How Functions Work?

This lesson helps us understand how a function works i.e., how a function is called, how it performs the task and how it returns the result.

WE'LL COVER THE FOLLOWING

Calling a function

Starting a function so that it achieves its task is called **calling a function**. The function call syntax is the following:

```
function_name(parameter_values)
```

The actual values that are passed to functions are called **function arguments**. Although the terms parameter and argument are sometimes used interchangeably in the literature, they signify different concepts.

The arguments are matched to the parameters one by one in the order that the parameters are defined. For example, the last call of `printMenu()` in the previous lesson uses the arguments `Return` and `1`, which correspond to the parameters `firstEntry` and `firstNumber`, respectively.

> **Note:** The type of each argument must match the type of the corresponding parameter.

Doing work

In previous chapters, we have defined expressions as entities that do work. Function calls are expressions as well: they do some work. Doing work means producing value or having an effect on the program state:

- **Producing a value:** Some operations only produce values. For example, a function that adds numbers would be producing the result of that addition. As another example, a function that makes a `Student` object by using the student's name and address would be creating a `Student` object.

- **Having side effects:** Effects are any changes in the state of the program or its environment. Some operations have only side effects. An example is how the `printMenu()` function above changes `stdout` by printing to it. As another example, a function that adds a `Student` object to a student container would also have a side effect: it would be causing the container to grow.
  In summary, operations that cause a change in the state of the program have side effects.

- **Having side effects and producing a value:** Some operations do both. For example, a function that reads two values from `stdin` and returns their sum would have side effects due to changing the state of `stdin` and also produce the sum of the two values.

- **No operation:** Although every function is designed as one of the three categories above, depending on certain conditions at the compile time or at the run time, some functions end up doing no work at all.

The return value

The value that a function produces as a result of its work is called its **return value**. This term comes from the observation that once the program execution branches into a function, it eventually returns to the point from where the function was called. Functions get called, and they return values.

Just like any other value, return values have types. The type of the return value is specified right before the name of the function, at the point where the function is defined. For example, a function that adds two values of type `int` and returns their sum also as an `int` would be defined as follows:

```
int add(int first, int second) {
  // ... the actual work of the function ...
}
```

The value that a function returns takes the place of the function call itself. For example, assuming that the function call `add(5, 7)` produces the value 12, then the following two lines would be equivalent:

```
writeln("Result: ", add(5, 7));
writeln("Result: ", 12);
```

In the first line above, the `add()` function is called with the arguments `5` and `7` before `writeln()` gets called. The value 12 that the function returns is in turn passed to `writeln()` as its second argument.

This allows passing the return values of functions to other functions in complex expressions:

```
writeln("Result: ", add(5, divide(100, studentCount())));
```

In the line above, the return value of `studentCount()` is passed to `divide()` as its second argument. Then, the return value of `divide()` is passed to `add()` as its second argument, and eventually,a the return value of `add()` is passed to `writeln()` as its second argument.

The `return` statement

The return value of a function is specified by the `return` keyword in the body of the function:

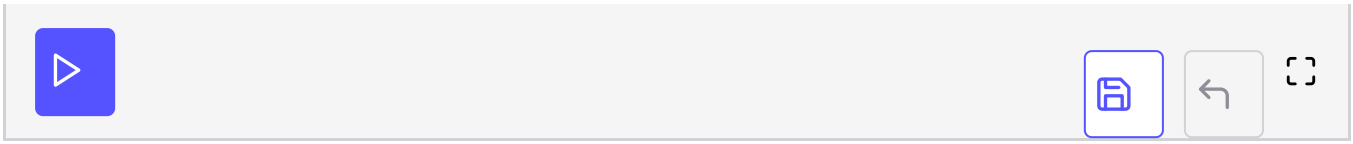
```
import std.stdio;

int add(int first, int second) {
    int result = first + second;
    return result;
}

void main() {

    writeln("Result: ", add(5, 7));
    writeln("Result: ", 12);
```

```
}
```



Defining the add function

A function produces its return value by taking advantage of statements, expressions and potentially by calling other functions. The function would then return that value by the `return` keyword, at which point the execution of the function ends.

It is possible to have more than one return statement in a function. The value of the first return statement that gets executed determines the return value of the function for a particular call:

```
int complexCalculation(int aParameter, int anotherParameter) {
    if (aParameter == anotherParameter) {
        return 0;
    }
    return aParameter * anotherParameter;
}
```

The function above returns 0 when the two parameters are equal, and the product of their values when they are different.

`void` functions

The `return` types of functions that do not produce values are specified as void. We have seen this many times with the `main()` function so far, as well as with the `printMenu()` function in the previous lesson. Since they do not return any value to the caller, their return types have been defined as `void`.

> **Note:** `main()` can also be defined as returning `int`.

Name of the function

The name of a function must be chosen so that it clearly communicates the purpose of the function. For example, the names `add` and `printMenu` were appropriate because their purpose was adding two numbers and printing a

menu, respectively.

A common guideline for function names is that they contain a verb like *add* or *print*. According to this guideline names like `addition()` and `menu()` would be less than ideal.
However, it is acceptable to name functions simply as nouns if those functions do not modify any values. For example, a function that returns the current temperature can be named as `currentTemperature()` instead of `getCurrentTemperature()`.
Coming up with names that are clear, short and consistent is part of the subtle art of programming.

In the next lesson, we will see how the quality of code can be improved with the help of functions.