Object Oriented Programming – I

Self Review For Students-Part 1

Introduction to Loops

Loops are a technique that allow us to execute a statement many times. This is what gives a computer program its true power because computers can execute statements in loops millions of times a second.

There are 4 basic loop structures in Java.

1. while loop
2. for loop
3. do-while loop
4. foreach loop

The first three are general purpose loops. The fourth is a special loop form that can only be used with arrays and Iterable types. We will cover foreach loops later in the term.

1. The while loop:

```
while (condition)
  loop-body-statement
```

loop-body-statement is a single statement, simple or compound, condition is any boolean expression

While loop behavior:

1) the condition is evaluated
2) if the condition is true:
2a) the loop body statement is executed
2b) repeat step 1
if the condition is false, go to the next statement of the program

Example 1: A silly example that prints "Hello" forever:

```
while (true)
  System.out.println("Hello");
```

Note that we can put anything that evaluates to a boolean inside the parentheses.

Example 2: A loop that prints "Hello" never:

```
while (false)
  System.out.println("Hello");
```

Example 3: An example that prints "Hello" 5 times: We need to keep track of how many times we have printed "Hello". So we need a variable to store that number.

```
int count = 0; // count stores the number of times we have printed "Hello"
while (count < 5) {
    System.out.println("Hello");
    count = count + 1;
```

```
} // note: at this point in the code, count stores 5
```

What does count remember? The number of times we have printed out "Hello". Are we using count correctly?

- At the beginning, we have not printed anything and count is 0.

- Each time we print "Hello", we add one to count.

So, our use of count matches what we want it to do. What is true at the end of the loop? count stores 5 .We wanted to print 5 times, and count stores 5 so we did print 5 times.

2. The for loop

```
for (initial statement; condition; increment statements)
        loop-body-statement
```

- the initial statement is a single statement
- increment statements are 0 or more statements, separated by commas (no terminating semicolons)

condition and loop-body-statement are the same as for while loops.

For loop behavior:

1) the initial statement is executed
2) the condition is evaluated
3) if the condition is true
   3a) the loop body statement is executed
   3b) the increment statements are executed
   3c) repeat step 2

if the condition is false, go to the next statement of the program

Example 4: Write a loop that prints "Hello" 5 times, but this time using a for loop.

```
// numHellos stores the number of times we have printed "Hello"
for (int count = 0; count < 5; count = count + 1)
        System.out.println("Hello");

// note: at this point in the code count does not exist
```

3. The do-while loop

```
do {
  loop-body-statement(s)
} while (condition);
```

do-while-loop behavior:

1) Execute the loop-body-statement(s)
2) Evaluate the condition
   If the condition is true, repeat step 1
   If the condition is false, go to the next statement of the program

The do-while loop should be avoided except in cases where you really need the body of the loop to execute before testing the condition.

The problem with the do-while loop is that, if you accidentally drop the "do", the rest of the code is still valid Java, but it does something entirely different! It becomes a compound statement followed by a while loop!

When to use a while loop and when to use a for loop? Your choice. The two are exactly the same, just ordered differently.

Some guidelines:

- for loop advantage is that the code that describes the loop is all in the header.
- for loops are good when the loop is controlled by variable(s) and there is a simple increment.
- while loop advantage is that the syntax is simpler.
- while loops are good when the increment is complicated

While Loop Example 1: Euclid's Greatest Common Divisor algorithm

We want to compute the greatest common divisor of two positive integers: a and b
If a % b = 0 then the greatest common divisor is b
Otherwise gcd(a,b) = gcd(b, a % b)
This gives us a loop. For example:
gcd(30, 12) = gcd(12, 6) = 6
gcd(24, 15) = gcd(15, 9) = gcd(9, 6) = gcd(6, 3) = 3

```
public static int gcd(int a, int b) {
        while (a % b ! = 0) {
                int r = a % b;
                a = b;
                b = r;
        }
        return b;
}
```

This algorithm seems to require a > b. (It does not, trace the loop and see what happens if b > a.)

While Loop Example 2: Computing a square root using Newton's method

A little background. Please look up Newton's method if you do not understand how it works. Let x1 be a guess for the square root of n.

Let f(x) = x * x - a (Note that f(x) = 0 when x is the square root of a.

Since the plot of f(x) is concave up, we can get a new guess by following the tangent to the curve from the current guess (x1, f(x1)) to (x2, 0). x2 is the new, better guess.

Using the slope formula were m is the slope of the tangent, we have
m = (0 - f(x1)) / (x2 - x1)
Note that m = f'(x1) = 2 * x1
2 * x1 = -f(x1) / (x2 - x1) or x2 = x1 - (x1 * x1 - a) / (2 * x1)
We then repeat setting x1 to be x2 and x2 to be the new, even better, guess. **When do we stop?**

Option1: is to stop when the difference in the guesses is really small: x1 - x2 < 1e-10.

Option2: is to stop when the best guess is really close the the square root: x1 * x1 - a < 1e-10.

A good option is to stop when the "relative error" is small instead of the "absolute error" (which is what we are doing). This is a good situation for a while loop because the increment is complicated. Here is the code:

```
public static double squareRoot(double a) {
        double x1 = ???;        // the square root guess of the previous iteration
        double x2 = ???;        // the best current guess for the square root
        while (x1 - x2 > 1e-10) {
                x1 = x2;
                x2 = x1 - (x1 * x1 - a) / (2 * x1);
        }
        return x2;
}
```

What should the initial x1 and x2 be? We should set x2 to be the initial "guess" for the squareroot and set x1 to be larger. If you know how Newton's method work, the initial guess does not have to be that good, just larger than the square root and not orders of magnitude away from the square root. So, using a as the initial value will work if a >= 1. If a < 1, you don't want to use a because then your guess will be smaller than the square root. Let us use a+1.

```
public static double squareRoot(double a) {
        double x1 = a+2;        // the square root guess of the previous iteration
        double x2 = a+1;        // the best current guess for the square root
        while (x1 - x2 > 1e-10) {
                x1 = x2;
                x2 = x1 - (x1 * x1 - a) / (2 * x1);
        }
        return x2;
}
```

It is possible to shorten the code. Can you see why this works?

```
public static double squareRoot(double a) {
        double x = a+1;         // the square root guess of the previous iteration
        while (x * x - a > 1e-20) {
                x = x - (x * x - a) / (2 * x);
        }
        return x;
}
```

Also, note that you could also do:

```
public static double squareRoot(double a) {
        double x = a+1;         // the square root guess of the previous iteration
        while (x * x - a > 1e-20) {
                x = (x * x + a) / (2 * x);
        }
```

```
        return x;
}
```

Remember Strings:

Recall that String is a class that represents text. A String object can not be changed once created. Some useful methods and operators:

+: creates a new String that is the result of concatentating two Strings

length(): returns the number of characters in the String

charAt(5): returns the 6th character of the String. (In Java, the first character is at index 0.)

String s = "Hello" s.length() → returns 5 s.charAt(1) → returns 'e'
"".length() → returns 0 "Hello".charAt(0) → returns 'H' "".charAt(0) → Error
"Hello".charAt("Hello".length()) → Error because it asks for the index 5 character in a string of only 5 characters.
"Hello there".charAt(5) → returns ' ' ("hello" + "there").length() → returns 10

For Loop Example 1: Write a method that determines if a String is a palindrome (the same forward as backwards). For example, "radar" is a palindrome. The trick is to first think how YOU would do it. What are the different steps you need?

- You read the string backwards and compare with how it is read forwards.
- That means you first compare the first character with the last, then the 2nd with the 2nd to last, and so on.
- So, we need two variables: one to keep track of where we are at the start of the string, and one to keep track of were we are at the end of the string:

```
        for (int first = 0, last = s.length() - 1; ???; first = first + 1, last = last - 1)
```

What test do we need?

```
        if (s.charAt(first) == s.charAt(last))
          // then keep going
        else
            // it can't be a palindrome
```

There is nothing to do if we need to keep going. The index as to be increased, but that is being done by the increment statement if the for loop. So, our code so far is:

```
public static boolean isPalindrome(String s) {
        for (int first = 0, last = s.length() - 1; ???; first = first + 1, last = last - 1)
                if (s.charAt(first) == s.charAt(last))
                        ; // then keep going
                else
                        return false;    // it can't be a palindrome
```

The empty statement is a little awkward. Sometimes, empty statements are not easy to avoid. Here we can easily avoid it by switching the then and else statements and doing the opposite condition. Since the else is an empty statement, we can drop it.

```
public static boolean isPalindrome(String s) {
        for (int first = 0, last = s.length() - 1; ???; first = first + 1, last = last - 1)
```

```
            if (s.charAt(first) != s.charAt(last))
                return false;    // it can't be a palindrome
```

What is the goal we want to achieve when we get to the end of the loop?

We want our loop to catch every time that s is not a palindrome, so if we get all the way to the end of the loop without returning, we know that s must be a palindrome.

```
public static boolean isPalindrome(String s) {
    for (int first = 0, last = s.length() - 1; ???; first = first + 1, last = last - 1){
        if (s.charAt(first) != s.charAt(last))
            return false;    // it can't be a palindrome
     }
     return true;        // the loop completed without finding a mismatch, so s is a palindrome
}
```

So, what is our subgoal for each iteration of the loop?

- This is what we want to have completed once we finish the loop body and the increment statements.
- For this loop, we want the characters of s from index 0 to first - 1 to exactly match the characters from s.length - 1 down to last + 1. (Do you see why we have the +1 and -1?)

This logic lets us see when we are supposed to stop the loop.

- If we stop when first == last (and the string has odd length) then we will have tested all but the very center character.
- If we stop when first == last + 1 (and the string has even length) then we will have tested every character.

So, we stop when first >= last, and our loop condition is the opposite: first < last.

```
public static boolean isPalindrome(String s) {
    // compare the characters at 0..(first-1) against the characters (length-1)..(last+1)
    // and exit the method if we find a mismatch
    for (int first = 0, last = s.length() - 1; first < last; first = first + 1, last = last - 1)
        if (s.charAt(first) != s.charAt(last))
        return false;    // it can't be a palindrome
    }
    return true;     // the loop completed without finding a mismatch, so s is a palindrome
}
```

Note that we only need one variable to keep track of which characters we are examining instead of two. Here is the same method, but using a single index variable instead of two.

```
public static boolean isPalindrome(String s) {
// compare the first "offset" characters against the last "offset" characters, exit the method if we find a mismatch
    for (int offset = 0; offset < s.length() / 2; offset = offset + 1)
        if (s.charAt(offset) != s.charAt(s.length() - 1 - offset))
            return false;    // it can't be a palindrome
    }
    return true;        // the loop completed without finding a mismatch, so s is a palindrome
}
```

NOTE Creating Strings:
When building a string using a loop, we may be tempted to use the + operator.

```
String result = "";
for (....) {
  ....
  result = result + c;
  ....
}
return result;
```

However, this is not a good solution. **The + operator creates a new String each time**. If we are dealing with a very long String, such as if we want to capitalize a DNA code of millions of characters, we will be creating a LOT of unnecessary Strings and using up our memory.

Java provides a StringBuilder class to create Strings. **StringBuilder** has all the same methods as String (charAt, length, etc) plus several others. Once useful one is **append** that adds new characters (or other values) to the end of the string being created.

How to create an empty StringBuilder? Just like you create any other initial instance:
StringBuilder result = new StringBuilder();

What do we return at the end? We can't return result because it is not a String. But Object has a method that returns a String representation, and every class inherits it from Object.

```
result.toString();
```

For Loop Example 2: Write a method that capitalizes every lower case letter in a string.

How would we do it? Look at each character one at a time.
- If it is lower case, capitalize it.
- If it is not lower case, keep it.

Note that we can not change the String so we must create a new String that is the same as the original, but with capital letters.
Finally, how to we test if letters are lower case, and how to make them upper case?
- Remember that char is a primitive type, and a char is just a number.
- Let us treat them as numbers.

That means we can test to see if the "number" of the letter is in the proper range.
- We can also use math to "shift" a value from one range to another.
- We must remember that that applying a primitive arithmetic operator will result in an int type so we will need to remember to typecast back to char.

(It is also possible to do this using built-in Java methods, but the math trick will work in a lot of languages.)

```
public static String capitalize(String s) {
        StringBuilder builder = new StringBuilder();
        for (int index = 0; index < s.length(); index = index + 1) {
```

```
            char c = s.charAt(index);
            if (c >= 'a' && c <= 'z')
                c = (char)(c - 'a' + 'A');
            builder.append(c);
        }
        return builder.toString();
}
```

Arrays:

An array is a collection of variables of the same type, stored in a contiguous chunk of memory. We can create an array to hold variables of any type:

int[] array1 → array1 is an array that will store ints
JFrame[] array2 → array2 is an array that will store JFrames
double[] array3 → array3 is an array that will store doubles
double[][] array4 → array4 is an array that will store double[]. That is, it is an array that stores arrays that store doubles.

→We can initialize the array using the new operator and stating how many variables (elements) will be in the array:

```
array1 = new int[100];
```

array1 now stores the address for an array that contains 100 int variables.

```
array2 = new JFrame[30];
```

array2 now stores the address for an array that contains 30 JFrame variables.

→Now, to access each element, we again use the square brackets. To store a value in the first element (variable) of the array:

```
array1[0] = 12;
```

NOTE: Remember, the we start counting from 0 in Java!

Each element of an array is a variable of the given type. So, all the rules of variables apply:
array1[2] = 3.15 ← Illegal! We are trying to store a double in a variable of type int
array1[2] = 7 ← Legal!
array1[2] = 'c' ← Legal! char is narrower than int so Java will automatically widen the 'c' to int
array1[2] = array3[2] ← Illegal! We are trying to store the value from a double variable into an int variable
array1[2] = (int)array3[2] ← Now it is legal because of the typecast.

Array disadvantage: Because the array is stored in contiguous memory, we cannot change the length (number of variables) of the array once it is created.

Array advantage: Because the array is stored in contiguous memory, we can access each element in a single step, no matter how large:

int[] a = new int[1000000];
For example, to access a[95436], we do not need to run through the array to the 95436'th entry.

We know the address of a (it is stored in the variable a), we know the byte size of the type of the array (for example, each int is 4 bytes) and we know which one we want (index 95436). So the address of a[95436] is: (address of a) + 4 * 95436

Loops and Arrays:

How do we fill an array? → Usually with a loop.

JFrame[] frames; // frames will store (the location to) an array that stores JFrames
frames = new JFrame[30]; // now frames has the address of a chunk of memory divided into 30 variables of type JFrame.
But no JFrame is yet stored in frames. To place a separate JFrame in each one:

```
for (int index = 0; index < frames.length; index = index + 1) {
        frames[index] = new JFrame();
}
```

If you have a small array, then you don't need a loop. You can enter the elements on at a time, or you can use an "array constructor" shortcut.

Our first method with arrays: Create a method that reverses the contents of an array**.** Each slot in the array is just a variable and so we will use variable assignment. We will need one extra variable to store values so we do not lose any values.

1. save the value in the back of the array, array[array.length - 1 - index]

2. store the value at the front of the array array[index] into the back of the array array[array.length - 1 - index]
- this overwrites the value in array[array.length - 1 - index]. Good thing we saved it!

3. store the saved value into array[index]

When do we stop the loop? You might be tempted to say when index reaches the end (array.length), but that is not correct. Can you see why?

```
public static void reverse(int[] array) {
        for (int index = 0; index < length / 2; index = index + 1) {
                int save = array[array.length - 1 - index];
                array[array.length - 1 - index] = array[index];
                array[index] = save;
         }
}
```

More on Arrays: "increasing" the size of an array

Create a method that takes an int array and an int. The array is full, but we need to put the int value at the end of the array. We can't change the size of an array, so instead, we must create a brand-new array, copy the values over, and then put the new value at the end.

```
public static int[] append(int[] array, int x) {
        int[] newarray = new int[array.length + 1];
        for (int i = 0; i < array.length; i = i + 1)
```

```
            newarray[i] = array[i];
        newarray[newarray.length - 1] = x;
        return newarray;
}
```

First, note that we needed a loop to do the copy. If we write "newarray = array;" we instead copy the address of array to newarray. That has the effect of having both newarray and array point to the same array.
Second, note that we copied x in outside the loop. We could have tried to also include x inside the loop, but that would mean an if statement and more complicated code. To keep your code simple, remember to have each loop accomplish one task.
Third, note that we had to return newarray. If we did not, we would lose the newarray. newarray is a local variable and so it is lost once the method ends. When the method ends, all local variables and method parameters are removed from memory. If we lose newarray, we lose the address to the new array. By returning the new address, we can have it outside the method.

Another incorrect assumption is to have an assignment statement: array = newarray. This would copy the address of newarray to array, but it has the same problem because array is a method parameter. Thus, array will be lost once the method ends.
Example:
myArray = append(myArray, 11); → will create a new array one larger than myArray, copy the data over, add 11 to the end, and then return the address of the new array, and that address is stored in myArray. myArray now points to a new array one larger than the array it originally pointed to. The old array is still in memory. Java will eventually de-allocate it if there are no other variables storing its address.

<u>Linear Searching in an Array:</u>
Here is a standard loop for searching for an item in an array:

```
public static int linearSearch(int x, int[] array) {
        for (int i = 0; i < array.length; i++) {
                if (x == array[i])
                        return i;
        } // at this point i = array.length so we checked all entries of array
        return -1;
}
```

What is the subgoal of the loop: It is what must be true after each iteration for the loop to continue: "x is not in the first i elements of array"

Is this the best search method we can write?
In the worst case, how many array locations does it inspect? →All of them. So, in the worst case, the time the method takes will be proportional to the length of the array.
On average, it will need to check half the elements in the array. So, in the average case the time the method takes is also proportional to the length of the array.

Can we write a faster loop? No! What is the proof? No matter how we decide to run through the array, until we look at every location, we can not be sure if x is not in the array. So, unless we know more information about the array, our linearSearch method is optimal.

Searching in a sorted array: Suppse the array is sorted in non-decreasing order. Now can we write a faster method? Proposed Algorithm (Binary Search)

1. Examine the middle element.
2. If the middle element is smaller than x, repeat on the upper half of the array.
3. If the middle element is larger than x, repeat on the bottom half of the array.

This algorithm looks more complicated to code, but is it fundamentally faster? We start with a list of n elements, and each time, we cut the number of elements we are considering in half.
n → n/2 → n/4 → n/8 → ... → 1
So, the number of iterations (and the number of elements of the array) is k where n / (2^k)) = 1.
k = log(base 2) n

How much an improvement is this over linear search?

Array size	#elements linear search check	#element binary search check
8	8	3
1000	1000	10
1,000,000	1,000,000	20
1,000,000,000	1,000,000,000	30
1,000,000,000,000,000	1,000,000,000,000,000	50

So, it is very worth it to write this method. Here is the first attempt we made in class. We are using a while loop because we do not know, at first, how many times we will need to iterate.

Binary Search, More on Reasoning About Loops
Suppose the array is sorted in non-decreasing order. Now can we write a faster method?
Proposed Algorithm (Binary Search):

1. Examine the middle element.
2. If the middle element is smaller than x, repeat on the upper half of the array.
3. If the middle element is larger than x, repeat on the bottom half of the array.

Here is our first incorrect attempt:

```
/** Return the index of x in a or -1 if x is not in array.
* Precondition: a is sorted in non-decreasing order */
public static int binarySearch(int x, int[] array) {
    int front = 0;        // stores the smallest index of array in the region still being considered
    int back  = array.length - 1; // the largest index of array in the region still being considered
    while (front != back) {
        int mid = (front + back) / 2;
        if (x < array[mid])
            back = mid;
        else if (x > array[mid])
```

```
            front = mid;
        else // only other case is x == array[mid]
            return mid;
    }
    return -1;
}
```

Loops are tricky to get completely correct so it is not surprising that our first attempt is not quite correct. Because of that, computer scientists have developed mathematical logic to reason about loops.
First, we have to decide what the goal of our loop is. That is something we -should- be able to do since we are writing the code. (Note, this is another reason why we want our loops to do just one task. It makes describing the goal easier.)
Given the goal for the loop, we must next determine what the -subgoal- for each loop iteration is. The subgoal (in formal math terms the subgoal is called the "loop invariant") is what we need to accomplish with each iteration of the loop in order to achieve the goal. Verifying that a loop is correct means that we verify that we are able to achieve the subgoal after each iteration, and we verify that the loop will eventually stop.

A correct subgoal has the following properties:
a) The subgoal is true after each iteration of the loop.
b) The subgoal is true just before the first iteration of the loop. (Technically, this can be thought of as the same as (a) by saying that if you have not entered the loop, then you are at the end of the "0th" iteration.)
c) The subgoal AND the loop condition being false logically implies the goal of the looo.

Let us check the reasoning of our loop.
1) What is the goal: At the end of the loop, x should not be in array. (Because if we reached the end of the loop, we did not return x.)
2) What is the subgoal for each loop iteration: After each iteration, if x is in array, it is in array[front...back] (i.e. it is between front and back, inclusive).

Now let us verify the subgoal. Recall the three properties we need:
a) The subgoal is true at the end of each loop iteration.
b) The subgoal is true immediately before the first iteration of the loop starts.
c) When the loop condition becomes false, the subgoal and the condition being false logically implies the goal.

Property b is easy to verify. Before we start the loop, front = 0 and back = array.length - 1. Of course, if x is in array it will be in array[0..length-1].
Property c is does not hold in our loop: When the loop condition becomes false, the subgoal and the condition being false implies the goal. The loop condition becomes false when front == back.

The subgoal is now: "If x is in array, it is in array[front...front]"
That does NOT logically imply the loop goal! x could still be in array if x is at array[front].

Using the logic reasoning, we see that our loop is incorrect. Maybe thorough testing would have found this problem, but the error will only occur in specific situations (do you see why?). Let us fix the method by changing the loop condition so that when it is false we do get the loop goal:

while (front <= back) {
Now, when the loop stops, back < front and so the loop subgoal becomes "if x is in array, it is in array[front..(front-1)]", and an array where the first index is larger than the last index is mathematically empty.

Next, we will check point (b): the subgoal must be true after each iteration. The way you verify this is check that if the loop subgoal is true at the start of the iteration, then it is still true at the end of the iteration. See if you can reason through the code and verify that we always achieve the loop subgoal.

Are we done? We verified the loop subgoal so the logic of our loop is correct, but we still must verify that the loop terminates. Having a logically correct loop does no good if the loop runs forever.
Are we guaranteed that the loop will terminate, or could it run forever? To guarantee that it does not run forever, we have to show that the gap between front and back decreases with each iteration. Thus, either front increases or back decreases. Is it possible to have a situation where front and back to do not change? YES!! Consider front = 5 and back = 6 and the element we are looking for is at index 6. Note that in this situation, mid = 5, array[mid] is smaller than the element, and so front = mid = 5. front never changed! **What is our fix?** To note that we know that the element is not at the array[mid] and so we will not include it. Since we know that front <= mid <= back, setting front = mid + 1 and back = mid - 1 guarantees that either front is increased or back is decreased on each iteration.

Here is the corrected code:

```
/** Return the index of x in a or -1 if x is not in array.
* Precondition: a is sorted in non-decreasing order */
public static int binarySearch(int x, int[] array) {
        int front = 0;   // stores the smallest index of array in the region still being considered
        int back = array.length - 1;  //the largest index of array in the region still being considered
        while (front <= back) {
                int mid = (front + back) / 2;
                if (x < array[mid])
                        back = mid - 1;
                else if (x > array[mid])
                        front = mid + 1;
                else // only other case is x == array[mid]
                        return mid;
        }
        return -1;
}
```

The moral: using the formal logic reasoning helps us design correct loops before we code them into the program.

Loop Testing

How to test loops?
Here is a good "rule of thumb" to follow to help you catch most of the usually bugs that can show up in loops:

1) Test 0, test 1, test many
2) Test first, test middle, test last

Consider the binarySearch method.
What does "test 0, tests 1, test many" mean?
One interpretation is test arrays of length 0, 1, and many. What does "test first, test middle, test last" mean? One interpretation is to test where we are searching for an element in at the front of the array, an element at the back, and an element somewhere in the middle.

Another interpretation is to test where we find the element quickly (the first iteration of the loop), and one where the loop runs to the end (the element is not in the array). So, this guide suggests that we have to test the following array lengths: 0, 1, and large, and we have to search for the middle element, elements not the middle, the element at the front, the element at the end, and elements that are not in the array: one smaller than all elements in the array, one larger than all elements in the array, and one that falls between the largest and smallest.

Java Shortcuts:

1) **Binary Operator shortcuts:** This shortcut works for all binary operators
   i += 3 ← shorthand for i = i + 3
   x /= 10 ← shorthand for x = x / 10

2) **Conditional Operator:** condition ? value1 : value2
   - if condition is true, the result is value1. Otherwise the result is value2

Instead of

```
if (x <= 10)
        y = 5;
else
        y = -5;
```

We can write: y = (x <= 10) ? 5 : -5;

3) **Switch Statements:** Multiple if statements that test equality of values:

if (month == 1)
  days = 31;
else if (month == 2)
  days = 28;
else if (month == 3)
  days = 31;
 	
switch (month) {
	switch (month) {
	  case 1:
	  case 3:
	  case 5:
	  case 7:
	  case 8:
	  case 10:
	  case 12:

```
        case 1:
            days = 31;
            break;
        case 2:
            days = 28;
            break;
```

```
            days = 31;
            break;
        case 2:
            days = 28;
            break;
        default:
            days = 30;
            break;
}
```

Note that break is used to exit execution. If a break is omitted on a case, execution "falls through" to the next case.

Important: the switch statement uses == to test equality. This works well with primitive, and it can work with objects -as long as you are testing whether two objects are the same address (Exception, starting in Java 7, the case will test the contents of String).

4) Increment and Decrement Shortcuts

i++ ← increments i and then returns the original value of i
++i ← increments i by 1 and then returns the new value of i
i-- ← decrements i and then returns the original value of i
--i ← decrements i by 1 and then returns the new value of i

Be careful that you use the ++ shortcut correctly! i++ and ++i have different effects.

Ex:
```
i = 5;
i = ++i;
```
1. i is incremented to 6
2. the value of i is returned and stored into i
The result is that i stores 6.

```
i = 5;
i = i++;
```
1. The value of i is returned and remembered for when the right side is done evaluating
2. The value of i is incremented to 6
3. The original remembered value of i is stored into i.
The result is that i stored 5.

5) Increment and loops:

```
for (int i = 0; i < a.length; i++)
    a[i] = i;

for (int i = 0; i < a.length; ++i)
    a[i] = i;
```
Both of the above loops do the same thing. a is assigned {0, 1, 2, 3, ...}

```
int i = 0;
while (i < a.length)
    a[i++] = i;

int i = 0;
while (i < a.length)
    a[++i] = i;
```

These two loops are different. The first assigns to a {1, 2, 3, ...}. The second skips over the first element to give {*, 1, 2, 3, ...} but then crash when it tries to store a value in a[a.length].

Here are two more

```
int i = 0;
while (i < a.length)
    a[i] = ++i;

int i = 0;
while (i < a.length)
    a[i] = i++;
```

The first assigns to a {1, 2, 3, 4, ...} and the second assigns to a {0, 1, 2, 3, ...}

What does this do?

```
int i = 0;
while (i < a.length-1)
    a[i] += ++a[++i];
```