Chapter 9

SUBPROGRAMS

Presented By:

BARCENILLA, Jomer Allan G.
PEREZ, Jameson Adriel S.
SUCALIT, Giovanni Ross P.

In this CHAPTER.

01

Introduction

02

Fundamentals of Subprograms

03

Design issues of Subprograms

04

Local Referencing Environments

05

Parameter - Passing Methods

06

Parameters that are Subprograms

In this CHAPTER.

07 Calling Subprograms Indirectly

08 Design Issues for Functions

09 Overloaded Subprograms

10 Generic Subprograms

11 User-Defined Overloaded Operators

12 Closures and Coroutines

01 INTRODUCTION

- Two fundamental abstraction facilities
  - Data abstraction
    - Discussed in Chapter 11
  - Process abstraction
    - Discussed in this Chapter

SUBPROGRAMS

 What are Subprograms?

- semi-independent set of instructions
- activated using a call
- can be used in different parts of the program

Characteristics of Subprograms

- Each subprogram has a single entry point
- The calling program is suspended during execution of the called subprogram
- The calling program resumes when the called subprogram’s execution terminates

Categories of Subprograms

Functions

Functions structurally resemble procedures but are semantically modeled on mathematical functions

- Has Return values
- Has Side Effects

Procedures

Procedures are collection of statements that define parameterized computations

- No Return values
- No Side Effects

TERMINOLOGIES

A *subprogram definition* describes the interface to and the actions of the subprogram abstraction

The examples in the next slides are subprogram definitions of a subprogram that gets the sum of two integers and returns the sum to the calling program.

TERMINOLOGIES

Subprogram definition in C

```c
int addition(int augend, int addend){
   return augend + addend;
}
```

TERMINOLOGIES

Subprogram definition in Java

```
public static int addition(int augend, int addend){
    return augend + addend;
}
```

TERMINOLOGIES

Subprogram definition in Python

```
def addition(augend, addend):
    return augend + addend
```

TERMINOLOGIES

A *subprogram call* is an explicit request that the subprogram be executed

TERMINOLOGIES

Subprogram call

C

```c
int sum = addition(5,10);
```

Python

```python
sum = addition(5, 10)
```

Java

```java
int sum = addition(5, 10);
```

TERMINOLOGIES

A *subprogram header* is the first part of the definition, including the name, the kind of subprogram, and the formal parameters.

TERMINOLOGIES

Subprogram header

C

```c
int addition(int augend, int addend){
    return augend + addend;
}
```

TERMINOLOGIES

Subprogram header

Java

```
public static int addition(int augend, int
addend){
   return augend + addend;
}
```

TERMINOLOGIES

Subprogram header

Python

```
def addition(augend, addend):
    return augend + addend
```

TERMINOLOGIES

The *parameter profile* (aka signature) of a subprogram is the number, order, and types of its parameters.

C

```c
int addition(int augend, int addend){
    return augend + addend;
}
```

Python

```python
def addition(augend, addend):
    return augend + addend
```

Java

```java
public static int addition(int augend, int addend){
    return augend + addend;
}
```

TERMINOLOGIES

The *protocol* is the subprogram's parameters profile and, if it is a function, its return type.

C

```c
int addition(int augend, int addend){
    return augend + addend;
}
```

Python

```python
def addition(augend, addend):
    return augend + addend
```

Java

```java
public static int addition(int augend, int addend){
    return augend + addend;
}
```

TERMINOLOGIES

A *subprogram declaration* provides the protocol, but not the body,of the subprogram.

In C and C++, these are called function prototypes

C

```c
int addition(int augend, int addend);

int addition(int augend, int addend){
    return augend + addend;
}
```

TERMINOLOGIES

A *formal parameter* is a dummy variable listed in the subprogram header and used in the subprogram.

C

```c
int addition(int augend, int addend){
    return augend + addend;
}
```

Python

```python
def addition(augend, addend):
    return augend + addend
```

Java

```java
public static int addition(int augend, int addend){
    return augend + addend;
}
```

TERMINOLOGIES

An *actual parameter* represents a value or address used in the subprogram call statement.

C

```c
int sum = addition(5,10);
```

Python

```python
sum = addition(5, 10)
```

Java

```java
int sum = addition(5, 10);
```

TERMINOLOGIES

```c
#include <stdio.h>

int add(int augend, int addend);

int main(){
    int x = add(10,5);
    printf("%d", x);
}

int add(int augend, int addend){
    return augend + addend;
}
```

Actual/Formal Parameter Correspondence

Positional

- The binding of actual parameters to formal parameters is by position: the first actual parameter is bound to the first formal parameter and so forth
- *Advantage:* Safe and effective

Actual/Formal Parameter Correspondence

Keyword

- The name of the formal parameter to which an actual parameter is to be bound is specified with the actual parameter
- *Advantage:* Parameters can appear in any order, thereby avoiding parameter correspondence errors
- *Disadvantage:* User must know the formal parameter's names

Actual/Formal Parameter Correspondence

```
def addition(augend, addend):
    return augend + addend
```

Positional

```
sum = addition(5,10);
```

Keyword

```
sum = addition(augend=5,addend=10);
```

Formal Parameter Default Values

In certain languages (e.g., C++, Python, Ruby, PHP), formal parameters can have default values (if no actual parameter is passed)

```
def addition(augend = 5, addend = 10):
    return augend + addend
```

Formal Parameter Default Values

In C++, default parameters must appear last because parameters are positionally associated (no keyword parameters)

```
int addition(int augend, int addend = 5){
   return augend + addend;
}
```

Variable Numbers of Parameters

- C# methods can accept a variable number of parameters as long as they are of the same type–the corresponding formal parameter is an array preceded by **params**

```
public static int addition(params int[] num)

public static string addString(params string[] names)

public void testScores(string name, params int[] num)
```

Variable Numbers of Parameters

- In Ruby, the actual parameters are sent as elements of a hash literal and the corresponding formal parameter is preceded by an asterisk.

```
def scores(*num)
    ...
end
```

```
def printAll(title, *chapters)
    ...
end
```

Variable Numbers of Parameters

- In Python, the actual is a list of values and the corresponding formal parameter is a name with an asterisk

```
def addition(*args):
    sum = 0
    for num in args:
        sum += num
    return sum
```

```
def multiplication(*args):
    product = 1
    for num in args:
        product *= num
    return product
```

Local Referencing Environments

- Local variables can be stack-dynamic
  - Advantages:
    - Support for recursion
    - Storage for locals is shared among some subprograms
  - Disadvantages:
    - Allocation/de-allocation, initialization time
    - Indirect addressing
    - Subprograms cannot be history sensitive

Local Referencing Environments

- Local variables can be static
  - Advantages and disadvantages are the opposite of those for stack-dynamic local variables

Local Referencing Environments: Examples

In most contemporary languages, locals are stack dynamic

In C-based languages, locals are by default stack dynamic, but can be declared `static`

The methods of C++, Java, Python, and C# only have stack dynamic locals

Semantic Models of Parameter Passing

IN MODE

Pass by Value

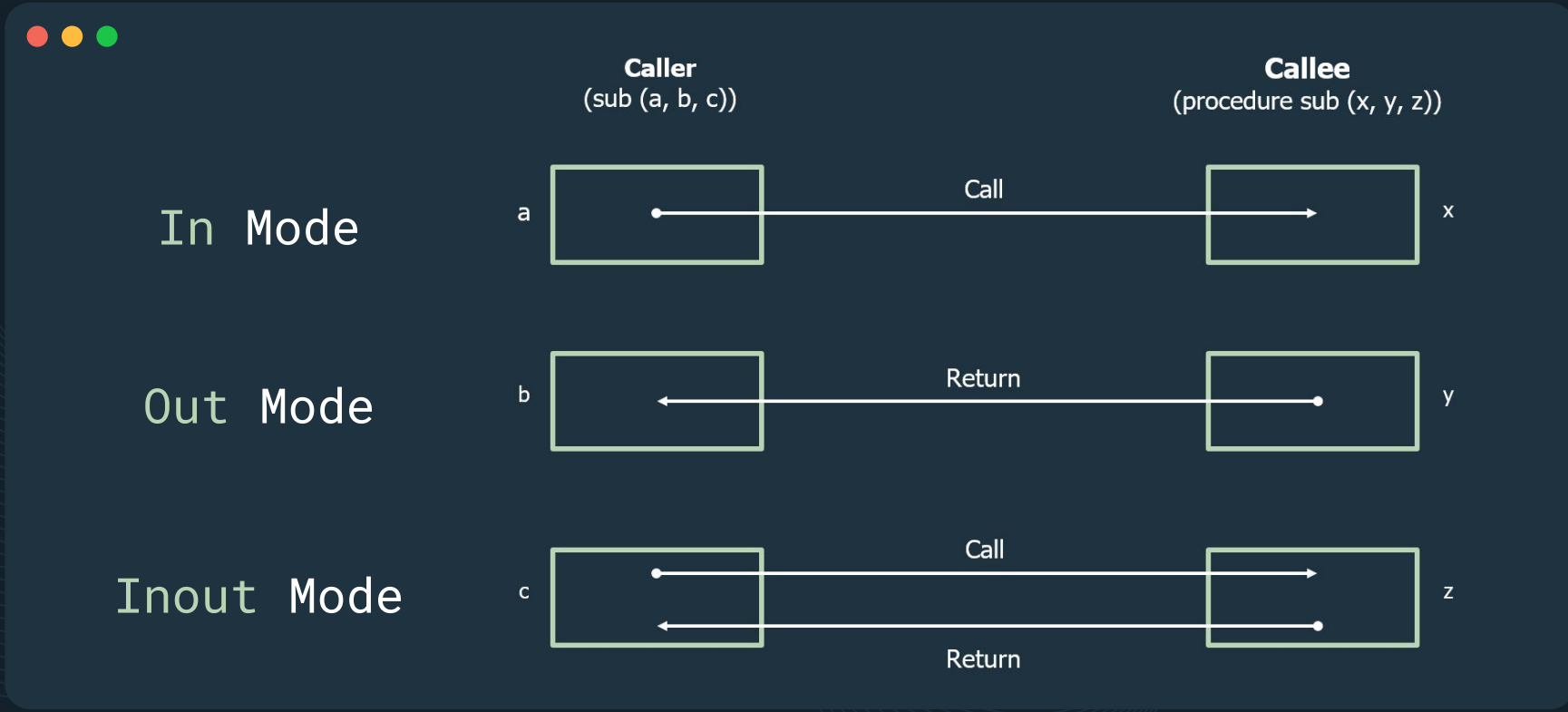
OUT MODE

Pass by Result

INOUT MODE

- Pass by Value-Result
- Pass by Reference
- Pass by Name

Models of Parameter Passing



Conceptual Models of Transfer

Physically move a value

Move an access path to a value

Pass-by-Value (In Mode)

The value of the actual parameter is used to initialize the corresponding formal parameter

- Normally implemented by copying
- Can be implemented by transmitting an access path but not recommended (enforcing write protection is not easy)

In Mode

Pass-by-Value (In Mode)

- Example:

Python

```python
def addition(augend, addend):
    print(augend + addend)
```

```python
x = 5
y = 10
addition(x, y)
```

C

```c
void addition(int augend, int addend){
    printf("%d", augend + addend);
}
```

```c
int x = 5;
int y = 10;
addition(x, y);
```

Pass-by-Value (In Mode)

- *Disadvantages*:
  - If by physical move: additional storage is required (stored twice) and the actual move can be costly (for large parameters)

```
def addition(augend, addend):
    print(augend + addend)
```

```
x = 5
y = 10
addition(x, y)
```

Pass-by-Value (In Mode)

- *Disadvantages*:
  - If by access path method: must write-protect in the called subprogram and accesses cost more (indirect addressing)

```
void addition(int **augend, int addend){
    printf("%d", **augend + addend);
}
```

```
int x = 5;
int *ptr = &x;
addition(&ptr, 10);
```

Pass-by-Result (Out Mode)

- When a parameter is passed by result, no value is transmitted to the subprogram; the corresponding formal parameter acts as a local variable; its value is transmitted to caller's actual parameter when control is returned to the caller, by physical move
  - Require extra storage location and copy operation

Out Mode

Pass-by-Result (Out Mode)

- Example:

Python

```
def lightSpeed():
    return 299792458
```

```
x = lightSpeed();
```

C

```
float gravity(){
    return 9.8;
}
```

```
float c = lightSpeed();
float m = 100.0;
float e = m * (c * c);
```

Pass-by-Result (Out Mode)

- Potential problems:
  - `sub(p1, p1);` whichever formal parameter is copied back will represent the current value of `p1`

C#

```
void Fixer(out int x, out int y) {
    x = 17;
    y = 35;
}
. . .
f.Fixer(out a, out a);
```

Pass-by-Result (Out Mode)

- Potential problems:
  - `sub(list[sub], sub);` Compute address of list[sub] at the beginning of the subprogram or end?

C#

```
void DoIt(out int x, int index) {
    x = 17;
    index = 42;
}
. . .
sub = 21;
f.DoIt(list[sub], sub);
```

Pass-by-Value-Result (Inout Mode)

- A combination of pass-by-value and pass-by-result
- Sometimes called pass-by-copy
- Formal parameters have local storage
- Disadvantages:
  - Those of pass-by-result
  - Those of pass-by-value

Inout Mode

Pass-by-Value-Result (Inout Mode)

- Example:

Python

```python
def addition(augend, addend):
    return augend + addend
```

```
x = 75;
y = 25;
sum = addition(x, y);
```

C

```c
int multiply(int x, int y){
    return x * y;
}
```

```
int x = 5;
int y = 7;
int product = multiply(x, y);
```

Pass-by-Reference (Inout Mode)

- Pass an access path
- Also called pass-by-sharing
- Advantage: Passing process is efficient (no copying and no duplicated storage)
- Disadvantages:
  - Slower accesses (compared to pass-by-value) to formal parameters
  - Potentials for unwanted side effects (collisions)
  - Unwanted aliases (access broadened)

```
fun(total, total);  fun(list[i], list[j]);  fun(list[i], i);
```

Pass-by-Reference (Inout Mode)

- Another issue:

  Can the passed reference be changed in the called subprogram?

  - In C, it is possible

  - But in some other languages, such as Pascal and C++, formal parameters that are addresses are implicitly dereferenced, which prevents such changes

Pass-by-Reference (Inout Mode)

- Example:

C

```c
void increment(int *x){
    *x += 1;
}


int x = 0;          x = 1
increment(&x);
```

Pass-by-Name (Inout Mode)

- Textual substitution
- Formals are bound to an access method at the time of the call, but actual binding to a value or address takes place at the time of a reference or assignment
- Allows flexibility in late binding
- Implementation requires that the referencing environment of the caller is passed with the parameter, so the actual parameter address can be calculated

Pass-by-Name (Inout Mode)

- In most languages parameter communication takes place thru the run-time stack
- Pass-by-reference are the simplest to implement; only an address is placed in the stack

Inout Mode

Pass-by-Name (Inout Mode)

- Example:

ALGOL 60

```
begin
    integer n;
    procedure p(k: integer);
        begin
            print(k);   10
            n := n+1;
            print(k);   11
        end;
    n := 10;
    p(n);
end;
```

Implementing Parameter-Passing Methods

Function header:

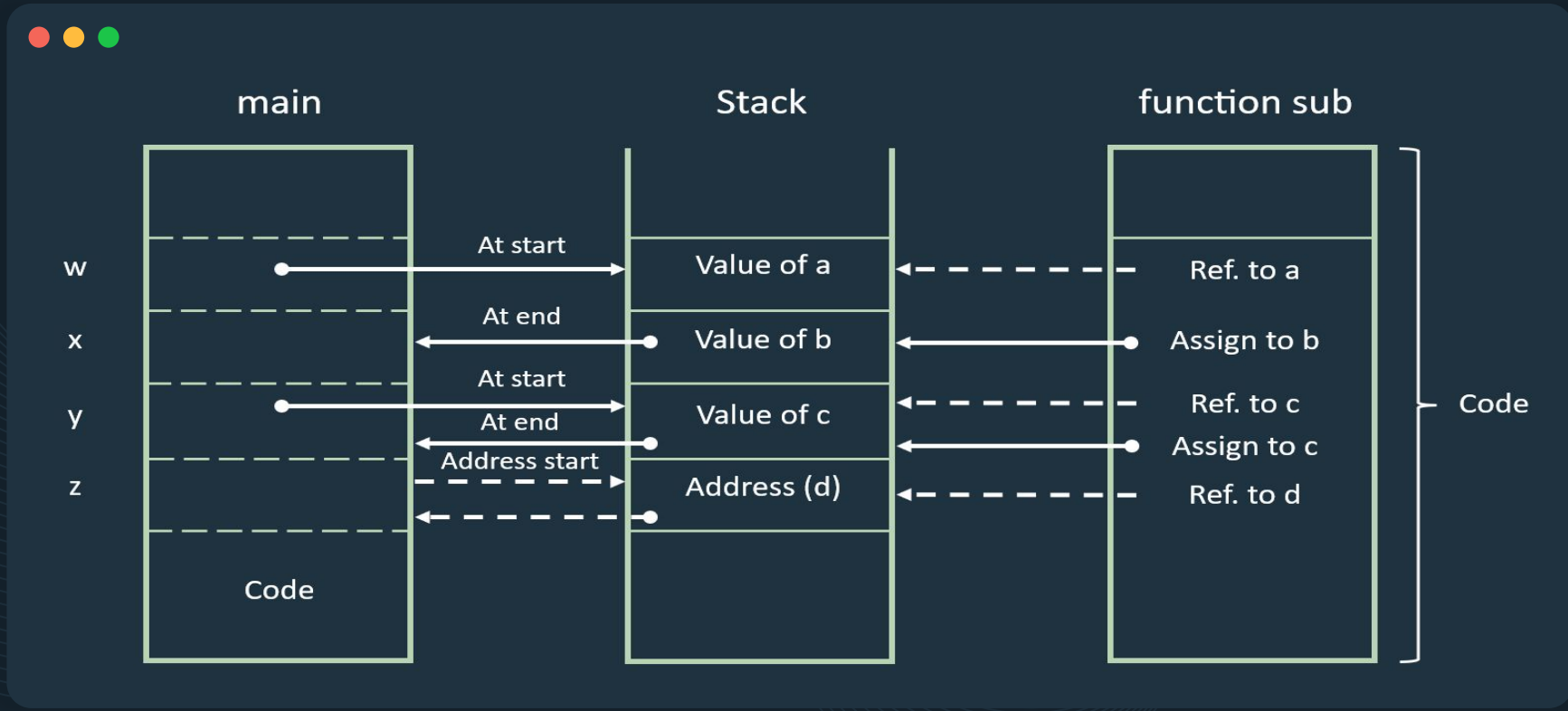
```
void sub(int a, int b, int c, int d);
```

Function call in main:

```
sub(w, x, y, z);
```

- Pass w by value
- Pass x by result
- Pass y by value-result
- Pass z by reference

Implementing Parameter-Passing Methods



Parameter Passing Methods of Major Languages

C

- Pass-by-value
- Pass-by-reference is achieved by using pointers as parameters

C++

A special pointer type called reference type for pass-by-reference

Java

- All non-object parameters are passed are passed by value
- Object parameters are passed by reference

Parameter Passing Methods of Major Languages

Fortran 95+

Parameters can be declared to be in, out, or inout mode

C#

Default method: pass-by-value

- Pass-by-reference is specified by preceding both a formal parameter and its actual parameter with `ref`

PHP

Very similar to C#, except that either the actual or the formal parameter can specify `ref`

Parameter Passing Methods of Major Languages

Swift

Default passing method is by value, but pass-by-reference can be specified by preceding the formal with `inout`

Perl

All actual parameters are implicitly placed in a predefined array named `@_`

Python and Ruby

Uses pass-by-assignment (all data values are objects); the actual is assigned to the formal

Type Checking Parameters

- Considered very important for reliability
- FORTRAN 77 and original C:
  - None
- Pascal and Java:
  - It is always required
- ANSI C and C++:
  - Choice is made by the user
  - Prototypes

Type Checking Parameters

- Relatively new languages Perl, JavaScript, and PHP:
  - Does not require type checking

- In Python and Ruby:
  - Variables do not have types (objects do)
  - Parameter type checking is not possible

Multidimensional Array as Parameters

- If a multidimensional array is passed to a subprogram and the subprogram is separately compiled, the compiler needs to know the declared size of that array to build the storage mapping function

Multidimensional Array as Parameters: C and C++

- Programmer is required to include the declared sizes of all but the first subscript in the actual parameter

```
void fun(int matrix[][10]){
    ...
}
```

```
int mat[5][10];
fun(mat);
```

Multidimensional Array as Parameters: C and C++

- Disallows writing flexible subprograms
- Solution: pass a pointer to the array and the sizes of the dimensions as other parameters; the user must include the storage mapping function in terms of the size parameters

```
void fun(int *mptr, int rows, int cols){
    ...
}
```

Multidimensional Array as Parameters: Java and C#

- Similar to Ada
- Arrays are objects; they are all single-dimensioned, but the elements can be arrays
- Each array inherits a named constant (`length` in Java, `Length` in C#) that is set to the length of the array when the array object is created

Multidimensional Array as Parameters: Java and C#

- Example:

Java

```
float sumer(float mat[][]) {
    float sum = 0.0f;
    for (int row = 0; row < mat.length; row++) {
        for (int col = 0; col < mat[row].length; col++) {
            sum += mat[row][col];
        }
    }
    return sum;
}
```

Design Considerations for Parameter Passing

- Two important considerations
  - Efficiency
  - One-way or two-way data transfer

- But the above considerations are in conflict
  - Good programming suggest limited access to variables, which means one-way whenever possible
  - But pass-by-reference is more efficient to pass structures of significant size

Parameters that are Subprogram Names

- It is sometimes convenient to pass subprogram names as parameters

- Issues:
  1. Are parameter types checked?
  2. What is the correct referencing environment for a subprogram that was sent as a parameter?

Parameters that are Subprogram Names: Referencing Environment

- Shallow binding: The environment of the call statement that enacts the passed subprogram
  - Most natural for dynamic-scoped languages
- Deep binding: The environment of the definition of the passed subprogram
  - Most natural for static-scoped languages
- Ad hoc binding: The environment of the call statement that passed the subprogram

Parameters that are Subprogram Names: Referencing Environment

JavaScript

```
function sub1() {
    var x;
    function sub2() {
        alert(x);
    };
    function sub3() {
        var x = 3;
        sub4(sub2);
    };
    function sub4(subx) {
        var x = 4;
        subx();
    };
    x = 1;
    sub3();
};
```

Shallow Binding:
The referencing environment of that execution is that of sub4, so the reference to x in sub2 is bound to the local x in sub4, and the output of the program is 4.

Parameters that are Subprogram Names: Referencing Environment

JavaScript

```
function sub1() {
    var x;
    function sub2() {
        alert(x);
    };
    function sub3() {
        var x = 3;
        sub4(sub2);
    };
    function sub4(subx) {
        var x = 4;
        subx();
    };
    x = 1;
    sub3();
};
```

Deep Binding:
The referencing environment of sub2's execution is that of sub1, so the reference to x in sub2 is bound to the local x in sub1, and the output is 1.

Parameters that are Subprogram Names: Referencing Environment

JavaScript

```
function sub1() {
    var x;
    function sub2() {
        alert(x);
    };
    function sub3() {
        var x = 3;
        sub4(sub2);
    };
    function sub4(subx) {
        var x = 4;
        subx();
    };
    x = 1;
    sub3();
};
```

Ad hoc Binding:
The binding is to the local x in sub3, and the output is 3.

Calling Subprograms Indirectly

- Usually when there are several possible subprograms to be called and the correct one on a particular run of the program is not known until execution (e.g., event handling and GUIs)

Calling Subprograms Indirectly

- In C and C++, such calls are made through function pointers

```
void fun(int x){
    print("%d", x);
}
```

```
void (*fptr)(int) = &fun;
(*fptr)(10);

void (*fptr)(int) = fun;
fptr(10);
```

Calling Subprograms Indirectly

- In C#, method pointers are implemented as objects called *delegates*
  - A delegate declaration:

```
public delegate int Change(int x);
```

    - This delegate type, named `Change`, can be instantiated with any method that takes an `int` parameter and returns an `int` value

Calling Subprograms Indirectly

- A method:

```
static int fun1(int x){ ... };
```

- Instantiate:

```
Change chgfun1 = new Change(fun1);
```

- Can be called with:

```
chgfun1(12); // fun1(12);
```

  - A delegate can store more than one address, which is called a *multicast delegate*

Design Issues for Functions

- Are side effects allowed?
  - Parameters should always be in-mode to reduce side effect (like Ada)
- What types of return values are allowed?
  - Most imperative languages restrict the return types
  - C allows any type except arrays and functions
  - C++ is like C but also allows user-defined types
  - Java and C# methods can return any type (but because methods are not types, they cannot be returned)
  - Python and Ruby treat methods as first-class objects, so they can be returned, as well as any other class

Overload Subprograms

- An *overloaded subprogram* is one that has the same name as another subprogram in the same referencing environment
  - Every version of an overloaded subprogram has a unique protocol
- C++, Java, C#, and Ada include predefined overloaded subprograms

Overload Subprograms

- In Ada, the return type of an overloaded function can be used to disambiguate calls (thus two overloaded functions can have the same parameters)

- Ada, Java, C++, and C# allow users to write multiple versions of subprograms with the same name

Generic Subprograms

- A *generic* or *polymorphic subprogram* takes parameters of different types on different activations
- Overloaded subprograms provide *ad hoc polymorphism*
- *Subtype polymorphism* means that a variable of type T can access any object of type T or any type derived from T (OOP languages)

Generic Subprograms

- A subprogram that takes a generic parameter that is used in a type expression that describes the type of the parameters of the subprogram provides *parametric polymorphism*
  - A cheap compile-time substitute for dynamic binding

Generic Subprograms

- C++
  - Versions of a generic subprogram are created implicitly when the subprogram is named in a call or when its address is taken with the & operator
  - Generic subprograms are preceded by a **template** clause that lists the generic variables, which can be type names or class names

```
template <class Type>
    Type max(Type first, Type second) {
        return first > second ? first : second;
    }
```

Generic Subprograms

- Java 5.0
  - Differences between generics in Java 5.0 and those of C++:
    1. Generic parameters in Java 5.0 must be classes
    2. Java 5.0 generic methods are instantiated just once as truly generic methods
    3. Restrictions can be specified on the range of classes that can be passed to the generic method as generic parameters
    4. Wildcard types of generic parameters

Generic Subprograms

- Java 5.0

```
public static <T> T doIt(T[] list) { ... }
```

  - The parameter is an array of generic elements (T is the name of the type)

  - A call:

```
doIt<String>(myList);
```

Generic Subprograms

- Java 5.0

  Generic parameters can have bounds:

```
public static <T extends Comparable> T doIt(T[] list) { … }
```

  The generic type must be of a class that implements the Comparable interface

Generic Subprograms

- Java 5.0
  - Wildcard types

    `Collection<?>` is a wildcard type for collection classes

```
void printCollection(Collection<?> c) {
   for (Object e: c) {
     System.out.println(e);
   }
 }
```

  - Works for any collection class

Generic Subprograms

- C# 2005
  - Supports generic methods that are similar to those of Java 5.0
  - One difference: actual type parameters in a call can be omitted if the compiler can infer the unspecified type
  - Another – C# 2005 does not support wildcards

Generic Subprograms

- F#
  - Infers a generic type if it cannot determine the type of a parameter or the return type of a function – *automatic generalization*
  - Such types are denoted with an apostrophe and a single letter, e.g., `'a`
  - Functions can be defined to have generic parameters

Generic Subprograms

- F#
  - Functions can be defined to have generic parameters

```
let printPair (x: 'a) (y: 'a) =
    printfn "%A %A" x y
```

  - `%A` is a format code for any type
  - These parameters are not type constrained

Generic Subprograms

- F#
  - If the parameters of a function are used with arithmetic operators, they are type constrained, even if the parameters are specified to be generic
  - Because of type inferencing and the lack of type coercions, F# generic functions are far less useful than those of C++, Java 5.0+, and C# 2005+

User-Defined Overload Operators

- Operators can be overloaded in Ada, C++, Python, and Ruby

- A Python example

```
def __add__ (self, second) :
  return Complex(self.real + second.real,
                 self.imag + second.imag)
```

Use: To compute `x + y`, `x.__add__(y)`

Closures

- A *closure* is a subprogram and the referencing environment where it was defined
  - The referencing environment is needed if the subprogram can be called from any arbitrary place in the program
  - A static-scoped language that does not permit nested subprograms doesn't need closures
  - Closures are only needed if a subprogram can access variables in nesting scopes and it can be called from anywhere
  - To support closures, an implementation may need to provide unlimited extent to some variables (because a subprogram may access a nonlocal variable that is normally no longer alive)

Closures

- C#
  - We can write a closure in C# using a nested anonymous delegate
  - `Func<int, int>` (the return type) specifies a delegate that takes an `int` as a parameter and returns and `int`

```
static Func<int, int> makeAdder(int x) {
    return delegate(int y) {return x + y;};
}
...
Func<int, int> add5 = makeAdder(5);
Console.WriteLine("Add 5 to 20: {0}", add5(20));
```

Closures

- JavaScript closure:

```
function makeAdder(x) {
  return function(y) {return x + y;}
}
...
var add10 = makeAdder(10);
var add5 = makeAdder(5);
document.write("add 10 to 20: " + add10(20) + "<br />");
document.write("add 5 to 20: " + add5(20) + "<br />");
```

  - The closure is the anonymous function returned by `makeAdder`

Closures

```
Output:
add 10 to 20: 30
add 5 to 20: 25
```

Closures

```
function outerFunction(outerVar){
    return function innerFunction(innerVar){
        console.log('Outer Variable: ' + outerVar)
        console.log('Inner Variable: ' + innerVar)
    }
}

var newFunction = outerFunction("out")
newFunction('in')
```

Closures

Output:

```
Outer Variable: out
Inner Variable: in
```

Closures

- In Python, closures use the lambda function also known as an anonymous function.
- It can take any number of parameters but can only have one expression.
- The resulting value of the evaluated expression is then returned.

Closures

Syntax: lambda argument(s): expression

Example:

```
x = lambda n:n*n
print(x(3))
```

Output: 9

Closures

```
def makeAdder(x):
    return lambda y: x+y

add10 = makeAdder(10)
print('Add 10 to 20: ' + str(add10(20)))

Output: Add 10 to 20: 30
```

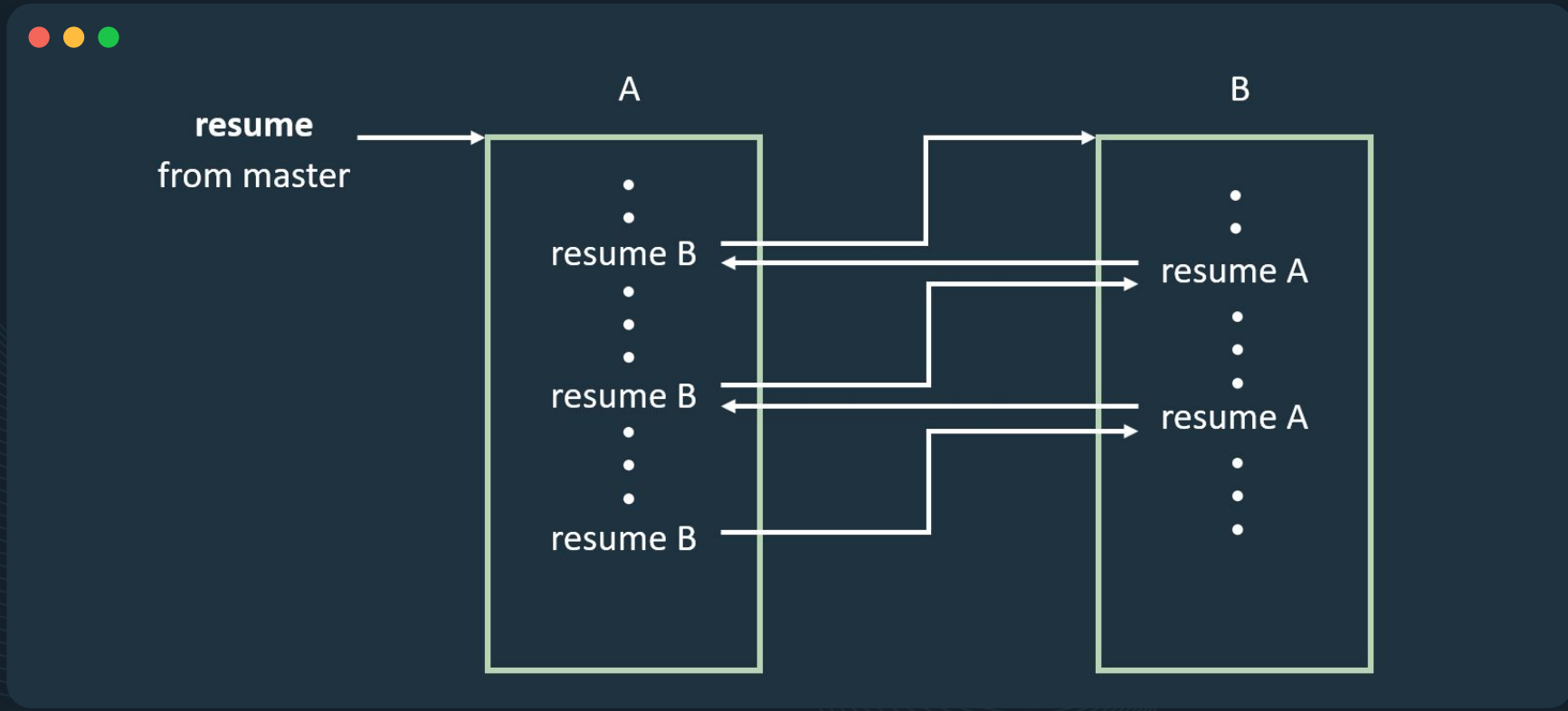
Coroutines

- A *coroutine* is a subprogram that has multiple entries and controls them itself – supported directly in Lua
- Also called *symmetric control:* caller and called coroutines are on a more equal basis
- A coroutine call is named a *resume*

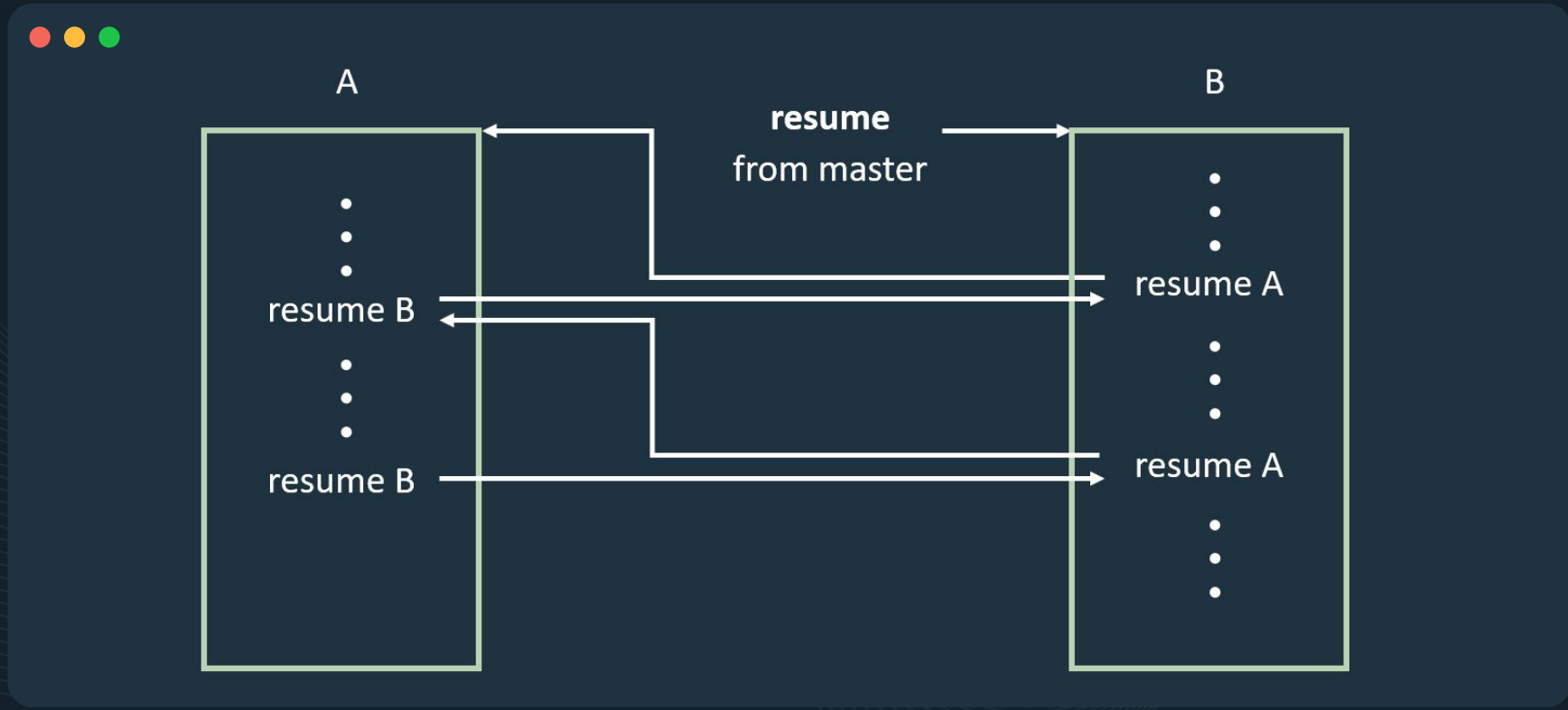
Coroutines

- The first resume of a coroutine is to its beginning, but subsequent calls enter at the point just after the last executed statement in the coroutine
- Coroutines repeatedly resume each other, possibly forever
- Coroutines provide *quasi-concurrent execution* of program units (the coroutines); their execution is interleaved, but not overlapped

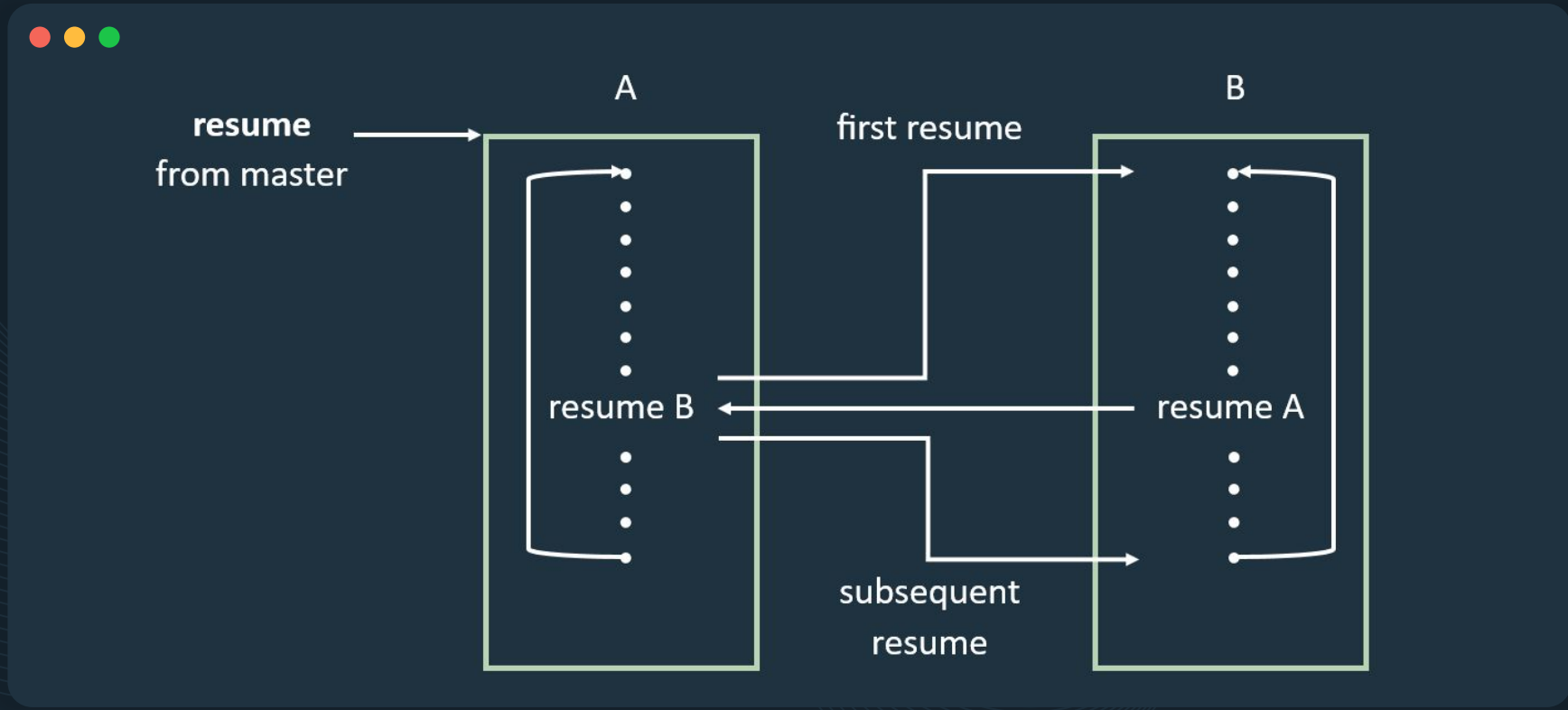
Coroutines Illustrated: Possible Execution Controls



Coroutines Illustrated: Possible Execution Controls



Coroutines Illustrated: Possible Execution Controls



Summary

A subprogram definition describes the actions represented by the subprogram

Subprograms can be either functions or procedures

Local variables in subprograms can be stack-dynamic or static

Three models of parameter passing: in mode, out mode, and inout mode

Some languages allow operator overloading

Summary

Subprograms can be generic

A closure is a subprogram and its referencing environment

A coroutine is a special subprogram with multiple entries

THANK YOU!

Do you have any questions?