

Control Abstraction

Dr.. Nguyen Hua Phung

HCMC University of Technology, Viet Nam

09, 2013

- 1 **Subprogram Definition**
- 2 **Subprogram Mechanisms**
 - Simple Call Return
 - Recursive Call
 - Exception
- 3 **Parameter Passing**
- 4 **Higher-order Functions**

- 1 **Subprogram Definition**
- 2 **Subprogram Mechanisms**
 - Simple Call Return
 - Recursive Call
 - Exception
- 3 **Parameter Passing**
- 4 **Higher-order Functions**

Subprogram definition consists of:

- **Specification**

- Subprogram name
- Parameters
 - input + output
 - order
 - type
 - parameter passing mechanisms: by value, by reference, by name,...
- Behaviour of the subprogram

- **Implementation:**

- Local data
- Collection of statements as **subprogram body**

```
def apply(interval: Int ,
           repeats: Boolean = true)
    (op: => Unit) {
  val timeOut = new javax.swing.AbstractAction() {
    def actionPerformed
      (e: java.awt.event.ActionEvent) = op
  }
  val t = new javax.swing.Timer(interval , timeOut)
  t.setRepeats(repeats)
  t.start()
}
```

```
def apply(interval: Int ,
           repeats: Boolean = true)
    (op: => Unit) {
  val timeOut = new javax.swing.AbstractAction() {
    def actionPerformed
      (e: java.awt.event.ActionEvent) = op
  }
  val t = new javax.swing.Timer(interval , timeOut)
  t.setRepeats(repeats)
  t.start()
}
```

- 1 How many subprogram definitions are there in the above code?
- 2 How many parameters are there in each subprogram definition?

An activation of a subprogram:

- is created when the subprogram is invoked
- is destroyed when the subprogram completed its execution

An activation of a subprogram:

- is created when the subprogram is invoked
- is destroyed when the subprogram completed its execution

An activation includes

- Static part: Code segment
- Dynamic part: Activation record
 - formal parameters
 - local data
 - return address
 - other links

1 Subprogram Definition

2 Subprogram Mechanisms

- Simple Call Return
- Recursive Call
- Exception

3 Parameter Passing

4 Higher-order Functions

- Simple Call-Return
- Recursive Call
- Exception Processing Handler
- Coroutines
- Scheduled Subprograms
- Tasks

```
int mul (Node root) {  
    if (root == null) return 1;  
    else root.val * mul(root.left)  
                * mul(root.right);  
}
```

Rewrite the above function to short-circuit the traversal if **val** of a node in the binary tree is 0?

Basic Features

- No recursion
- Explicit Call Site
- Single Entry Point
- Immediate Control Passing
- Single Execution

- Be able to call recursive
 - Direct Recursive Call
 - Indirect Recursive Call (Mutual Recursive)
- Other features same as Simple Call-Return

- May have no explicit call site

Exception Processing Handler

- May have no explicit call site
- Used in

Exception Processing Handler

- May have no explicit call site
- Used in
 - Event-Driven Programming

Exception Processing Handler

- May have no explicit call site
- Used in
 - Event-Driven Programming
 - Error Handler

Exception Processing Handler

- May have no explicit call site
- Used in
 - Event-Driven Programming
 - Error Handler

Exception Processing Handler

- May have no explicit call site
- Used in
 - Event-Driven Programming
 - Error Handler

Example,

```
class EmptyExcp extends Throwable {int x=0;};

int average(int [] V) throws EmptyExcp(){
    if (length(V)==0) throws new EmptyExcp();
    else ...
};
...
try {...
    average(W);
    ...
}
catch (EmptyExcp e) {write("Array_empty"); }
```

A language must specify:

- *which* exceptions can be handled and *how* they can be defined
- *how* an exception can be raised
- *how* an exception can be handled

- Java: subclass of *Throwable*
- Ada: values of a special type
- C++: any value

Raising exception

- By user interaction
(Click, MouseMove, TextChange, ...)

Raising exception

- By user interaction
(Click, MouseMove, TextChange, ...)
- By operating system

Raising exception

- By user interaction
(Click, MouseMove, TextChange, ...)
- By operating system
- By an object (Timer)

Raising exception

- By user interaction
(Click, MouseMove, TextChange, ...)
- By operating system
- By an object (Timer)
- By programmer (throw)

```
object Timer {  
  def apply(interval: Int,  
            repeats: Boolean = true)  
    (op: => Unit) {  
    val timeOut = new javax.swing.AbstractAction() {  
      def actionPerformed  
        (e: java.awt.event.ActionEvent) = op  
    }  
    val t = new javax.swing.Timer(interval, timeOut)  
    t.setRepeats(repeats)  
    t.start()  
  }  
}  
Timer(2000) { println("Timer went off") }  
Timer(10000, false) { println("10 seconds are over!") }
```

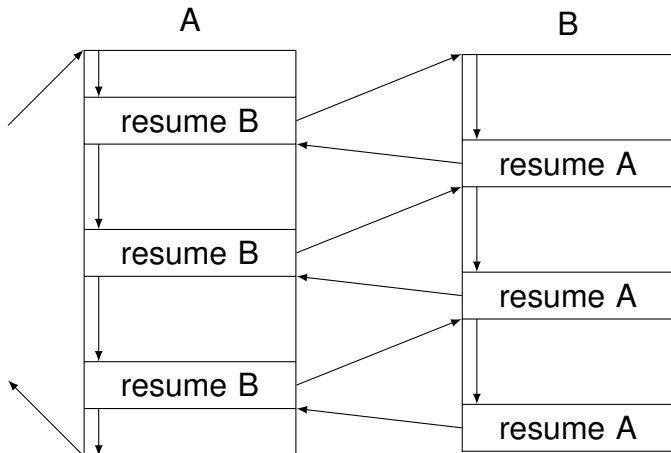
- Define *the protected block* to intercept the exception for being handled
- Define exception handler associated with the protected block

- Define *the protected block* to intercept the exception for being handled
- Define exception handler associated with the protected block

Termination Semantic

- non-resumable (common) + stack unwinding
- resumable
 - at the statement causing the error
 - after the statement causing the error

A coroutine may postpone its execution and control is back to caller. Its execution later is resumed at the place it postponed.



- able to execute concurrently with other tasks
- run on multi-processor machine or
- single processor machine using time sharing

- able to execute concurrently with other tasks
- run on multi-processor machine or
- single processor machine using time sharing

Issue?

- able to execute concurrently with other tasks
- run on multi-processor machine or
- single processor machine using time sharing

Issue?

- Synchronization
 - Race condition
 - Deadlock
- Communication


```
val pa = (0 until 10000).toArray.par  
pa.map(_ + 1)  
pa.map { v => if (v % 2 == 0) v else -v }  
pa.fold(0) { _ + _ }  
var a = 0  
pa foreach { a += _ }
```

- The execution of callee is NOT started when it is invoked
 - scheduled by time
CALL A AT TIME = CURRENT_TIME + 10
 - scheduled by priority
CALL B WITH PRIORITY 7
- Controlled by a scheduler

1 Subprogram Definition

2 Subprogram Mechanisms

- Simple Call Return
- Recursive Call
- Exception

3 Parameter Passing

4 Higher-order Functions

Definition

- Formal parameters: `int foo(float x, bool& y);`
 - just a simple name
 - close to a variable declaration
 - combine with symbols relating to parameter passing mechanism
- Actual parameters/Arguments: `foo(4*a, b)`
 - an expression

Definition

- Formal parameters: `int foo(float x, bool& y);`
 - just a simple name
 - close to a variable declaration
 - combine with symbols relating to parameter passing mechanism
- Actual parameters/Arguments: `foo(4*a, b)`
 - an expression

Formal-Actual Corresponding

- by position
`int foo(float a, int b) \Leftarrow foo(x+1, y-2)`
- by name
`int foo(float a, int b) \Leftarrow foo(b = x+1, a = y-2)`

- Input-Output
 - By value-result
 - By reference
 - By name
- Input Only
 - By value
 - By constant reference
- Output Only
 - By result
 - As a result of a function

- Pass by value-result

- Pass by value-result

caller

a

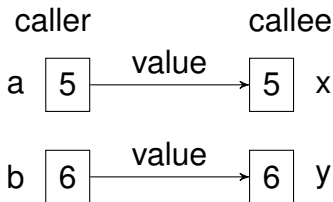
5

b

6

- Pass by value-result

`findMax(a,b) ⇒ int findMax(int x,int y) {...}`



- Pass by value-result

int findMax(int x,int y) {...}

caller

a

5

b

6

callee

7

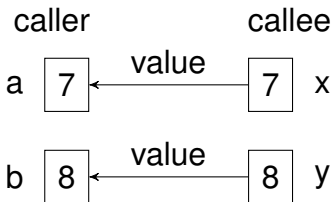
 x

8

 y

- Pass by value-result

`findMax(a,b) \Leftarrow int findMax(int x,int y) {...}`



- Pass by value-result

caller

a

7

b

8

- Pass by value-result
- Pass by reference

- Pass by value-result
- Pass by reference

caller

a

5

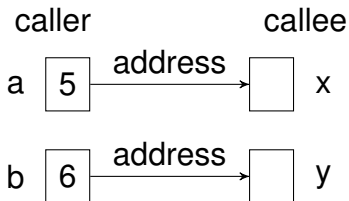
b

6

- Pass by value-result

- Pass by reference

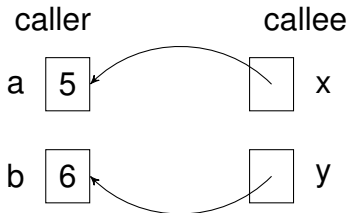
`findMax(a,b) \Rightarrow int findMax(int& x,int& y) {...}`



- Pass by value-result

- Pass by reference

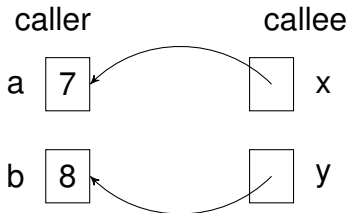
int findMax(**int**& x,**int**& y) {...}



- Pass by value-result

- Pass by reference

int findMax(**int**& x,**int**& y) {...}



- Pass by value-result

- Pass by reference

findMax(a,b) \Leftarrow

caller

a

7

b

8

- Pass by value-result
- Pass by reference
- Pass by name

- Pass by value-result

- Pass by reference

- Pass by name

`findMax(a,b) \Rightarrow int findMax(int \Rightarrow x,int \Rightarrow y) {...}`

- Pass by value-result

- Pass by reference

- Pass by name

int findMax(**int**⇒ x,**int**⇒ y) {...}

a ≡ x

b ≡ y

```
void swap(int x, int y) {  
    int t = x;  
    x = y;  
    y = t;  
}  
void main() {  
    int a[] = {2, 1, 0}; i = 0;  
    swap(i, a[i]);  
    cout << i << a[0] << a[1] << a[2];  
}
```

- Pass by value

- Pass by value

caller

a

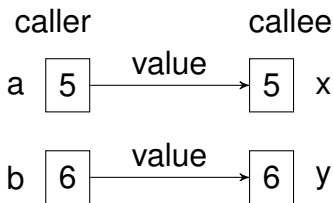
5

b

6

- Pass by value

`findMax(a,b) ⇒ int findMax(int x,int y) {...}`



- Pass by value

int findMax(int x,int y) {...}

caller

a

5

b

6

callee

7

 x

8

 y

- Pass by value

findMax(a,b) \Leftarrow

caller

a

5

b

6

- Pass by value
- Pass by constant reference

- Pass by value
- Pass by constant reference

caller

a

5

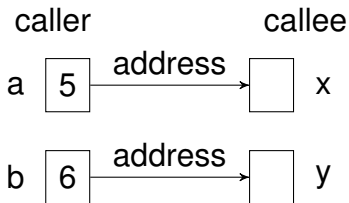
b

6

- Pass by value

- Pass by constant reference

`findMax(a,b) \Rightarrow int findMax(const int& x,const int& y) {...}`

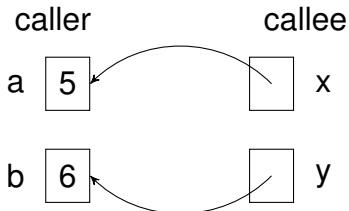


- Pass by value

- Pass by constant reference

int findMax(const int& x, const int& y)

{...}

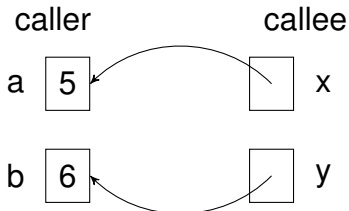


- Pass by value

- Pass by constant reference

int findMax(const int& x, const int& y)

{...}



- Pass by value
- Pass by constant reference
`findMax(a,b) \Leftarrow`

caller

a

5

b

6

- Pass by result

- Pass by result

caller

a

5

b

6

- Pass by result

`findMax(a,b) ⇒ int findMax(int x,int y) {...}`

caller

a

5

b

6

callee

x

y

- Pass by result

int findMax(int x,int y) {...}

caller

a 5

b 6

callee

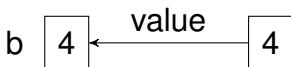
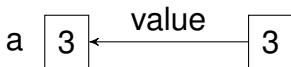
3 x

4 y

- Pass by result

`findMax(a,b) ⇐`

caller



- Pass by result
- As a result of a function

- Pass by result
- As a result of a function
`int foo() ... return 0;`
No actual parameter: `foo() + 1`


```
type VECT = array [1..3] of integer;  
type VECTPTR = @VECT;  
procedure SUB1;  
var A,B:VECT;P,Q:VECTPTR;  
begin  
    A[1] := 7;A[2] := 8; A[3] := 9;  
    B[1] := 7;b[2] := 8; B[3] := 9;  
    P := @A; Q := @B;  
    SUB2(P,Q);  
end;  
procedure SUB2(R:VECTPTR;var S:VECTPTR)  
begin  
    R^[1] := R^[1] + 10;    // 1  
    S^[1] := S^[1] + 10;    // 2  
    if ... then R := S;     // 3  
    else S := R;            // 4  
end;
```

Which storage (variable) is changed and what is its new value after changing after the following instruction is executed?

- a. // 1
- b. // 2
- c. // 3
- d. // 4

1 Subprogram Definition

2 Subprogram Mechanisms

- Simple Call Return
- Recursive Call
- Exception

3 Parameter Passing

4 Higher-order Functions

A function is *higher-order* when it accepts functions

- as its input parameters (fairly common)

A function is *higher-order* when it accepts functions

- as its input parameters (fairly common)
- as its out parameters (less common - but required in functional programming)

A function is *higher-order* when it accepts functions

- as its input parameters (fairly common)
- as its out parameters (less common - but required in functional programming)

A function is *higher-order* when it accepts functions

- as its input parameters (fairly common)
- as its out parameters (less common - but required in functional programming)

Example, in `stdlib.h` of C, there is a built-in sorting function

```
void qsort(void *base, size_t nmemb, size_t size,  
          int (*compar)(const void *, const void *));
```

A function is *higher-order* when it accepts functions

- as its input parameters (fairly common)
- as its out parameters (less common - but required in functional programming)

Example, in `stdlib.h` of C, there is a built-in sorting function

```
void qsort(void *base, size_t nmem, size_t size,  
          int(*compar)(const void *, const void *));
```

```
int  
int_sorter(const void *first_arg, const void *second_arg)  
    int first = *(int*)first_arg;  
    int second = *(int*)second_arg;  
    if (first < second) return -1;  
    else if (first == second) return 0;  
    else return 1;  
}
```


A function is *higher-order* when it accepts functions

- as its input parameters (fairly common)
- as its out parameters (less common - but required in functional programming)

Example, in `stdlib.h` of C, there is a built-in sorting function

```
void qsort(void *base, size_t nmemb, size_t size,  
          int(*compar)(const void *, const void *));
```

```
int main() {  
    int array[10], i;  
    /* fill array */  
    for ( i = 0; i < 10; ++i )  
        array[i] = 10 - i;  
    qsort(array, 10, sizeof(int), int_sorter);  
    for (i = 0; i < 10; ++i)  
        printf ("%d\n" ,array[i]);  
}
```

What is non-local environment?

- Deep binding
- Shallow binding

Example,

What is non-local environment?

- Deep binding
- Shallow binding

Example,

```
int x = 1;
int f(int y){ return x+y; }

int g (int h(int b)){
    int x = 2;
    return h(3) + x; //shallow binding
}

...
{int x = 4;
  int z = g(f); //deep binding
}
```

What is non-local environment?

- Deep binding
- Shallow binding

Example, Static scope: $z =$

```
int x = 1;
int f(int y){ return x+y; }

int g (int h(int b)){
    int x = 2;
    return h(3) + x; //shallow binding
}

...
{int x = 4;
  int z = g(f); //deep binding
}
```

What is non-local environment?

- Deep binding
- Shallow binding

Example, Static scope: $z = 6$

```
int x = 1;
int f(int y){ return x+y; }

int g (int h(int b)){
    int x = 2;
    return h(3) + x; //shallow binding
}

...
{int x = 4;
  int z = g(f); //deep binding
}
```

What is non-local environment?

- Deep binding
- Shallow binding

Example, Dynamic scope + Deep binding: $z =$

```
int x = 1;
int f(int y){ return x+y; }

int g (int h(int b)){
    int x = 2;
    return h(3) + x; //shallow binding
}
...
{int x = 4;
  int z = g(f); //deep binding
}
```

What is non-local environment?

- Deep binding
- Shallow binding

Example, Dynamic scope + Deep binding: $z = 9$

```
int x = 1;
int f(int y){ return x+y; }

int g (int h(int b)){
    int x = 2;
    return h(3) + x; //shallow binding
}

...
{int x = 4;
  int z = g(f); //deep binding
}
```

What is non-local environment?

- Deep binding
- Shallow binding

Example, Dynamic scope + Shallow binding: $z =$

```
int x = 1;
int f(int y){ return x+y; }

int g (int h(int b)){
    int x = 2;
    return h(3) + x; //shallow binding
}
...
{int x = 4;
  int z = g(f); //deep binding
}
```


What is non-local environment?

- Deep binding
- Shallow binding

Example, Dynamic scope + Shallow binding: $z = 7$

```
int x = 1;
int f(int y){ return x+y; }

int g (int h(int b)){
    int x = 2;
    return h(3) + x; //shallow binding
}

...
{int x = 4;
  int z = g(f); //deep binding
}
```

Exercise

```
var X:real;
  procedure SUB2(X,Y:real;function F(U:real):real);
  var Z:real;
  begin
    Z := abs(X - Y);
    Z := (F(X) + F(Y))* Z / 2;
    write(Z); // ??? in static-scoping language
  procedure SUB1;
  var Y:real;
    function FUNC(V:real):real;
    begin
      FUNC := X * V + Y;
    begin
      Y := 1;
      SUB2(0,1,FUNC);
    begin
      X := 3;
      SUB1;
    end.
```

```
object FileMatcher {  
  private def filesHere =  
    (new java.io.File(".")).listFiles  
  
  def filesEnding(query: String) =  
    for (file <- filesHere;  
        if file.getName.endsWith(query))  
      yield file  
  
  def filesContaining(query: String) =  
    for (file <- filesHere;  
        if file.getName.contains(query))  
      yield file  
  
  def filesRegex(query: String) =  
    for (file <- filesHere;  
        if file.getName.matches(query))  
      yield file  
}
```

Example in Scala

```
object FileMatcher {  
  private def filesHere =  
    (new java.io.File(".")).listFiles  
  
  def filesEnding(query: String) =  
    for (file <- filesHere;  
        if file.getName.endsWith(query))  
      yield file  
  
  def filesContaining(query: String) =  
    for (file <- filesHere;  
        if file.getName.contains(query))  
      yield file  
  
  def filesRegex(query: String) =  
    for (file <- filesHere;  
        if file.getName.matches(query))  
      yield file  
}
```

```
object FileMatcher {  
  private def filesHere =  
    (new java.io.File(".")).listFiles  
  
  def filesMatching(query: String ,  
    matcher: (String , String) => Boolean) = {  
    for (file <- filesHere;  
      if matcher(file.getName, query))  
      yield file  
  }  
  
  def filesEnding(query: String) =  
    filesMatching(query , _.endsWith(_))  
  
  def filesContaining(query: String) =  
    filesMatching(query , _.contains(_))  
  
  def filesRegex(query: String) =  
    filesMatching(query , _.matches(_))  
}
```

What returns as functions

What returns as functions

- Code

What returns as functions

- Code
- Environment

What returns as functions

- Code
- Environment

Example,

What returns as functions

- Code
- Environment

Example,

```
void→int F () {  
    int x = 1;  
    int g () {  
        return x+1;  
    }  
    return g;  
}  
void→int gg = F();  
int z = gg();
```

What returns as functions

- Code
- Environment

Example,

```
void→int F () {  
    int x = 1;  
    int g () {  
        return x+1;  
    }  
    return g;  
}  
void→int gg = F();  
int z = gg();
```

What returns as functions

- Code
- Environment

Example,

```
void→int F () {  
    int x = 1;  
    int g () {  
        return x+1;  
    }  
    return g;  
}  
void→int gg = F();  
int z = gg();
```

What returns as functions

- Code
- Environment

Example,

```
void→int F () {  
    int x = 1;  
    int g () {  
        return x+1;  
    }  
    return g;  
}  
void→int gg = F();  
int z = gg();
```

main

gg

z

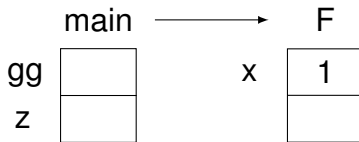


What returns as functions

- Code
- Environment

Example,

```
void->int F () {  
    int x = 1;  
    int g () {  
        return x+1;  
    }  
    return g;  
}  
void->int gg = F();  
int z = gg();
```

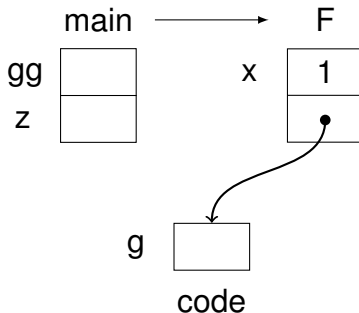


What returns as functions

- Code
- Environment

Example,

```
void->int F () {  
    int x = 1;  
    int g () {  
        return x+1;  
    }  
    return g;  
}  
void->int gg = F();  
int z = gg();
```

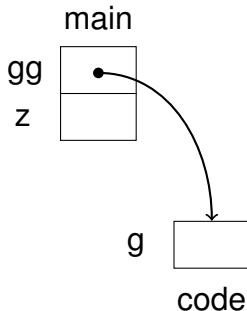


What returns as functions

- Code
- Environment

Example,

```
void->int F () {  
    int x = 1;  
    int g () {  
        return x+1;  
    }  
    return g;  
}  
void->int gg = F();  
int z = gg();
```

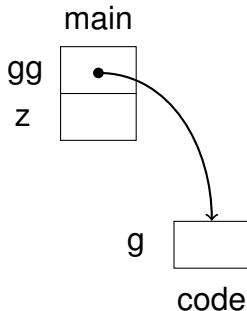


What returns as functions

- Code
- Environment

Example,

```
void->int F () {  
    int x = 1;  
    int g () {  
        return x+1;  
    }  
    return g;  
}  
void->int gg = F();  
int z = gg();
```

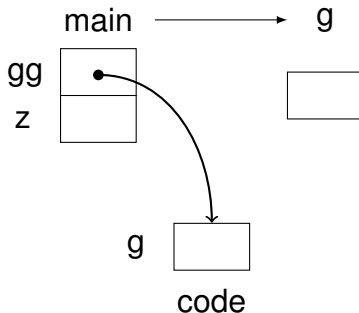


What returns as functions

- Code
- Environment

Example,

```
void->int F () {  
    int x = 1;  
    int g () {  
        return x+1;  
    }  
    return g;  
}  
void->int gg = F();  
int z = gg();
```

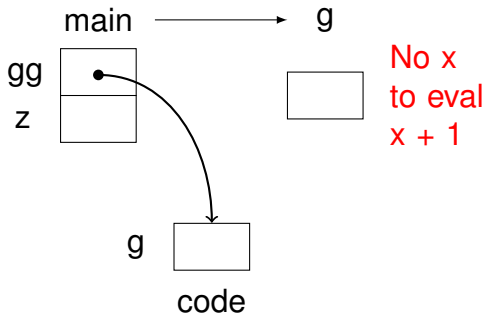


What returns as functions

- Code
- Environment

Example,

```
void->int F () {  
    int x = 1;  
    int g () {  
        return x+1;  
    }  
    return g;  
}  
void->int gg = F();  
int z = gg();
```



- 1 How to keep the environment for function as results?
- 2 Does the same problem happen for function as parameter? If your answer is Yes, please give an example.

► Skip Scala Example

- Subprogram mechanisms
 - Simple Call-Return
 - Recursive Call
 - Exception
 - Coroutine
 - Scheduled Call
 - Tasks
- Parameter Passing
- Higher-order Functions