----------------- CHAPTER 9 -------------------

SUBPROGRAMS

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

Characteristics of Subprograms

- single entry point
- calling program is suspended, only one executing program at a time
- resume upon subprogram termination

Categories of Subprograms

Functions	**Procedures**
Semantically modeled on math functions. Structurally resemble procedures	Collection of statements that define parameterized computations
Has return value **Has** side effects	**No** return value **No** side effects

Terminologies

Subprogram Definition – interface description and subprogram abstraction actions

e.g. ***int addition(int val1, int val2) {return val1 + val2;}***

Subprogram Call – explicit request of subprogram execution by a caller

e.g. int sum = ***addition(5, 10)***;

Subprogram Header – first part of a definition, includes name, type, formal parameters.

e.g. ***int addition (int val1, int val2)*** {}

Parameter Profile (signature) – the number, order, and types of the parameters.

e.g. int addition (***int val1, int val2***) {}

Protocol – parameter profile and return type

e.g. ***int*** addition ***(int val1, int val2)*** {}

Subprogram Declaration – function prototype

e.g. ***int addition (int val1, int val2);***

Formal Parameter – dummy variable in subprogram header

e.g. int addition(***int val1, int val2***)

Actual Parameter – value/address used in subprogram call statement

e.g. int sum = addition***(5, 10)***;

Actual/Formal Parameter Correspondence

Positional – first formal, first actual
safe and effective

Keyword – name specific. Appear in any order
Must know formal parameter names

***Python**, the actual is a list of values while formal is a name with asterisk

LOCAL REFERENCING ENVIRONMENTS

Stack Dynamic Local Variables

- ➔ *Support for recursion*
- ➔ *Shared local storage*
- ➔ Indirect addressing, cannot history sens
- ➔ alloc/dealloc, initialization time

Static Local Variables

- ➔ *Direct addressing*
- ➔ *Fast alloc/dealloc*
- ➔ No recursion
- ➔ No sharing of local storage

- **C** stack-dynamic locals, can be *static*
- **Java**, **Python**, stack dynamic locals only

Parameter Passing

In Mode - *Pass by Value*

- Value of actual is used to initialize formal
- Implemented by **copying**. Stored twice.

e.g. void addition(***int val1, int val2***) {}

Out Mode - *Pass by Result*

- No value given. Returns a value
- Requires extra storage and copy operation

e.g int addition() { ***return sum;*** }

Inout Mode - *Pass by Value-Result*

- Formal parameters have local storage
- Passes value. Returns a value

e.g. ***int*** addition (***int val1, int val2***) { ***return sum****;* }

Inout Mode - *Reference*

- Pass an access path (pass by sharing)
- Efficient pass process
- Slower access, unwanted side effects & alias

e.g. void increment (***int *x***) {}

Inout Mode - *Name*

- Textual substitution
- Flexibility in late binding
- Usually used in assembly language

*C, ***pass by value*** & ***reference*** using pointers
*Java, ***value*** non-objects. ***reference*** objects
*Python, ***pass by assignment***

TYPE CHECKING PARAMETERS
- Significant in reliability
- **C**, user-defined prototypes
- **Python**, no variable type. Parameter type not possible
- **Java**, always required

Multidimensional Array as Parameters
- Size of array is required, programmer job
- Declare size except first in actual parameter

e.g void addition(***int val[ ][10]***){}
- **C**, no flexible subprogram - pass pointer & dimensions as parameter

e.g. void addition(***int *ptr, int row, int col***) {}
- **Java** arrays are objects. 1D but elements can be arrays. Each array inherits *length*.

Considerations for Parameter Passing
➔ Limited access to variables, one-way whenever possible

➔ Pass by reference is more efficient on structures

Shallow binding - environment of call statement that enacts the passed subprogram
* natural for dynamic-scoped language (Python)
Deep binding - environment of the definition of the passed subprogram
* natural for static-scoped language ©
Ad hoc binding - environment of the call statement that passed the subprogram

* **C** allows types except arrays, functions
* **Java** can return any type, except methods
* **Python** can return any type, including methods

OVERLOADED SUBPROGRAMS
- Same subprogram name in same referencing environment. Versions have unique protocols

GENERIC SUBPROGRAMS (Polymorphic)
- Takes different parameters of different types on different activations
- Overloaded subprograms provide **ad hoc polymorphism**
- **Subtype polymorphism** - a variable of a type can access any object of that type or any derivative

USER-DEFINED OVERLOAD OPERATORS
- Can be done in Python

CLOSURES
- Subprogram + referencing environment where it was defined
- Use ***lambda*** in Python (anonymous)

e.g. ***x = lambda n : n * n***

COROUTINES
- Subprogram with multiple entries
- ***Symmetric control*** - equal basis
- Call is named a ***resume***
- Provide ***quasi-concurrent execution***. Interleaved but not overlapped

----------------- CHAPTER 10 -------------------

GENERAL SEMANTICS OF CALLS & RETURNS
- **Subprogram Linkage**; call + return operations
- Rules that govern how subprograms are called & executed

Call Semantics
- Parameter-passing methods
- Stack-dynamic allocation of local variables
- Save execution status of calling program
- Transfer control & arrange return
- Access to nonlocal variables

Return Semantics
- Return parameter values
- Deallocate stack-dynamic locals
- Restore execution status of caller
- Return control to calling program

IMPLEMENTING SIMPLE SUBPROGRAMS
- Static local variables
- No nesting, no recursion

Call Actions
- Save execution status;	either
- Parameter passing;	caller
- Pass return address;	caller
- Transfer control;	caller

Return Actions
- Return values;	called
- Accessible value;	called
- Restore execution status;	either
- Transfer control;	called

Required Storage
- Caller Status Information
- Parameters
- Return Address
- Return Values (functions)
- Temporaries (variables)

* *Linkage actions* can occur in the beginning or end. **Prologue** or **Epilogue**. Only epilogue in simple subprograms.

SUBPROGRAM PARTS
- Actual code (constant)
- Non-code (locals + changeable data)
- Both are fixed, statically allocated

ACTIVATION RECORD - layout of non-code part
ACTIVATION RECORD INSTANCE
- Collection of data
- One active version at a time
- One instance at a time

CONSTRUCTION OF COMPLETE SUBPROGRAM
- Not done entirely by compiler
- Machine code + reference list made during compilation
- **Linker** (loader) compiles executable program

STACK-DYNAMIC SUBPROGRAMS
- Support recursion
- Complex subprogram linkage

Complex Activation Record
- Implicit alloc/dealloc of locals
- Simultaneous instance of subprograms
- **A.R.** known at compile time (also size)
- **A.R.I.** is dynamically created

Local Variables - bound to storage within ARI
Parameters - value/address provided by caller
Dynamic Link - access nonlocal variables
Return Address - pointer to instruction after call

ACTIVATION RECORD
- **Stack** (run-time stack) last in, first out
- Another A.R. = another A.R.I.
- **Environment Pointer** to base of ARI of executing program. Control execution.

Caller Actions
- Create ARI
- Save execution status
- Computer and pass parameters
- Pass return address to the called
- Transfer control to the called

Prologue Actions (Called)
- Save old EP in stack as dynamic link
- Create new value for EP
- Allocate local variables

Epilogue Actions (Called)
- Formal to actual parameters
- Value move for ease-of-access to caller
- EP-1 and set EP to old dynamic link
- Restore execution status of caller
- Transfer control back to caller

*Subprogram is active until execution ends
*inactive = no more local scope
= no more referencing environment
= no more ARI

DYNAMIC CHAIN - collection of dynamic links in the stack at a given time (call chain).

LOCAL OFFSET - access to local variables using local offset. Determined at compile time.

NESTED SUBPROGRAMS
- Stack-dynamic local variables
- Allows for nesting
- Python
- All variables can be **non-locally** accessed

Process of Locating Non-Local Reference
- Find correction ARI
- Determine correct offset within ARI
- Only **nonlocal variables** are visible and accessible in **static ancestor scopes**
- Guaranteed by **static semantic rules**

STATIC SCOPING
Local Variables -
Parameters -
Static Link *(Static Scope Pointer)* - points to bottom of ARI, appears below parameters
Dynamic Link -
Return Address -

- **Static Chain**. Lineage of activation record instances. Connects all static ancestors.
- **Static Depth**. Integer indicating nesting depth from outermost scope.
- **Chain Offset**/**Nesting Depth**. SDR - SDD

(chain offset, local offset) - reference to variable

STATIC CHAIN MAINTENANCE
- Modified for each call and return

At Return
- Trivial
- Ends → ARI is removed from stack
- New top ARI → caller of above subprogram

At Call
- ARI of static parent must be found
- Static link points to most recent ARI of parent
- Dynamic link points to preceding ARI

Two Methods
- Search Dynamic Chain
- Treat calls like variable declaration

Treat Calls like Variable Declaration
- Compiler determines subprogram caller
- Nesting depth - No. of Enclosing Scopes

Evaluation of Static Chain
- Nonlocal reference is ***slow***
- ***Time-critical*** code is difficult

BLOCKS

- User-specified local scopes for variables
- Lifetime of locals begin when control enters block

2 Methods

- Parameter-less blocks that are always called from the same location. Each block has AR and ARI during execution.
- Max storage is statically determined, can be allocated after locals in AR.

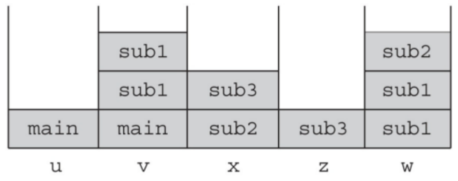
IMPLEMENTING DYNAMIC SCOPING

Deep Access

- Nonlocal reference is found by searching ARI
- Length of chain can't be statically determined
- ARI must have variable names

Shallow Access

- Put locals in central place
- One stack for each variable name



(The names in the stack cells indicate the program units of the variable declaration.)

---------------- CHAPTER 12 -----------------

Terms	**Definition**
ADTs	classes
Reference variables	holds a reference to information related to that variable
Class instances	objects
Derived class / subclass / child class	class that inherits
Parent class / superclass / base class	class that was inherited from
Members	together refers to variables and methods
Methods	subprograms that define operations on objects
Messages	- calls to methods - 2 parts: 1. method name 2. destination object - subprograms require data passed as parameter - data worked on by messages are usually within the object itself
Message protocol / message interface	entire collection of methods of an object

MAJOR LANGUAGE FEATURES

1. Abstract data types
2. **Inheritance** - central theme in OOP and languages that support it
3. **Polymorphism** / dynamic binding of method calls to methods

INHERITANCE

- *Problems with ADTs*
  1. difficult to reuse due to always needing changes
  2. independent and at same level
- *Solves ADTs problems*
  1. allows new classes to inherit common entities from existing ones, modify them, and add new ones
  2. allows reuse without requiring any changes to the reused ADT
  3. defines classes in a hierarchy
- *Complicated due to access controls to encapsulated entities where a class can hide entities*
  1. from its subclasses or clients
  2. for its clients while allowing its subclasses to see them
- **Method overriding** - *subclass can modify an inherited method*
  1. overridden method - method overridden
- *Ways a subclass can differ from its parent*
  1. Can add variables and/or methods
  2. Method overriding
  3. Methods or variables with private access are not visible in subclass
- **Disadvantage** > Creates tight coupling among class complicating maintenance

Kinds of Variables in a Class

1. *Class variables* - belong to class
2. I*nstance variables* - belong to an object

Kinds of Methods in a Class

1. *Class methods* - called by prefixing the method with either class name or a variable that references one of their instances
2. *Instance methods* - accept messages to objects

Levels of Accessibility

1. *Private* - not visible to subclasses
2. *Protected* - visible to subclasses, but not clients
3. *Public* - visible to clients

Dynamic Binding

- *<u>Polymorphic variable</u>* - can be defined in a class that is able to reference objects and any of its descendants
    - binding to correct method will be dynamic when overridden methods are called through a polymorphic variable
    - makes a statically typed language a little bit of dynamically typed
    - **Purpose** > allows software systems to be more easily extended during both development and maintenance
- Concepts
    - abstract method - one with no definition
    - abstract class - one that includes at least 1 virtual method
        - cannot be instantiated
- Sometimes called **dynamic dispatch**

DESIGN ISSUES FOR OOP LANGUAGES

1. Exclusivity of Objects
    a. *<u>Everything is an Object</u>*
        i. Elegant & uniform language
        ii. Slower message passing for simple operations
    b. *<u>Objects to Complete Typing System</u>*
        i. Fast operations on simple objects
        ii. Confusing type system
    c. *<u>Primitives=Imperative, Structured=Objects</u>*
        i. Fast operations on primitives
        ii. Confusion of type system
2. Subclass Subtype
    a. **Principle of Substitution** - no error
    b. **Subclass** - inherit to promote code reuse
    c. **Subtype** - inherit interface & behavior
3. Single / Multiple Inheritance
    a. **Shared Inheritance** - have same parent
    b. Sometimes convenient and valuable
    c. *Complex Language Implementation*
    d. *Potential Inefficiency (cost in binding)*
    e. *Complex Maintenance*
    f. **Interface** = good alternative
4. Object Alloc/Dealloc
    a. *<u>ADT behavior = Anywhere allocation</u>*
        i. Allocated from run-time stack
        ii. Explicitly create on heap (new)
    b. *<u>All heap-dynamic = Pointer access</u>*
        i. Simplified assignment
        ii. Implicit dereferencing
    c. *<u>Stack-dynamic = subtype problem</u>*
        i. **Object Slicing** - trunking of excess
5. Dynamic / Static Binding
    a. *<u>Message to Method Binding = Dynamic</u>*
        i. None = no advantage
        ii. All = efficient
    b. User Specified Binding
    c. Static Binding is faster
6. Nested Classes
    a. Information Hiding = main purpose
    b. Visibility of members in nesting/nested
7. Initialization of Objects
    a. Explicit or Implicit
    b. Subclass object initialization?

SUPPORT FOR OOP IN JAVA

- General Characteristics
    - Object data except primitives
    - Primitive has wrapper class - **Boxing**
    - Heap-dynamic objects (new)
    - **Finalize** - implicit reclaim of storage
- Inheritance
    - Single inheritance = ***extends***
    - Multiple Inheritance = ***implements***
    - Methods & Classes can be ***final***
    - All subclass are subtype
- Dynamic Binding
    - Dynamic message to methods
    - Except **final**, **static**, **private**
- Nested Classes
    - All hidden except nesting class
    - **Inner class** - non static nested class
    - Can be anonymous
    - **Local nested class** - defined in a method of its nesting class
- Evaluation
    - No support for procedural prog
    - No parentless classes
    - Dynamic Binding is normal
    - Interface = Multiple inheritance

IMPLEMENTING OO CONSTRUCTS

1. Storage Structure for Instance Variables
    a. **Class Instance Records** - store states of objects. Static (compile time)
    b. Class has parent, subclass instance variables are added to parent CIR
    c. Efficient due to static nature
2. Dynamic Binding of Messages to Methods
    a. Dynamic Binding = need entry in CIR
    b. **Virtual Method Table** (vtable) name of storage structure for list of DBM
    c. Method calls represented as **offsets** from beginning of vtable