Joanna Klukowska
joannakl@cs.nyu.edu

Lecture 1:
Elementary Programming and Control Structures

Contents

Joanna Klukowska
joannakl@cs.nyu.edu

1 Elementary Programming

1.1 Problem, algorithm, pseudocode, code

We write programs to solve real life **problems** (well, maybe not at the beginning, but eventually). Writing such a program involves:

- designing an algorithm (using natural language and pseudocode);
- writing source code that implements the algorithm.

An **algorithm** describes how the problem is solved by listing ordered steps that need to be taken (think of a cook book recipe, that is an algorithm for preparing a specific dish). An algorithm can be described by a natural language (preferably English for this course) and/or **pseudocode** which resembles code, but is not language specific). The **source code** should be obtained by implementing each step of the algorithm using proper syntax of a specific programming language (Java in this course).

WARNING: You should NEVER start by writing source code and then figuring out the algorithm once most of it is written.

The initial description of the algorithm gives you an easy way of documenting the source code.

1.2 Variables : storage and memory, types, naming convention

A **variable** represents a value stored in the computer's memory. Each variable has:

- name;
- type;
- memory location;
- value.

Variable declaration syntax: `type variableName;`

Variable assignment syntax: `variableName = variableValue;`

Name of a variable

Names should be descriptive, i.e., they should indicate what the variable is used for.

Names of variables have to follow rules for naming identifiers (elements such as classes, methods and variables in the Java program). An **identifier** is a sequence of characters that

- consists of letters, digits, underscores (_), and dollar signs ($)
- starts with a letter, an underscore (_), or a dollar sign ($) (cannot start with a digit)
- cannot be a reserved word (see Appendix A, "Java Keywords," for a list of reserved words)
- cannot be true, false, or null
- can be of any length.

Example:
valid names: `$$$$2`, `area`, `Area`, `_showMessage`, `numOfStudents`,

invalid names: `2pounds`, `a+b`, `public`, `int`

Joanna Klukowska
joannakl@cs.nyu.edu

Type of a variable

Java provides several built-in **primitive data types** :

- byte
- short
- int
- long
- float
- double
- boolean
- char

For more on primitive data types in Java see: http://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html. Later on in the semester we will learn how to create **user defined data types**.

Memory location of a variable

The **memory** for a variable is allocated when you declare it. The size of the chunk of memory associated with a given variable depends on its type and differs in different programming languages. Memory location of a variable cannot change after the variable is created.

Value of a variable

The value of a variable is stored in the memory associated with given variable. The computer memory can only store sequences of zeros and ones (binary numbers). The actual value of a variable is determined using such sequence of zeros and ones and the type of the variable. The exact same sequence of bits represents a different value when it is stored in memory locations associated with variables of different types.

1.3 Reading input from console

`System.in` refers to standard input; by default the keyboard input.

`System.out` refers to standard output; by default the output you see on the screen.

We saw `System.out` used previously to print text and numbers on the screen:

```
System.out.print("Enter a number for radius: ");

System.out.println("Enter a number for radius: ");
```

In order to read the input from the user we need a variable/object of type `Scanner` (`Scanner` is not one of the primitive data types. It is a class and we refer to variables whose type is a class as objects.) `Scanner` is the class that provides many ways of reading the input from console. To create a `Scanner` object in your program just write:

```
Scanner input = new Scanner(System.in);
```

To use `Scanner`, you need to import the java.util package, i.e., you need the line

```
import java.util.Scanner;
```

or

```
import java.util.*;
```

at the beginning of your file (the second option imports everything from the java.util package).

The non-primitive types come with tools that can be used with objects. Some of the tools provided by the `Scanner` class are:

Joanna Klukowska
joannakl@cs.nyu.edu

- `nextByte()` reads an integer of the byte type
- `nextShort()` reads an integer of the short type
- `nextInt()` reads an integer of the int type
- `nextLong()` reads an integer of the long type
- `nextFloat()` reads a number of the float type
- `nextDouble()` reads a number of the double type
- `next()` reads a string that ends before a whitespace character
- `nextLine()` reads a line of text (i.e., a string ending with the Enter key pressed).

To use these tools, you specify the name of the object, followed immediately by a dot (.), followed by one of the names above, for example

```
Scanner input = new Scanner(System.in);
double number = input.nextDouble();
```

For more (maybe even too much information at this point) see http://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html.

Code Example

See the file `QuadraticEquationSolver01.java` in the source code.

```
1  public class QuadraticEquationSolver01 {
2
3    public static void main(String[] args) {
4      //print the "welcome screen"
5      System.out.println("I solve quadratic equations of the form: ");
6      System.out.println("  ax^2 + bx + c = 0 \n");
7
8      //the values of a,b and c need to be stored in variables
9      double a, b, c;
10     //we also need variables to store our two solutions
11     double x1, x2;
12
13     //we need to read the values of a, b and c from the user
14     //so we need to use the Scanner object
15     Scanner in = new Scanner (System.in );
16
17     //ask the user to enter the three values
18     System.out.println("Enter the values of a, b and c " +
19         "and I will tell you the solutions.");
20     System.out.print(" a = ");
21     a = in.nextDouble();
22     System.out.print(" b = ");
23     b = in.nextDouble();
24     System.out.print(" c = ");
25     c = in.nextDouble();
26
27
28     //we are ready to compute the first solution
29     x1 = (-b - Math.sqrt( b * b - 4 * a * c  )) / (2 * a);
30     //and similarly, we compute the second solution
```

Joanna Klukowska
joannakl@cs.nyu.edu

```
31      x2 = (-b + Math.sqrt( b * b - 4 * a * c  )) / (2 * a);
32
33      //after hard work, we can show the user her/his solutions
34      System.out.println("\nThe solutions are:\nx1 = " + x1 + "\nx2 = " + x2);
35
36      System.out.println("\n\nThank you for crunching numbers!\n\n");
37
38      in.close();
39    }
40  }
```

1.4 Named constants

A **named constant** is an identifier that represents a permanent value, i.e., a variable whose value does not change.

Syntax:

```
final datatype CONSTANTNAME = value;
```

The value of a constant has to be assigned during its declaration. `final` is the keyword that indicates that the value is constant and will not be changed.

Benefits of using constants: You assign the value once, without need to repeat it all over the code. If you need to change the value, it is done in one place. Constants' descriptive names (assuming you use one) make programs easier to read.

1.5 Assignment operator =

In Java, **the variable has to be on the left hand side** of the assignment operator and **the value to be assigned to that variable has to be on the right hand side** of the assignment operator:

```
variable = value;
```

For example, you can write:

```
int radius;
radius = 5;}
```

but NOT

```
5 = radius; // This is WRONG
```

The compiler interprets the second expression as trying to change the value of the numeral 5 and generates an error.

1.6 Mathematical operations: numerical operators, order of operations, shorthand operators, type conversions, casting, integer vs. double division

Binary numeric operators

- `+` addition: 34 + 6, 34.5 + 2.1
- `-` subtraction: 34 - 6, 34.5 - 2.1
- `*` multiplication: 34 * 6, 34.5 * 2.1
- `/` integer division: 34 / 5, result: 6
- `/` real division: 34.0 / 5, result: 6.8
- `%` modulus/Remainder: 34 / 5, result: 4

Joanna Klukowska
joannakl@cs.nyu.edu

WARNING: There is no exponentiation operator in Java. You can use repeated multiplications or the method `Math.pow(a,b)` to compute a^b.

Order of operations is the same as in math: multiplication, division and modulus are performed in order from left to right before addition and subtraction. You can use parenthesis in order to change that behavior.

Shorthand assignment operators:

- `+=` addition assignment: x += 8; short for: x = x + 8;
- `-=` subtraction assignment: x -= 8; short for: x = x - 8;
- `*=` multiplication assignment: x *= 8; short for: x = x * 8;
- `/=` integer division assignment: x /= 8; short for: x = x / 8;
- `/=` real division assignment: x /= 8; short for: x = x / 8;
- `%=` modulus/Remainder assignment: x %= 8; short for: x = x % 8;

2 Selection Statements

2.1 Boolean data type and comparison operators

Boolean data type is used for a variable whose value can be either `true` or `false`. In Java the `boolean` type is used for such variables:

```
boolean isGreater = true;
```

Comparison operators compare two values and produce (or return) the value `true` or `false`
(assume `radius` is equal to `5`, in the examples below):

- `<` less than: `radius < 0` is `false`
- `<=` less than or equal: `radius <= 0` is `false`
- `>` greater than: `radius > 0` is `true`
- `>=` greater than or equal: `radius >= 0` is `true`
- `==` equal to: `radius == 0` is `false`
- `!=` not equal to: `radius != 0` is `true`

WARNING : `==` (double equal sign) is used for comparison, `=` (single equal sign) is used for assignment; confusing these two results in a common logical errors that may not caught by the compiler.

2.2 Different types of selection statements in Java

2.2.1 `if ...` statement

Syntax:

```
if ( Boolean expression ) }
  //statements to execute if Boolean-expression is true;
}
```

Joanna Klukowska
joannakl@cs.nyu.edu

Example:

```
if (radius >= ) {
  area = radius * radius * Math.PI;
}
```

2.2.2 `if ... else ...` statement, a.k.a. two way `if` statement

Syntax:

```
if ( Boolean expression ) {
  //statements to execute if Boolean-expression is true;
else {
  //statements to execute if Boolean-expression is false;
}
```

2.2.3 `if ... else if ... else ...` statement, a.k.a. multi-way `if` statement

Syntax:

```
if ( Boolean expression  1 ) {
   //statements to execute if Boolean expression  1 is true;
else {
  if ( Boolean expression 2 ) {
     //statements to execute if Boolean expression 2 is true;
  }
  else {
    if ( Boolean expression 3 ) {
       //statements to execute if Boolean expression 3 is true;
    }
    else {
        if ...
                else {
                  //statements to execute if all Boolean expressions are false
                }
    }
  }
}
```

Alternative, better, notation:

```
if ( Boolean expression  1 ) {
   //statements to execute if Boolean expression  1 is true;
else if ( Boolean expression 2 ) {
  //statements to execute if Boolean expression 2 is true;
}
else if ( Boolean expression 3 ) {
  //statements to execute if Boolean expression 3 is true;
}
else if ... {

}
else {
  //statements to execute if all Boolean expressions are false
}
```

Joanna Klukowska
joannakl@cs.nyu.edu

2.2.4 `switch` statements

Syntax:

```
switch ( expression ) {
  case value1: //statements to execute if expression has value1
  case value2: //statements to execute if expression has value2
  case ...
  case valueN: //statements to execute if expression has valueN
  default: //statements to execute if expression
           //does not match any of the above values

}
```

If only one of the cases should be executed when the match is made, the last statement in that case should be `break;`. Otherwise `switch` statement follows a fall through behavior and the next case is entered; this repeats until either a `break;` statement is encountered or the end of the `switch` statement is reached.

2.3 Logical operators

Logical operators are used in combination with comparison operators and boolean variables to produce more **compound Boolean expressions**. The logical operators in Java are:

- `!` not
- `&&` and
- `||` or

Truth tables

a	b	!a	a && b	a||b
false	false	true	false	false
false	true	true	false	true
true	false	false	false	true
true	true	false	true	true

Short-circuit evaluation :

- If the first operand of the `&&` operator is `false`, the second operand is not evaluated.
- If the first operand of the `||` operator is `true`, the second operand is not evaluated.

Example:
There is a problem with the following code if the user enters `0` for the value of `y`.

```
1  Scanner in = new Scanner( System.in );
2
3  int x;
4  int y;
```

Joanna Klukowska
joannakl@cs.nyu.edu

```
5
6   System.out.println("Enter values of integers x and y. \n"
7           + "I will give you x/y.");
8   System.out.print ("x = ");
9   x = in.nextInt();
10  System.out.print ("y = ");
11  y = in.nextInt();
12
13  if ( x%y != 0)
14    System.out.println("x/y = " + x/y + ", remainder " + x%y );
15  else
16    System.out.println("x/y = " + x/y );
17
```

Short circuit evaluation in Java lets us fix this by adding an extra check in the if statement as follows:

```
1   if ( y != 0 && x%y != 0)
2     System.out.println("x/y = " + x/y + ", remainder " + x%y );
3   else if ( y != 0 )
4     System.out.println("x/y = " + x/y );
5   else
6     System.out.println("Sorry, cannot divide by zero. ");
7
```

2.4 Common errors

Forgetting braces (especially for the previous Python users):

```
if ( radius >= 0 )
   area = radius * radius * PI;
   System.out.println( area );
```

WRONG

```
if ( radius >= 0 ){
   area = radius * radius * PI;
   System.out.println( area );
}
```

CORRECT

In Java a block of code has not be surrounded by the curly braces, not just indented!

The only exception to this rule is when the block of code consists of a single statement.

Semicolon at the end of the if line:

```
if ( radius >= 0 );
{
   area = radius * radius * PI;
   System.out.println( area );
}
```

USUALLY WRONG, equivalent to code on the right

```
if ( radius >= 0 ){
   //do nothing
}
area = radius * radius * PI;
System.out.println( area );
```

The semicolon right after the if condition indicates the end of the if block of code, i.e., if condition is true, execute the empty block.

Dangling else ambiguity:

Joanna Klukowska
joannakl@cs.nyu.edu

```
int i = 1, j = 2, k = 3;
if ( i > j )
   if (i > k )
      System.out.println("A");
   else
      System.out.println("B");
```

OR

```
int i = 1, j = 2, k = 3;
if ( i > j )
   if (i > k )
      System.out.println("A");
else
   System.out.println("B");
```

The `else` always matches the `if` **immediately** before it, unless curly braces are used to change that. To achieve the above *desired* behavior without ambiguity use:

```
int i = 1, j = 2, k = 3;
if ( i > j ){
   if (i > k ){
      System.out.println("A");
   }
   else {
      System.out.println("B");
   }
}
```

```
int i = 1, j = 2, k = 3;
if ( i > j ){
   if (i > k ){
      System.out.println("A");
   }
}
else {
   System.out.println("B");
}
```

Redundant testing of Boolean values:

```
boolean even = true
if ( even == true )
   System.out.println("Even");
else
   System.out.println("Odd");
```

Correct, but redundant

```
boolean even = true
if ( even )
   System.out.println("Even");
else
   System.out.println("Odd");
```

BETTER STYLE

Comparison of Boolean variables to `true` or `false` is redundant and may, in fact, reduce the readability of the code.

Forgetting the break statement in the case of switch statement:

```
switch ( score ){
  case 5: System.out.print("A");
  case 4: System.out.print("B");
  case 3: System.out.print("C");
  case 2: System.out.print("D");
  case 1: System.out.print("F");
  default: System.out.print("Not" +
                "a valid score");
```

Most likely INCORRECT. This compiles, but when score has a value of `5`, the printout is `ABCDFNot a valid score`.

```
switch ( score ){
  case 5: System.out.print("A");
          break;
  case 4: System.out.print("B");
          break;
  case 3: System.out.print("C");
          break;
  case 2: System.out.print("D");
          break;
  case 1: System.out.print("F");
          break;
  default: System.out.print("Not" +
                "a valid score");
```

Prints `A`, when `score` is `5`.

This is known as the **fall-through behavior**.

Joanna Klukowska
joannakl@cs.nyu.edu

2.4.1 Code examples

See `QuadraticEquationSolver02.java` for an improved version of out QuadraticEquationSolver01 program.

3 Looping

3.1 Syntax for the three looping statements

3.1.1 `while` loop

```
while (loop-continuation-condition) {
  //body of the loop}
  statements;
}
```

body of the loop - block of statements inside the loop statement

iteration = repetition - one-time execution of the body of the loop

loop-continuation-condition Boolean expression that controls the number of iterations; when it becomes `false`, the program control turns to the statement that follows the loop

Example:
The following loop computes the sum of numbers from 0 to 100.

```
int count = 0;
int sum = 0;
while (count <= 100 ) {
  sum = sum + count;
  count++;
}
```

3.1.2 `do ... while` loop

```
do {
  //body of the loop}
  statements;
} while (loop-continuation-condition);
```

Use this loop instead of the `while` loop if the body of the loop needs to be executed at least once. The condition is checked **after** the first loop iteration.

NOTE: there is a semicolon after the closing parenthesis for the loop-continuation-condition!

3.1.3 `for` loop

```
for (initial-action, loop-continuation-condition; action-after-each-iteration ) {
  //body of the loop}
  statements;
}
```

initial-action - a list of zero or more comma separated variable declaration/assignment expressions; executed only once

action-after-each-iteration - a list of zero or more statements to be executed after each iteration

Joanna Klukowska
joannakl@cs.nyu.edu

Use this loop when the number of iterations is known up front. The variables used in the `loop-continuation-condition` should be modified only by the `action-after-each-iteration` statements. If you need to modify them inside the loop, you should be using one of the other two loops.

Example:
The following loop computes the same sum of numbers from 0 to 100 but uses the **`for`** loop instead of the **`while`** loop.

```
int sum = 0;
for (int count = 0; count <= 100; count++ ) {
  sum = sum + count;
}
```

3.2 Sentinel controlled repetition

sentinel value - value that indicates the end of the input

Example:
Compute the sum of positive values entered from the command line. Number zero, 0, indicates the end of input.

```
do {
   read a number from the standard input (user)
   add number to the sum
} while ( number read is not zero )
```

Note: the above is pseudocode, not actual Java code.

3.3 Nested loops

Loop statements can be nested. You can mix different types of loops when nesting them.

Example:
How many times will the message "nesting loops" be printed by the following loop?

```
for ( int i = 0; i < 100; i++ ) {
  for ( int j = 0; j < 5; j++ ) {
    System.out.println("nesting loops");
  }
}
```

Note: the variable used as a counter in the inner loop is different than the one in the outer loop.

3.4 `break` and `continue` statements used to alter behavior of loops

`break` - when this statement is executed, the loop is immediately terminated, the program control turns to the statement that follows the loop

`continue` - when this statement is executed, the current iteration of the loop is immediately terminated, the program control turns to the first statement of the body of the loop in the next iteration

It is generally advisable to avoid using these statements unless absolutely necessary. It is much easier to read the code when the loop continuation condition determines the behavior of the loop. But sometimes the **`break`** and **`continue`** statements come in handy.

Example:
Compute the sum of positive values entered from the command line. Number zero, 0, indicates the end of input. The sum should not exceed 100, so stop if the user keeps on entering the numbers. Do not add multiples of 10 to the sum.

Joanna Klukowska
joannakl@cs.nyu.edu

```
do {
  read a number from the standard input (user)
  if the number is a multiple of 10
    continue
  if (sum + number > 100 )
    break
  add number to the sum
while (number read is not zero );
```

3.5 Common looping errors

Infinite loop If the loop-continuation-condition never becomes false, the loop never terminates. This means your program will end up running forever.

One possible infinite loop:

```
int x = 0;
while (x < 100000 )
   System.out.println ( x )
   x = x + 10000;
```

INCORRECT, the increment happens outside of the loop

```
int x = 0;
while (x < 100000 ){
   System.out.println ( x )
   x = x + 10000;
}
```

CORRECT

Another one:

```
int x = 0;
while (x < 100000 ){
   System.out.println ( x )
   x = x - 10000;
}
```

Most likely INCORRECT, but is it an infinite loop

correct version depends on what the program is supposed to do

Off by one error Write a loop that prints "looping" 100 times:

```
int counter = 0;
while (counter <= 100 ){
   System.out.print ("looping")
   counter++;
}
```

INCORRECT, it prints 101 lines

```
int counter = 0;
while (counter < 100 ){
   System.out.print ("looping")
   counter++;
}
```

CORRECT

Using a variable declared inside the `initial-action` of the `for` loop outside of the loop You can declare a variable inside the `initial-action` clause of the `for` loop. The scope of that variable is only within the body of the `for` statement. It should not be used outside.

Joanna Klukowska
joannakl@cs.nyu.edu

`for(int i=0; i< 100; i++)`
`  System.out.print ("looping")`
`System.out.println("i = " + i );`
	`int i;`
	`for( i=0; i< 100; i++)`
	`  System.out.print ("looping")`
	`System.out.println("i = " + i );`

INCORRECT, `i` is no longer in scope	CORRECT

Adding a semicolon at the end of `for/while` line, before the body of the loop results in an empty body of the loop. The remaining code is executed regardless of the loop condition.

`for(int i=0; i< 100; i++);` `  System.out.print ("looping")`	`for(int i=0; i< 100; i++)` `  System.out.print ("looping")`

`int i = 0;`
`while( i< 100 );`
`{`
`  System.out.print ("looping")`
`  i++;`
`}`
	`int i = 0;`
	`while( i< 100 )`
	`{`
	`  System.out.print ("looping")`
	`  i++;`
	`}`

INCORRECT, the body of the loop is empty	CORRECT

Forgetting the semicolon at the end of the `do ... while` loop is a compilation error, so usually it is easier to find.

`int i = 0;`
`do {`
`  System.out.print("looping");`
`  i++;`
`}while ( i < 100)`
	`int i = 0;`
	`do {`
	`  System.out.print("looping");`
	`  i++;`
	`}while ( i < 100);`

INCORRECT, but luckily for us, this does not compile	CORRECT

Using `break/continue` statements in place of `loop-continuation-condition` is a very bad style and reduces program readability.

`int number = 0;`
`while ( true ){`
`  number++;`
`  if (number > 100)`
`  break;`
`}`
	`int number = 0;`
	`while ( number <= 100 ){`
	`  number++;`
	`}`

INCORRECT, because it is hard to read	CORRECT

4 Formatting console output

We saw `System.out.print()` and `System.out.println()` methods - they both print text to the console. `System.out.printf()` allows us to print formatted text.

Syntax:

```
System.out.printf( format, item1, item2, ..., itemN);
```

Joanna Klukowska
joannakl@cs.nyu.edu

format is a string that consists of substrings and **format specifiers**.

item1...itemN are values (numbers, characters, strings, Boolean values), either variables or constants that will be substituted for the format specifiers in the format string (there should be the same number of items as format specifiers in the format string).

Simple format specifier consists of a percent sign followed by a letter: `b, c, d, f, e`, or `s` that indicates the type of the item corresponding to the format specifier.

- **`\%b`** a Boolean value
- **`\%c`** a character
- **`\%d`** a decimal integer
- **`\%f`** a floating point number
- **`\%e`** a number in scientific notation
- **`\%s`** a string

Example:

```
System.out.printf("Area of circle with radius %f is %f.", radius, area);
System.out.printf( "Given that x is %b and y is %b, the value of x && y is %b.",
                    x, y, x && y );
System.out.printf("Student: %s, tuition: $%f.", name, tuition );
```

Output:

```
Area of circle with radius 5.250000 is 86.587594.
Given that x is true and y is false, the value of x && y is false.
Student: John Smith, tuition: $25036.210000.
```

Format specifier can include field width and precision in addition to the conversion code (indicating the type of the value):

`%W.PC`

where **`W`** is the field width, **`P`** is the precision and **`C`** is the conversion code.

Example:

- **`%5c`** Output the character and add four spaces before the character item.
- **`%6b`** Output the Boolean value and add one space before the false value and two spaces before the true value.
- **`%5d`** Output the integer item with width at least 5. If the number of digits in the item is < 5, add spaces before the number. If the number of digits in the item is > 5, the width is automatically increased.
- **`%10.2f`** Output the floating-point item with width at least 10 including a decimal point and two digits after the point. Thus there are 7 digits allocated before the decimal point. If the number of digits before the decimal point in the item is < 7 add spaces before the number. If the number of digits before the decimal point in the item is > 7 the width is automatically increased.
- **`%12s`** Output the string with width at least 12 characters. If the string item has less than 12 characters, add spaces before the string. If the string item has more than 12 characters, the width is automatically increased.

4.0.1 Code example:

see `PrintfExample.java`