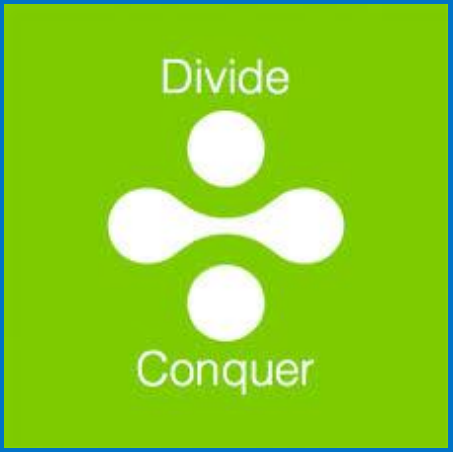
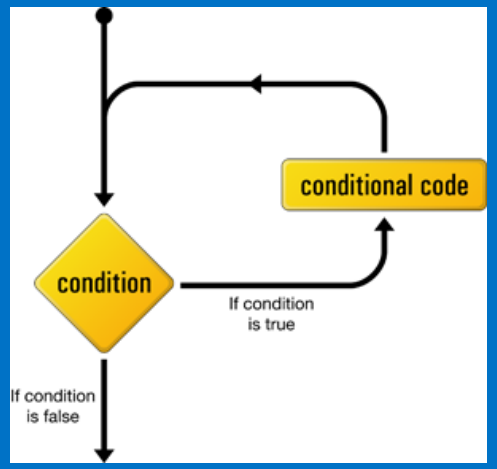




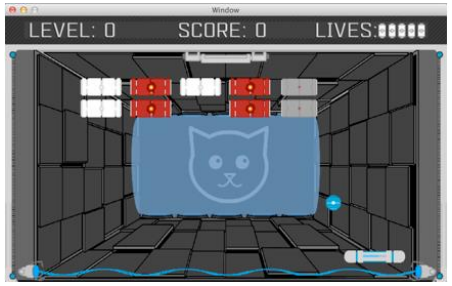
CSS 142

Lecture 8

aretik@uw.edu

TODAY'S CONTENT

1. **Midterm | Solutions | Feedback**
2. **Using and Debugging Loops**
3. **Pseudocode | Algorithms**
4. **Assertions | Random Generator**
5. **Coding Style and Commenting**



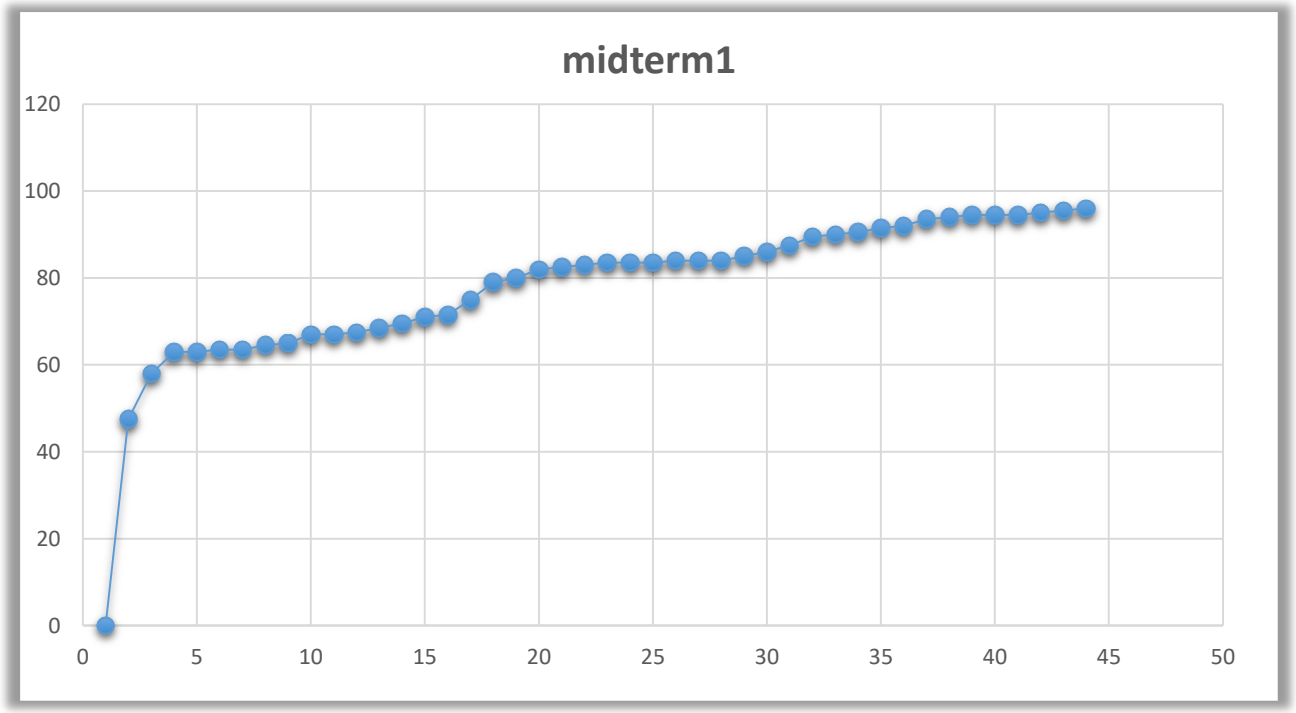
NEXT:

- **HW4: due end of Friday**
- **Wedn Reading 9.1, 10.1, 10.2**
- **Mon Reading 4.1 Classes**

EXAM:
review solutions

+ see a separate document

Midterm 1: feedback



Average Score:	**77.81**
High Score:	**96**
Median Score:	**82.5**

Midterm 1: feedback

Review solution

Key mistakes:

- `for` loops
- Infinite loops
- Booleans (Q12)

More on Wedn

Lexicographic

In mathematics, the **lexicographic** or **lexicographical** order (also known as lexical order, dictionary order, alphabetical order) is a generalization of the way the alphabetical order of words is based on the alphabetical order of their component letters.

The Java String class provides the .compareTo () method in order to lexicographically compare Strings. It is used like this > "apple".compareTo ("banana").

The return of this method is an int which can be interpreted as follows:

returns < 0 then the String calling the method is lexicographically first (comes first in a dictionary)

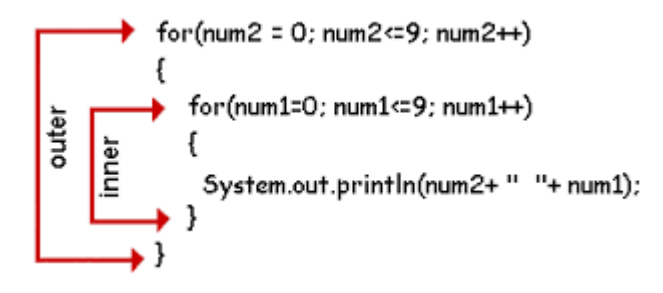
returns == 0 then the two strings are lexicographically equivalent

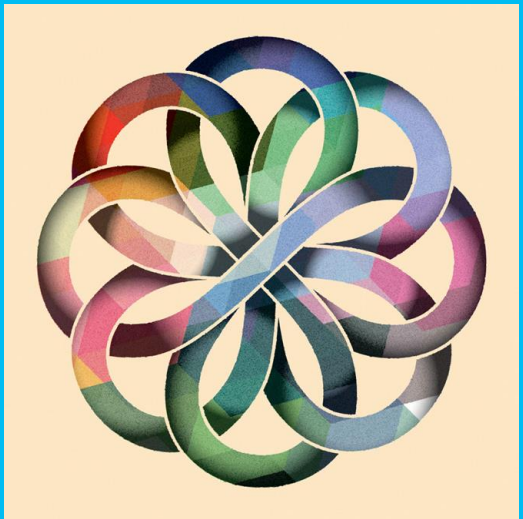
returns > 0 then the parameter passed to the compareTo method is lexicographically first.

More specifically, the method provides the first non-zero difference in ASCII values.

Thus "computer".compareTo ("comparison") will return a value of (int) 'u' - (int) 'a' (21). Since this is a positive result, the parameter ("comparison") is lexicographically first.

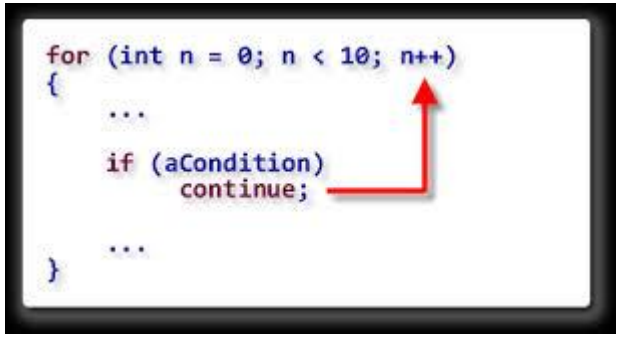
USING LOOPS


for(num2 = 0; num2<=9; num2++)
{
for(num1=0; num1<=9; num1++)
{
System.out.println(num2+ " "+ num1);
}
}
outer
inner

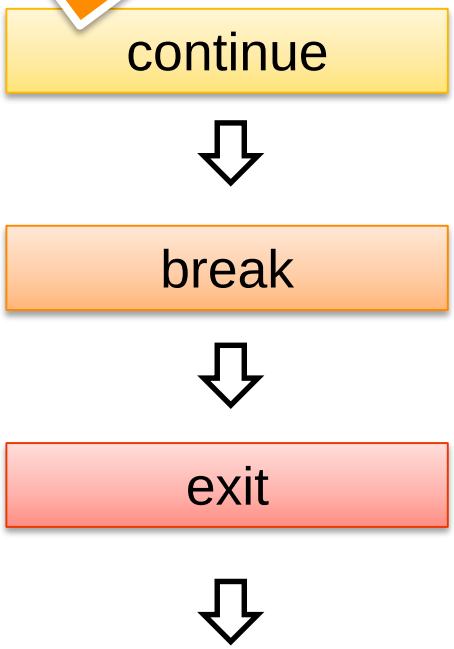


The `break` and `continue` Statements

- The `break` statement consists of the keyword `break` followed by a semicolon
  - When executed, the `break` statement ends the nearest enclosing switch or loop statement
- The `continue` statement consists of the keyword `continue` followed by a semicolon
  - When executed, the `continue` statement ends the current loop body iteration of the nearest enclosing loop statement
  - Note that in a `for` loop, the `continue` statement transfers control to the *update* expression
- When loop statements are nested, remember that any `break` or continue statement applies to the **innermost,** containing loop statement



Recommended not to use **continue;**



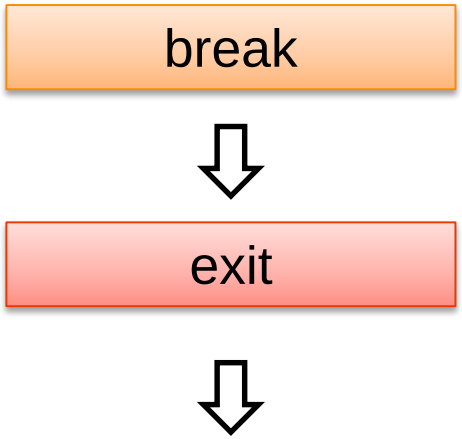
The `exit` Statement

- A `break` statement will end a loop or switch statement, but will not end the program

```
Initialization;
while (condition)
   {
   Statement 1 ;
   Statement 2 ;
   Statement 3 ;
   ..........
   ..........
   if ( If Condition)
     break;

   Statement N-1 ;
   Statement N ;
   Increment;
   }

OutsideStatement 1;
```



- The `exit` statement will immediately end the program as soon as it is invoked:
  - `System.exit(0);`
- The `exit` statement takes one integer argument
  - By tradition, a zero argument is used to indicate a normal ending of the program

The Labeled `break` Statement

- There is a type of `break` statement that, when used in nested loops, can end any containing loop, not just the innermost loop
- If an enclosing loop statement is labeled with an *Identifier,* then the following version of the break statement will exit the labeled loop, even if it is not the innermost enclosing loop:

```
break someIdentifier;
```

- To label a loop, precede it with an *Identifier* and a colon:

```
someIdentifier:
```

```
topLoop:
 do
    {….
  while …..
       if ….
              for….
                 if …

                    break topLoop;
                   ……
       }
```

Algorithms

a process or set of rules to be followed in calculations or other problem-solving operations, especially by a computer.

Algorithms and Pseudocode

- The hard part of **solving a problem** with a computer program is not dealing with the syntax rules of a programming language
- Rather, coming up with the underlying **solution method** is the most difficult part
- An ***algorithm*** is a set of precise instructions that lead to a solution
  - An algorithm is normally written in ***pseudocode***, which is a **mixture of programming language and a human language**, like English
  - Pseudocode must be precise and clear enough so that a good programmer can convert it to syntactically correct code
  - However, pseudocode is much less rigid than code: One needn't worry about the fine points of syntax or declaring variables, for example

Examples

Pseudocode

- For example, for making a cup of tea:

```
Organise everything together;
Plug in kettle;
Put teabag in cup;
Put water into kettle;
Wait for kettle to boil;
Add water to cup;
Remove teabag with spoon/fork;
Add milk and/or sugar;
Serve;
```

Pseudocode to Calculate the Sum & Average fo 10 Numbers

```
begin
        initialize counter to 0
        initialize accumulator to 0
        loop
                read input from keyboard
                accumulate input
                increment counter
        while counter < 10
        calculate average
        print sum
        print average
end
```

Exercise

1. Write a pseudocode to calculate the sum of numbers from 1 to x.
X to be provided by a user

Print the results

2. Implement the sum calculation in Java

5 mins; work in pairs

Exercise

1. Write a pseudocode to calculate the sum of numbers from 1 to x. x to be provided by a user

Print the results

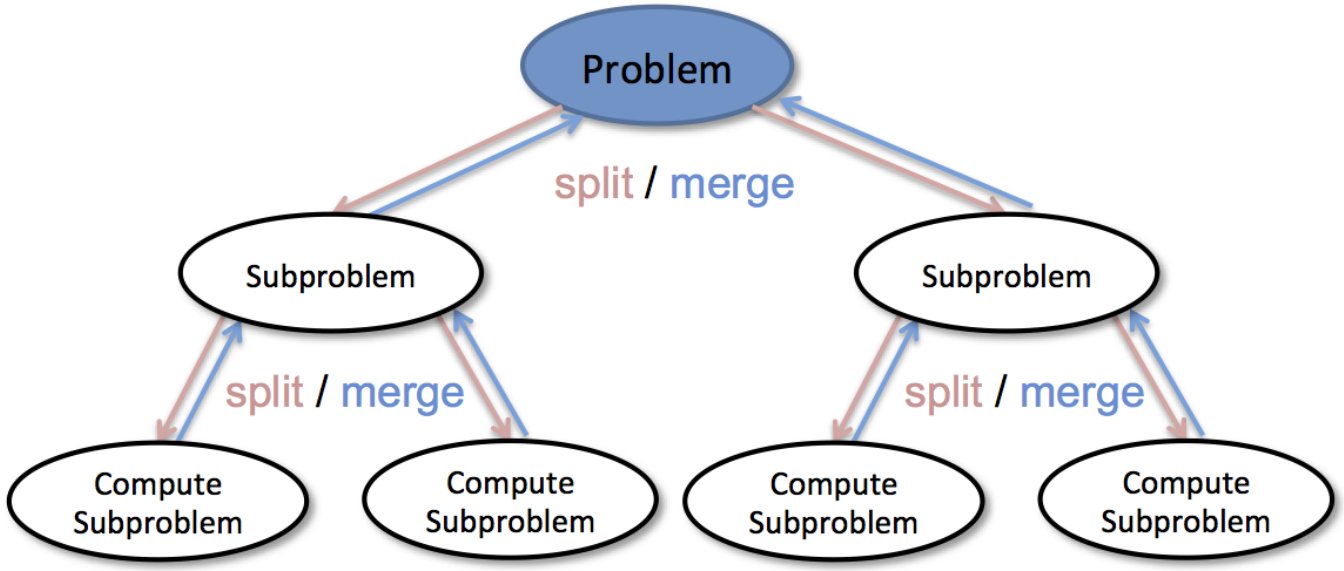
2. Implement the sum calculation in Java

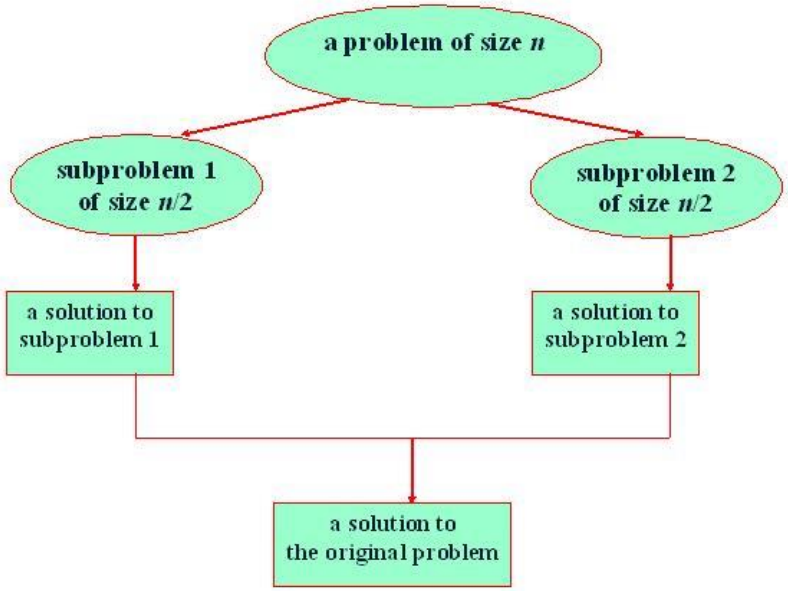
Pseudocode to Calculate the Sum & Average fo 10 Numbers

```
begin
        initialize counter to 0
        initialize accumulator to 0
        loop
                read input from keyboard
                accumulate input
                increment counter
        while counter < 10
        calculate average
        print sum
        print average
end
```

What's an alternative?
Will you do the same way if it's 1M or 1B?

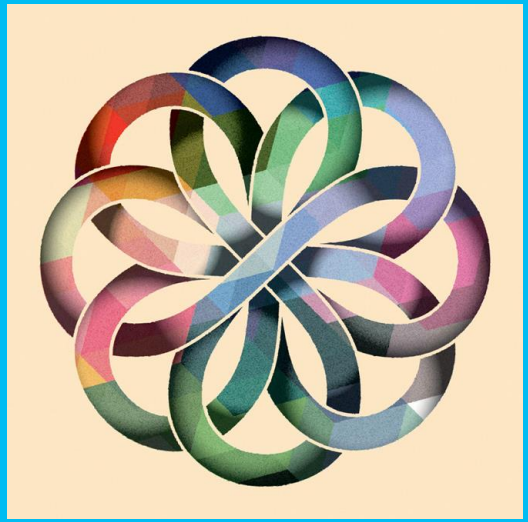
Divide and Conquer Algorithms





Example: **binary search**
We will look at that in arrays.

DEBUGGING LOOPS



See also Debugging week 2

Loop Bugs

- The two most common kinds of loop errors are unintended ***infinite*** *loops* and ***off-by-one*** *errors*
  - An off-by-one error is when a loop repeats the loop body one too many or one too few times
    - This usually results from a carelessly designed Boolean test expression
  - Use of **==** in the controlling Boolean expression can lead to an infinite loop or an off-by-one error
    - This sort of testing works only for characters and integers, and should never be used for floating-point

Tracing Variables

- ***Tracing variables*** involves watching one or more variables change value while a program is running
- This can make it easier to discover errors in a program and debug them
- Many *IDE*s (*Integrated Development Environments*) have a built-in utility that allows variables to be traced without making any changes to the program
- Another way to trace variables is to simply insert temporary output statements in a program

```
System.out.println("n = " + n);  // Tracing n
```

  - When the error is found and corrected, the trace statements can simply be commented out

```
// System.out.println("n = " + n);  // Tracing n
```

General Debugging Techniques

- Examine the system as a **whole** – don't assume the bug occurs in one particular place
- Try different **test cases** and **check the input values**
- **Comment out** blocks of code **to narrow down** the offending code
- Check common pitfalls
- **Take a break and come back later**
- **DO NOT** make random changes just hoping that the change will fix the problem!

Divide And Concur

Debugging Example (1 of 9)

- The following code is supposed to present a menu and get user input until either ‘a’ or ‘b’ is entered.

```
String s = "";
char c = ' ';
Scanner keyboard = new Scanner(System.in);

do
{
  System.out.println("Enter 'A' for option A or 'B' for option B.");
  s = keyboard.next();
  s.toLowerCase();
  c = s.substring(0,1);
}
while ((c != 'a') || (c != 'b'));
```

This and next 8 examples are from Savitch Absolute Java, 6e Chapter 3

Debugging Example (2 of 9)

Result: Syntax error:

```
c = s.substring(0,1);    : incompatible types
found:  java.lang.String
required: char
```

- Using the “random change” debugging technique we might try to change the data type of `c` to `String`, to make the types match
- This results in more errors since the rest of the code treats `c` like a `char`

- First problem: substring returns a String, use charAt to get the first character:

```
String s = "";
char c = ' ';
Scanner keyboard = new Scanner(System.in);

do
{
  System.out.println("Enter 'A' for option A or 'B' for option B.");
  s = keyboard.next();
  s.toLowerCase();
  c = s.charAt(0);
}
while ((c != 'a') || (c != 'b'));
```

Now the program compiles, but it is stuck in an infinite loop. Employ tracing:

```
do
{
   System.out.println("Enter 'A' for option A or 'B' for option B.");
   s = keyboard.next();
   System.out.println("String s = " + s);
   s.toLowerCase();
   System.out.println("Lowercase s = " + s);
   c = s.charAt(0);
   System.out.println("c = " + c);
}
while ((c != 'a') || (c != 'b'));
```

Sample output:

```
Enter 'A' for option A or 'B' for option B.
A
String s = A
Lowercase s = A
c = A
Enter 'A' for option A or 'B' for option B.
```

From tracing we can see that the string is never changed to lowercase. Reassign the lowercase string back to `s`.

- The following code is supposed to present a menu and get user input until either ‘a’ or ‘b’ is entered.

```
do
{
  System.out.println("Enter 'A' for option A or 'B' for option B.");
  s = keyboard.next();
  s = s.toLowerCase();
  c = s.charAt(0);
}
while ((c != 'a') || (c != 'b'));
```

However, it’s still stuck in an infinite loop. What to try next?

- Could try the following “patch”

```
do
{
   System.out.println("Enter 'A' for option A or 'B' for option B.");
   s = keyboard.next();
   s = s.toLowerCase();
   c = s.charAt(0);
   if ( c == 'a')
        break;
   if (c == 'b')
        break;
}
while ((c != 'a') || (c != 'b'));
```

This works, but it is ugly! Considered a coding atrocity, it doesn’t fix the underlying problem. The boolean condition after the while loop has also become meaningless. Try more tracing:

```
do
{
   System.out.println("Enter 'A' for option A or 'B' for option B.");
   s = keyboard.next();
   s = s.toLowerCase();
   c = s.charAt(0);
   System.out.println("c != 'a' is " + (c != 'a'));
   System.out.println("c != 'b' is " + (c != 'b'));
   System.out.println("(c != 'a') || (c != 'b')) is "
                 + ((c != 'a') || (c != 'b')));
}
while ((c != 'a') || (c != 'b'));
```

Sample output:

```
Enter 'A' for option A or 'B' for option B.
A
c != 'a' is false
c != 'b' is true
(c != 'a') || (c != 'b')) is true
```

From the trace we can see that the loop's boolean expression is true because `c` cannot be not equal to `'a'` and not equal to `'b'` at the same time.

- Fix: We use && instead of ||

```
do
{
  System.out.println("Enter 'A' for option A or 'B' for option B.");
  s = keyboard.next();
  s = s.toLowerCase();
  c = s.charAt(0);
}
while ((c != 'a') && (c != 'b'));
```

- Even better: Declare a boolean variable to control the do-while loop. This makes it clear when the loop exits if we pick a meaningful variable name.

```
boolean invalidKey;
do
{
   System.out.println("Enter 'A' for option A or 'B' for option B.");
   s = keyboard.next();
   s = s.toLowerCase();
   c = s.charAt(0);
   if (c == 'a')
         invalidKey = false;
   else if (c == 'b')
         invalidKey = false;
   else
         invalidKey = true;
}
while (invalidKey);
```

Assertion Checks

- An *assertion* is a sentence that says (asserts) something about the state of a program
  - An assertion must be either true or false, and should be true if a program is working properly
  - Assertions can be placed in a program as comments
- Java has a statement that can check if an assertion is true
  - `assert Boolean_Expression;`
  - If assertion checking is turned on and the `Boolean_Expression` evaluates to `false`, the program ends, and outputs an *assertion failed error message*
  - Otherwise, the program finishes execution normally

Assertion Checks

- A program or other class containing assertions is compiled in the usual way
- After compilation, a program can run with assertion checking turned on or turned off
  - Normally a program runs with assertion checking turned off
- In order to run a program with assertion checking turned on, use the following command (using the actual **`ProgramName`**):
  - `java -enableassertions ProgramName`

Preventive Coding

- **Incremental Development**
  - Write a little bit of code at a time and test it before moving on
- **Code Review**
  - Have others look at your code
- **Pair Programming**
  - Programming in a team, one typing while the other watches, and periodically switch roles

Generating Random Numbers

- The Random class can be used to generate pseudo-random numbers
  - Not truly random, but uniform distribution based on a mathematical function and good enough in most cases
- Add the following import
- ```
  import java.util.Random;
  ```
- Create an object of type Random
  - ```
    Random rnd = new Random();
    ```

Generating Random Numbers

- To generate random numbers use the nextInt() method to get a random number from 0 to n-1

```
int i = rnd.nextInt(10);    // Random number from 0 to 9
```

- Use the nextDouble() method to get a random number from 0 to 1 (always less than 1)

```
double d = rnd.nextDouble();    // d is >=0 and < 1
```

Simulating a Coin Flip

Display 3.11

```
import java.util.Random;
public class CoinFlipDemo
{
   public static void main(String[] args)
   {
      Random randomGenerator = new Random();
      int counter = 1;

      while (counter <= 5)
      {
         System.out.print("Flip number " + counter + ": ");
         int coinFlip = randomGenerator.nextInt(2);
         if (coinFlip == 1)
             System.out.println("Heads");
         else
             System.out.println("Tails");
         counter++;
      }
   }
}
```

Sample Dialogue (output will vary)

```
Flip number 1: Heads
Flip number 2: Tails
Flip number 3: Heads
Flip number 4: Heads
Flip number 5: Tails
```

Comments and Style

Some reminders:

Every file should have Javadoc comments with appropriate @tags.

- **@name**
- **@version**
- **@date**
- Javadoc comments start with "/**".
- Block comments start with "/*"
- No line of code should be longer than 80 characters.

also
see two documents

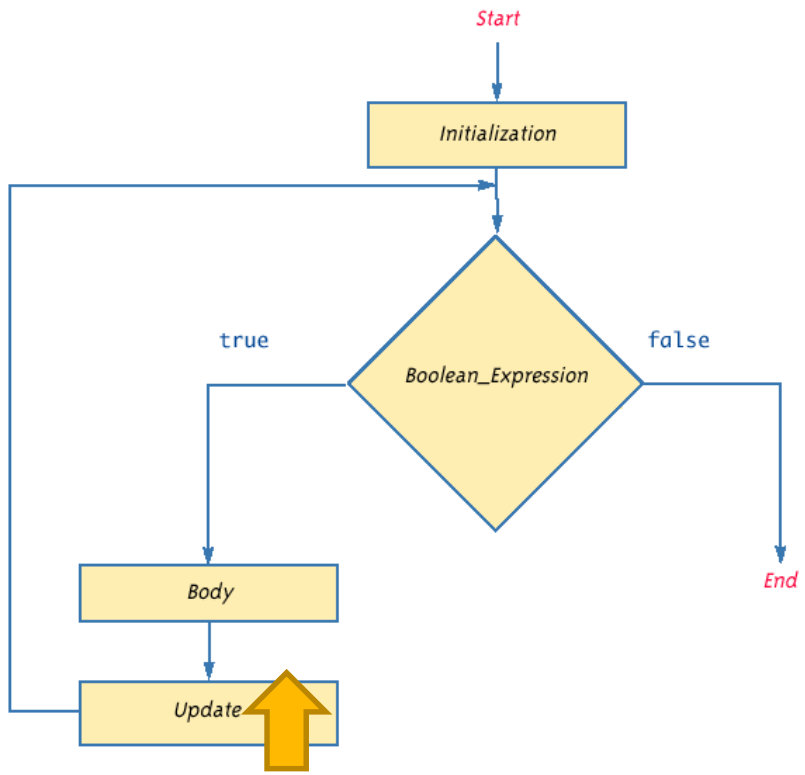
Appendix:

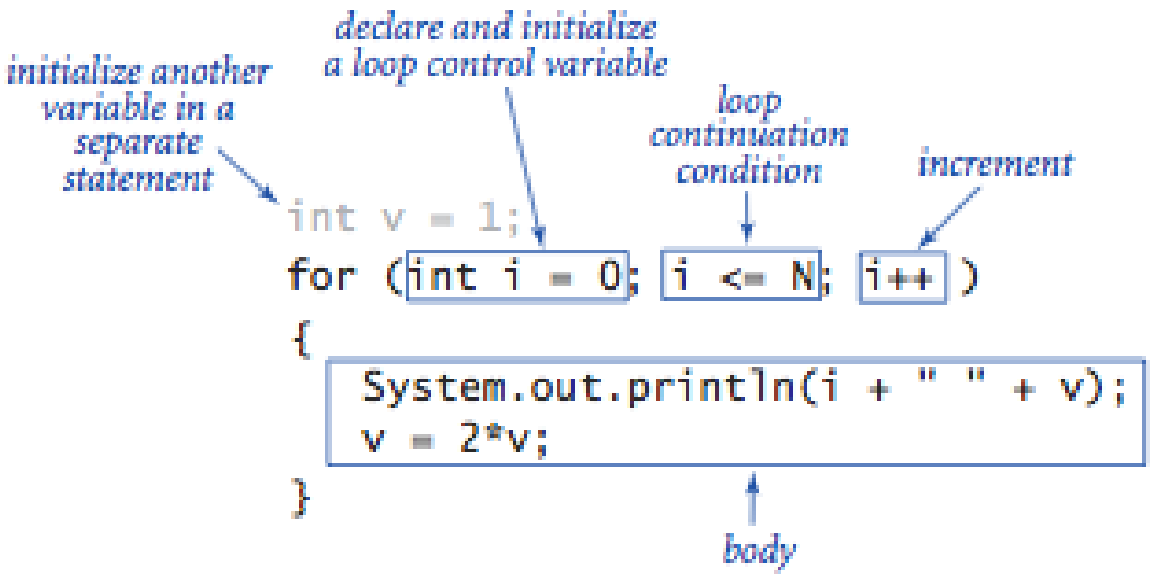
Recap on loops

Semantics of the `for` Statement

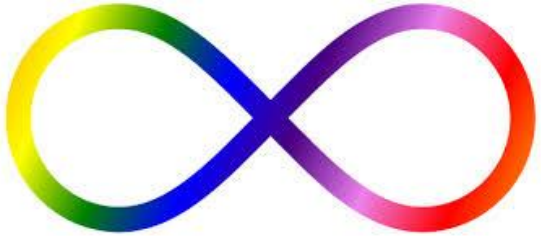
Display 3.9 Semantics of the for Statement

```
for (Initialization; Boolean_Expression; Update )
        Body
```





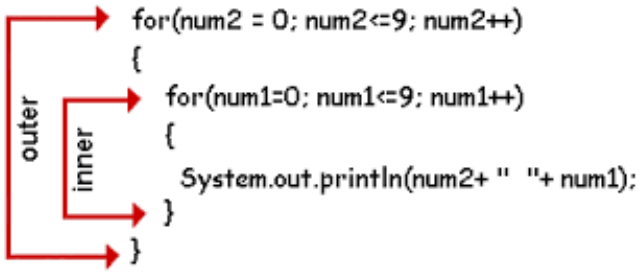
Infinite Loops



- A **`while`**, **`do-while`**, or **`for`** loop should be designed so that the value tested in the Boolean expression is changed in a way that eventually makes it false, and terminates the loop
- If the Boolean expression remains true, then the loop will run forever, resulting in an *infinite loop*
  - Loops that check for equality or inequality (**`==`** or **`!=`**) are especially prone to this error and should be avoided if possible

Why?

Nested Loops



- Loops can be *nested*, just like other Java structures
  - When nested, the **inner** loop iterates from beginning to end for each single iteration of the **outer** loop

```
int rowNum, columnNum;
for (rowNum = 1; rowNum <=3; rowNum++)
{
  for (columnNum = 1; columnNum <=2; columnNum++)
    System.out.print(" row " + rowNum + " column " + columnNum);
    System.out.println();
}
```

How many rows and columns?

Need to Traverse A String?

```
for(int r = 0; r < string.length(); r++)
{
        System.out.println( string.charAt(r));
}
```

Or, backwards…

```
for(int r = string.length() – 1; r >= 0; r--)
{
        System.out.println( string.charAt(r));
}
```

A Running Sum

```
int sum = 0;
for( int j = 0; j <= 20; j++ )
  {
      sum += j;   //same as sum = sum + j
  }
=================================
for( int k = 0; k <= 30; k++ )
 {
      if( k % 2 == 0 ) {
              sum = sum + k;
    }
 }
```

Home Work 4

Need to pay attention
to
Coding Style and
Commenting
(see docs)

Hands-on: Class Activity (HoA)

NEXT WEEK