Java Foundation with Data Structures

Lecture 5 : Functions, Variables and Scope

ₛFunctions

A Function is a collection of statements designed to perform a specific task. Function is like a black box that can take certain input(s) as its parameters and can output a value which is the return value. A function is created so that one can use it as many time as needed just by using the name of the function, you do not need to type the statements in the function every time required.

Defining Function

return_type function_name(parameter 1, parameter 2, ...) {

    statements;

}

Function

Inputs
(Parameters of the
function)

Output
(Return
value by the
function)

Collection of Statements
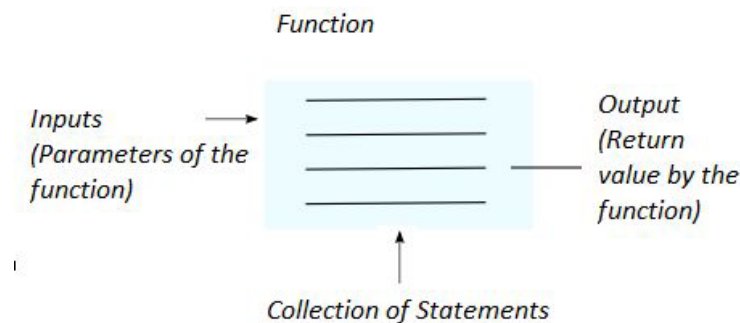
- ● **return type:** A function may return a value. The *return type* of the function is the data type of the value that function returns. Sometimes function is not required to return any value and still performs the desired task. The return type of such functions is *void.*

Example:

Following is the example of a function that sum of two numbers. Here input to the function are the numbers and output is their sum.

```
1.  public static int findSum( int a, int b){
2.          int sum = a + b;
3.          return  sum;
4.  }
```

Function Calling

Now that we have read about how to create a function lets see how to call the function. To call the function you need to know the name of the function and number of parameters required and their data types and to collect the returned value by the function you need to know the return type of the function.

Example

```
1.  public static int findSum( int a, int b){
2.      int sum = a + b;
3.      return  sum;
4.  }
5.  public static void main () {
6.      int a = 10, b = 20;
7.      int  c= findSum (a, b); // function findSum () is called using its name and
        by knowing
8.      System.out.print(c);// the number of parameters and their data type.
9.  }                        // integer c is used to collect the returned value by
        the function
```

Output:
30

IMPORTANT POINTS:
*   *Number of parameter* and their *data type* while calling must match with function signature. Consider the above example, while calling function *findSum ()* the number of parameters are two and both the parameter are of integer type.

*   It is okay not to collect the return value of function. For example, in the above code to find the sum of two numbers it is right to print the return value directly.

"System.***out***.print(c);"

- **void return type functions:** These are the functions that do not return any value to calling function. These functions are created and used to perform specific task just like the normal function except they do not return a value after function executes.

Following are some more examples of functions and their use to give you a better idea.

Function to find area of circle

1. **public static double** findArea(**double** radius){
2.   **double** area = radius*radius*3.14;   //return type is double
3.   **return** area;
4. }
5.
6. **public static void** main(String[] args) {
7.   **double** radius = 5.8;
8.   **double** c = *findArea*(radius);
9.   System.***out***.print(c);
10. }
11.

Function to print average

1. **public static void** printAverage(int a, int b){ *//return type of the function is void*
12.   **int** avg = (a + b) / 2;
13.   System.***out***.print(avg);
2. }   *// This function does not return any value*
3.
4. **public static void** main () {
5.   **int** a = 15, b = 25;
6.   printAverage (a, b);
7. }

Why do we need function?

- **Reusability:** Once a function is defined, it can be used over and over again. You can call the function as many time as it is needed, which saves work. Consider that you are required to find out the area of the circle, now either you can apply the formula every time to get the area of circle or you can make a function for finding area of the circle and invoke the function whenever it is needed.
- **Neat code:** A code created with function is easy to read and dry run. You don't need to type the same statements over and over again, instead you can invoke the function whenever needed.
- **Modularisation –** Functions help in modularising code. Modularisation means to divides the code in small modules each performing specific task. Functions helps in doing so as they are the small fragments of the programme designed to perform the specified task.
- **Easy Debugging:** It is easy to find and correct the error in function as compared to raw code without function where you must correct the error (if there is any) everywhere the specific task of the function is performed.
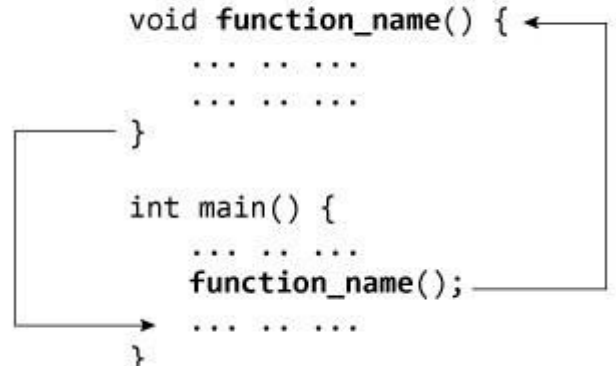
How does function calling works?

Consider the following code where there is a function called findsum which calculates and returns sum of two numbers.

```
//Find Sum of two integer numbers
1.  public static int findSum( int a, int b){
2.      int sum = a + b;
3.      return  sum;
4.  }
5.  public static void main () {
6.      int a = 10, b = 20;
7.      int  c= findSum (a, b);
8.      System.out.print(c);
9.
```

The function being called is called **callee(**here it is findsum function) and the function which calls the callee is called

```
#include <iostream>

void function_name() {  ←
    ... .. ...
    ... .. ...
}

int main() {
    ... .. ...
    function_name();
    ... .. ...
}
```

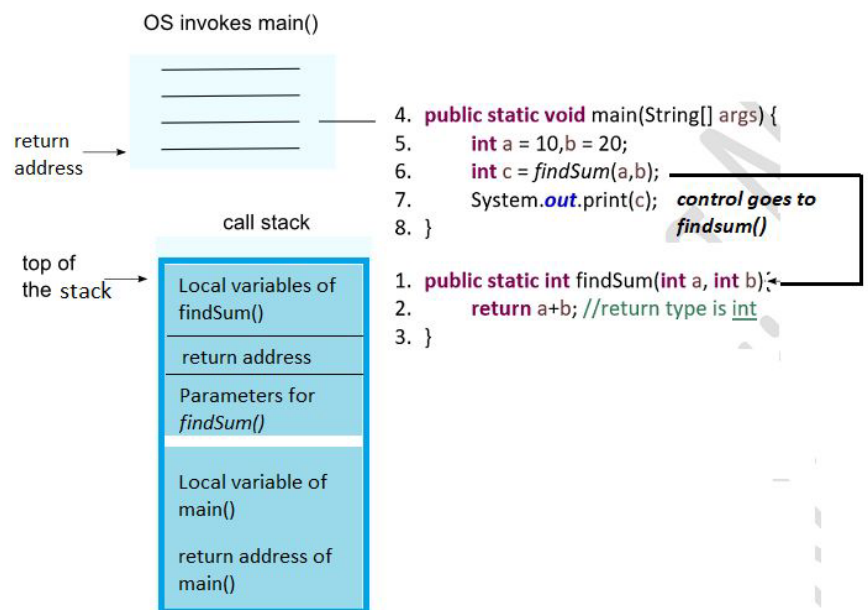the **caller** (here main function is the caller) .
When a function is called, programme control goes to the entry point of the function. Entry point is where the function is defined. So focus now shifts to callee and the caller function goes in paused state .

For Example: In above code entry point of the function *findSum ()* is at line number 3. So when at line number 9 the function call occurs the control goes to line number 3, then after the statements in the function *findSum ()* are executed the programme control comes back to line number 9.

Role of stack in function calling (call stack)

A call stack is a storage area that store information about the *active* function and paused functions. It stores parameters of the function, return address of the function and variables of the function that are created statically.

Once the function statements are terminated or the function has returned a value, the call stack removes all the information about that function from the stack.

OS invokes main()

return address

call stack

top of the stack → Local variables of findSum()

return address

Parameters for findSum()

Local variable of main()

return address of main()

```
4. public static void main(String[] args) {
5.       int a = 10,b = 20;
6.       int c = findSum(a,b);
7.       System.out.print(c);   control goes to
8. }                            findsum()

1. public static int findSum(int a, int b){
2.       return a+b; //return type is int
3. }
```

Benefits of functions

- **Modularisation**
- **Easy Debugging:** It is easy to find and correct the error in function as compared to raw code without function where you must correct the error (if there is any) everywhere the specific task of the function is performed.
- **Neat code:** A code created with function is easy to read and dry run.

Variables and Scopes

Local Variables

Local variable is a variable that is given a local scope. Local variable belonging to a function or a block has its scope only within the function or block inside which it is declared. Scope of a variable is part of a programme for which this variable is accessible.

Example:

```
1.  #include<iostream>
2.  using namespace std;
3.  public static void main(){
4.      int  a = 10;
5.      System.out.print(a);
6.
7.  }
```

Output
5
In the above code the variable *a* declared inside the block after if statement is a local variable for this block.

Lifetime of a Variable

The lifetime of a variable is the time period for which the declared variable has a valid memory. Scope and lifetime of a variable are two different concepts, scope of a variable is part of a programme for which this variable is accessible whereas lifetime is duration for which this variable has a valid memory.

Loop variable

Loop variable is a variable which defines the loop index for each iteration.

Example

"for (int *i* = 0; i < 3; i++) {   // variable *i* is the loop variable
        ….....;
        …......;
        statements;
    } "
For this example, variable *i* is the loop variable.

Variables in the same scope
Scope is part of programme where the declared variable is accessible. In the same scope, no two variables can have name. However, it is possible for two variables to have same name if they are declared in different scope.

Example:

1. **public static void** main(String[] args) {
2.   **int** a = 10;
3.   **double** <u>a</u> = 5; // two variables with same name, the code will not compile
4.   System.**out**.println(a);
5. }

For the above code, there are two variables with same name **a** in the same scope of main () function. Hence the above code will not compile.

Pass by value:

When the parameters are passed to a function by pass by value method, then the formal parameters are allocated to a new memory. These parameters have same value as that of actual parameters. Since the formal parameters are allocated to new memory any changes in these parameters will not reflect to actual parameters.

Example:
    //Function to increase the parameters value

1. **public static void** increase(**int** x, **int** y){
2.   x++;
3.   y = y + 2;
4.   System.**out**.println(x + ":" + y); // *x and y are formal parameters*
5. }
6. **public static void** main(String[] args) {
7.   **int** a = 10;
8.   **int** b = 20;
9.   *increase*(a,b);
10.   System.**out**.println(a + ":" + b); // *a and b are actual parameters*
11.
12.}

Output:
11: 22
10: 20

For the above code, changes in the values of *x* and *y* are not reflected to *a* and *b* because x and y are formal parameters and are local to function increment so any changes in their values here won't affect variables a and b inside main.