Walk through 2 - Java Tutorial

Topic Elements of programming in Java

1. Introduction

In the first lab you wrote and run your first Java program (Hello World) using Eclipse. In the next four labs including this lab you will learn in more details the basic structure of a Java program that you need for implementing algorithms and data structures described in 2C03.

2. Lab Objectives

The objective of this lab is to learn:

- Primitive data types in Java
- Conditions and Loops
- Arrays

3. Lab Setup

Before beginning this lab, you should have:

1. The Eclipse and Java Runtime Environment on your computer.
2. Completed Lab walk-through 1 from the previous week.
3. Read chapter 1, page 3-20 in your textbook.
4. **Before you start doing this lab, in your Eclipse workspace create a project named 2XB3_Lab2 inside a package named cas.lab2.wt. Add classes to this project as you walk through this instruction.**

Note

For each task you are given the class template and helper codes and comments. The TA will give you a few minutes to complete the task on your own. Then you can verify your work, as the TA will walk you through the steps to complete the task.

YOU NEED TO HAVE YOUR OWN STORAGE DEVICE (E.G., A USB KEY) IN ORDER TO CREATE THE ECLIPSE WORKSPACE.

IT IS YOUR RESPONSIBILITY TO KEEP YOUR STORAGE DEVICE SAFE (WITH NECESSARY BACKUPS) FOR FUTURE USE OF THE PROJECTS YOU ARE CREATING IN THE LABS.

4. Lab Exercise

4.1 Built-in types of data

A *data type* is a set of values and a set of operations defined on them. For example, we are familiar with numbers and with operations defined on them such as addition and multiplication. There are eight different built-in types of data in Java, mostly different kinds of numbers. We use the system type for strings of characters so frequently that we also consider it here.

type	*set of values*	*common operators*	*sample literal values*
`int`	integers	`+ - * / %`	`99 12 2147483647`
`double`	floating-point numbers	`+ - * /`	`3.14 2.5 6.022e23`
`boolean`	boolean values	`&& || !`	`true false`
`char`	characters		`'A' '1' '%' '\n'`
`String`	sequences of characters	`+`	`"AB" "Hello" "2.5"`

Terminology. We use the following code fragment to introduce some terminology:

```
int a, b, c;
a = 1234;
b = 99;
c = a + b;
```

The first line is a *declaration statement* that declares the names of three *variables* using the *identifiers* a, b, and c and their type to be int. The next three lines are *assignment statements* that change the values of the variables, using the *literals* 1234 and 99, and the *expression* a + b, with the end result that c has the value 1333.

4.2 Comparison

The *comparison* operators are *mixed-type* operations that take operands of one type (e.g., int or double) and produce a result of type boolean. These operations play a critical role in the process of developing more sophisticated programs.

Task 1: Write a Java program that tests whether an integer corresponds to a leap year in the Gregorian calendar.

You start with …..

```
/*************************************************************
 *  Compilation:  javac LeapYear.java
 *  Execution:    java LeapYear n
 *  Prints true if n corresponds to a leap year, and false otherwise.
 *  Assumes n >= 1582, corresponding to a year in the Gregorian
calendar.
 *  % java LeapYear 2004
 *  true
 *  % java LeapYear 1900
 *  false
 *  % java LeapYear 2000
 *  true
 *************************************************************/
public class LeapYear {
    public static void main(String[] args) {
        int year = Integer.parseInt(args[0]);
        boolean isLeapYear;
        // divisible by 4
        isLeapYear = (year % 4 == 0);
        // divisible by 4 and not 100
        isLeapYear = isLeapYear && (year % 100 != 0);
        // divisible by 4 and not 100 unless divisible by 400
```

```
        isLeapYear = isLeapYear || (year % 400 == 0);
        System.out.println(isLeapYear);
    }
}
```

4.3 Library method and APIs

Many programming tasks involve using Java library methods in addition to the built-in operators. An *application programming interface* is a table summarizing the methods in a library.

- *Printing strings to the terminal window.*

`void System.out.print(String s)`	*print s*
`void System.out.println(String s)`	*print s, followed by a newline*
`void System.out.println()`	*print a newline*

- *Converting strings to primitive types.*

`int Integer.parseInt(String s)`	*convert s to an int value*
`double Double.parseDouble(String s)`	*convert s to a double value*
`long Long.parseLong(String s)`	*convert s to a long value*

- *Mathematical functions.*

`public class Math`

`double abs(double a)`	*absolute value of a*
`double max(double a, double b)`	*maximum of a and b*
`double min(double a, double b)`	*minimum of a and b*
`double sin(double theta)`	*sine of theta*
`double cos(double theta)`	*cosine of theta*
`double tan(double theta)`	*tangent of theta*
`double toRadians(double degrees)`	*convert angle from degrees to radians*
`double toDegrees(double radians)`	*convert angle from radians to degrees*
`double exp(double a)`	*exponential (e^a)*
`double log(double a)`	*natural log ($\log_e a$, or $\ln a$)*
`double pow(double a, double b)`	*raise a to the b th power (a^b)*
`long round(double a)`	*round a to the nearest integer*
`double random()`	*random number in $[0, 1)$*
`double sqrt(double a)`	*square root of a*
`double E`	*value of e (constant)*
`double PI`	*value of π (constant)*

You can call a method by typing its name followed by *arguments*, enclosed in parentheses and separated by commas. Here are some examples:

method call	*library*	*return type*	*value*
`Integer.parseInt("123")`	`Integer`	`int`	`123`
`Double.parseDouble("1.5")`	`Double`	`double`	`1.5`
`Math.sqrt(5.0*5.0 - 4.0*4.0)`	`Math`	`double`	`3.0`
`Math.log(Math.E)`	`Math`	`double`	`1.0`
`Math.random()`	`Math`	`double`	*random in* [0, 1)
`Math.round(3.14159)`	`Math`	`long`	`3`
`Math.max(1.0, 9.0)`	`Math`	`double`	`9.0`

4.4 Type conversion

We often find ourselves converting data from one type to another using one of the following approaches.

- *Explicit type conversion*. Call methods such as Math.round(), Integer.parseInt(), and Double.parseDouble().
- *Automatic type conversion.* For primitive numeric types, the system automatically performs type conversion when we use a value whose type has a larger range of values than expected.
- *Explicit casts.* Java also has some built-in type conversion methods for primitive types that you can use when you are aware that you might lose information, but you have to make your intention using something called a *cast*.
- *Automatic conversions for strings.* The built-in type String obeys special rules. One of these special rules is that you can easily convert any type of data to a String by using the + operator.

Task 2: Write a Java program that reads an integer command-line argument *n* and prints a "random" integer between 0 and *n*−1. **Hint:** You will be reading the argument as a String.

You start with …

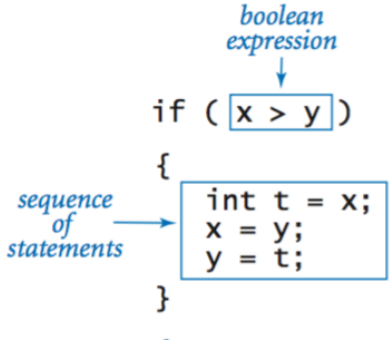
```
public class RandomInt {
    public static void main(String[] args) {
        // a positive integer
        int n = Integer.parseInt(args[0]);

        // a pseudo-random real between 0.0 and 1.0
        double r = Math.random();

        // a pseudo-random integer between 0 and n-1
        int value = (int) (r * n);

        System.out.println(value);
    }
}
```

4.5 Conditions and Loops

In the programs that we have examined to this point, each of the statements is executed once, in the order given. Most programs are more complicated because the sequence of statements and the number of times each is executed can vary. We use the term *control flow* to refer to statement sequencing in a program.

If statements.
Most computations require different actions for different inputs.

- The following code fragment uses an if statement to put the smaller of two int values in x and the larger of the two values in y, by exchanging the values in the two variables if necessary.



Task 3: Write a Java program Flip.java that uses Math.random() and an if-else statement to print the results of a coin flip.

We start with…

```
public class Flip {

    public static void main(String[] args) {

        // Math.random() returns a value between 0.0 and 1.0
        // so it is heads or tails 50% of the time
        if (Math.random() < 0.5) System.out.println("Heads");
        else                     System.out.println("Tails");
    }
}
```

The table below summarizes some typical situations where you might need to use an if or if-else statement.

While loops.
Many computations are inherently repetitive. The while loop enables us to execute a group of statements many times. This enables us to express lengthy computations without writing lots of code.

- The following code fragment computes the largest power of 2 that is less than or equal to a given positive integer *n*.

Task 4: Write a Java program TenHellos.java that prints "Hello World" 10 times.
We start with….

```
public class TenHellos {
    public static void main(String[] args) {

        // print out special cases whose ordinal doesn't end in th
        System.out.println("1st Hello");
        System.out.println("2nd Hello");
        System.out.println("3rd Hello");
```

```
        // count from i = 4 to 10
        int i = 4;
        while (i <= 10) {
            System.out.println(i + "th Hello");
            i = i + 1;
        }

    }
}
```

For loops.
The *for loop* is an alternate Java construct that allows us even more flexibility when writing loops.

- *For notation.* Many loops follow the same basic scheme: initialize an index variable to some value and then use a while loop to test an exit condition involving the index variable, using the last statement in the while loop to modify the index variable. Java's for loop is a direct way to express such loops.
- *Compound assignment idioms.* The idiom i++ is a shorthand notation for i = i + 1.
- *Scope.* The *scope* of a variable is the part of the program that can refer to that variable by name. Generally the scope of a variable comprises the statements that follow the declaration in the same block as the declaration. For this purpose, the code in the for loop header is considered to be in the same block as the for loop body.

4.6. Arrays

An array stores a sequence of values that are all of the same type. We want not just to store values but also to be able to quickly access each individual value. The method that we use to refer to individual values in an array is to number and then *index* them—if we have *n* values, we think of them as being numbered from 0 to *n*−1.

Arrays in Java
Making an array in a Java program involves three distinct steps:

- Declare the array name.
- Create the array.
- Initialize the array values.

We refer to an array element by putting its index in square brackets after the array name: the code a[i] refers to element i of array a[]. For example, the following code makes an array of n numbers of type double, all initialized to 0:

```
double[] a;                     // declare the array
a = new double[n];              // create the array
for (int i = 0; i < n; i++)     // elements are indexed from 0 to n-1
   a[i] = 0.0;                  // initialize all elements to 0.0
```

Typical array-processing code.
The following are typical examples of using arrays in Java.

create an array with random values	`double[] a = new double[n];`
	`for (int i = 0; i < n; i++)`
	`  a[i] = Math.random();`

print the array values, one per line	`for (int i = 0; i < n; i++)` `  System.out.println(a[i]);`

find the maximum of the array values	`double max = Double.NEGATIVE_INFINITY;`
	`for (int i = 0; i < n; i++)`
	`  if (a[i] > max) max = a[i];`

compute the average of the array values	`double sum = 0.0;`
	`for (int i = 0; i < n; i++)`
	`  sum += a[i];`
	`double average = sum / n;`

reverse the values within an array	`for (int i = 0; i < n/2; i++)`
	`{`
	`  double temp = a[i];`
	`  a[i] = a[n-1-i];`
	`  a[n-i-1] = temp;`
	`}`

copy sequence of values to another array	`double[] b = new double[n];`
	`for (int i = 0; i < n; i++)`
	`  b[i] = a[i];`

Programming with arrays.
We consider a number of important characteristics of programming with arrays.

- *Zero-based indexing.* We always refer to the first element of an array a[] as a[0], the second as a[1], and so forth. It might seem more natural to you to refer to the first element as a[1], the second value as a[2], and so forth, but starting the indexing with 0 has some advantages and has emerged as the convention used in most modern programming languages.
- *Array length.* Once we create an array, its length is fixed. You can refer to the length of an a[] in your program with the code a.length.
- *Default array initialization.* For economy in code, we often take advantage of Java's default array initialization convention. For example, the following statement is equivalent to the four lines of code at the top of this page:

  double[] a = new double[n];
- The default initial value is 0 for all numeric primitive types and false for type boolean.
- *Memory representation.* When you use new to create an array, Java reserves space in memory for it (and initializes the values). This process is called *memory allocation*.
- *Bounds checking.* When programming with arrays, you must be careful. It is your responsibility to use legal indices when accessing an array element.
- *Setting array values at compile time.* When we have a small number of literal values that we want to keep in array, we can initialize it by listing the values between curly braces, separated by a comma. For example, we might use the following code in a program that processes playing cards.

```
String[] SUITS = {
          "Clubs", "Diamonds", "Hearts", "Spades"
      };
```

```
String[] RANKS = {
    "2", "3", "4", "5", "6", "7", "8", "9", "10",
    "Jack", "Queen", "King", "Ace"
};
```

- After creating the two arrays, we might use them to print a random card name such as Queen of Clubs, as follows.
-

```
int i = (int) (Math.random() * RANKS.length);
int j = (int) (Math.random() * SUITS.length);
System.out.println(RANKS[i] + " of " + SUITS[j]);
```

- *Setting array values at run time.* A more typical situation is when we wish to compute the values to be stored in an array For example, we might use the following code to initialize an array of length 52 that represents a deck of playing cards, using the arrays RANKS[] and SUITS[] just defined.

```
String[] deck = new String[RANKS.length * SUITS.length];
for (int i = 0; i < RANKS.length; i++)
    for (int j = 0; j < SUITS.length; j++)
        deck[SUITS.length*i + j] = RANKS[i] + " of " + SUITS[j];
System.out.println(RANKS[i] + " of " + SUITS[j]);
```

Shuffling and sampling.

Now we describe some useful algorithms for rearranging the elements in an array.

- *Exchange.* Frequently, we wish to exchange two values in an array. Continuing our example with playing cards, the following code exchanges the card at position i and the card at position j:

```
String temp = deck[i];
deck[i] = deck[j];
deck[j] = temp;
```

- *Shuffling.* The following code shuffles our deck of cards:

```
int n = deck.length;
for (int i = 0; i < n; i++) {
   int r = i + (int) (Math.random() * (n-i));
   String temp = deck[r];
   deck[r] = deck[i];
   deck[i] = temp;
}
```

- Proceeding from left to right, we pick a random card from deck[i] through deck[n-1] (each card equally likely) and exchange it with deck[i].

Task 5: Let's put everything together and write a Java program Deck.java that contains the full code for creating and shuffling a deck of cards.

We start with…

```
/*************************************************************************
 *  Compilation:  javac Deck.java
 *  Execution:    java Deck
 *
```

```
 *  Deal 52 cards uniformly at random.
 *
 *  % java Deck
 *  Ace of Clubs
 *  8 of Diamonds
 *  5 of Diamonds
 *  ...
 *  8 of Hearts
 *
 *****************************************************************/

public class Deck {
    public static void main(String[] args) {
        String[] SUITS = {
            "Clubs", "Diamonds", "Hearts", "Spades"
        };

        String[] RANKS = {
            "2", "3", "4", "5", "6", "7", "8", "9", "10",
            "Jack", "Queen", "King", "Ace"
        };

        // initialize deck
        int n = SUITS.length * RANKS.length;
        String[] deck = new String[n];
        for (int i = 0; i < RANKS.length; i++) {
            for (int j = 0; j < SUITS.length; j++) {
                deck[SUITS.length*i + j] = RANKS[i] + " of " +
SUITS[j];
            }
        }

        // shuffle
        for (int i = 0; i < n; i++) {
            int r = i + (int) (Math.random() * (n-i));
            String temp = deck[r];
            deck[r] = deck[i];
            deck[i] = temp;
        }

        // print shuffled deck
        for (int i = 0; i < n; i++) {
            System.out.println(deck[i]);
        }
    }

  }
```

Matrix operations.
Typical applications in science and engineering involve implementing various mathematical operations with matrix operands. For example, we can *add* two *n*-by-n matrices as follows:

```
double[][] c = new double[n][n];
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        c[i][j] = a[i][j] + b[i][j];
    }
}
```

Similarly, we can *multiply* two matrices. Each entry c[i][j] in the product of a[] and b[] is computed by taking the dot product of row i of a[] with column j of b[].

```
double[][] c = new double[n][n];
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++)  {
        for (int k = 0; k < n; k++)  {
            c[i][j] += a[i][k]*b[k][j];
        }
    }
}
```

Task 6: Write a code fragment Transpose.java to transpose a square two-dimensional array in place without creating a second array.

We start with….

```
public class Transpose {

    public static void main(String[] args) {

        // create n-by-n matrix
        int n = Integer.parseInt(args[0]);
        int[][] a = new int[n][n];
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                a[i][j] = n*i + j;
            }
        }

        // print out initial matrix
        System.out.println("Before");
        System.out.println("------");
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                System.out.printf("%4d", a[i][j]);
            }
            System.out.println();
        }

        // transpose in-place
        for (int i = 0; i < n; i++) {
            for (int j = i+1; j < n; j++) {
                int temp = a[i][j];
```

```
                a[i][j] = a[j][i];
                a[j][i] = temp;
            }
        }

        // print out transposed matrix
        System.out.println();
        System.out.println("After");
        System.out.println("------");
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                System.out.printf("%4d", a[i][j]);
            }
            System.out.println();
        }

    }
}
```

Task 7. Submit your work

Once all tasks are completed, you should submit your Eclipse project. Follow the instructions below for submission:

- Include a .txt file named last_name_initials.txt in the root of the project containing on separate lines: Full name, student number, any design decisions/assumptions you feel need explanation or attention.
- After checking the accuracy and completeness of your project, save everything and right-click on the name of the project, select Export->General->Archive File.
- Ensure that just your project has a check-mark beside it, and select a path to export the project to. The filename of the zipped project must follow this format: *macID_Lab2.zip.* Check the option to save the file in .zip format. Click Finish to complete the export.
- Go to Avenue and upload your zipped project to 'Lab Walk-through 2 – Lab Section X)
- IMPORTANT: You MUST export the FULL Eclipse project. Individual files (e.g. java/class files) will NOT be accepted as a valid submission.

4. Practice Problems

1 What do each of the following print?

a. System.out.println(2 + "bc");
b. System.out.println(2 + 3 + "bc");
c. System.out.println((2+3) + "bc");
d. System.out.println("bc" + (2+3));
e. System.out.println("bc" + 2 + 3);

Explain each outcome.

2. A physics student gets unexpected results when using the code
double force = G * mass1 * mass2 / r * r;
to compute values according to the formula $F = Gm_1m_2 / r^2$. Explain the problem and correct the code.\

3. Write a program SpringSeason.java that takes two int values m and d from the command line and prints true if day d of month m is between March 20 (m = 3, d =20) and June 20 (m = 6, d = 20), false otherwise.

Loop and conditions

4. Suppose a gambler makes a series of fair $1 bets, starting with $50, and continue to play until she either goes broke or has $250. What are the chances that she will go home with $250, and how many bets might she expect to make before winning or losing? Write a Gambler.java program that is a simulation that can help answer these questions. It takes three command-line arguments, the initial stake ($50), the goal amount ($250), and the number of times we want to simulate the game.

Arrays

5. ***Sampling without replacement***. In many situations, we want to draw a random sample from a set such that each member of the set appears at most once in the sample. Write a Java program Sample.java that takes two command-line arguments m and n, and creates a *permutation* of length n whose first m entries comprise a random sample. See the textbook for details.

6. ***Random walkers*.** Suppose that n random walkers, starting in the center of an n-by-n grid, move one step at a time, choosing to go left, right, up, or down with equal probability at each step. Write a program RandomWalkers.java to help formulate and test a hypothesis about the number of steps taken before all cells are touched.