**Lecture #1**

Variables: store information needed by the program

* Must have a type
  + int – stores numbers without a fractional part
  + float, double – stores any number with or without fraction, double being more precise
  + char – individual character (think letter, digit, punctuation)
  + String – text data
  + boolean – stores true or false
* Must have a value, but the value can change as the program runs
  + If not provided, primitive types are assigned as zero and null for strings

Literal values

* Every primitive type (and Strings) can be initialized using a literal of the type

|  |  |  |
| --- | --- | --- |
| Type | Literal value | Example initialization |
| String | “Hello World” | String greeting = “Hello World”; |
| char | ‘A’ | Char letter = ‘A’; |
| boolean | true | boolean flag = true; |
| int | 1024 | int kilo = 1024; |
| double | 3.154159 | double pi = 3.14159 |

* Note:
  + char literal is in single quotes while String literal is in double quotes
  + String can consist of a single character, but char cannot consist of multiple characters

Naming variables

* Must have a name
  + Start with alphabet or underscore
  + Can include alphabetics, digits, and underscores
  + Should indicate the purpose of the variable in the program
  + Strongly encourage variables to be lowercase
    - If multi-worded, make the first letter of the name lowercase and subsequent words start with a capitol letter
      * Ex: fileName, remoteUserAddress

Assignment

* The ‘=’ operator assigns value on the right, to the variable on the left
* Left hand side must be a variable
* Right hand side can be a variable or expression
  + Ex: int x = 0, y = 3, z = 4;

Named Constants

* Magic number: numeric constant that appears without explanation
  + Ex: answer = rate\*1.67; //significance of 1.67?
* Make it a named constant using the keyword “final”

**final** float RATE\_INCREASE = 1.67;

answer = rate\*RATE\_INCREASE;

* Value of a named constant cannot be assigned to another value
* Strongly encourages all capitols for constants
  + Multi-word constant names can have an underscore between words

Conversions

* Implicit conversion

float op float = float

float op int = float

int op float = float

int op int = int

Shortcuts

* x = x+5; is equivalent to x += 5;
* x = x\*10; is equivalent to x \*=10;
* x= x/10; is equivalent to x /=10;
* x= x%3; is equivalent to x %=10;
* x= x+1; is equivalent to
  + x++; //post increment
  + ++x; //pre increment
  + x +=1;

Console I/O

* Input from the standard input device and output to the standard output device
* Keyboard is the standard input device
* Screen/monitor is the standard output device
* Scanner class reads from the keyboard and text files
* System.out.print() or System.out.println() or System.out.format() to write text on the standard output device

double pi = 3.14159265358979323846264338; // to 26 places

// BUT only up to 15 places max can be stored

System.out.println( "pi=" + pi ); // prints 3.14159265358979323846264338

// use format() to force max width of 6 places counting dot with 4 coming after the dot. The 59 rounds to 6

System.out.format( "pi=%6.4f", pi ); // prints 3.1416

* System.in is the official name of the keyboard device

Scanner kbd = new Scanner(system.in);

System.out.print(“Enter your name: “);

String name = kbd.next(); //stop, wait until use types and hits return

System.out.println(“You enttered “ + name);

//don’t enter more than one token (spaces)

**Lecture #2**

Boolean type and its operators

boolean b = true; //stores the truth value true in b

b = false; //overwrites b with the value false

versus

int i = 10;

b = i<20; //where the expression i<20 is true

And: &&

Or: ||

Logical negative (not): !

Equality: ==

Inequality: !=

Less than: <

Less than or equal: <=

Greater than: >

Greater than or equal: >=

Short-circuiting happens when the result can be determined before the entire expression has been examined

DeMorgan’s law: any expression can be equivalently expressed by multiplying a NOT through the boolean expression and changing || to && or && to ||

Negation of a conjunction is the disjunction of the negations

!(p&&q) -> !p || !q

Negation of a disjunction is the conjunction of the negations

!(p||q) -> !p&&!q

If statements

“and” examples

if (age >6 && age <19) {

System.out.println(“You should be in School!”);

}

“or” examples

if (letter == ‘A’ || ‘B’ || ‘C’)

* Simple conditional: if

if (age<21) {

System.out.println(“too young to drink”);

}

* Two-way branch: if else

if (age<18) {

System.out.println(“too young to drink”);

}

else {

System.out.println(“draft or bottle?”);

}

* Three-way branch: if else/if else

if (age<18) {

System.out.println(“too young to drink”);

}

else if (age<70) {

System.out.println(“draft or bottle”);

}

else {

System.out.println(“How about some Geritol?”);

}

Negating the test

if (<boolean expression>) {

//nothing in the if part

}

else

do something

Also equal to

if (!<boolean expression>)

do something

**Lecture #3**

Loops

* Used to repeat an action
* Must have a stop condition
* For, while, do/while

While loops

1. Check test
2. If true,
   1. Execute the body
   2. When the body has finished, go to step 1
3. If false, exit loop

* Test is checked at the very beginning, and then each time after the entire loop body has been executed
* Technically just an if statement that keeps going back to the test and quits looping at first failure of the test

Do loop

* is a variant of the while that waits till the bottom to test
* Error checking; run the program first before determining if it passes the test

For loop

1. Execute the initiating statement
2. Execute the test
3. When the test is true
   1. Execute the body
   2. Execute the update
   3. Return to step 2
4. Otherwise when the test is false, exit the loop

* Nested for loops

**Lecture #4**

Arrays

* Collection of values of the same type stored consecutively in memory

int arr[] = new int[5];

* + Two-part declaration:
    - Right side of the assignment
      * “new”
        + Allocates chunk of memory for 5 ints
        + Brings back the address for that chunk of memory

Throws exception if out of memory

* + - Left side
      * int arr[]
        + Creates a reference variable named “arr” that contains the address of the first cell in the memory chunk

arr points to or references the beginning of the array, the [0] cell

* + - “arr” is the reference variable
    - The chunk of memory is the object
    - References point to objects
* Use “.length” to determine number of cells in the array

int arr[] = new int[5];

for (int i=0; i<arr.length; ++i;

arr[i] = i\*2; //fills the array to be [-]--> [0][2][4][6][8]

//println(arr.length) to show the number 5

* Array discipline
  + When declaring an array, declare an int named count to track number of values entered into the array
  + Initialize count to zero and use it to represent
    - Number of values put into the array so far
    - Index position of where the nect value should be stored
* Passing an array
  + Must be to a method that is written to receive an array

public static void main( String[] args ) {

int arrCnt = 0;

int arr[] = new int[5];  
for (arrCnt=0 ; arrCnt<arr.length ; arrCnt++)

arr[arrCnt] = arrCnt \* 2; printArray( arr, arrCnt);

}  
private static void printArray( int[] array, int cnt ) {

for (int i=0 ; i < cnt ; ++i System.out.println( array[i] );

System.out.println();

}

* + This is only passing a copy of the address of where the array starts, not a copy of the actual data values (passing reference)
    - Copying the actual array and sending it to the method would be very memory inefficient

Filling an array from a file

File

45 23 56 21 76 13

int[] arr = new int[10];  
int arrCnt= 0;  
while ( infile.hasNextInt() )

arr[arrCnt++] = infile.nextInt(); printArray( arr, arrCnt );

arrCnt is now 6. The last value is stored at arr[arrCnt-1]