**ICS 102 Lab10: Static methods**

**Objectives:** 1. Learning to pass and return primitive values and 1D-arrays to static methods.

2. Learning method overloading.

3. Learning methods with variable number of parameters.

3. Learning how to handle invalid arguments to methods.

1. **The general form of a method:**

**<access-specifier> <static> returnType methodName(<parameters>)**

**{**

**statement**

**statement**

**statement**

**…**

**…**

**statement**

**}**

Where **< >** denotes parts that may be missing in the definition

**access-specifier**: None 🡪 the method can be invoked by code anywhere in the class package.

**public** 🡪 the method can be invoked by code in all packages.

**private** 🡪 the method can only be invoked by code in its own class.

**protected** (Not in ICS 102 Syllabus)

**static**: if present, it indicates that the method belongs to the class and it is shared by all objects of the class

**returnType**: a primitive type, a class type, or **void** if the method returns no value

**parameters**: a list of zero or more variable declarations, separated by comma.

The call to a **static** void method is a statement, it has the general form:

**ClassName.methodName(argumentList);**

Where:

* **ClassName** is the class name in which the method is defined.
* **argumentList** is a list of zero or more expressions that are convertible to the types of the corresponding parameters.

Note:

* The ClassName may be ommited if the method call is in the same class as the called method.
* For a method whose number of arguments is fixed, the number of arguments must be equal to the number of corresponding parameters.
* It is a compile error to call a non-static method directly from a static method: **non-static method cannot be referenced from a static context.**

The call to a static method that returns a single value is an expression, it has the general form:

**ClassName.methodName(argumentList)**

**Note:** A Java method:

* Cannot be nested inside another method.
* Can return zero or one value. The returned value can be a reference to an object.
* Passes each argument call by value.

**2. void static methods and static methods returning primitive values**

Consider the following Java program that displays the sum of two numbers and a table of numbers from 1 to 10, their squares and square roots:

**public class StaticMethodExamples01 {**

**public static void main(String[] args) {**

**int num1 = 4, num2 = 8;**

**int sum = num1 + num2;**

**System.out.printf("%d + %d = %d%n%n", num1, num2, sum);**

**System.out.printf("%6s%10s%16s%n%n", "Number", "Square", "Square Root");**

**for(int k = 1; k <= 10; k++)**

**System.out.printf("%6d%10d%16.2f%n", k, k\*k, Math.sqrt(k));**

**}**

**}**

We can place the addition operation and the displaying of the table in separate static methods:

**public class StaticMethodExamples {**

**public static void main(String[] args) {**

**int num1 = 4, num2 = 8;**

**int sum = sum2(num1, num2); // method call to a non-void method is an expression**

**System.out.printf("%d + %d = %d%n%n", num1, num2, sum);**

**displayTable(); // method call to a void method is a statement**

**}**

**private static int sum2(int x, int y){**

**return x + y;**

**}**

**private static void displayTable( ){**

**System.out.printf("%6s%10s%16s%n%n", "Number", "Square", "Square Root");**

**for(int k = 1; k <= 10; k++)**

**System.out.printf("%6d%10d%16.2f%n", k, k\*k, Math.sqrt(k));**

**}**

**}**

**3. Passing 1D array references to methods**

An array is passed to a method by using the array reference variable as an argument, the corresponding

parameter is declared as a reference to a one-dimensional array.

Example:

**public class Passing1DArraysToMethods{**

**public static void main(String[] args){**

**double[] grades = {80.0, 65.5, 40.0, 72.5, 90.0, 35.8, 56.5};**

**System.out.print("The grades are: ");**

**displayArray(grades);**

**double average = calculateAverage(grades);**

**System.out.println("\nThe average is " + average);**

**}**

**private static double calculateAverage(double[] x){**

**double sum = 0;**

**for(int k = 0; k < x.length; k++)**

**sum += x[k];**

**return sum / x.length;**

**}**

**private static void displayArray(double[] array){**

**for(int k = 0; k < array.length; k++)**

**System.out.print(array[k] + " ");**

**}**

**}**

**Note:** In Java an array reference is passed to a method **call by value**. If we pass a copy of an array reference to a method, the method can modify the contents of the array. However, the method cannot change the original reference to refer to a new array, because the method only has a copy of the reference.

Example:

**public class CallExample04 {**

**public static void main(String[] args) {**

**int[] x = {1, 2, 3, 4, 5};**

**System.out.println("Before modification: ");**

**System.out.println("x[1] = " + x[1] + ", x[4] = " + x[4]);**

**modify(x);**

**System.out.println("After modification: ");**

**System.out.println("x[1] = " + x[1] + ", x[4] = " + x[4]);**

**}**

**private static void modify(int[] array){**

**array[1] = 35;**

**array[4] = 22;**

**}**

**}**

Output:

**Before modification:**

**x[1] = 2, x[4] = 5**

**After modification:**

**x[1] = 35, x[4] = 22**

**4. Methods returning 1D array references**

Example:

# import java.util.Scanner;

# public class ReturnArrayReference {

# 

# private static double[] readArray(){

# Scanner scanner = new Scanner(System.in);

# System.out.printf("Enter number of elements: ");

# int size = scanner.nextInt();

# double[] x = new double[size];

# for(int index = 0; index < size; index++){

# System.out.printf("Enter element x[%d]: ", index);

# x[index] = scanner.nextDouble();

# }

# return x;

# }

# 

# private static void printArray(double[] array){

# for(int k = 0; k < array.length; k++)

# System.out.printf("%.1f ", array[k]);

# }

# 

# public static void main(String[] args) {

# double[] values;

# values = readArray();

# System.out.printf("%nThe array is:%n");

# printArray(values);

# }

# }

**Note:** It is a good Java programming practice for a Java method that returns a reference to an array to return a reference to a non-partial array.

Example:

**// Finds the even elements of an array**

**import java.util.Scanner;**

**public class Example4 {**

**public static void main(String[] args) {**

**Scanner scanner = new Scanner(System.in);**

**int[] values, evenValues;**

**values = readArray();**

**evenValues = getEvenValues(values);**

**if(evenValues.length == 0)**

**System.out.printf("There are no even values in the array.");**

**else if (evenValues.length == 1)**

**System.out.printf("The even value in the array is: %d", evenValues[0]);**

**else{**

**System.out.printf("%nThe even values in the array are:%n");**

**printArray(evenValues);**

**}**

**}**

**private static int[] readArray(){**

**Scanner scanner = new Scanner(System.in);**

**System.out.printf("Enter number of elements: ");**

**int size = scanner.nextInt();**

**int[] x = new int[size];**

**for(int index = 0; index < size; index++){**

**System.out.printf("Enter element x[%d]: ", index);**

**x[index] = scanner.nextInt();**

**}**

**return x;**

**}**

**private static void printArray(int[] array){**

**for(int k = 0; k < array.length; k++)**

**System.out.printf("%d ", array[k]);**

**}**

**private static int[] getEvenValues(int[] x){**

**int count = 0, m = 0;**

**for(int k = 0; k < x.length; k++){**

**if(x[k] % 2 == 0){**

**count++;**

**}**

**}**

**int[] z = new int[count];**

**for(int k = 0; k < x.length; k++){**

**if(x[k] % 2 == 0){**

**z[m] = x[k];**

**m++;**

**}**

**}**

**return z;**

**}**

**}**

**5. Method overloading**

Method Overloading is a feature that allows a class to have two or more methods having the same name, if their parameter lists are different (i.e., if their signatures are different).

Parameter lists could differ in:

1. Number of parameters.

2. Data type of parameters.

3. Sequence of Data type of parameters.

What is not included in a method signature is the return type of the method and the names of the parameters.

Examples of method overloading:

**public class StaticOverloadingTest {**

**public static void main(String args[]) {**

**greet("Muhsin");**

**greet("Yusuf", "Good Morning");**

**}**

**private static void greet(String name){**

**System.out.println("Hello " + name);**

**}**

**private static void greet(String name, String greeting){**

**System.out.println(greeting + " " + name);**

**}**

**}**

**-----------------------------------------------------------------------------------------------**

**public class DisplayOverloading{**

**private static void display(char c){**

**System.out.println(c);**

**}**

**private static void display(int c){**

**System.out.println(c);**

**}**

**private static void display(char c, int num) {**

**System.out.println(c + " "+num);**

**}**

**private static void display(int num, char c){**

**System.out.println("I am the fourth definition of method display" );**

**System.out.println("These were passed to me " + num + " and " + c );**

**}**

**public static void main(String[] args){**

**display(2, 'A');**

**display('B');**

**display(2);**

**display('G', 4);**

**}**

**}**

**6. Methods with variable number of arguments**

In JDK 5, Java has included a feature that simplifies the creation of methods that need to take a variable number of arguments. This feature is called **varargs** and it is short-form for variable-length arguments. A method that takes a variable number of arguments is a **varargs** method.

Prior to JDK 5, variable-length arguments could be handled in two ways. One by using overloaded methods and another by putting the arguments into an array, and then passing this array to the method. Both of them are potentially error-prone and require more code. The **varargs** feature offers a simpler, better option.

**Syntax of varargs :**

A variable-length argument is specified by ellipsis i.e. three dots after the data type:

returnType ***methodName***(**dataType ...** **variableName**){ // method body}

This syntax tells the compiler the method ***methodName*** can be called with zero or more arguments. As a result, ***variableName*** is implicitly declared as an array of type **dataType[ ].**

**// Java program to demonstrate varargs**

**class Test1 {**

**// A method that takes variable number of integer arguments.**

**private static void fun(int ... a){**

**System.out.println("Number of arguments: " + a.length);**

**for (int i: a)**

**System.out.print(i + " ");**

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

**}**

**// Driver code**

**public static void main(String args[]){**

**// Calling the varargs method with different number of parameters**

**fun(100);         // one parameter**

**fun(1, 2, 3, 4);  // four parameters**

**fun();            // no parameter**

**}**

**}**

**Output:**

Number of arguments 1

100

Number of arguments 4

1 2 3 4

Number of arguments 0

## Rules for varargs:

* There can be only one variable argument in a method.

Example: **void** method(String ... a, **int** ... b){}//Compile time error

* Variable argument (varargs) must be the last argument in a method.

Examples: **void** method(**int** ... a, String b){}//Compile time error

int nums(int a, float b, double ... c) { } // valid

* All actual parameters must have types that are upward convertible to the **varags** formal parameter type.

Example:

For the varags method: **private static void fun(double ... x){// method body}**

The call **fun(10, 12, 27.2, -5.4)** is valid, but the call **fun(10, "big", 15.7)** is not valid because the

type **String** is not convertible to **double** the type of the varags **x**.

**6. Handling invalid arguments to a method**

It is good programming practice to make a called method throw an exception incase of invalid arguments and then let the calling method handle the exception.

Example:

**import java.util.\*;**

**public class Circle2{**

**private static double area(double radius)throws IllegalArgumentException {**

**if(radius < 0)**

**throw new IllegalArgumentException("Negative radius");**

**else**

**return Math.PI \*radius \* radius;**

**}**

**public static void main(String[] args){**

**Scanner scanner = new Scanner(System.in);**

**double circleArea = 0, radius;**

**boolean inputError;**

**do{**

**System.out.printf("Enter in cm radius >= 0: ");**

**try{**

**radius = scanner.nextDouble();**

**circleArea = area(radius);**

**inputError = false;**

**}**

**catch(IllegalArgumentException e){**

**System.out.println(e);**

**inputError = true;**

**}**

**catch(InputMismatchException e){**

**System.out.println(e);**

**inputError = true;**

**scanner.nextLine(); // clear the input buffer**

**}**

**} while(inputError);**

**System.out.printf("Circle area = %.2f sqr cm %n", circleArea);**

**}**

**}**

**Lab tasks**

**Task01**

For any integer *n* ≥ 0, ***n* !** , the factorial of n, is defined as the product *n* \* *n* - 1 \* *n* − 2 … \* 2 \* 1 where **0!** is defined to be 1

The factorial of **n** can be approximated, as a value of type **double**, by the formula:



Write a Java program that calls a static **void** method **instruction** that displays the following message:

This program computes the approximate value of the factorial using the formula proposed by R. W. Gosper.

To use this program, the input should be non-negative. Also use values not greater than 143

The program then prompts for and reads an integer, computes, and finally displays the factorial of the input in the format shown in the sample program run below.

The factorial must be computed in a static method **factorial** that uses the above formula. The method must throw an **IllegalArgumentException** if n < 0 or if n > 143.

Note:

* The method factorial must not contain any input or output statements.
* Your main method must handle both **java.util.InputMismatchException** and **java.lang.IllegalArgumentException**.

Sample program runs:

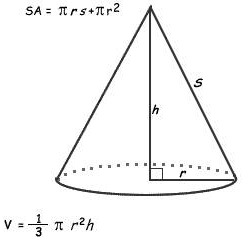
|  |
| --- |
|  |
|  |
|  |
|  |

**Task 02**

Write a complete Java program that prompts for and reads the volume, in cubic cm, and height, in cm, of a cone, it then calculates and displays the surface area of the cone in square cm.

Note: Your main method must:

* Use two static methods **getRadius** and **getSurfaceArea**. There must be no input or output statements in these two methods. Each of these methods must throw **IllegalArgumentException** if any of its arguments is invalid.
* Handle both **java.util.InputMismatchException** and **java.lang.IllegalArgumentException**.



Sample program runs:

|  |
| --- |
|  |
|  |
|  |
|  |

**Task 03**

Write a static method that receives a time in seconds and returns the equivalent time in hours, minutes, and seconds. Write an interactive main method to test your method. The method must throw an **IllegalArgumentException** if time < 0.

**Hint:** Use integer division and remainder.

**Note:** Since in Java a method can return zero or one value; to return three values from a Java method, return a 3-element 1D-array reference (or return a **String**, in which case use a **String** scanner in the calling program to get the returned values).

Sample program runs:

|  |
| --- |
|  |
|  |
|  |
|  |
|  |

**Task 04**

Write a Java program that prompts for and reads the size **n** of a 1D-array **x** of type double. Your program must validate the input by looping and displaying an error message until a valid value is entered. If the size **n** is valid, the program prompts for and reads **n** values into the array. It then passes this array to a static method **reverseArray** that creates and returns an array reference to an array which is the reverse of the array **x**. The program finally prints the reversed array on the screen using a static method **printArray**.

Note: The method **reverseArray** must not modify the array whose reference is passed to the method.

Sample program run:

|  |
| --- |
|  |

**Task 05**

Write a Java program that prompts for and reads values into an integer array **x** of size 10. It then prompts for and reads an integer value **v**. It finally passes the array **x** and the value **v** to a static method **largerElements** that returns a reference to an array containing the elements of **x** that are larger than **v**, if any. The **main** method then prints the larger elements.

**Note**: The array whose reference is returned to the **main** must not be a partial array.

Sample program runs:

|  |
| --- |
|  |
|  |
|  |
|  |