Introduction to Programming

with Java, for Beginners

Debugging
Algorithmic Thinking
Scope

Debugging

- It is highly unlikely that you will write code that will work on the first go

- Bugs or errors
  - Syntax
    - Fixable if you learn to read compiler error messages
  - Semantic
    - No easy fix
      - Use print statements to our advantage

Syntax Error

- Use the Dr Java tool to your advantage
  - Keywords turn blue
  - Comments turn green
  - { } matching

- Reading compiler Errors
  - Turn on line numbers (In DrJava (Edit Preferences -> Display)
  - Learn common syntax errors
    - Missing Semicolon at end of a statement (line no. indicated)
    - Methods should be with class definition
    - For loop as three statements (each ending in semi-colon)
    - Using variable that are not assigned

Debugging with System.out.println

```java
public static int sumOdd(int n){
  //sum positive odd numbers upto n
  // e.g. sum(3) = 4
  int sum = 0;
  for (int i = 1; i <= n; i = i + 1){
    sum = sum + i;
    System.out.println(i + " : " + sum);
  }
  return sum;
}
```

Result of print

<table>
<thead>
<tr>
<th>i</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

Remember to comment out print statement when you are done testing
Example DRY Principle

```java
public static int sumOdd(int n){
    //sum positive odd numbers up to n
    // e.g. sum(3) = 4
    int sum = 0;
    for (int i = 1; i <= n; i = i + 1){
        if(isOdd(i)) {
            sum = sum + i;
        }
    }
    return sum;
}
```

Formulating a Solution

- First Think *Algorithmically*
  - Well defined step by step procedure
  - Use *psuedocode* to write out your steps
    - English like code
    - It allows the designer to focus on the logic of the algorithm without being distracted by details of language syntax
- Then *Translate* the solution into programming language
  - Put together the components we have so far
    - declarations, assignments, control structures

Example: Fibonacci sequences

- A Fibonacci sequence is an infinite list of integers
- The first two numbers are given
  - Usually (but not necessarily) these are 1 and 1
- Each subsequent *number* is the sum of the two preceding numbers:
  1 1 2 3 5 8 13 21 34 55 89 144 ...
- Let’s write instructions to compute the sequence as long a *number* less than 1000

Starting the Fibonacci sequence

- We need to initialize two numbers in the sequence
  - Set first to 1
  - Set second to 1
- We need to print these out:
  - Print first and second
- We need to compute and print the next number:
  - Set next to sum of first & second; print next
Taking the next step

- Now what?
  - Need to add second and next
    - nextnext = second + next
  - What if the sequence is too long
    - I do want to make 100s of variables to hold each item
- The sequence so far is: first second next
- Do I see a pattern emerging?

Preparing to make many steps

- We need to make these moves:

\[
\begin{array}{ccc}
\text{first} & \text{second} & \text{next} \\
1 & 1 & 2 \\
1 & 2 & 3 \\
2 & 3 & 5 \\
3 & 5 & 8 \\
\end{array}
\]

- We can do it like this:
  - Set first to second
  - Set second to next
- We can put these statements in a loop and do them as many times as we please

Complete Psuedocode

- Set first to 1
- Set second to 1
- Print first and second
- while next number < 1000
  - Set next to sum of first & second
  - print next
  - Set first to second
  - Set second to first next

Psuedocode Rules

- Can use words such as while, if else-if
  - E.g. for 1 to n
- Do not specify data declarations or types
- Use Words that specify an action such as set, reset, increment, compute, calculate, add, sum, multiply, print, getinput
- Use indentation for block of code i.e. { .. }

Translate it to Programming Language

```java
int first = 1;
int second = 1;
int next = 0;
System.out.print(first + " ");
System.out.print(second + " ");
while (next < 1000) {
    next = first + second;
    System.out.print(next + " ");
    first = second;
    second = next;
}
```

Scope

- **Scope** means the area of code in which an entity is known (or alive)
  - Mainly concerned with variables and methods
  - Which parts of the program can access them?

- Sometimes scope is explicitly designated with a keyword
  - private: known only within the class
  - public: known outside of (and within) the class
  - Note that Methods have explicit scope

- Other times it is implicitly designated by location

Implicit Scope: Method Parameters

- A method parameter is an “input variable”
- **Scope**: the method in which it is defined
- No other method can access (read/write) it

```java
public int absoluteValue(int n) {
    if (n < 0) {
        return -n;
    }
    else {
        return n;
    }
}
```

Implicit Scope: Local Variables

- A “local variable” is defined within a method body { }
  - They are inherently private to the method in which they are defined
  - We don’t use public/private for local variables

- It may be defined in a block { } within a method body

- **Scope**: point of declaration to end of closest enclosing block
Example of Local Variables

```java
public int isLarger(int x, int y) {
    if (x > y) {
        int larger = x;
    } else {
        int larger = y;
    }
    return larger;
}
```

Another Example

```java
int fibonacci(int limit) {
    int first = 1;
    int second = 1;
    while (first < limit) {
        System.out.print(first + " ");
        int next = first + second;
        first = second;
        second = next;
    }
    System.out.println();
}
```

For Loop Special Case

- The `for` loop is a special case
- You can declare variables in the `for` statement
- The scope of those variables is the entire `for` loop
- This is true even if the loop is not a block i.e. without `{ }

```java
for (int x = 1; x <= 10; x = x + 1){
    System.out.println(x);
}
```

Nested for loops

- Loop inside a loop
- Just like nested if statements
- After the inner loop count reaches the limit, the inner loop variable is re-initialized to initial value

```java
void multiplicationTable() {
    for (int i = 1; i <= 10; i++) {
        for (int j = 1; j <= 10; j++)
            System.out.print(" " + i * j);
        System.out.println();
    }
}
```
Return statements in loops

- Return statement should last statement before ending method
- Having return statements within loops will cause compiler to throw syntax error
  - This because the compiler does not know whether the statement is reachable or not
- Always use variable to store the value and then finally return that value

Pass by Value

- By default, copies of parameter are sent to a method

```java
public static void main(String[] args){
    int x = 0;
    System.out.println("In main: x = "+x);
    foo(x);
    System.out.println("In main: x = "+x);
}
```

Output:
```
In main: x = 0
In foo: x = 0
In foo: x = 5
In main: x = 0
```

```java
public static void foo(int x){
    System.out.println("In foo: x = "+x);
    x = 5;
    System.out.println("In foo: x = "+x);
}
```