUTPUT x
  INCREMENT x BY 2
ENDWHILE
ASSERTION: x >= 11
```

Logical Expression Not Affected by a Loop

- None of the statements in the loop will cause the logical expression to change to **`false`**

```
INPUT x
WHILE x < 0 OR x > 10 DO
  OUTPUT "Invalid Input"
ENDWHILE
ASSERTION: 0 <= x <= 10
```

```
x := 0
WHILE x < 11 DO
  OUTPUT x
ENDWHILE
ASSERTION: x >= 11
```

Corrected:

```
INPUT x
WHILE x < 0 OR x > 10 DO
  OUTPUT "Invalid Input"
  INPUT x
ENDWHILE
ASSERTION: 0 <= x <= 10
```

Corrected:

```
x := 0
WHILE x < 11 DO
  OUTPUT x
  INCREMENT x BY 2
ENDWHILE
ASSERTION: x >= 11
```

Typographical Error in Java

- The insertion of a semi colon in the wrong place has caused the body of the loop to fall outside of the loop

```java
evensSum = 0;
nextNo = 0;
while(nextNo <= 100);
{
    evensSum = evensSum + nextNo;
    nextNo += 2; // add two to nextNo
} // Assertion: nextNo > 100
System.out.println(evensSum);
```

- The loop ends after the semi-colon following the while loop i.e., there are no statements in the loop. The boolean expression will just continually be checked.

DO-WHILE

Pseudo Code:

```
DO
    Body of loop
WHILE boolExpression

ASSERTION: boolExpression is false
```

Java:

```
do
{
    statements;
} while(boolExpression);
// boolExpression is false
```

- Repetition is controlled by a logical expression at the bottom of the loop
    - If the boolean expression is **true** the body of the loop is executed

DO-WHILE (2)

- The expression is repeatedly checked after the last statement is executed
  - If the boolean expression is false then program execution jumps to the first statement after the body of the loop
  - The loop must be executed at least once because the logical expression is evaluated at the bottom of the loop
- Generic Properties
  - The fact that it is a `DO-WHILE` loop is clearly stated
  - The boolean expression is clear
  - Which statements are contained within the loop are clear

Example: Algorithm

```
DO
    INPUT age
WHILE age <= 0 OR age >= 110
ASSERTION: 0 < age < 110
```

- What is potentially wrong with this algorithm?

Example: Java

```java
int age;
Scanner sc = new Scanner(System.in);

do
{
    System.out.println("Enter Age");
    age = sc.nextInt();
} while((age <= 0) || (age >= 110));
// Assertion: 0 < age < 110
```

- Nothing is wrong with the logic, however, it does not give the user an indication of what went wrong

Example: Solved

- In your programs use the following template:
    - Note: We will modify it slightly when we cover submodules, and again when we cover Exceptions

```
prompt := "Enter value between " lower " and " upper
outputPrompt := prompt
DO
    OUTPUT outputPrompt + lower + upper
    INPUT num
    outputPrompt := "Error please enter an number
                     in the valid range" + newline
                     + prompt
WHILE((num < lower) OR (num > upper))
ASSERTION: lower <= value <= upper
```

sqrt - Pseudo

```
squareRoot := number / 2.0
COMMENT: First "guess"

DO
    t := squareRoot
    COMMENT: Next guess will be closer

    squareRoot := (t + (number / t)) / 2.0 (REALS)
WHILE((t - squareRoot) IS NOT 0.0)
ASSERTION: When (t - squareRoot) is 0, we cannot get
           any closer
```

- How do we check for 0.0 here?

sqrt - Java

```
double t;
double squareRoot = number / 2.0;

do
{
    t = squareRoot;
    squareRoot = (t + (number / t)) / 2.0;
} while(Math.abs(t - squareRoot) > 0.0000000001);
```

Loop Equivalency

- A WHILE loop can be expressed as a DO-WHILE loop

```
WHILE x < 10 DO
    INCREMENT x BY 2
ENDWHILE
ASSERTION: x >= 10
```

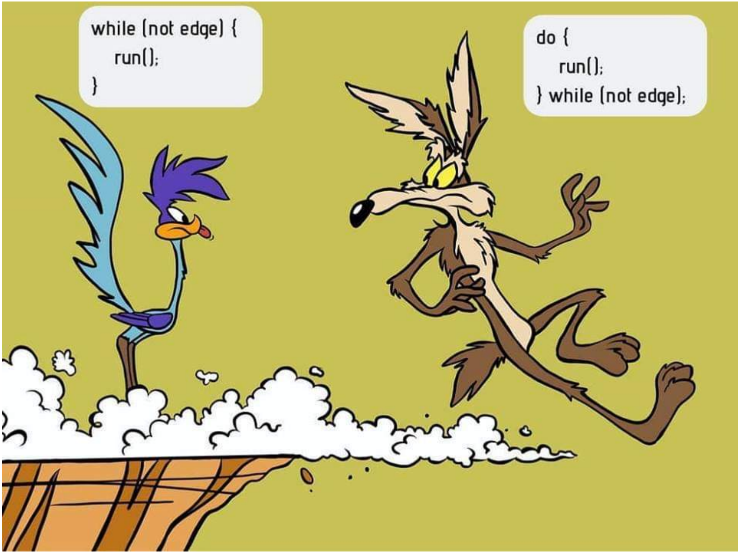
```
IF x < 10 THEN
    DO
        INCREMENT x BY 2
    WHILE x < 10
    ASSERTION: x >= 10
ENDIF
ASSERTION: x >= 10
```

- A DO-WHILE loop can be expressed as a WHILE loop

```
DO
    INCREMENT x BY 2
WHILE x < 10
ASSERTION: x >= 10
```

```
INCREMENT x BY 2
WHILE x < 10 DO
    INCREMENT x BY 2
ENDWHILE
ASSERTION: x >= 10
```

Think before you design



FOR Loop

- Consider the WHILE loop below:

```
count := 1
WHILE count <= 10 DO
    OTHER_ACTIONS
    INCREMENT count BY 1
ENDWHILE
```

- This loop will repeat **exactly** 10 times
- The first time through the loop, **count** will be 1
- The contents of the **count** variable will increase by 1 at the end of each iteration
- The final time through the loop **count** will be 10, until it reaches the last statement "**INCREMENT count BY 1**" which will terminate the loop

FOR Loop (2)

- A more general form of this WHILE loop would be:

```
count := startVal
WHILE count <= stopVal DO
    OTHER_ACTIONS
    INCREMENT count BY incVal
ENDWHILE
```

- The number of times the loop will iterate is:

$$\frac{\texttt{stopVal} - \texttt{startVal}}{\texttt{incVal}} + 1$$

- The first time through the loop **count** will be initialised to **startVal**
- The contents of **count** will increase by **incVal** after each iteration
- The last time through the loop **count** will be set to **stopVal**

FOR Loop (3)

- This kind of a loop is extremely useful
- It is a special case of the WHILE loop
- Pseudo Code:

```
FOR count := startVal TO stopVal CHANGEBY increment
    OTHER_ACTIONS
ENDFOR
```

- The variable **increment** can positive or negative
- **count** is known as the FOR loop **index**
- Generic Properties:
  - It should be clear that it is a FOR loop
  - FOR loop index should be clear
  - Start, end and increment values for the index should be clear
  - The statements contained within the loop should be clearly indicated

Properties of a FOR Loop

- Loop index should always be a local variable
- Loop index should never be a Real number (i.e., must always be a discrete value)
- Loop index should never be explicitly modified inside the loop
- Number of iterations is: $\frac{\texttt{stopVal} - \texttt{startVal}}{\texttt{incVal}} + 1$
  - Note: A zero or negative value above, implies the loop will not iterate
- The value of the loop index is undefined outside of the loop
- For loop is never executed if:
  - Positive increment and **stopVal** $<$ **startVal**
  - Negative increment and **startVal** $<$ **stopVal**

FOR Loops in Java

- Java inherits its FOR loop syntax from C
    - This means that the Java compiler will hardly ever tell you if you are doing something inappropriate with a FOR loop
- Syntax:

```
for(initialisation; booleanExpression; increment)
{
    body_of_loop;
}
```

- Example:

```
sum = 0;
for(count = 1; count <= 10; count++)
{
    sum += count;
}
```

Declaring Loop Indexes

- Normally good programming practice says declare all local variables at the start of the method block
- A loop index is the one exception because it should never be referred to outside of the for loop
- Java allows us to declare our variables anywhere so:

```
int sum = 0;
for(int count = 1; count <= 10; count++)
{
    sum += count;
}
```

- This means an attempt to refer to the loop index outside of the for loop will most likely incur a compiler error

Index as a Real

```
int x = 1;
for(double ii = 0.0; ii < 1.0; ii += 0.1)
{
    System.out.println(ii + ” and x is ” + x++);
}
```

- How many times should this loop iterate?
- 0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9 -
- Finish here because 1.0 is not less than 1.0, thus 10 times

Index as a Real: Actual Output

```
0.0 and x is 1
0.1 and x is 2
0.2 and x is 3
0.30000000000000004 and x is 4
0.4 and x is 5
0.5 and x is 6
0.6 and x is 7
0.7 and x is 8
0.79999999999999999 and x is 9
0.89999999999999999 and x is 10
0.99999999999999999 and x is 11
```

- This is **11** times! Do **not** use Real numbers as an index in a FOR loop

Abuse of FOR Loops in Java

- There are very few constraints as to what can be done in Java’s **for()** loops
  - Here are some examples of poor coding:

```java
for(int sum = 0, int count = 1; count <= 10; sum += count++);
```

  - What does this do?

```java
for(z = 2.45, p = 3.5, r = -4; frog != kermit; s -= 3.5, t += 4)
{
    ...
}
```

Abuse of Loops

- There are three statements that *can* be used in loops that you should **never** use:
  - **break** exit loop
  - **continue** skip to next iteration of the loop
  - **goto** go to LABEL (but not in Java)

```
for( ; ; )                    // Infinite Loop
{
    ...
    if(cond1) continue;       // Start the next Iteration
    else if(cond2) break;     // Exit the loop now
    else if(cond3) goto FRED; // Go to label FRED (Somewhere)
    ...                       // Neither cond1 or cond2 true
}
```

- It is very typical for a programming language to allow for poor algorithm design and programming style
- You are **much** cleverer than a compiler, **never** rely on the compiler for deciding what is or isn't best practice

FOR Loop Example: Algorithm

```
FOR index = 0 TO myArray LENGTH CHANGEBY 1
    OUTPUT myArray[index]
ENDFOR
```

FOR Loop Example: Java

```
/**************************************************
 * ASSERTION: Contents of array output to screen   *
 **************************************************/
for(int ii = 0; ii < myArray.length; ii++)
{
    System.out.println(myArray[ii]);
}
```

FOR Loop Example (2): Algorithm

```
ASSERTION: if n is 0 or negative, then nFactorial is 1

ALGORITHM:
    nFactorial := 1
    FOR ii := 2 TO n CHANGEBY 1
        nFactorial := nFactorial * ii
    ENDFOR

ALTERNATE ALGORITHM:
    nFactorial := 1
    FOR ii := n DOWNTO 2 CHANGEBY -1
        nFactorial := nFactorial * ii
    ENDFOR
```

FOR Loop Example (2): Java

```
/*******************************************************
 * ASSERTION: if n 0 or negative, then nFactorial is 1 *
 *******************************************************/

long nFactorial = 1;
for(int ii = 2; ii <= n; ii++)
{
    nFactorial *= (long)ii;
}

// ------------------------------------------------------

long nFactorial = 1;
for(int ii = n; ii >= 2; ii--)
{
    nFactorial *= (long)ii;
}
```

FOR Loop Example (3): Algorithm

```
ALGORITHM:
    nChooseR := n! / ((n - r)! * r!)
```

- We will need 3 FOR loops, one for each factorial calculated
  - We will simplify this more when we cover Submodules (next Lecture)

FOR Loop Example (3): Java

```
int nChooseR;
long bottom;
long nMinusRFact = 1;
long rFactorial = 1;
long nFactorial = 1;

for(int ii = 2; ii <= (n - r); ii++)
{
    nMinusRFact *= (long)ii;
}
for(int ii = 2; ii <= r; ii++)
{
    rFactorial *= (long)ii;
}
for(int ii = 2; ii <= n; ii++)
{
    nFactorial *= (long)ii;
}
bottom = nMinusRFact * rFactorial;
nChooseR = (int) (nFactorial / bottom);
```

Nesting Loops

- Any control structure can be nested inside any other control structure
  - `IF-THEN-ELSE` inside loop, loop inside loop
- Must be careful of algorithm efficiency
- Nesting one loop inside another increases the number of processing steps at an exponential rate
- Good use of indentation is essential

Nested Loop Example

- Write an algorithm that will input a number between 1 and 12 (inclusive) from the user, then output all of the times tables (1 to 12) between 1 and the input number
    - i.e., If the user inputs 3, the program will output:

```
Enter a number in the range 1 to 12: 3
The 1 Times Table
1 x 1 = 1
...
1 x 12 = 12
The 2 Times Table
2 x 1 = 2
...
2 x 12 = 24
The 3 Times Table
3 x 1 = 3
...
3 x 12 = 36
```

Nested Loop Example: Algorithm

```
outputPrompt := "Enter a number in the range 1 to 12: "
DO
    OUTPUT outputPrompt
    INPUT num
    outputPrompt := "Error: Please enter a number in the range
                     1 to 12 only"
WHILE(num < 1) OR (num > 12)
ASSERTION: num in the range 1 to 12 inclusive

FOR table := 1 TO num CHANGEBY 1
    OUTPUT "The " table " Times Table"
    FOR number := 1 TO 12 CHANGEBY 1
        OUPTUT table " x " number " = " (table * number)
    ENDFOR
    ASSERTION: table Times Table is output to the user
ENDFOR
ASSERTION: one to n Times Table is output to the user
```

Nested Loop Example: Java

```java
int num;
Scanner sc = new Scanner(System.in);
String outputPrompt = "Enter a number in the range 1 to 12: "
do
{
    System.out.print(outputPrompt);
    value = sc.nextInt();
    outputPrompt = "Error: Please enter a number in the range
                   1 to 12 only: ";
} while((value < lower) || (value > upper);
for(int table = 1; table <= n; table++)
{
    System.out.println("The " + table + " Times Table");
    for(int number = 1; number <= 12; number++)
    {
        System.out.println(table + " x " + number + " = "
                           + (table + number));
    }
}
// Comments have been omitted to save space.
```

Nested Loops and Algorithm Complexity

- Algorithm Complexity is an indication of the efficiency of an algorithm
- It attempts to show the rate of increase in processing steps as a function of the amount of data being processed
- Algorithm complexity will be covered in much more detail in DSA (`COMP1002`)
- For the moment consider the previous example where two `FOR` loops were nested

Algorithm Complexity

```
1 for(int table = 1; table <= n; table++)
2 {
3     System.out.println("The " + table + " Times Table");
4     for(int number = 1; number <= 12; number++)
5     {
6         System.out.println(table + " x " + number + " = "
7                               + (table * number));
8     }
9 }
```

- When the user inputs 6, how many times did the statement **`System.out.println(table + ... );`** (*Line 6*) execute?
- How about when the user inputs 12?

Next Week

- The next Lecture will cover:
    - Submodules