Fundamentals of Programming 2

Lecture 3

Aurelia Power, TU Dublin – Blanchardstown Campus, 2019

* Notes based on the Java Oracle Tutorials, Deitel & Deitel, and Horstman

If-Else: Possible Problems

- Java allows you to associate one single statement to be executed with an if or an else block **without enclosing them inside a pair of { }**
- Consider the following block of code:

```
if(a >= 5)
        System.out.println(a + " >= 5");
else
        System.out.println(a + "  < 5");
```

This will be executed if the value of a is greater or equal to 5, for example, if a is 7

This will be executed if the value of a is less than 5, for example, if a is 3

- But if you have more than one statements to be associated with the **if** clause, you must use the **{ }**, otherwise one of 2 problems is going to occur:

1. If you have only the if clause, but you want to execute/not execute 2 or more statements conditionally, then, if the condition is false, only the first subsequent statement will be skipped, not all of them:

X

```
if(a >= 5)
        System.out.println(a + " >= 5");
        System.out.println(a + " > 3");
```

What will be the output if a is 2?

If a is 2, only this statement will be skipped , but not the second one

This will be executed, so it will output: 2 > 3, which is not true…

If-Else: Possible Problems

2. The code will NOT compile, if we also have an else following, because you cannot place any other statement between the if block and the else block;

X

```
if(a >= 5)
        System.out.println(a + " >= 5");
        System.out.println(a + " > 3");
else
        System.out.println(a + " < 5");
```

This line will NOT compile: we can only have 1 statement associated with an **if,** if there is a subsequent else and you choose not use the braces.

- To address the 2 issues, we must enclose the 2 statements inside the **{ }**

√

```
if(a >= 5) {
        System.out.println(a + " >= 5");
        System.out.println(a + " > 3");
}
else{
        System.out.println(a + " < 5");
}
```

So what will be the output now if a is 2?

Dangling Else Problem and nesting of if/else statements

- So…Java always associates an **else** with the **previous if** unless told otherwise by use of the **{ }** braces:

EXAMPLE – assume that we want to output x <= 7 if x is smaller than 7, irrespective of the value of y;

```
if  ( x > 7 )
      if  ( y > 7 )
            System.out.println(  "x and y are > 7" );
else
      System.out.println(  "x  <= 7" );
```

The else clause is also dependent on whether y is > than 7; but we don't want this… so to fix it, we use **{ }** to specify which if the else will be associated with

What will it output if x is 5 and y is 10?? Nothing, but we want it to output x <= 7
What will it output if x is 10 and y is 5?? x <= 7 – which is not true

---

```
if  ( x > 7 )    {
      if  ( y > 7 )
            System.out.println(  "x and y are > 7" );
}else{
            System.out.println(  "x <= 7" );
}
```

- Now… What will it output if x is 5 and y is 10??

x <= 7 – which is true

- What will it output if x is 10 and y is 5??

Nothing which is what we want

An Example

```java
/**
 * demonstrates how nested if works by applying it to a guessing game
 */
import java.util.Scanner;
/**
 * @author Aurelia Power
 *
 */
public class NestedIfElseDemo {
    public static void main(String[] args) {
        int correctAnswer = 7;
        Scanner sc = new Scanner(System.in);

        //prompt user to input his guess
        System.out.print("Guess a number between 1 and 10: ");
        int userGuess = sc.nextInt();

        // if the guess is the same as the correct answer praise the user
        if(userGuess == correctAnswer){
            System.out.println("Well done... you are correct :)");
        }else{
            //otherwise, print a message saying that the guess is wrong
            System.out.println("your guess is wrong :(");
            //in addition, output whether the guess was too high
            if(userGuess > correctAnswer){
                System.out.println("your guess was too high");
            }else{ // or whether the guess was too low
                System.out.println("your guess was too low");
            }//end inner else
        }//end outer else
    }//end main method
}
```

Possible Outputs:

- **When the user guess is smaller than the correct answer, for instance, 1**

Guess a number between 1 and 10: 1
your guess is wrong :(
your guess was too low

- **When the user guess is the same as the correct answer, thus, 7**

Guess a number between 1 and 10: 7
Well done... you are correct :)

- **When the user guess is bigger than the correct answer, for instance, 10**

Guess a number between 1 and 10: 10
your guess is wrong :(
your guess was too high

The morale… always use { } until experienced enough

Nesting if/else can be re-written as a series of if/if-else/else…

```
public static void main(String[] args) {
    int correctAnswer = 7;
    Scanner sc = new Scanner(System.in);

    //prompt user to input his guess
    System.out.print("Guess a number between 1 and 10: ");
    int userGuess = sc.nextInt();

    // if the guess is the same as the correct answer praise the user
    if(userGuess == correctAnswer){
        System.out.println("Well done... you are correct :)");
    }else if(userGuess > correctAnswer){
        //print a message saying that the guess is wrong
        System.out.println("your guess is wrong :(");
        //in addition, output whether the guess was too high
        System.out.println("your guess was too high");
    }else{
        //again, print a message saying that the guess is wrong
        System.out.println("your guess is wrong :(");
        //in addition, output whether the guess was too low
        System.out.println("your guess was too low");
    }//end  else

    // close the Scanner object
    sc.close();
}//end main method
```

EXERCISE:

- Choose your own name for the class
- Run the program with at least 5 values

NOTE: it is recommended to close the Scanner object invoking the method close, once you have finished taking input, to avoid leaking resources issues:

`sc.close();`

Furthermore… Nesting if/else can be re-written as a series of ifs…

```
public static void main(String[] args) {
    int correctAnswer = 7;
    Scanner sc = new Scanner(System.in);

    //prompt user to input his guess
    System.out.print("Guess a number between 1 and 10: ");
    int userGuess = sc.nextInt();

    // if the guess is the same as the correct answer praise the user
    if(userGuess == correctAnswer){
        System.out.println("Well done... you are correct :)");
    }
    if(userGuess > correctAnswer){
        //print a message saying that the guess is wrong
        System.out.println("your guess is wrong :(");
        //in addition, output whether the guess was too high
        System.out.println("your guess was too high");
    }
    if(userGuess < correctAnswer){
        //again, print a message saying that the guess is wrong
        System.out.println("your guess is wrong :(");
        //in addition, output whether the guess was too low
        System.out.println("your guess was too low");
    }//end  else

    // close the Scanner object
    sc.close();
}//end main method
```

EXERCISE:

- Choose your own name for the class
- Run the program with at least 5 values

NOTE:

- Any selection construct, no matter how complex, can be written using only carefully-written basic single selection, that is, only if statements.
- So any if/else (nested or otherwise) can be re-written using ifs only

Your turn…

- What is the **output** of the following blocks of code for the following values of input: **happy**, **sad**, **angry**.

```
if( input.equals("happy"))
  System.out.println("☺");
else
  System.out.println("☹");
```

```
if(!input.equals("happy"))
  System.out.println("☺");
else
  System.out.println("☹");
```

```
if( input.equals("Sad"))
  System.out.println("☺");
else
  System.out.println("☹");
```

```
if( input.equals("happy"))
  System.out.println("☺");
  System.out.println("☺");
else
  System.out.println("☹");
```

```
if( !input.equals("happy"))
  System.out.println("☺");
else
  System.out.println("☹");
  System.out.println("☺");
```

```
if( input.equals("angry"))
  System.out.println("😠");
else{
  System.out.println("☹");
  System.out.println("☺");
}
```

The 'switch' statement: Multiple Selection

- Another construct that we can use when we have multiple choices, is the **switch** statement.
- However, we can only use it when:
  - only one variable is being checked in each condition
  - the check involves specific values of that variable (e.g. 'A', 'B') and not ranges (e.g. >39) ;

- EXAMPLE: year 1 is divided into 6 groups, and I want to know the day and time when a specific group has the FOP2 lab;
  - If you are in group 1, you are on Monday from 9 – 11
  - If you are in group 2, you are on Monday from 11 – 13
  - If you are in group 3, you are on Monday from 14 – 16
  - If you are in group 4, you are on Monday from 16 – 18
  - If you are in group 5, you are on Tuesday from 9 – 11
  - If you are in group 6, you are on Tuesday from 11 – 13

NOTE: this are not your actual lab times!!! ☺

```java
import java.util.Scanner;
/**
 * demonstrates how to use the switch construct to model the lab times problem
 */
/**
 * @author aura power
 */
public class SwitchDemo {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        //prompt the user to enter group number and store that input into a variable
        System.out.print("Enter group number: ");
        int groupNumber = sc.nextInt();
        /* check the groupNumber against the listed cases; for each case there is
           a different message to output; when a match is found, the corresponding
           statement will be executed and then the switch block will be exited */
        switch(groupNumber) {
            case 1: System.out.println("Monday from 9 – 11"); break;
            case 2: System.out.println("Monday from 11 - 13"); break;
            case 3: System.out.println("Monday from 14 – 16"); break;
            case 4: System.out.println("Monday from 16 – 18"); break;
            case 5: System.out.println("Tuesday from 9 – 11"); break;
            case 6: System.out.println("Tuesday from 11 – 13"); break;
            default: System.out.println("No such group!!");
        } // end switch
        // close the input stream
        sc.close();
    } // end main
} // end class
```

Switch Statement

- The **switch structure** contains a series of case labels, such as possible group numbers (as int), and an optional default case, for when the variable checked does not match any of the cases.

- The only **data types** that can be checked in a switch are:
  - byte, short, char, and int primitive data types.
  - String
  - It also works with *enumerated types* (not covered here).
  - A few special classes that wrap certain primitive types: Character, Byte, Short, and Integer (not covered yet).

- The **break** statement causes the program control to exit the switch structure, and continue with what is after the switch block, if anything.

- Advantages of the switch
  - it is clear and elegant
  - that all its branches will test the same variable …in our case, the **groupNumber**

The Switch Structure – more about break…

- The break statement is used because otherwise the cases in a switch statement would "*all run together*".
- So, if we forget to put the break statements, all the cases that are after a match is found will also be executed.
- In other words, execution falls through to the next branch, and so on, until a break or the end of the switch is reached.
- **Note**: there is no need to put a break after the last case/default, as the switch will exit naturally

Example of fall through:

```
case 1: System.out.println("you are on Monday from 9 – 11");
case 2: System.out.println("you are on Monday from 11 - 13");
case 3: System.out.println("you are on Monday from 14 - 16");
case 4: System.out.println("you are on Monday from 16 - 18");
case 5: System.out.println("you are on Tuesday from 9 – 11");
case 6: System.out.println("you are on Tuesday from 11 – 13"); break;
default: System.out.println("no such group!!");
```

If the group number is 4, then the output will be:

you are on Monday from 16 - 18
you are on Tuesday from 9 – 11
you are on Tuesday from 11 – 13

If the group number is 7, then the output will be:

you are on Tuesday from 14 – 16
no such group!!

The 'switch' statement: a 2nd example with String

```
public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    // prompt user to enter the role of a staff member and store the input
    System.out.print("Enter the role: ");
    String role = sc.next();

    /* check the role against the listed cases,
     * print the corresponding name when a match is found,
     * then exit the switch */
    switch(role){
        case "manager": System.out.println("John"); break;
        case "deputy": System.out.println("Helen"); break;
        case "director": System.out.println("Alicia"); break;
        case "cook": System.out.println("Brendan"); break;
        case "assistant": System.out.println("Chris"); break;
        default: System.out.println("no such role!!");
    } // end switch

    // close the Scanner
    sc.close();
}
```

Exercise 1: Choose a name for the class, and make sure you import the necessary class, then run it by testing all the cases.

Exercise 2: Add 2 more cases, one for cleaner, and for accountant, then test them.

Like in Python, we have 2 types of errors:

1. Syntax errors (e.g., when we forget to put all braces, or the semicolons, misspellings, etc.) are caught by the compiler.

2. Logic errors has its effect at execution/run time; and as we add more to our programs and have more selection and repetition structures, we will also are more likely to encounter logic errors.

- fatal logic error causes a program to fail and terminate prematurely. Examples include InputMismatchException when you try to input, let's say, a name where a whole number is expected, or when you have an input loop and the JVM runs out of memory...

- nonfatal logic error allows a program to continue executing but causes it to produce incorrect results. We discuss several of these cases: wrong order of the if statements, not grouping the statements using braces, wrong boolean condition in the if header, and so on…

Abbreviated Assignment Expressions

Java, like Python, provides several assignment operators for abbreviating assignment expressions that uses the same variable on both sides of the assignment, using the following **format**:

variable operator= expression;

Example 1: c = c + 3;
Can be abbreviated with the addition & assignment operator **+=** as: c **+=** 3;

Example 2: c = c - 3;
Can be abbreviated with the subtraction & assignment operator **-=** as: c **-=** 3;

Example 3: c = c * 3;
Can be abbreviated with the multiplication & assignment operator ***=** as: c ***=** 3;

Example 4: c = c / 3;
Can be abbreviated with the division & assignment operator **/=** as: c **/=** 3;

Example 5: c = c % 3;
Can be abbreviated with the modulus & assignment operator **%=** as: c **%=** 3;

Increment and Decrement Operators

- Java also provides the unary mathematical operators (not found in Python!!):

1. **Increment operator ++** - causes a variable to be increased by 1;

2. **Decrement operator --** - causes a variable to be decreased by 1;

- If the unary increment or decrement operator is placed **after** the variable it is referred to as the **post**-increment and **post**-decrement operator, respectively:

  a++- post-incrementing variable a by 1
  a-- - post-decrementing variable a by 1

- If the unary increment or decrement operator is placed **before** the variable it is referred to as the **pre**-increment and **pre**-decrement operator, respectively.

  ++a – pre-incrementing variable a by 1
  --a – pre-decrementing variable a by 1

NOTE: Attempting to use the increment or decrement operator on an expression other than one to which a value can be assigned is a syntax/compilation error. For example, writing ++(x + 1) will not compile, because (x + 1) is not a variable.

Increment and Decrement Operators

- **Pre-incrementing (pre-decrementing)** a variable causes the variable to be incremented (decremented) by 1, then the new value of the variable is used in the expression/another statement in which it appears.

Example of using pre-incrementing/decrementing inside an expression: assume a is 3, b is 7;
-- a + b → will yield a result of 9, because a is first decremented by 1 (so it becomes 2) then the value of b is added;
10 + ++b → will yield a result of 18, because b is first incremented by 1 (so it becomes 8) then 10 is added to it;

- **Post-incrementing (post-decrementing)** the variable causes the current value of the expression is first computed, then the variable is incremented (decremented) by 1.

Example of using pre-incrementing/decrementing inside an expression: assume a is 3, b is 7;
a-- + b → will yield a result of 10, because a is first added to the value of b, then decreased by 1;
10 + b++ → will yield a result of 17, because 10 is first added to b, then b is increased by 1;

Increment and Decrement Operators

- If **pre-incrementing** and **post-incrementing** are used on their own outside an expression or another statement, they both increment the respective variable by 1 and yield the same result:

  Example: assume a is 3 and b is 7

```
++a;
b++;
System.out.println("a is: " + a + "b is: " + b);
// will output a is: 4 b is: 8
```

- Similarly, if **pre-decrementing** and **post-decrementing** are used on their own outside an expression or another statement, they both decrement the respective variable by 1 and yield the same result:

  Example: again, assume a is 3 and b is 7

```
--a;
b--;
System.out.println("a is: " + a + "b is: " + b);
// will output a is: 2 b is: 6
```

- **So, outside an expression/statement, it does not matter whether we use pre or post operations, because they yield the same result!!!**

Example with re-setting the values of variables

```
int a = 3, b = 7;
int exp = -- a  + b ;
int exp2 = 10 + ++b ;
System.out.println("if a is 3 and b is 7, -- a  + b is " + exp);
System.out.println("if a is 3 and b is 7, 10 + ++b  is " + exp2);

// reset the values of variables a and b to 3 and 7
a = 3; b = 7;
exp = a--  + b ;
exp2 = 10 + b++;
System.out.println("if a is 3 and b is 7, a-- + b is " + exp);
System.out.println("if a is 3 and b is 7, 10 + b++ is " + exp2);

// reset the values of variables a and b to 3 and 7
a = 3; b = 7;
++a;
b++;
System.out.println("if a is 3 and b is 7, then, after ++a and b++ outside "
        + "expressions, a is: " + a + " and b is: " + b);

// reset the values of variables a and b to 3 and 7
a = 3; b = 7;
--a;
b--;
System.out.println("if a is 3 and b is 7, then, after --a and b-- outside "
        + "expressions, a is: " + a + " and b is: " + b);
```

Output:

if a is 3 and b is 7, -- a + b is 9
if a is 3 and b is 7, 10 + ++b is 18
if a is 3 and b is 7, a-- + b is 10
if a is 3 and b is 7, 10 + b++ is 17
if a is 3 and b is 7, then, after ++a and b++ outside expressions, a is: 4 and b is: 8
if a is 3 and b is 7, then, after --a and b-- outside expressions, a is: 2 and b is: 6

Exercise:

Declare a class (with a name of your choice), the main method, and then place this code in there; then run the program.

Example without re-setting the values of variables

```
// initial value of variable x
int x = 5;
System.out.println("Initial value of x is: " + x);

// pre-incrementing and pre-decrementing
System.out.println("Value of x when pre-incrementing is: " + ++x);
System.out.println("Value of x after pre-incrementing is: " + x);
System.out.println("Value of x when pre-decrementing is: " + --x);
System.out.println("Value of x after pre-decrementing is: " + x);

// post-incrementing and post-decrementing
System.out.println("Value of x when post-incrementing is: " + x++);
System.out.println("Value of x after post-incrementing is: " + x);
System.out.println("Value of x when post-decrementing is: " + x--);
System.out.println("Value of x after post-decrementing is: " + x);
```

Exercise:

Declare a class (with a name of your choice), the main method, and then place this code in there; then run the program.

Output:

Initial value of x is: 5
Value of x when pre-incrementing is: 6
Value of x after pre-incrementing is: 6
Value of x when pre-decrementing is: 5
Value of x after pre-decrementing is: 5
Value of x when post-incrementing is: 5
Value of x after post-incrementing is: 6
Value of x when post-decrementing is: 6
Value of x after post-decrementing is: 5

Iteration/Repetition

- Like in Python, we can use repetition/iteration constructs to repeatedly execute blocks of code.

- Like in Python, the programming structure that is used to control the repetitions is often called a **loop**.

- There are 4 **types of loops** in Java that allow you to repeatedly perform a task:

  - **while** loop;
  - **for** loop;
  - **do…while** loop;
  - **enhanced for** loop (covered later).

The ‘while’ Loop

- Like in Python, the **while loop** is typically used to account for a *non-fixed number of iterations: sentinel controlled repetition*
- But, like in python, the while loop is quite versatile and it can also account for *fixed-number iterations: counter controlled repetition*

The while syntax

```
while ( /* test/guard/Boolean condition*/ )
{
        // instruction(s) to be repeated go here
}
```

The WHILE loop – Input validation

- Like in Python, one typical application of the while loop is input validation!!
- In this case, we want the user to be able to re-enter the value until correct value has been entered.

- For instance, if we want the user to input only positive integers for age:
- So, as long as the input is negative (that is, smaller than 0), the program should do the following:
    1. Re-display the prompt to enter age
    2. And take input again and assign to the variable age
- To do so, we can use the **while loop** that should carry out 1 and 2 over and over again until the user finally enters a positive integer…
- Then the program can continue its execution as planned.

```java
import java.util.Scanner;

/**
 * uses the while loop to allow user to re-enter a positive integer value for age
 */
/**
 * @author aura power
 *
 */
public class AgeDemo {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        //prompt the user to enter his/her age
        System.out.print("Enter your age: ");
        int age = sc.nextInt();
        while(age < 0) { // as long as the value is negative
            // prompt user to enter a positive value
            System.out.print("You must enter a positive value: ");
            // allow user to re-enter value
            age = sc.nextInt();
        }
        // output the value of age back to the user
        System.out.println("Your age is: " + age);
        //close the input scanner
        sc.close();
    } // end main
}// end class
```

SAMPLE OUTPUT:

Enter your age: -10

You must enter a positive value: -15

You must enter a positive value: -34

You must enter a positive value: 34

Your age is are 34

What would happen if we formulated the condition of the while as age > 0??

The program would not allow user to input positive integers, and re-display the prompt until the user enters a negative int !!! – which does not make sense.

The WHILE loop – Another Example

- We can add further conditions to age check..
- For instance, we can safely assume that age cannot also be greater than 300
- In this case, we want the user to be able to re-enter the value until the age is positive, but also not greater than 300!!!
- So, as long as the input is negative or greater than 300, the program should do the following:
    1. Re-display the prompt to enter age
    2. And take input again and assign to the variable age
- To do so, we can modify the **while loop** to carry out 1 and 2 over and over again until the user finally enters a positive integer no greater than 300…

while(age < 0 || age > 300)

EXERCISE: implement the above program separately, as a different class

What would happen if we have && instead of || ??
The condition will always be false, because a number cannot be in the same time < 0 and > 300; it can only be one or the other

The WHILE loop – Yet Another Example…

Assume we want to know how many years will take to double our account balance (if we left it untouched!!!)

Let's break it down…

1. First we need to store the account balance
2. Then we need to store the interest rate
3. We then need a variable to keep track of how many years will take to double the balance
4. We also need to calculate the interest using the following formula: balance * rate / 100
5. We then add the interest to the balance until the balance is doubled; when is doubled, we exit the loop… so this is the boolean condition that we need to state in the header.

```java
public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    //prompt the user to enter his/her account balance and store that
    System.out.print("Enter your balance: ");
    double balance = sc.nextDouble();
    //prompt the user to enter interest per year and store that
    System.out.print("Enter the interest rate per year: ");
    double interestRate = sc.nextDouble();
    //compute the balance to be reached .. in our case, doubled
    final double TARGET = balance * 2;
    // number of years that takes to double the balnce..initially is 0
    int year = 0;

    // as long as the balance is smaller than the target of doubled balance
    while(balance < TARGET){
        // we compute the interest
        double interest = balance * interestRate / 100;
        // we add the interest to the balance
        balance = balance + interest;
        //we also increase the number of years
        year++;
    }
    // We output the final value of years
    System.out.println(year);
    //close the scanner
    sc.close();
```

EXERCISE: write import statement and put this code into a class

SAMPLE OUTPUT:

Enter your balance: 10000

Enter the interest rate per year: 5

15

While – another example

- Find the first power of 3 larger than 100:

```
int product = 3;
while ( product <= 100 )
    product = 3 * product;
```

- Each iteration multiplies the **product** by 3, so **product** takes on the values 9, 27, 81 and 243 successively.
- When variable **product** becomes 243, the while-statement condition—product <= 100—becomes false.
- Repetition terminates. The final value of product is 243.
- Program execution continues with the next statement after the while statement (if any).

Your Turn…

- Assuming that a is 2 and n is 4, what is the output of the following block of code?

```
int r = 1;
int i = 1;
while ( i <= n ) {
        r = r * a;
        i++;
}
System.out.println(r+ " " + i + " " + a+ " " + n);
```

Use a table to keep track of values.

- Trace the following code. What error do you observe?

```
int n = 1;
while (n != 30) {
        System.out.println(n);
        n = n + 10;
}
```

Use a table to keep track of values

- What is the output of the following loop?

```
int n = 5;
while (n >= 0) {
        n--;
        System.out.print(n);
}
```

Use a table to keep track of values.

- Trace the following code. What error do you observe?

```
int lower = 0;
int upper = 3;
while (lower >= upper) {
        System.out.println(lower);
        ++ lower;
}
```

Your Turn…

- Assuming that a is 2 and n is 4, what is the output of the following block of code?

```
int r = 1;
int i = 1;
while ( i <= n   | |  r <= n) {
        r = r * a;
        i++;
}
System.out.println(r+ " " + i + " " + a+ " " + n);
```

Use a table to keep track of values.

- Trace the following code. What error do you observe?

```
int n = 0;
while (n > 3 && n < 7) {
        System.out.println(n);
        ++ n;
}
```

- What is the output of the following loop? What error do you observe?

```
int n = 4;
while (n % 2 == 0) {
        n++;
        System.out.print(--n);
}
```

Use a table to keep track of values.

- Trace the following code. What error do you observe?

```
int k = 0;
while (k < 10) {
        System.out.println(k);
        k *= 2;
}
```

The do/while Repetition Structure

- The **do/while** repetition structure is similar to the while structure.

BUT…

- In the while structure the loop condition is tested at the beginning of the loop **before** the body of the loop is performed.
- The **do/while** structure tests the loop-continuation condition **after** the body of the loop is performed.
- Therefore, in the case of **do/while** structure, **the body of the loop is always executed at least once**.

The while syntax

```
do{
    // instruction(s) to be repeated go here
} while ( /* test/guard/Boolean condition*/ );
```

An example of a menu driven program

```
int choice;
System.out.println("*** FOP2 Lab Times***");
do
{
   System.out.println("Enter one of the following choices");
   System.out.println("1: TIME FOR GROUP 1");
   System.out.println("2: TIME FOR GROUP 2");
   System.out.println("3: TIME FOR GROUP 3");
   System.out.println("4: QUIT PROGRAM");
   choice = input.nextInt();
   System.out.println();
   switch(choice)
   {
     case 1: System.out.println("10.00 a.m ");break;
     case 2: System.out.println("1.00 p.m ");break;
     case 3: System.out.println("11.00 a.m ");break;
     case 4: System.out.println("Goodbye ");break;
     default:  System.out.println("Options 1-4 only!");
   }
} while (choice != 4);
```

Exercise:
Write a class that contains the main method, and then put the code in the main; execute the program with various input

Your turn:
What is the output of the following block of code?

```
int k = 3;
do{
      System.out.print(--k);
} while (k >= 0);
```

Use a table to keep track of the variable.

The ‘for’ loop syntax

- The for loop is typically used to implement counter-controlled repetition (but can be used with sentinel controlled repetition too – not covered);
- That is, usually the for loop is used when a value runs from a start point to an end point with a constant increment/decrement;

set a counter to some initial value (usually zero or one)

condition under which the loop may continue

changes the counter value each time round the loop

```
for( /*initial state*/;/*guard*/;/*progress*/ )
{
    // instruction(s) to be repeated go here
}
```

‘while’ vs ‘for’

```
int counter = 1; // Initialize the counter
while (counter <= 10) //Check counter is less or equal to 10
{
    System.out.println(counter); //output the counter
    counter++; // Update the counter
}
```

- The above while loop can be written as a for loop in a more compact way; but what does it output?

```
for (int counter = 1; counter <= 10; counter++){
    System.out.println(counter);
}
```

All the numbers from 1 to 10 inclusively, each on a different line

The ‘for’ loop: an example

Initial State/ Initialisation

Guard (loop-continuation condition which is a boolean)

Progress/Update

```
for(int counter = 1; counter <= 10; counter++) {
        System.out.println(counter);
}
```

- The **initialization** is executed once, before the loop body is entered;
- The **continuation condition** is checked before each iteration;
- The **update** is executed after all the statements in the body of the loop are executed;
- When the guard becomes false, the statements in the loop are no longer executed.

Some String methods and how to use the for loop to iterate through the string

- **Note 1:** the length() method returns the number of characters in a given string, including white spaces and special characters;

For example, String s = "Hi Ann!"; System.out.println(s.length()); //outputs 7

- **Note 2:** the charAt(i) returns the character at position i, but the indexing of characters in a String begins at 0 (same as in Python) → so the first index is 0, the last index is length()-1

For example, String s = "Hi Ann!"; System.out.println(s.chatAt(0)); //outputs **H**
System.out.println(s.chatAt(3)); //outputs **A**

0	1	2	3	4	5	6
H	i		A	n	n	!

```
for (int k = 0; k < s.length(); k++)
{
      System.out.print(s.charAt(k) + " ");
}
```

OUTPUT:
H i A n n !

```
for (int k = s.length() -1; k >= 0; k--)
{
      System.out.print(s.charAt(k) + " ");
}
```

OUTPUT:
! n n A i H

Break and continue statements

- **break** is used to exit/terminate a switch or a loop earlier; can be used with any type of loops;

```
for(int i = 0; i<3; i++){
    if(i == 2) {
        break;
    }
    System.out.print(i + " ");
}
```

OUTPUT:
0 1
- Because when i becomes 2, the loop is exited completely, that is no more iterations…

- **cotinue** is used only in loops to skip instructions in the given iteration; can be used with any loop type;

```
for(int i = 0; i<=3; i++){
    if(i==2) {
        continue;
    }
     System.out.print (i + " ");
}
```

OUTPUT:
0 1 3
- Because when i becomes 2, the instructions after the continue are skipped, in this case the instruction to print i; but it will go back to check the guard of the loop, and if it still stands, it will continue to execute the body…

Break and continue statements more examples

```
int i = 0;
while( i < 3){
        System.out.print(i + " ");
        if(i == 2) {
              break;
        }
        i++;
 }
```

What is the OUTPUT?
0 1 2

```
int i = 0;
do{
       System.out.print(i + " ");
       if(i == 2) {
             continue;
       }
       i++;
}while( i < 5);
```

What is the OUTPUT?

- **This is an infinite loop, because the i++ will never be executed again once i becomes 2;**
- **One solution is to place the i++; before the if.**

1. How many numbers does this loop print?

```
for (int n = 10; n >= 0; n--) {
      System.out.println(n);
}
```

2. Write a for loop that prints all even numbers between 10 and 20 (inclusive).

3. Write a for loop that computes the sum of the integers from 1 to n, where n is inputted by user.

4. What does the following block of code outputs? Re-write this code using a while loop.

```
int s = 0;
for (int i = 1; i <= 10; i++) {
      s = s + i;
}
```

5. What does the following block of code outputs?

```
String s = "Aurelia";
for(int k = 1; k < s.length(); k+=3){
      System.out.print(s.charAt( k ));
}
```

Nested loops

- When the body of a loop contains another loop, the loops are nested.
- A typical use of nested loops is printing a table with rows and columns:
  - The outer loop iterates over all the rows
  - The inner loop iterates over all columns in the current row.
- For instance, we can display in tabular form all the powers of x from1 to 3, where x ranges from 1 to 3 also:

```
for(int i = 1; i <= 3; i++){ //this loop prints the header of the table
      System.out.print("x at the power of " + i + "\t");
}
System.out.println();
for(int i = 1; i <= 3; i++){ // the outer loop
      int power = 1;
      for(int j = 1; j <= 3; j++){ // the inner loop
            power = power * i;
            System.out.print(power+ "\t\t\t");
      }
      System.out.println();
}
```

```
x at the power of 1     x at the power of 2     x at the power of 3
1                       1                       1
2                       4                       8
3                       9                       27
```

Nested loops

- You can have any combination of nested loops:

```
for(int i = 1; i <= 3; i++){ // the outer loop is a for loop
    int j = 0;
    while( j < 3){ // the inner loop is a while loop
        if(j%2 == 0){
          System.out.print("^");
        } //end if
        else{
          System.out.print("_");
        } //end else
        j++;
    } //end inner while
    System.out.println();
} //end outer for
```

OUTPUT:

```
^_^
^_^
^_^
```

Nested loops

- You can have any combination of nested loops:

```
Scanner sc = new Scanner(System.in);
System.out.print("Enter a number or a char/string to exit: ");
while(sc.hasNextInt()){
    int n = sc.nextInt();
    for(int i = n; i >= 0; i--){
        System.out.print(i + " ");
        if(i == 0){
           System.out.print("BOOOM!!");
        } //end if
    } //end inner for
    System.out.println();
    System.out.print("Enter a number or a char/string to exit: ");
} //end outer while
sc.close();
```

```
Enter a number or a char/string to exit: 5
5 4 3 2 1 0 BOOOM!!
Enter a number or a char/string to exit: 7
7 6 5 4 3 2 1 0 BOOOM!!
Enter a number or a char/string to exit: 25
25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 BOOOM!!
Enter a number or a char/string to exit: jrthu
```

Generating random numbers

- We will use the method called **nextInt** from the Random class which is also found in the **java.util** package, so we must import it;
- The method nextInt(int x) returns a pseudo-random integer between 0 (inclusive) and the specified number x (exclusive); the logic is similar to the function randint in Python

```
import java.util.Random;
public class RandomGenerator{
    public static void main(String[] args) {
        Random rand = new Random();
        for (int n = 1; n <= 10; n ++) {
            System.out.print(rand.nextInt(7) );
        } //end for
    } //end main
} // end class
```

OUTPUT: any 10 numbers between 0 and 7 (note that 7 is not inclusive);
For example: 1 6 2 4 1 5 0 0 3 4