3 Control flow

Objects are *very* important to effective programming. It is beneficial to learn to thing of programming in terms of objects first - in order to work more effectively with an *object oriented* language such as Java. However, many of the tools that make programming powerful precede wide spread use of objects in programming.[10] It is important to understand and affect flow in a programming language, if we are to use computers effectively.

3.1 Order matters

First, it is important to consider that actions are executed in order. Imagine if we are creating and eating a virtual peanut butter sandwich, the following order should succeed:

- open the bread
- take out to slices of bread
- spread peanut butter on the bread
- place peanut butter faces together
- eat sandwich

However, if we were to try:

- spread peanut butter on the bread
- open the bread
- take out to slices of bread
- place peanut butter faces together
- eat sandwich

The program (or *algorithm*) would fail on the first step - because no bread is available.

Consider the following two examples using objects:

Example 1

```
Person suzy = new Person("Suzy", "Jenkins", 12);
suzy.sayHi();}
```

[10] Newer is not always better.

Example 2

```
suzy.sayHi();
Person suzy = new Person("Suzy", "Jenkins", 12);
```

Example 1 succeeds because the *suzy* **object** is created before the method `sayHi()` is called. The second example fails because `suzy.sayHi()` tries to call the *suzy* object's `sayHi()` method before there is a suzy object.

In short, commands are executed in the order written.

Such a *sequential order* is the default order of completion, i.e. one thing then the next.

The other types of order are *selection* and *repetition*.

3.2 Selection structures

3.2.1 if statements

The most common and versatile of selection statements in Java is the **if** statement. The basic structure of the *if* statement is ***if*(some condition)** *{What happens if the condition is true;}* For example, the following statement is **true** so it will run once and exit.

```
if(true){
  System.out.println(``This will print once and end.'');
}
```

By contrast, the following code begins has the integer i set to 1, so it executes because ***i*** is less than 10, then quits.

```
int i = 1;
  if(i<10){
    System.out.println(``This will print once and end.'');
  }
```

Common **equality and relational operators** for Java are:

- $<$ less than - e.g. `if (a < b){ action();}` executes if a is less than b
- $>$ greater than - e.g. `if (a > b){ action();}` executes if a is greater than b
- $==$ equal to - e.g. `if (a == b){ action();}` executes if a is equivalent to b

- != not equal to - e.g. `if (a != b){ action();}` executes if a is not equal to b
- <= less than or equal to - e.g. `if (a <= b){ action();}` executes if a is less than *or equal to* b
- >= greater than or equal to - e.g. `if (a >= b){ action();}` executes if a is greater than *or equal to* b

3.2.2 if else statements

In Java, **if** is commonly combined with **else**. Perhaps most straight forward is specifying one condition and having a default action if that condition is not true. For example, the following code will print "pass" if the **integer** *grade* is *greater than* or *equal to* 70.

```
if (grade >= 70){
  System.out.println(``pass'');
} else {
  System.out.println(``fail'');
}
```

Indeed, we can connect several *if*, *else if* statements to construct more complex logic. The careful structuring of increasingly complex *if*, *else if* statements can create more nuanced programs.

Consider, for example, a program that will assign a letter grade based on a numeric score:

```
if (grade >= 90) {
  System.out.println(``A'');
} else if (grade >= 80) {
  System.out.println(``B'');
} else if (grade >= 70) {
  System.out.println(``C'');
} else {
  System.out.println(``fail'');
}
```

Notice especially, that the final **else** serves to define what to do if all the preceding criteria are not met. A grade will be greater than or equal to 90, 80, 70 or it is failing.

A common exercise asked of job applicants is to design *if* to determine whether a number is divisible by 3, 5, or both. Only one response should be printed. If a number is divisible by 3 and 5 the specified response (and only specified response) is printed - most failed programs print all three responses or the divisible by 3 or divisible by 5 response.

3.3 Repetition AKA *looping*

One of the great advantages of computers is that they are very good at doing things - repeatedly and very fast. In Java, we use *for*, *while*, and *do while* loops. The most archetypal of loops in Java is the ***while*** loop. Its use and syntax is very straight forward. Basically, as long as some condition is met, the specified action continues. The basic structure of a **while** loop is while (some condition is met){ do this action(); }.

For example, the following `while` loop will print "it's so true" continously because the condition always evaluates to **true**.

```
while(true){
  System.out.println(``it's so true'');
}
```

How then might a *while* loop exit? In the example below, the *while* loop counts up to 10 and then exits because the integer ***i*** is no longer less than 10. Note the *++* - it means that 1 is added to the variable ***i*** each time the loop repeats (i.e. each time it *iterates*).

```
int i =0;
while(i < 10){
  i++;
}
```

3.4 *for* statements

It turns out that it is very useful to have computers *iterate*[11] through various lists of commands. In many programming languages, including Java, the *for* loop is used to count *up* or emphdown and then exit at a specified value. The basic syntax of a for loop is:

for (initial value; condition; increment){ action ;}

For example, the following *for* loop starts at 0, will continue as long as i is less than 10, and the value of i goes up by 1 each *iteration*. During each iteration the value of i will be printed[12].

```
for (int i=0; i<10; i++){
  System.out.println(i);
}
```

A very common use for *for* loops are as counters - either counting up or down. Here we see a *for* loop that counts down from 100 and stops at 1. **i >0** is used to stop the counter before 0 and ***i***– is used to count down rather than up.

```
for (int i=100; i>0; i--){
  System.out.println(i);
}
```

[11] iterate is much like *count through* or do something in a *step-by-step* manner.
[12] Note the lack of quotation marks around *i*.

3.4.1 Doing things, e.g. math, with *for* loops

Using Java's basic mathematical operators, we can begin *iterating* through many different calculations.

In Java, the basic mathematical operators are:

- **addition - 2+2 == 4**
- **subtraction - 2-2 == 0**
- **multiplication - 6*3 == 18**
- **division - 6/3 == 2**
- ***modulo* or remainder - 248%2 == 0**

These mathematical operators can leverage the values created by *for* loops. For example, the following *loop* will *iterate* through the values of i up to 100 - showing the result of the number i multiplied by 5 i.e. 1 x 5, 2 x 5, 3 x 5, 4 x 5, etc.

```
for (int i = 0; i <= 100; i++){
  System.out.println(i * 5);
}
```

Though this is not the most exciting program but illustrates the underlying utility of a *for* loop. Moreover, *for* loops can become even more useful combined with other *selection* and *repetition* such as *if* statements or additional *for* loops.

In the following example, we can print every even number between 0 and 1000 as even or odd by using the *modulo* % operator. Namely, a number divided by 2 with no remainder is even; otherwise, it is odd.

```
for (int i = 0; i < 1000; i++){
  if(i%2==0){
    System.out.println(i + `` is even'');
  } else {
    System.out.println(i + ``is odd'');
  }
}
```

Similarly, using *nested* ***for*** loops - *for* loops inside other *for* loops can be useful. In the example below, each *for* loop counts up to 12 - thereby creating a 12 x 12 multiplication table. [13]

```
for (int i = 0; i <= 12; i++) {
  for (int j = 0; j <= 12; j++) {
```

[13] *for* loops can use any variable. They commonly use i to refer to *index* variable (from math) and nested *for* loops use the subsequent letters in the alphabet i.e., j, k, l, m, and n.

```
      System.out.println(i + "x" + j + " = " + (i * j));
    }
  }
```

The following table shows the values of i and j for each iteration.

Iteration	i	j	**Iteration**	i	j
it. 1	0	0	*it. 25*	1	11
it. 2	0	1	*it. 26*	1	12
it. 3	0	2	*it. 27*	2	0
it. 4	0	3	*it. 28*	2	1
it. 5	0	4	*it. 29*	2	2
it. 6	0	5	*it. 30*	2	3
it. 7	0	6	*it. 31*	2	4
it. 8	0	7	*it. 32*	2	5
it. 9	0	8	*it. 33*	2	6
it. 10	0	9	*it. 34*	2	7
it. 11	0	10	*it. 35*	2	8
it. 12	0	11	*it. 36*	2	9
it. 13	0	12	*it. 37*	2	10
it. 14	1	0	i*it. 38*	2	11
it. 15	1	1	i*it.39*	2	12
it. 16	1	2	i*it.40*	3	0
it. 17	1	3	i*it. 41*	3	1
it. 18	1	4	i*it. 42*	3	2
it. 19	1	5	i*it. 43*	3	3
it. 20	1	6	i*it. 44*	3	4
it. 21	1	7	i*it. 45*	3	5
it. 22	1	8	i*it. 46*	3	6
it. 23	1	9	i*it. 47*	3	7
it. 24	1	10	*it. 48*	3	8
...	...	...	...	...	...

3.5 Iteration practice

Of course, we as people use computers and for loops to compute faster and more extensively than we as humans can. **However**, it is important that we develop basic proficiency and understanding of how for loops work. Examine the following code and and complete the accompanying table.

Practice 1.

```
for(int i=0;i<5;){
  for(int j=0; j<5; j++){
    i++;
}}
```

Iteration	i	j	**Iteration**	i	j
iteration 1			iteration 5		
iteration 2			iteration 6		
iteration 3			iteration 7		
iteration 4			iteration 8		

Practice 2.

```
for (int i=0; i<3; ){
  for (int j=0; j<3;j++){
    if(j%2==0){i++;
    }
  }
}
```

Iteration	i	j	**Iteration**	i	j
it. 0			*it. 5*		
it. 1			*it. 6*		
it. 2			*it. 7*		
it. 3			*it. 8*		
it. 4			*it. 9*		

Practice 3.

```
for(int i=0; i<5;i++){
  for(int j=0; j<5; j++){
    for(int k=0; k<5;k++){
```

```
        int a= i+j+k;
      }
    }
  }
```

It.#	i	j	k	a	**It.**#	i	j	k	a
it 0					*it 10*				
it 1					*it 11*				
it 2					*it 12*				
it 3					*it 13*				
it 4					*it 14*				
it 5					*it 15*				
it 6					*it 16*				
it 7					*it 17*				
it 8					*it 18*				
it 9					*it 19*				

Practice 4.

```
  int a=0;
  for (int i=0; i<4; i++){
    for (int j=0; j<4; j++){
      if (j%2 ==0){
      a++;
    }
  }
  }
System.out.println(a);
```

It.#	i	j	a	**It.**#	i	j	a
it 0				*it 5*			
it 1				*it 6*			
it 2				*it 7*			
it 3				*it 8*			
it 4				*it 9*			

Practice 5.
Consider the following code and answer the following questions ***before*** filling in the table:

- What will the final output of k be?
- Why?
- Looking at the for loops, how would you justify your assertion?

```
int a = 1;
int k = 0;
for(int i = 0; i < 10; i++) {
  for(int j = 0; j < 3; j++){
    a++;
    if((a % 2) == 0){
    k++;
  }
}
}
  System.out.println(k);
```

It.#	i	j	k	a	**It.#**	i	j	k	a
it 0					*it 16*				
it 1					*it 17*				
it 2					*it 18*				
it 3					*it 19*				
it 4					*it 20*				
it 5					*it 21*				
it 6					*it 22*				
it 7					*it 23*				
it 8					*it 24*				
it 9					*it 25*				
it 10					*it 26*				
it 11					*it 27*				
it 12					*it 28*				
it 13					*it 29*				
it 14					*it 30*				
it 15					***Final***				

Practice 6.

Imagine - you have been given 200 cinder blocks to wall off a flower bed - each measuring 1 ft.

The flower bed must have a minimum width and length of at least 1 ft. [14]

Design an algorithm to:

- Print all possible combinations of width, length, and area
- Print the greatest possible area at the completion of the function

Write your algorithm here. Label and explain the major components.

[14] Assume a rectangular flower bed. Perimeter is the sum of all sides. Area is width multiplied by length.

Practice 7.

Train a leaves New York going 100mph. Train b leaves Florida going 200mph. The distance from Florida to New York is 1000 miles.

- Design an algorithm using *nested* ***for*** loops to determine where the two trains intersect.
- Show the distance that each train is travelling per hour in a table.
- Explain in your own words how your algorithm calculates when the trains pass each other.