Programming Languages

CS 315

Homework 3

Kutay Tire

22001787

Section 2

Nested Subprogram Definitions

Code Segment

```
print("Part 1 - Nested Subprograms");
 print("First Demo");
 int foo(x, y) {
  print("x: $x y: $y");
  int bar() {
   print("In bar x: $x y: $y");
   y = x + y;
   print("In bar x: $x y: $y");
   return x + y;}
  void foobar() {
   print("Calling foobar: ");
   bar();}
  var afterBar = bar();
  print("In foo x: $x y: $y");
  foobar();
  return afterBar;}
 foo(3,4);
 print("Second Demo");
 void out() {
  var element = 1;
  void inner1() {
   print("In inner1 element: $element");}
  void inner2(Function f) {
   f();}
  inner2(inner1);}
 out();
 print("Note that inner1() function cannot be accessed directly as it is defined in out()\n");
```

Output

```
Part 1 - Nested Subprograms
First Demo
x: 3 y: 4
In bar x: 3 y: 4
In bar x: 3 y: 7
In foo x: 3 y: 7
Calling foobar:
In bar x: 3 y: 7
In bar x: 3 y: 10
Second Demo
In inner1 element: 1
Note that inner1() function cannot be accessed directly as it is defined in out()
```

Explanation

In Dart, inner functions can have access the variables and values of the outer function. However, they cannot be accessed outside the function they are declared. In the first part of the code, a case is demonstrated where the values of local variables are changed inside a nested subprogram. As it can be seen from the output, the value of y is changed to 7.

Another attribute is that if the scope allows, an inner sub-function can call another inner sub-function. Here, the bar () method is called inside the foobar () method. In the second demo, a function is used as a parameter of another inner sub-program. Finally, an inner function cannot be called outside the enclosing function. Here, the compiler gives an error if inner1 () is tried to called outside out () function.

Nested subprograms in Dart can be useful for organizing and modularizing code, and they can also help to reduce redundancy and improve code reuse. However, it is important to use them appropriately and avoid overusing them, as they can also make code more difficult to read and understand if used excessively.

Scope of Local Variables

Code Segment

```
print("Part 2 - Scope of Local Variables");
 print("First Demo");
 var global = 5;
 // Note that global variables can be accessed with in loop. However, the index variable cannot
 // be accessed outside the for-loop.
 for (int index = 0; index < 4; index++) {
  print("index is $index");
```

```
    global++;
    print("global is $global");}
print("Second Demo");
//print(index); --> This will give an error.
var a = 0;
while (a < 5) {
    int localb = 0; // Since localb is defined in loop, the value resets to 0 after each iteration.
    var localc = 0;
    print("localb inside while-loop is $localb");
    a++;;
    while (localb < 3) {
      int innerlocald = 5;
      localc++;
      print("localc is $localc");
      localb++;}
    // print(innerlocald); --> scope of innerlocald ends after the second while loop.
  }
  //print(localb); --> scope of localb ends after the while loop.
  print("The value of global a is: $a"); // However, a can be accessed as it is global.

  // Sample of shadowing
  var t = 0;
  for (int i = 0; i < 3; i++) {
    var t = 2;
    print(t);
  }
```

Output

```
Part 2 - Scope of Local Variables
First Demo
index is 0
global is 6
index is 1
global is 7
index is 2
global is 8
index is 3
global is 9
Second Demo
localb inside while-loop is 0
localc is 1
localc is 2
localc is 3
localb inside while-loop is 0
localc is 1
localc is 2
localc is 3
localb inside while-loop is 0
localc is 1
localc is 2
localc is 3
localb inside while-loop is 0
localc is 1
localc is 2
localc is 3
localb inside while-loop is 0
localc is 1
localc is 2
localc is 3
The value of global a is: 5
2
2
2
```

Explanation

In Dart, the scope of a local variable refers to the part of the program where the variable is visible. Local variables are variables that are defined within a function or block of code, and are only accessible within that function or block of code. In this code sample, the first sample shows the local variable **index**. It is only accessible with in the for-loop whereas the global

variable **global** can also be accessed. In the second demo, the value of **localb** resets to 0 after each iteration as it is declared locally. Therefore, its lifetime ends after an iteration can it is recreated. Furthermore, the **localc** variable can be accessed within the while-loop as its scope extends the block, too.

Note that trying to access these values outside their blocks gives compilation errors. The variables **localb** or **innerlocald** cannot be accessed outside the loop since their lifetimes end with the loop. One final case is shadowing in Dart. As it can be seen from the final sample, the global variable **t** is shadowed inside the for-loop since the local variable **t** is used. Local variable are collected via an automatic garbage collector at the end when their scopes end.

Parameter Passing Methods

Code Segment

```
print("Part 3 - Parameter Passing Methods");

print("\nDart allows passing parameters by value. \n");

void functionPassByValue(String str, int num, Function fcn) {


   int res = fcn(num);

   str = str + " Tire";

   num = num + res;

   print("Inside the func. the value of num is $num and the value of str is $str");}


int myfunc(int x) {

   return x + 1;}


var str = "Kutay";

var number = 5;


print("Before func. call, the value of num is $number and the value of str is $str");

functionPassByValue(str, number, myfunc);

print("After func. call, the value of num is $number and the value of str is $str");


print("\nCase of pass-by-reference using list as it is not a primitive type.");

void increment(List<int> numbers) {

   for (int i = 0; i < numbers.length; i++) {
```

```
    numbers[i]--;}}
List<int> numbersReference = [1, 2, 3, 4, 5];
print("Before: $numbersReference");
increment(numbersReference);
print("After: $numbersReference");
```

Output

```
Part 3 - Parameter Passing Methods

Dart allows passing parameters by value.

Before func. call, the value of num is 5 and the value of str is Kutay
Inside the func. the value of num is 11 and the value of str is Kutay Tire
After func. call, the value of num is 5 and the value of str is Kutay

Case of pass-by-reference using list as it is not a primitive type.
Before: [1, 2, 3, 4, 5]
After: [0, 1, 2, 3, 4]
```

Explanation

In Dart, pass by value is used by default for primitive types. As a result, the values of an integer or a string does not change after a function call. Here, the **str** value is still “Kutay” even though it is made as “Kutay Tire” inside the loop. Similarly the value of **number** does not change and is still 5 after the function.

However, pass-by-reference can also be imitated. For instance, even though it is not shown, wrapper classes of Dart can be used to achieve this. Here, a list is given as a parameter since it is not a primitive type. After the function call, the elements inside the method are changed.

Keyword and Default Programs

Code Segment

```
print("Part 4 - Keywords and Default Parameters");
print("\nFirst Demo");
int operation1(int x, int y, {int z = 5, int k = 3}) {
   return ((x + y) * z) - k; }
// Parameters are accessed by keyword rather than order.
```

```
print(operation1(3,4)); // Optional part omitted

print(operation1(3,4, k:4));

print(operation1(3,4, k:4, z:3)); // Since it is named with {}, the order of optional parameters does not matter


print("\nSecond Demo");

int operation2(int x, int y, [int z = 5, int k = 3]) {

   return ((x + y) * z) - k;}


// Parameters are accessed by order.

print(operation2(3,4)); // Optional part omitted

print(operation2(3,4,4)); // z = 4 and the value of k is 3 by default

print(operation2(3,4,2,1)); // Since it is positional with [] the order of optional parameters matter.

print(operation2(3,4,1,2)); // z = 1, k = 2
```

Output

```
Part 4 - Keywords and Default Parameters

First Demo
32
31
17

Second Demo
32
25
13
5
```

Explanation

In Dart, default values for function parameters will take on the default value if no value is provided when the function is called. This can be useful in some cases, but it can also lead to confusion if the default value is not clearly documented or if the default value is not appropriate for all cases.

For the first demo, parameters are accessed via keywords and this is achieved in Dart via “{}” symbols. This means that parameters are passed by indicating the name of the keyword and

keyword parameters have to be passed via their respective keywords. In the first print, the values for z and k are default as they are omitted. In the second print statement, the value of k is given as 4 and z again takes the default value. In the third print statement, both z and k are given variables. Note that since it is by keyword, writing k first does not give any compilation errors.

In the second demo, parameters are accessed via order rather than keywords. This is achieved by using "[]" symbols in function definition. Since these are optional parameters, the user does not have to specify a value when calling a function unless there are no default values. In the first print statement, the optional parts are omitted and z and k take the default values. In the second print statement, the third argument which is 4 is automatically assigned to z. In the third and fourth print arguments, z and k take the values 1 and 2.

Closures

Code Segment

```
print("Part 5 - Closures");

print("\nFirst Demo");

(int number) {

   print("The value is $number");

}(5);

print("\nSecond Demo");

var names = ['Kutay', 'İpek', 'Doruk'];

Function addToList(String name) {

     return () {

     names.add(name);

     print(names);

   };}

var addName = addToList('Simge');

addName();

print("\nThird Demo");

Function addTo(int number1) {

   return (int number2) => number1 + number2;}

var element = addTo(4);

print(element(8));
```

Output

```
Part 5 - Closures

First Demo
The value is 5

Second Demo
[Kutay, İpek, Doruk, Simge]

Third Demo
12
```

Explanation

In Dart, a closure is a function object that has access to variables in its lexical scope and they can be assigned to variables. They can also be passed as arguments to other functions. The first demo samples a closure by assigning the value 5 to **number** variable. In the second and third demos, the functions **addToList** and **addTo** return closures. These closures can act like functions by taking parameters. For example, the **element** closure takes the value of 8 as a parameter and 12 is printed at the end. Closures can also hold references for variables.

Language Evaluation

Dart was a readable and a writable language in terms of the attributes asked. Nested subprograms supported a syntax that is very easy to understand. The only issue was the use of “{}” and “[]” for optional arguments as they were not very familiar to me. Furthermore, their roles were not very clear at first glance. Closures were not also very easy to comprehend. Even though it is not shown in the samples, the cases of nested closures could be really difficult to understand. Nevertheless, it was an easy language to write.

Learning Strategy

For this homework, I started by learning the principles of Dart. I checked out how nested subprograms and scoping works initially. I was familiar with the scoping from the midterm so it was not a huge problem. However, I had to learn how default parameter passing works in Dart. I usually gained these information from the official documentations of the language and sample programs in sites like GeeksforGeeks. I also learnt the basics of closures from the book and then learnt the specific cases for Dart like using arrow-syntax. Finally, I wrote and compiled the code with the help of online compiler. When I stuck on a problem, I consulted

my friends on whether they faced a similar problem. From time to time, I ran small test programs to understand a concept better or see possible result.

URLs of Online Compiler

Dart: https://dartpad.dev/?