**Lab 6 Notes**

**Part A**

1. Demo to the class Write a Java program to print a face.

public class TeaCup {

public static void main(String[] args)

{

System.out.println(" +\"\"\"\"\"+ ");

System.out.println("[| o o |]");

System.out.println(" | ^ | ");

System.out.println(" | '-' | ");

System.out.println(" +-----+ ");

}

}

Sample Output:

+"""""+

[| o o |]

| ^ |

| '-' |

+-----+

**Part B**

**Goals for today:**

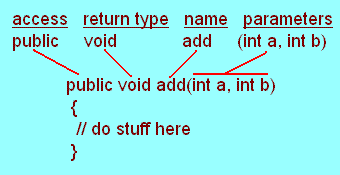
Learn the basics of declaring and passing parameters to methods

use return values to send data between methods

Example:

access return type name parameters

public void add (int a, int b)



The following list describes the method declaration piece by piece:

visibility: The visibility of a method determines whether the method is available to other classes. The options are

public: Allows any other class to access the method

static: This optional keyword declares that the method is a static method, which means that you can call it without first creating an instance of the class in which it’s defined. The main method must always be static, and any other methods in the class that contains the main method usually should be static as well.

return-type: After the word static comes the return type, which indicates whether the method returns a value when it is called — and if so, what type the value is. If the method doesn’t return a value, specify void.

If you specify a return type other than void, the method must end with a return statement that returns a value of the correct type. For more information, see return Statement.

method-name: Now comes the name of your method. The rules for making up method names are the same as the rules for creating other identifiers: Use any combination of letters and numbers, but start with a letter.

parameter list: You can pass one or more values to a method by listing the values in parentheses following the method name. The parameter list in the method declaration lets Java know what types of parameters a method should expect to receive and provides names so that the statements in the method’s body can access the parameters as local variables.

This would be written

public void add(int a, int b)

{

// do stuff here

}

Since the return type is void, you will have to write what you want the method to do inside of the method such as by printing it out from the method

public void add(int a, int b)

{

System.out.println(a+b);

}

To use the above method, you would simply call that method and inserting the two integers that will go into the integer parameters:

add(7, 4);

If we want to use the results for something else within our main method, we will need a return type. Since we are adding integers, int return type is adequate:

public int add(int a, int b)

{

// do stuff here

// return type required

}

Since we have an integer return type, we need to have a return statement that will return an int. We can use a local variable (c) to hold the result and return the value of that local variable. Note that c must be the same datatype as the return type - which is int in this example:

public static int add(int a, int b)

{

int c = a+b;

return c;

}

This can also be shortened to do away with the local variable:

public static int add(int a, int b)

{

return a + b;

}

3.

In order to use the above with the int return type, In the main method you can either set a variable to the result:

int temp = add(5, 7);

or you can print out the result by calling it in your print statement:

System.out.println( add(3,3));

A parameter allows you to pass in a value to a method as you call it.

public static void name(type name) { // declare

methodName(expression); // call

Example:

public static void squared(int num) {

System.out.println(num + " times " + num + " is " + (num \* num));

}

...

squared(3); // 3 times 3 is 9

squared(8); // 8 times 8 is 64

Solving "Parameter Mystery" problems example

* Note the values passed, the order in which they are passed, and the order the method prints them.
* public class MysterySoda {
* public static void main(String[] args) {
* String soda = "coke";
* String pop = "pepsi";
* String pepsi = "soda";
* // #1 = "coke", #2 = "pepsi", #3 = "soda"
* carbonated(soda, pop, pepsi);
* }
* // #1 #2 #3
* public static void carbonated(String coke, String soda, String pop) {
* // #2 #3 #1
* System.out.println("say " + soda + " not " + pop + " or " + coke);
* }
* }
* output: // say#2pepsi not#3soda or#1coke

## **Passing information between methods**

What if you want the behavior of a *method* to vary depending on a value of a certain variable in your main *method*? It is possible to *pass* several values to *method*, and receive up to one value back. You may for example have several numbers you would like some calculation performed on. You could pass these values to a *method* and then receive the result back.

Let's start with how to give, or *pass*, a value to a *method*. This is done by giving the *method* an *argument*, which goes between the round brackets () after the *method* name. Look at the following example, an explanation follows.

/\*\*

\* Purpose: Example of program with two methods.

\* The main method uses the printSquare method.

\* The printSquare takes an double argument and prints the

\* value of the argument squared to the screen.

\*/

public class ArgExample

{

// method to print square of a value to screen

public static void printSquare(double y)

{

System.out.println(y\*y);

return;

}

// main method

public static void main(String[] argv)

{

double x = 5.0;

printSquare(3.0); // call to print the square of 3.0

printSquare(x); // call to print the square of x

}

}

The *method* printSquare is *declared* as public static void printSquare(double y). Unlike before, the brackets that follow the *method* name now contain a variable declaration, in this case for a variable y which is of *type* double. The variable y exists within this method and can be used here for calculations etc..

Because the variable declaration for y is in the round brackets, whenever the *method* printSquare is called the *method* expects a double value to be in the round brackets in the call statement. So the call for printSquare may look like:

printSquare(3.0);

In the printSquare *method* y then takes this value, for the case above 3.0, so the value 9.0 is printed to the screen.

Within the main *method* a variable x is declared and given a value. When the call printSquare(x) is used, y takes whatever value x has in the main *method* at the time, in this case 5.0, so 25.0 is printed to the screen.

It is possible for a *method* to have several arguments. These should be separated by commas, e.g.

public static void calculate(double x, double y, int i, short n)

{

//method code goes here

return;

}

then to call the *method* simply enter values, or variables, of the expected *type*:

calculate(3.0, result, 2, num) // where result and num are

// double and short variables respectively