# Exercise Notes

## Java Development Environment

Two acronyms you seer with Java are the Java Runtime Environment (JRE) and the Java Development Kit (JDK). Platforms (operating systems) that want to run already compiled and packaged Java programs require a JRE. If you want to develop in Java, then you need the JDK (which has the JRE in it as well). You can check that a JRE is available on your system by typing “java-version” at a command prompt, and that a JDK is available by typing “javac -version” on the command prompt. You may have multiple versions of JREs and JDKs installed on your host, if so the one in effect is the one pointed to by the JRE\_HOME, JDK\_HOME, and PATH environment variables.

It is recommended you use the JDK version 21 or later, though version 17 or better should work. The [JDK documentation](https://docs.oracle.com/en/java/javase/21/) should be useful, the API Documentation under Specifications in particular.

It is also recommended that you get used to using the JDK on the command-line during these exercises rather than in an IDE. IDEs can obscure the version of Java you are using, compile-time and runtime flags, and the use of the CLASSPATH environment variable.

## Java Console I/O

Java does not provide a simple mechanism for reading values from the keyboard. Therefore, we will use a (slightly modified) class provided by the authors of the CoreJava book named Console. Console provides a set of methods that read bytes from the keyboard and convert them into characters, Strings, integers, or doubles.

public class Console

{

public static void printPrompt(String prompt);

public static char readChar(); // including whitespace

public static String readString(); // read until a newline

public static String readString(String prompt);

public static String readWord(); // read until whitespace

public static int readInt(String prompt);

public static int readInt();

public static double readDouble(String prompt);

public static double readDouble();

}

Console.class is already present in your classes subdirectory. You may use Console in any program for this training. An example use of Console might look like:

{

int x;

float y;

x = Console.readInt("Enter an Integer :");

y = Console.readDouble ("Enter a Double :");

char ch = Console.ReadChar(); // get next character from keyboard

}

# Exercises part 1

## Exercise #1 Hello World

Write a Java program that prints your name.

## Exercise #2 Output and loops

Write a program that prints a table of Celsius to Fahrenheit values. The table should display values from 0 to 100 degrees Celsius in increments of 5 degrees. The conversion from Celsius to Fahrenheit is:

Fahrenheit = 9/5 \* Celsius + 32

Be careful of integer division. As a warning, the above equation will compute the wrong value if written directly as a Java statement (try it and see). Fixing the problem requires a floating expression. However, as part of this exercise make the variable containing the Fahrenheit value an integer. This exercise demonstrates Java strong typing. An integer expression may be implicitly converted to a double, since no information would be lost. However, converting the double back to an integer requires an explicit conversion, since information may be lost.

An example table is shown below:

Celsius Fahrenheit

0 32

5 ...

10 ...

... ...

100 212

## Exercise #3 Input and if statement

Write a program that computes a paycheck. The input to the program is the employee's name, the hourly pay rate, and the number of hours worked. The program should output the employee's name and their paycheck amount. The amount is computed as hours worked times the hourly pay. Overtime is computed at 1.5 times hourly pay for all hours over 40. An example is shown below:

Enter employee name: Bob

Enter pay rate: 10

Enter hours worked: 41

Paycheck for Bob is $415.0

Next, modify the program such that it continually reads names, pay rates, and numbers of hours worked and prints paycheck amounts. The program should terminate when a pay rate of 0 is entered.

Finally, modify the program such that it terminates when the name "DONE" is entered. Recall when comparing two objects, the == operator compares the references to see if they refer to the same object. Therefore, we must use a method equals() defined on String objects. An example is shown below:

if (name.equals("Java"))

## Exercise #4 Arrays

Write a program that reads a set of integer values from the keyboard, displays their average, then displays the min and the max values. Use an array to solve this problem. First, read all the values into an array using a loop. Then use separate loops to compute the average and find the min and max values.

The user determines the size of the data set. The program starts by prompting the user for the size of their data set. Then, it reads that number of integers from the keyboard. The array should be exactly the size necessary. One purpose of this exercise is to illustrate that Java array sizes can be established at run-time.

## Exercise #5 Switch statements

Write a program that reads characters entered from the keyboard, counts the number of commas, dollar signs, and carriage returns, then prints the counts after the input has terminated. The user terminates the input with a '#' character. Use a switch statement rather than an if statement to check the different character values. An algorithm for solving the problem is given below:

initialize counter values

read a character

while (character not a '#')

{

if character matches one we're looking for, increment appropriate counter

read another character

}

print the counter values

Use the readChar() method inside Console to read a character one at a time.

## Exercise #6 Strings

You are given the following data file containing information regarding customers for a bulk mailing.

Bob\*\*\*\*\*\*\*caSmith\*\*\*\*\*12345San Luis\*\*113 S. 1st Street\*\*\*

Joe\*\*\*\*\*\*\*AzLuftwitz\*\*98726Ajo\*\*\*\*\*\*\*98 North 10th Pl.\*\*\*

Sue\*\*\*\*\*\*\*MNAnastwitz\*81726Duluth\*\*\*\*155 S. Elm\*\*\*\*\*\*\*\*\*\*

Al\*\*\*\*\*\*\*\*FlJohnson\*\*\*92812Melbourne\*15 West B St.\*\*\*\*\*\*\*

END

The fields are in the following order: first name, state, last name, zip, city, address. Each field is a fixed size. The list of entries ends with a line "END". For each entry in the file, generate a mailing label of the form:

Last\_Name, First\_Name

Address

City, State Zip

Also, you do not want to send the mailing to people whose last name contain “witz”, so do not generate a label for those people. Finally, the program that generated the name list did not always generate State abbreviations with two capitol letters. Make the state names all-caps before printing them.

Do not worry about reading from the file. Write the program as if it reads from the keyboard (Console.readString()) and then redirect input on the command line:

% java Main < input.txt

This exercise requires the use and manipulation of Java's String type and you will need to discover several methods available on Strings. Some that might be useful in this exercise include substring(), replace(), trim(), toUpper(), and lastIndexOf(). All methods on every Java type are documented in the JDK API documentation. One purpose of this exercise is for you to gain experience using the API documentation. Java's String has many methods available for programmers to manipulate. Browse the JDK documentation and locate java.lang.String. Read the method documentation for the methods described above and use them to solve the above problem.

# Exercises part 2

## Exercise #1 Class with instances

Write an Employee class that tracks an employee's name, department number, and annual salary. The Employee class uses the following public interface:

public void setInfo(String name, int dept, double salary)

public String getName()

public int getDept()

public void display()

public double computeBonus()

SetInfo() sets the Employee object to contain the information provided by the parameters. Display() writes the state of the Employee object (all of it's data) to System.out. Finally, computeBonus() determines the employee's bonus based on department number: departments less than 1000 receive a 1% bonus, others receive a 5% bonus.

Next, write a main class (one with a main() method) that creates a single Employee object, initializes it, displays it, and prints its bonus to the screen. Execute this simple main program to verify the Employee class is correct.

Finally, modify main to create an array of 3 employee objects. Call setInfo() on each object separately. Then call display() and computeBonus() on each object from within a loop. Some code fragments are shown below:

Employee list[] = new Employee[3];

list[0].setInfo("Sue", 400, 10000.0);

list[1].setInfo(...);

for (int i = 0; i < list.length; i++)

list[i].display();

## Exercise #2 Constructors

Add constructors to the solution from the previous exercise. First, add a default constructor to Employee that sets the name to "NONE", department to -1, and salary to 0. Verify the constructor works by removing one of the setInfo() calls in the main program. That employee should have default values instead of those provided by setValue() when main is executed.

Next, add a non-default constructor with the following signature:

public Employee (String name, int dept, double salary)

and modify all the calls in main() to use the constructor to initialize the object rather than setInfo(). When finished, main() should have no calls to setInfo() for this part of the exercise.

## Exercise #3 Class Associations

This exercise demonstrates object associations, where one object (company) contains links to other objects (employees). Links allow objects to invoke methods on the associated objects, as company must invoke methods on employee objects to implement its methods.

Using your Employee class from the previous exercise, write a class named Company with the following interface:

void addEmployee(Employee e)

Employee getEmployee(String name)

void displayAll()

double totalBonus()

Company declares an array of 10 employees and an integer size initialized to zero. Each time addEmployee() is called, the employee object is added to the array of employees and the size incremented. Displayall() simply calls display() for all the employee objects in the company and totalBonus() returns the sum of bonuses for the all employees. Finally, write a simple main program that creates a company object, adds a few employees, and invokes getEmployee(), displayAll(), and totalBonus().

When copying the Employee class from the previous exercise, you may use either the .class or .java file. The Java compiler must have access to Employee's definition to verify Company invokes Employee methods correctly. The compiler can use either the .class file or the .java file for this purpose.

## Exercise #4 The toString() Method

To motivate this exercise, write and execute the following main() program using the Employee class developed previously:

{

Employee e = new Employee("Bob", 1000, 10000.0);

System.out.println("Employee is " + e);

}

Every data item in Java can convert to a String. We've already discovered the fact that all the primitive data types implicitly convert when used as a String. Objects also convert to String. However, they are explicitly converted through the method toString(). A default toString() is provided for all objects (by the class Object) which displays the object's class and the object's location in the virtual machine. For most applications, this default conversion to a String is rather ugly and not desired. Add a method toString() to the Employee class which takes no parameters and returns a String. Have the method return the employee's name from toString(). Re-execute the program and verify the println now prints the employee's name.

## Exercise #5 Overloading Methods

Write a class named Adder with the following interface:

double add(double n1, double n2)

int add(int n1, int n2)

Within each add method write a print statement displaying which add() method was called and return the proper result. Write another class named Main that creates an Adder object and invokes the following methods:

Adder object = new adder();

System.out.println(object.add(10.0, 20.0));

System.out.println(object.add(10, 20));

The program should execute correctly and display that each call to add called the appropriate overloaded version. Now add the following statement to main():

System.out.println(object.add(10.0F, 20.0F));

Which of the overloaded add() methods is called? Is the program now ambiguous since it does not know which add() method to call (will it even compile)? Execute the program again and see which add() is invoked.

## Exercise #6 Static methods and data

Bear with this exercise. It is more difficult to describe than it is to write. Modify the Employee class so that methods write to a common log file. Each method in Employee should send a message to the log file indicating what they are doing:

logFile.println("Computing bonus for " + this);

The log file is declared as a static PrintStream object within the Employee class. Since it static, all Employee objects will reference it and write to the same location. PrintStream is contained in the java.io package, so you should import that package at the top of your file.

A client program assigns the log file dynamically by invoking a static method on the class. Add a method setLogFile() to Employee which reassigns the stream to a new stream:

public static void setLogFile(PrintStream newFile)

Finally, use the main program given below to test your new Employee class. It is provided in a given file D2Ex6\_Main.java. The program first sets the log file to System.out (the screen). It then creates and initializes 3 Employee objects, displays them, and computes their bonuses. Finally, the program opens "out.txt" and changes Employee's log file to the file by calling setLogFile. After execution, check that out.txt received the proper output.

Compiling the main program may generate *deprecation* warnings. Deprecation warnings mean the program uses methods from the API classes that are obsolete and will not be available in future releases of the APIs. Ignore these warnings for this assignment.

import java.io.\*;

class Main {

public static void main(String args[]) {

// Begin by logging to the screen

Employee.setLogFile(System.out);

Employee list[] = new Employee[3];

for (int i = 0; i <list.length; i++)

list[i] = new Employee();

list[0].setInfo("Joe", 400, 10000.0);

list[1].setInfo("Sue", 1400, 20000.0);

list[2].setInfo("Ann", 500, 30000.0);

for (int i = 0; i < list.length; i++)

list[i].display();

for (int i = 0; i < list.length; i++)

System.out.println(list[i].computeBonus());

// Now log to the file named "out.txt"

Try {

FileOutputStream file = new FileOutputStream("out.txt");

PrintStream out = new PrintStream(file);

Employee.setLogFile(out);

for (int i = 0; i < list.length; i++)

list[i].display();

for (int i = 0; i < list.length; i++)

System.out.println(list[i].computeBonus());

}

catch(Exception e) {e.printStackTrace();}

}

}

# Exercises part 3

## Exercise #1 Subclasses

The figure below shows a class hierarchy for employees. For this exercise, implement the part time and full time subclasses. The Employee class is provided. Employee is an abstract class which contains an abstract method computeBonus(). Each subclass must implement the abstract method computeBonus(). Part time employees receive a bonus that is 1% of their salary. Full time employees receive a bonus that is 3% of their salary plus $100 times their current number of options.

Both employee subclasses override the display method. The new display methods first prints the type of employee (part time of full time) and then call Employee's display() to display the employee's properties. Full time employee should also display the number of options the full time employee has with the company before calling Employee's display(). Finally, full time employees have options. The class contains methods to increase and obtain the available options. Not all methods are used in this exercise; some will be used in subsequent exercises.

Test your subclasses with a main program. The program should create 5 employees in an array as shown below:

Employee list[] = new Employee[5];

list[0] = new PartTimeEmployee();

list[1] = new FullTimeEmployee();

...

list[0].setInfo("Bob", 100, 10000.0);

list[1].setInfo("Sue", 200, 20000.0);

...

for (each employee)

list[i].display();

After creating the Employee objects, call display() on each as shown above. Then increase the options on all full time employees by 100. Use another for loop to iterate through all the employees in list. Use the *instanceof* operator to test if the Employee is full time before adding options. After increasing the options, call display on each Employee again to verify the options were increased. Finally, call computeBonus() on each Employee and print out the bonus value.

## Exercise #2 Subclass constructors

Employee in the previous example provides two constructors, a default constructor and a non-default constructor. Neither of the subclasses from the previous exercise defines their own constructor. Therefore, the compiler created a default constructor that simply called the parent's (Employee's) default constructor.

Start this exercise by removing (or commenting out) Employee's default constructor. Recompile the code for the subclasses. The compiler should generate an error stating that Employee does not contain a default constructor. Since Employee contains a constructor, it requires initialization. Therefore, the default constructor created by the compiler is in error because the parent's default constructor does not exist.

Next, place comments around Employee's non-default constructor and recompile. Everything should compile successfully. Neither Employee nor its subclasses define constructors, so the compiler creates default constructors for both classes which simply call the parent's default constructor.

Now let's add constructors to the subclasses. Remove the comments around Employee's non-default constructor so that it is the only constructor. Since Employee requires initialization, subclasses must invoke its constructor. Add the following constructors to the subclasses:

public PartTimeEmployee (String name, int dept, double salary)

public FullTimeEmployee (String name, int dept, double salary, int options)

Finally, modify main() so the it calls the new constructors, rather than the setInfo() methods.

# Exercises part 4

Participants may choose two different sets of exercises for Day 4. Exercise #1 and #2 are exercises that emphasize Vectors and GUI events. Exercise #3 and #4 also emphasize Vectors and GUI events, but continue the Bar Chart program from yesterday. Exercises #3 and 4 should be considered more advanced and more difficult than #1 and #2. Participants should write either {1, 2} or {3, 4}, then continue with exercise #5.

## Exercise #1 Vector

This exercise uses the Employee class and subclasses developed in exercise #2 from part 3. Copy those files (.class or .java) to the current directory and write a Company class that supports the following interface:

public void addEmployee(Employee a

public void deleteEmployee(String name)

public void displayAllEmployees()

The Company class stores Employee objects in a Vector. AddEmployee() adds the employee object to the Vector and deleteEmployee() removes an object from the Vector. The delete method will need to loop through each employee object individually comparing the name of the employee with the name provided by the caller. The employee class provides a method getName() which returns the employee's name. Finally, displayAll() calls display() for all employees stored in the Vector.

Test your Company class with the provided main program:

class Main

{

public static void main(String args[])

{

Company company = new Company();

company.addEmployee(new PartTimeEmployee("Bob", 100, 10000.0));

company.addEmployee(new FullTimeEmployee("Sue", 200, 20000.0, 20));

company.addEmployee(new FullTimeEmployee("Sal", 300, 30000.0, 30));

company.addEmployee(new PartTimeEmployee("Al", 400, 40000.0));

company.addEmployee(new FullTimeEmployee("Joe", 500, 50000.0, 50));

company.displayAllEmployees();

company.deleteEmployee("Bob");

company.deleteEmployee("Joe");

System.out.println();

company.displayAllEmployees();

}

}

## Exercise #2 Vector (BarChart Program)

Modify the given program such that is uses Vector rather than arrays. BarChart should declare three Vectors rather than three arrays for the data, colors, and labels. The set methods should also accept parameters that are Vectors, not arrays. Finally, BarChartFrame should use Vectors for storing the values entered in the Frame and sent the BarChart.

Currently, BarChartFrame initializes the arrays in the declarative section of the class. Vectors cannot be initialized in that manner. Therefore, you must move the initialization of those three Vectors into the constructor (or better, a separate method called from the constructor that initializes the Vectors). Only populate the three Vectors with one or two items, not the six items that the arrays contained.

The conversion to Vectors is mostly straight forward, except when retrieving a value (Vector.elementAt(i)). ElementAt() returns an Object, which you must convert to the appropriate type before using. The three Vectors contain Color, String (label), and Integer (data values) objects. Color and String are directly passed to their appropriate methods. The Integer is used as the Y value in fillRect(), which requires a primitive integer type. Use the following code to obtain the primitive integer value from an Integer object:

Integer intobj = (Integer) data.elementAt(i); //Vector returns an Object

int barLength = intobj.intValue(); //get int from the Integer

g.fillRect(100, yposition, barLength, barWidth);

This exercise demonstrates that vectors are more convenient to use, except when primitive values are manipulated. You will encounter a similar problem creating the integer Vector. Simply populate the Vector as:

data.addElement(new Integer(90));

labels.addElement("PPV");

colors.addElement(Color.red);

## Exercise #3 Files

Write a program that reads values from a file named "graph.txt" and prints them to the screen. Each line of the input file contains three values: a number, a string label, and a string color. Read the integer value as an integer and the other two values as strings. Use a string or stream tokenizer tokenizer object to help parse the file. You may use either a stream (FileInputStream) or a reader (FileReader) to access the file. The tokenizer requires a Reader object, so if a stream is used, an InputStreamReader object must be used as bridge as shown in the tokenizer example.