Java Module 1 - Halfway Quick Notes

Types

Primitive Types

(Fast, have no methods, can use operators +/-=%)

```
int x = 4;                  // A basic whole number
long y = 9000000000;        // A large whole number
short z = 1000;             // A small whole number
byte b = 200;               // A very small whole number
float weight = 2.5;         // A decimal number
double height = 3.1415926;  // A large decimal numberm
boolean isValid = false;    // A true or false value
char input = 'F';           // A single character (letter, symbol, digit,
etc.)
```

String Type

(can use one operator + for string concat)

```
String name = "Mike";
```

String - **CONCATENATION**

```
String message = "Hello " + name;
```

String - **COMPARISON**

```
if (name.equals("Mike")) {
    // ...
}
```

String - **CASE INSENSITIVE** comparison

```
if (name.equalsIgnoreCase("mike")) {
    // ...
}
```

String - Check if a string **STARTS** with something:

```
if (name.startsWith("Mik")) {

}
```

String - Check if a string **ENDS** with something:

```
if (name.endsWith("ke")) {

}
```

String - **FIND** a substring within a string

```
int ikPos = name.indexOf("ik");  // will be 1, since ik starts in the 1 pos
int xxPos = name.indexOf("xx"); // will be -1 for "not found"
```

String - Check if a string **CONTAINS** a substring

```
//
if (name.contains("ik")) {

}
```

String - make a new copy with something **REPLACED**

```
String newName = name.replace("ik", "x"); // newName will be "Mxe"
```

String - convert to **LOWERCASE** or **UPPERCASE**

```
String lowName = name.toLower(); // mike
String upName = name.toUpper(); // MIKE
```

String - **SPLIT** a string

```
String vegetables = "carrot,tomato,cucumber,lettuce";
String[] vegetableArray = vegetables.split(",");
```

String - **JOIN** an string array back into a single string

```
String vegetablesString = String.join(",", vegetablesArray);
```

String - Get a **SUBSTRING**

```
String name = "Mike";
// we give the start index, and one after the end index
String secondTwo = name.subString(2, 4);
```

String - Get a **ENDING SUBSTRING** (just give the start index)

```
String end = name.subString(2);  // "ke"
```

String - Get a **STARTING SUBSTRING** (0, one after end index)

```
String starting = name.subString(0, 3);  // "Mi"
```

BigDecimal Type

BigDecimal - **CREATING**

```
BigDecimal price = new BigDecimal("5.99");
BigDecimal price2 = BigDecimal.valueOf(5.99);
```

BigDecimal - Comparing for **EQUALS**

```
if (price.compareTo(price2) == 0) {
    // ...
}
```

BigDecimal - Comparing for **GREATER THAN**

```
if (price.compareTo(price2) > 0) {
    // ...
}
```

BigDecimal - Comparing for **LESS THAN**

```
if (price.compareTo(price2) < 0) {
    // ...
}
```

BigDecimal - Doing BigDecimal **MATH**

- can not use operators like +-/=%
- must use big decimals for other numbers in calculation

```
BigDecimal halfPrice = price.divide(BigDecimal.valueOf(2));
BigDecimal totalPrices = price.add(price2);
BigDecimal tax = price.multiply(BigDecimal.valueOf(0.0575));
```

LocalDate type

LocalDate - **CREATING** dates

```
LocalDate date = LocalDate.of(1981, 1, 21);
```

LocalDate - Getting **PARTS** of the date

```
int year = date.getYear();
int month = date.getMonthValue();
int day = date.getDayOfMonth();
```

LocalDate - Comparing dates for **EQUAL**

```
LocalDate date2 = LocalDate.of(1987, 7, 24);
if (date.equals(date2)) {

}
```

LocalDate - Comparing dates for **GREATER/AFTER**

```
if (date2.isAfter(date)) {
    // ...
}
```

LocalDate - Comparing dates for **LESS/BEFORE**

```
if (date2.isBefore(date)) {

}
```

LocalDate - Doing date **MATH**

```
LocalDate fiveMore = date.addDays(5);
LocalDate fiveLess = date.minusDays(5);
```

Array Types

Array - **DECLARING** arrays

```
String[] names = new String[5];  // with a size and default values
int[] scores = new int[] {1, 2, 3, 4, 5};   // with inital values

// indeces  start at 0.  So these 5 element arrays have indece 0-4
```

Array - **GETTING** and **SETTING** values

```
String firstName = names[0];       // get item 0
scores[1] = 533;                   // set item 1 to 533
```

Array - Getting the **SIZE** of an array

```
int totalNames = names.length;
```

Array - Using size to get the **LAST** element of an array

```
// (the last index is always 1 less than the length):
int lastNameInArray = names[names.length - 1];
```

Array - **LOOPING** through an array by index

```
for (int i = 0; i < names.length; i++) {
    // Each time through is goes up 1 (0 first time, then 1, then 2...)
    // All the way to names.length

    // so names[i] refers to the name corresponding to this time through
    // the array.  (names[0], names[1], names[2], ...)

    int currentName = names[i];
}
```

ArrayList Type

Like an array, but you can add/remove

ArrayList - **DECLARING** an ArrayList

```
List<String> names = new ArrayList<>(); // a list of strings
// NOTE: We have to use the "boxed" uppercase type when using non-array
// collection types to hold primitives
List<Double> scores = new ArrayList<>(); // a list of doubles
```

ArrayList - **ADDING** and **REMOVING** items

```
names.add("Mike");
names.remove("Mike");
```

ArrayList - **LOOPING** through an ArrayList with for each

```
for (String name : names) {
    // loop will repeat for each item, storing current item
    // in name variable
    // ...
}
```

ArrayList - Checking if an ArrayList **CONTAINS** an item

```
if (names.contains(name)) {
    // ...
}
```

HashSet Type

Like an ArrayList, but no duplicates:

HashSet - **DECLARING** a HashSet

```
Set<String> validNames = new HashSet<>();
```

HashSet - **ADDING** to a HashSet

```
validNames.add("Mike");
validNames.add("Steve");
validNames.add("Mike");
// Only 2 items, second mike was ignored (no duplicates in maps!)
```

HashMap Type

List an array list, but every item has a lookup key

HashMap - **DECLARING** a HashMap

```
Map<String, double> scores = new HashMap<String, double>();p
```

HashMap - **ADDING** to or **UPDATING** a HashMap

```
scores.put("Mike", 5.0);
scores.put("Steve", 2.2);
scores.put("Mike", 4.0);  // OK, just updates "Mike" from 5 to 4
```

HashMap - **REMOVING** from a HashMap (done by key)

```
scores.remove("Mike");
```

HashMap - **GETTING** an item by key

```
double mikeScore = scores.get("Mike");
```

HashMap - Checking if a **KEY EXISTS**

```
if (scores.containsKey("Mike")) {
    // ...
}
```

HashMap - **LOOPING** through a hashmap's entries

```
// Looping (awkward/unusual for hashmaps)
for (Map.Entry<String, double> scoreEntry : scores.entrySet()) {
    String currentKey = scoreEntry.getKey();
    double currentValue = scoreEntry.getValue();

    // ...
}
```

Queue Type

Like an array list, but we always read/remove items at same time, and always from the front of the line (FIFO)

Queue - **DECLARING** a Queue

```
Queue<String> customers = new LinkedList<String>();
```

Queue - **ADDING** to a queue

```
customers.offer("Mike");
customers.offer("Steve");
customers.offer("Brian");
```

Queue - **LOOPING** through a queue

```
while (customers.size() > 0) {
    // Poll reads AND removes the FIRST item (Mike first, etc.)
    String currentCustomer = customers.poll();
    System.out.println("Now serving " + currentCustomer);
}
```

Stack Type

Like an array list, but we always read/remove items at same time, and always from the back of the line (LIFO)

Stack - **DECLARING** a Stack

```
Stack<String> tasks = new Stack<String>();
```

Stack - **ADDING** to a Stack

```
tasks.push("Make dinner");
tasks.push("Go to store");
tasks.push("Put gas in car");
```

Stack - **LOOPING** through a Stack

```
while (tasks.size() > 0) {
    // Pop reads AND removes the LAST item (gas first, etc.)
    String currentTask = tasks.pop();
    System.out.println("Now doing task: " + currentTask);
}
```

Conversions

Upcasting/Downcasting primitive numbers:

```
int x = 10;
long z = x;     // No cast needed when going to a bigger type

double speed = 33;
int nSpeed = (int) speed;   // Explicitly cast when going to a smaller type
```

Converting Strings and primitive numbers:

```
double speed = 55.5;
String strSpeed = Double.toString(speed);  // Use the uppercase type's
toString()

String userInput = "125";
int userInputAsNumber = Integer.parseInt(userInput);

String userInput = "234234.44";
double userInputAsDecimal = Doulbe.parseDouble(userInput);
```

Converting Big Decimals

```
// From a string
String userInput = "12345";
BigDecimal bigNumber = new BigDecimal(userInput);

// From any kind of primitive number
int x = 5;
BigDecimal bigNumber2 = BigDecimal.valueOf(x);

// To a String
BigDecimal bigNumber3 = new BigDecimal("123");
String s = bigNumber3.toString();

// To a primitive number
BigDecimal bigNumber4 = new BigDecimal("34.2");
double doubleVersion = bigNumber4.doubleValue();
int intVersion = bigNumber4.intValue();
```

Expressions

Arithmetic expressions

combine values to get a new value:

```
int x = 5;
int y = 2;
int product = x * y;
int intDivision = x / y;
int remainder = x % y;
int oneHundred = x * 20;
int biggerExpression = x + 10 * (y + 14);

String firstName = "Mike";
String lastName = "Lambert";
String concat = firstName + " " + lastName;

boolean allTestsPass = (x == 5) && (z == 10);

// using an expression in a var..
int age = 15;
int ageIn5Years = age + 5;

if (ageIn5Years > 18) {
    // ...
}

// using the expression directly
if (age + 5 > 18) {
    // ...
}

// Function calls can be part of expressions:
int x = 10 + getSquareRoot(25);
```

Comparison / Boolean Expressions

Usage

```
int x = 5;
int y = 3;

// Boolean expression (comparison) in an if statement
if (x == y) {
    // ....
}

// Storing result of boolean expression in a boolean
boolean areTheyEqual = (x == y);

// Boolean expression as a loop condition
while (x == y) {
    // ...
}
// Middle part of for statement is the loop condition
for (int i = 0; i == 10; i++) {
    // ...
}
```

PRIMITIVE COMPARISONS (double ==) **do not use with strings**

```
double x = 5.0;
double y = 2.2;

if (x == y) {   // we use == ONLY IF we are comparing primitives
    // ...
}
```

OBJECT COMPARISONS (String, BigDecimal, etc.)

```
BigDecimal x = new BigDecimal("5.0");
BigDecimal y = new BigDecimal("2.0");

if (x.equals(y)) {  // we use the .equals() method on objects to compare
them
    // ...
}

String userInput = scanner.nextLine();

if (userInput.equals("yes")) {
    // ...
}

// Case-insensitive comparison:
if (userInput.equalsIgnoreCase("yes")) {
    // ...
}
```

COMBINING with AND, OR, XOR:

```
if (x == 5 && y == 3) {
    // AND (only happens if BOTH pass)
}

if (x == 5 || y == 3) {
    // OR (happens if either or both pass)
}

if (x == 5 ^ y == 3) {
    // XOR (happens if one and only one pass - not neither, not both)
}

if ((x == 5) && (y == 3 || userInput.equals("yes"))) {
    // Chaining several, and using parenthese to control grouping
}
```

Loops

While loops

```
String userInput = "";

while (!userInput.equals("quit")) {
    // ...
    userInput = Scanner.nextLine();
}
```

For Loops

BASIC COUNTING to a fixed number:

```
// for (INITIALIZATION, CONDITION, POST-ITERATION STEP)
for (int i = 1; i <= 10; i++) {
    System.out.println(i);
}
```

Counting from 0 to array length (**STANDARD ARRAY LOOP**):

```
String[] students = new String[] { "Mike", "Steve", "Brian" };

for (int i = 0; i < students.length; i++) {
    String currentStudent = students[i];
    // ...
}
```

Visiting **EVERY THIRD ITEM** in an array:

```
String[] students = new String[] {
    "Mike", "Steve", "Brian",
    "Janet", "Scott", "Sharon",
    "Dave", "Joel", "Brandon"
};

// Mike, Janet, Dave...
for (int i = 0; i < students.length; i += 3) {
    String currentStudent = students[i];
    // ...
}
```

Doing something different based on **ARRAY INDEX MODULUS**:

```
String[] orders = new String[55];

// Fill with S,M,L,S,M,L, etc.
for (int i = 0; i < orders.length; i++) {
    if (i % 3 == 0) {
        orders[i] = "S";
    } else if (i % 3 == 1) {
        orders[i] = "M";
    } else {        // i % 3 must be 2...
        orders[i] = "L"
    }
}
```

Looping **BACKWARDS**:

```
String[] students = new String[] { "Mike", "Steve", "Brian" };

for (int i = students.length - 1; i >= 0; i--) {
    String currentStudent = students[i];
    // ...
}
```

Methods (aka Functions)

Method declaration and calling.

```
// The form for declaring a method:
//    public static RETURNTYPE NAME(TYPE ARGNAME, TYPE ARGNAME, ...)

// The square function "takes" one parameter.  An int, that will be
// referred to as "x" for the body of the method.
public static int square(int x) {
    int result = x * x;

    // Here, we return the result.  We go back to where
    // we were called from and replace the call with this value
    return result;
}

public static void main(String[] args) {
    // Here, we "call" square and pass 5 as the first parameter
    // this transfers control to the square function.  When it
    // returns, we wil replace "square(5)" with the value that
    // was returned from square
    int fiveSquared = square(5);
}
```

Void methods

```
// This method has a return type of void.  This means
// it is just used to go do something, and doesn't have
// any return value that it gives back to the caller.
public static void promptUser(String name) {
    System.out.println("Hello " + name);

    // We can put return with nothing like this:
    return;

    // But we could also ommit this return for void
    // methods, and if we reach the end of the body, the return
    // is implied
}

public static void main(String[] args) {
    // Here, we are calling the function and we aren't
    // storing its return value in a variable, because it
    // doesn't return anything
    sayHi("Mike");
}
```