CS49K Programming Languages

Chapter 6: Control Flow

LECTURE 8: PROGRAM STRUCTURE

DR. ERIC CHOU

IEEE SENIOR MEMBER

Objectives

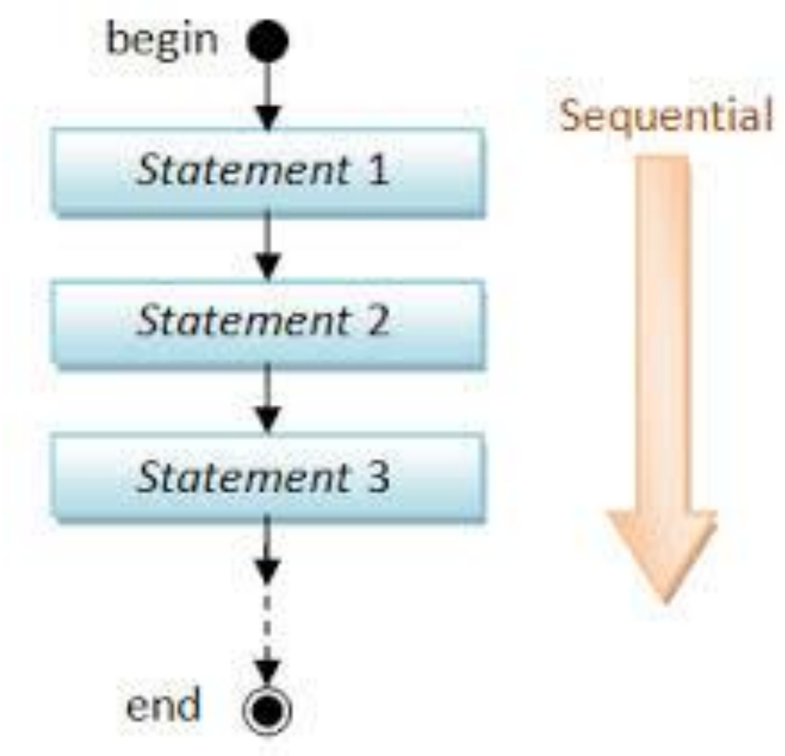
- Programming Paradigm
- Sequential Programming
- Expression Evaluation
- Control Structures (Selection, Iteration, Function, Exception, and Recursion)

Programming Paradigm

SECTION 1

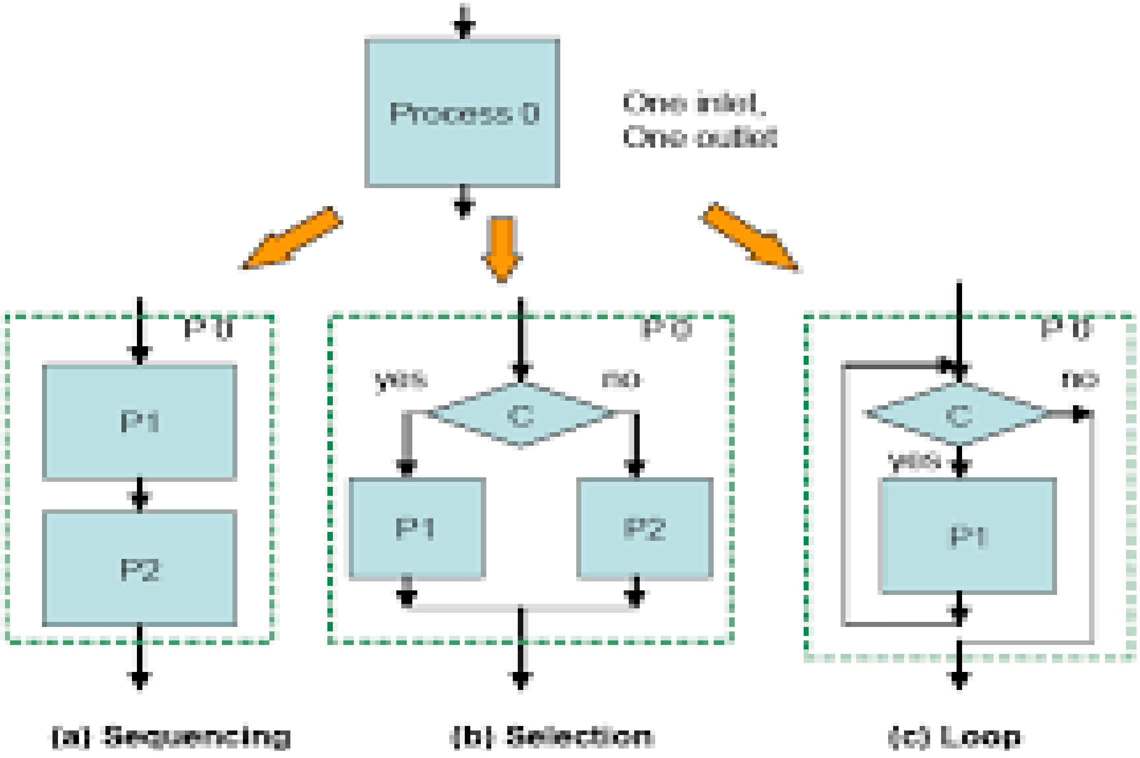
Single-Thread/Single Processor Programming

Sequential Programming



Single-Thread/Single Processor Programming

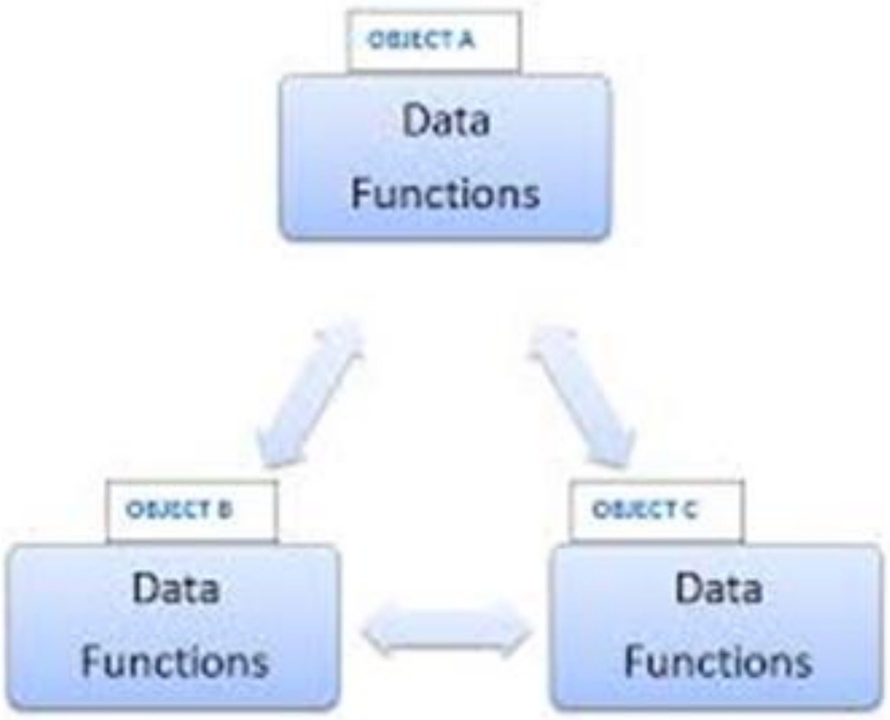
Structured Programming



Structured Programming

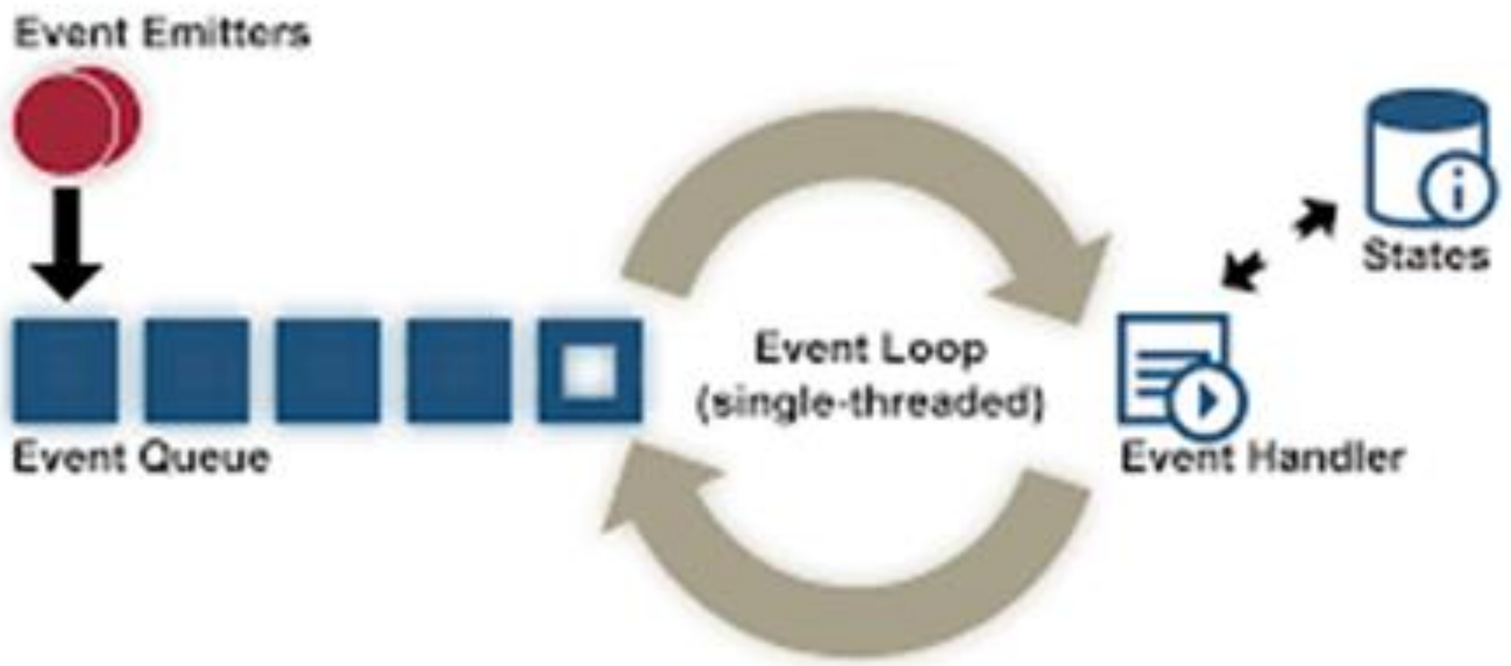
Concurrent Programming

Object-Oriented Programming



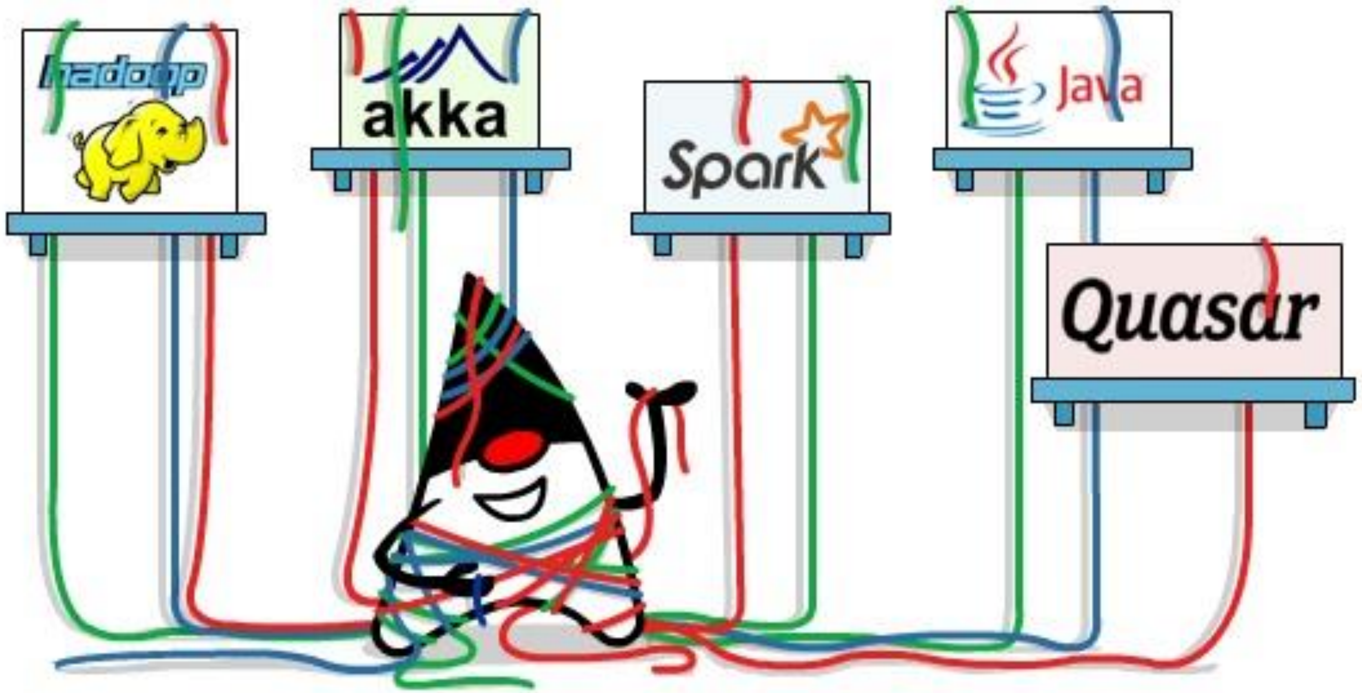
Concurrent Programming

Event-Loop Programming



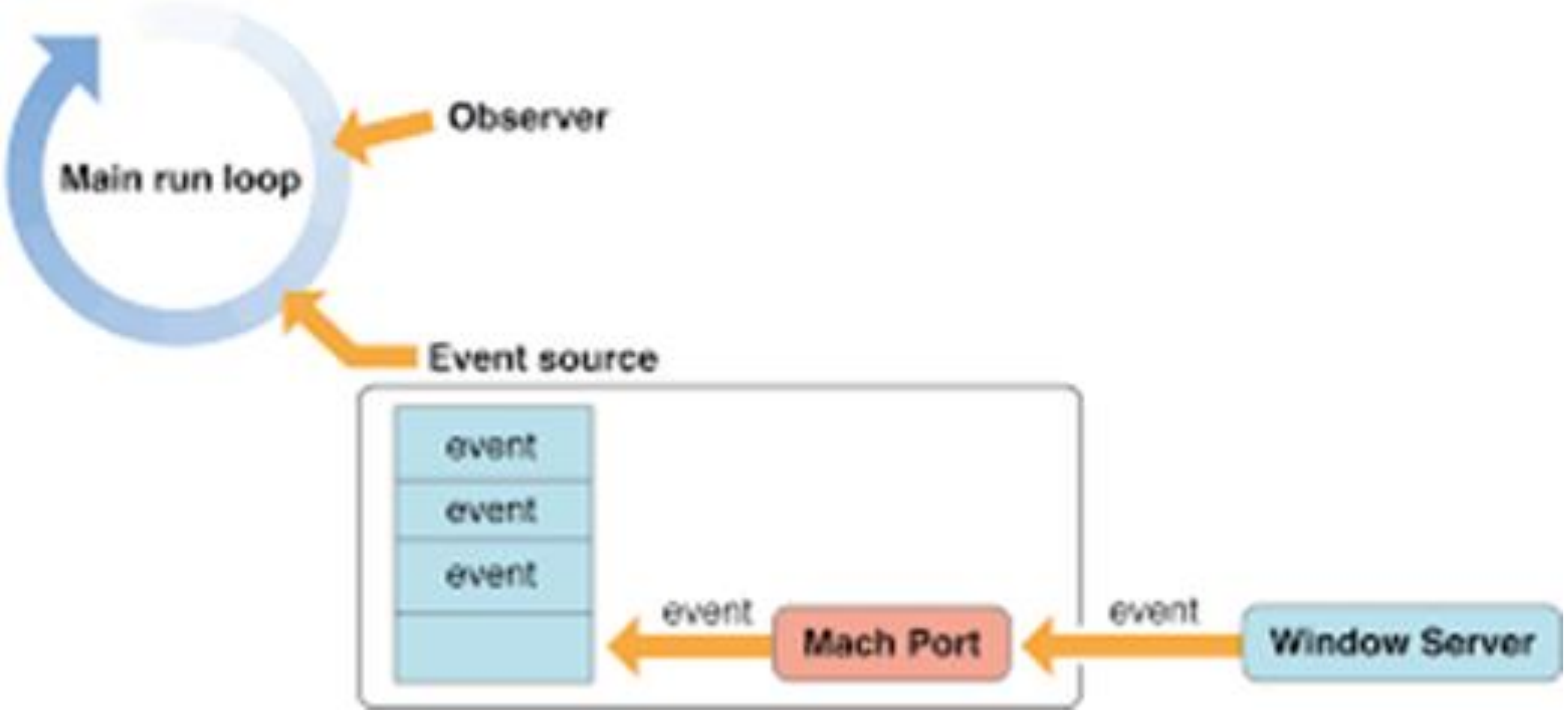
Concurrent Programming

Multithreading/Multiprocessing



Concurrent Programming

Event-Driven Programming (Event Queue)



Sequential Programming

SECTION 2

Control Flow

Basic paradigms for control flow

- **Goto's (Branch, Jump)**
- **Sequencing**
- Selection
- Iteration
- Procedural Abstraction
- Recursion
- Concurrency
- Exception Handling and Speculation
- Non-determinacy

Unstructured Control Flow

Assembly, COBOL, Fortran

- Sequencing
- Goto’s (Branch, and Jump)
- Code Section or Segment (COBOL)

Unstructured Control Flow

- Unstructured control flow: the use of goto statements and statement labels to implement control flow
  - Generally considered bad
  - Most can be replaced with structures with some exceptions
    - Break from a nested loop (e.g. with an exception condition)
    - Return from multiple routine calls
  - Java has no goto statement (supports labeled loops and breaks)
- Language Feature to support unstructured control flow: Sequencing, Branch on Condition, and Goto Labels.

Sequencing

The execution of statements and evaluation of expressions is usually in the order in which they appear in a program text.

- Sequencing
  - specifies a linear ordering on statements
    - one statement follows another
  - very imperative, Von-Neumann
- A compound statement is a delimited list of statements
  - A compound statement is called a block when it includes variable declarations
  - C, C++, and Java use **{** and **}** to delimit a block
  - Pascal and Modula use `begin ... end`
  - Ada uses `declare ... begin ... end`

Assembly Jump and C++ goto

Assembly jump	C++ goto

<pre>   mov eax,3
        jmp lemme_outta_here
        mov eax,999   ; <- not executed!
lemme_outta_here:
        ret</pre>
	<pre> int x=3;
	        goto quiddit;
	        x=999;
	quiddit:
	        return x;</pre>

Assembly Branch and Jumps

Here's how to use compare and jump-if-equal ("je"):

```
        mov eax,3
        cmp eax,3 ; how does eax compare with 3?
        je lemme_outta_here  ; if it's equal, then jump
        mov eax,999  ; <- not executed *if* we jump over it
lemme_outta_here:
        ret
```

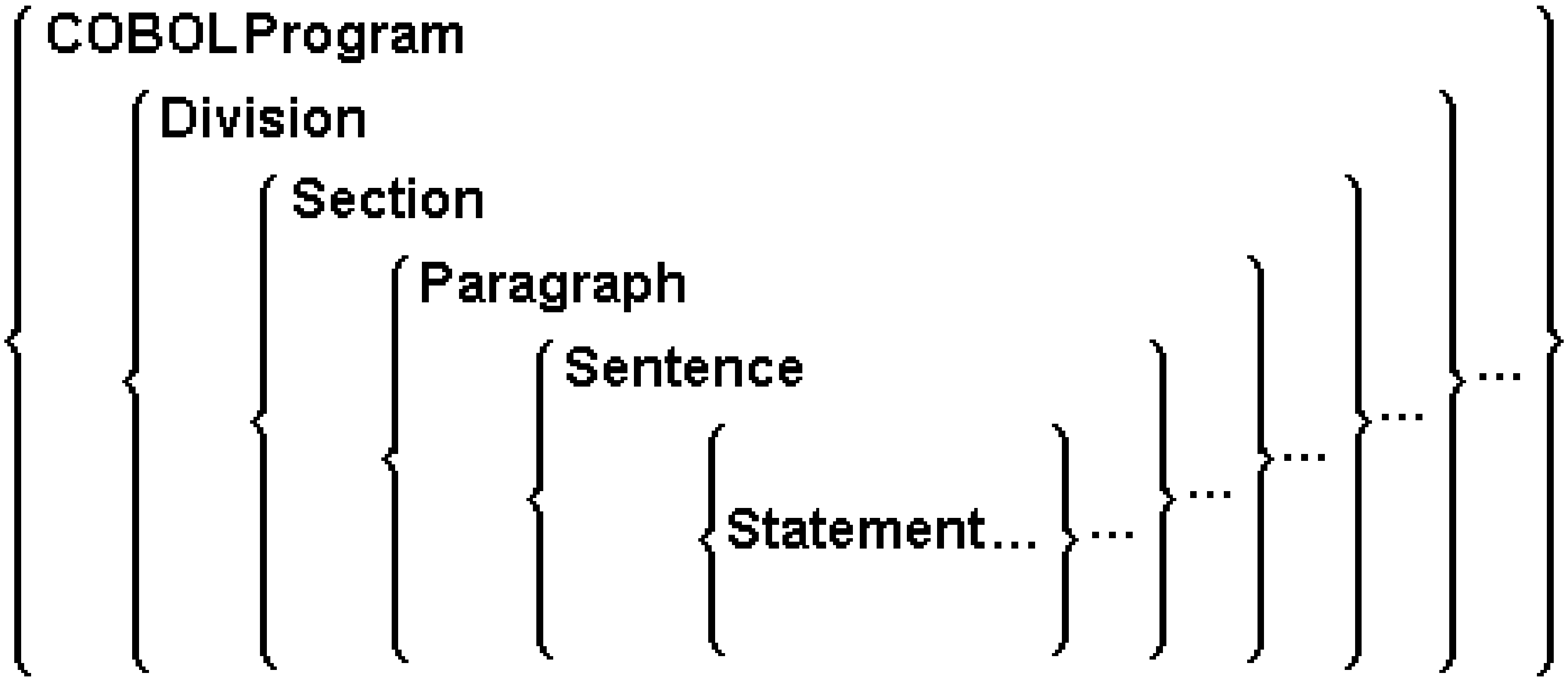
Here's compare and jump-if-less-than ("jl"):

```
        mov eax,1
        cmp eax,3 ; how does eax compare with 3?
        jl lemme_outta_here  ; if it's less, then jump
        mov eax,999  ; <- not executed *if* we jump over it
lemme_outta_here:
        ret
```

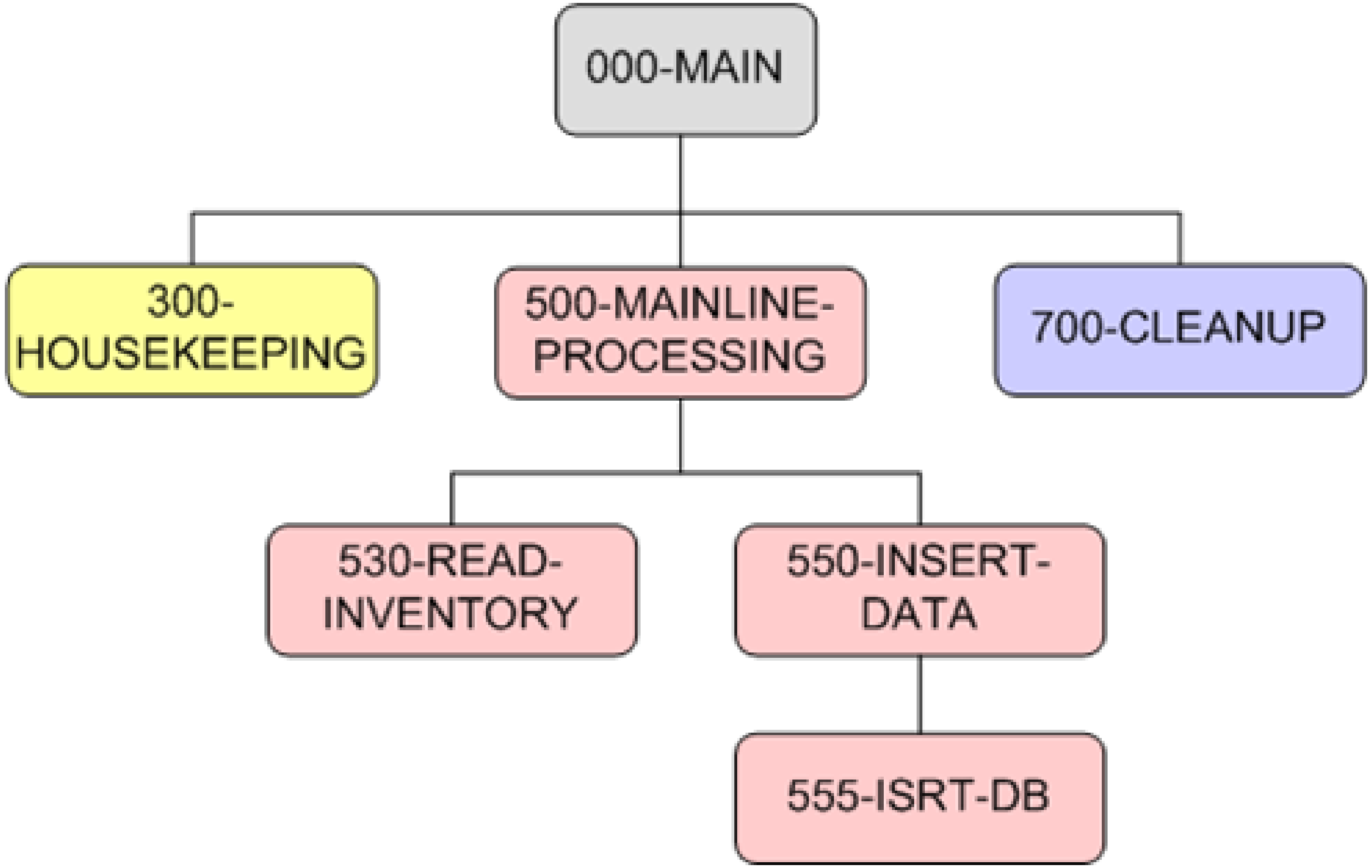
Instruction	**Useful to...**
jmp	Always jump
ja	Unsigned >
jae	Unsigned >=
jb	Unsigned <
jbe	Unsigned <=
jc	Unsigned overflow, or multiprecision add
jecxz	Compare ecx with 0 (Seriously!?)
je	Equality
jg	Signed >
jge	Signed >=
jl	Signed <
jle	Signed <=
jne	Inequality
jo	Signed overflow

Code Section

COBOL (Implemented by Jump and Branch)



Code Section
COBOL



Code Section

COBOL

```
EDIT         SYSADM.DEMO.SRCLIB(IMSPGM01) - 01.10          Columns 00001 00066
Command ===>                                                      Scroll ===> CSR
PROCEDURE DIVISION.
000-MAIN.
    ENTRY 'DLITCBL' USING INVENTORY-PCB-MASK

    MOVE '000-MAIN' TO WS-PARA-NAME
    PERFORM 800-DEBUG

    PERFORM 300-HOUSEKEEPING
    THRU    300-HOUSEKEEPING-EXIT

    PERFORM 500-MAINLINE-PROCESSING
    THRU    500-MAINLINE-PROCESSING-EXIT
    UNTIL   AT-END

    PERFORM 700-CLEANUP
    THRU    700-CLEANUP-EXIT

    PERFORM 999-GOBACK.
```

Expression Evaluation I

Continue/Exit Condition

SECTION 3

Control Flow

Basic paradigms for control flow

- Sequencing
- **Selection**
- **Iteration**
- **Procedural Abstraction**
- **Recursion**
- Concurrency
- Exception Handling and Speculation
- Non-determinacy

Expression Evaluation

Infix	Postfix	Prefix
((A * B) + (C / D))	((A B *) (C D /) +)	(+ (* A B) (/ C D))
((A * (B + C)) / D)	((A (B C +) *) D /)	(/ (* A (+ B C)) D)
(A * (B + (C / D)))	(A (B (C D /) +) *)	(* A (+ B (/ C D)))

- Infix, prefix operators
- Precedence, associativity (see Figure 6.1)
  - C has 15 levels - too many to remember
  - Pascal has 3 levels - too few for good semantics
  - Fortran has 8
  - Ada has 6
    - Ada puts *and & or* at same level
  - **Lesson**: when unsure, use parentheses!

Expression Evaluation

Figure 6.1 Operator precedence levels in Fortran, Pascal, C, and Ada. The operator s at the top of the figure group most tightly.

Fortran	Pascal	C	Ada
		++, -- (post-inc., dec.)	
**	not	++, -- (pre-inc., dec.), +, - (unary), &, * (address, contents of), !, ~ (logical, bit-wise not)	abs (absolute value), not, **
*, /	*, /, div, mod, and	* (binary), /, % (modulo division)	*, /, mod, rem
+, - (unary and binary)	+, - (unary and binary), or	+, - (binary)	+, - (unary)
		<<, >> (left and right bit shift)	+, - (binary), & (concatenation)
.eq., .ne., .lt., .le., .gt., .ge. (comparisons)	<, <=, >, >=, =, <>, IN	<, <=, >, >= (inequality tests)	=, /=, <, <=, >, >=
.not.		==, != (equality tests)	
		& (bit-wise and)	
		^ (bit-wise exclusive or)	
		| (bit-wise inclusive or)	
.and.		&& (logical and)	and, or, xor (logical operators)
.or.		|| (logical or)	
.eqv., .neqv. (logical comparisons)		?: (if...then...else)	
		=, +=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |= (assignment)	
		, (sequencing)	

Expression Evaluation

- Ordering of operand evaluation (generally none)
- Application of arithmetic identities
  - distinguish between commutativity, and (assumed to be safe)
  - associativity (known to be dangerous)

  `(a + b) + c works if a~=maxint and b~=minint and c<0`

  `a + (b + c) does not`

  - inviolability of parentheses

Expression Evaluation

Short-circuiting

- Consider `(a < b) && (b < c)`:
  - If **a >= b** there is no point evaluating whether **b < c** because **(a < b) && (b < c)** is automatically false
- Other similar situations

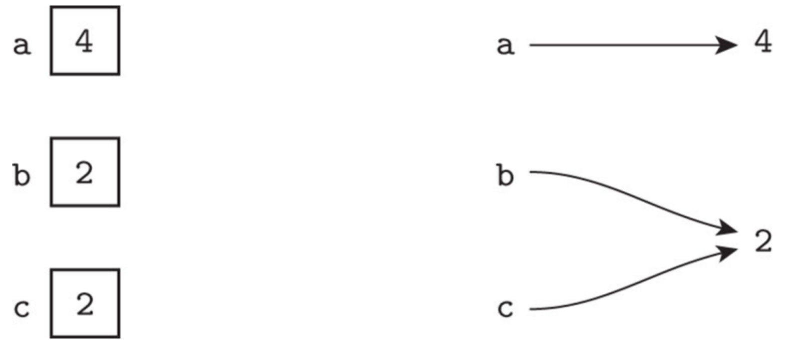
```
if (b != 0 && a/b == c) ...
if (*p && p->foo) ...
if (f || messy()) ...
```

- Can be avoided to allow for side effects in the condition functions

Expression Evaluation

Variables as values vs. variables as references (reference model)

- value-oriented languages
    - C, Pascal, Ada
- reference-oriented languages
    - most functional languages (Lisp, Scheme, ML)
    - Clu, Smalltalk
- Algol-68 kinda halfway in-between
- Java deliberately in-between
    - built-in types are values
    - user-defined types are objects - references

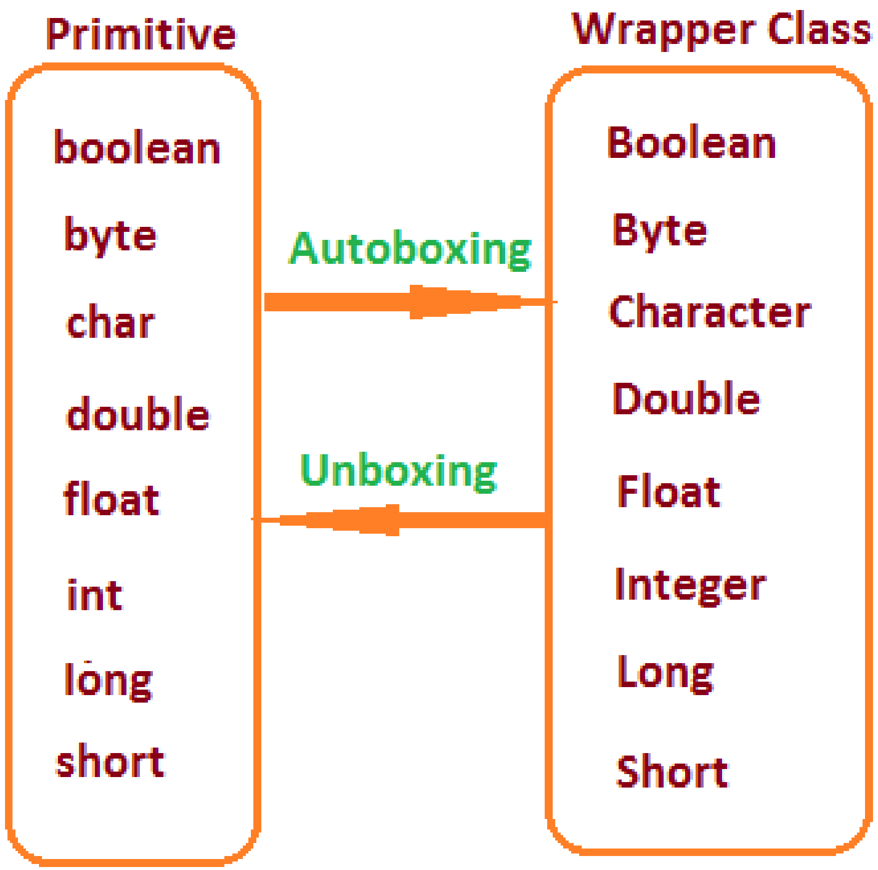


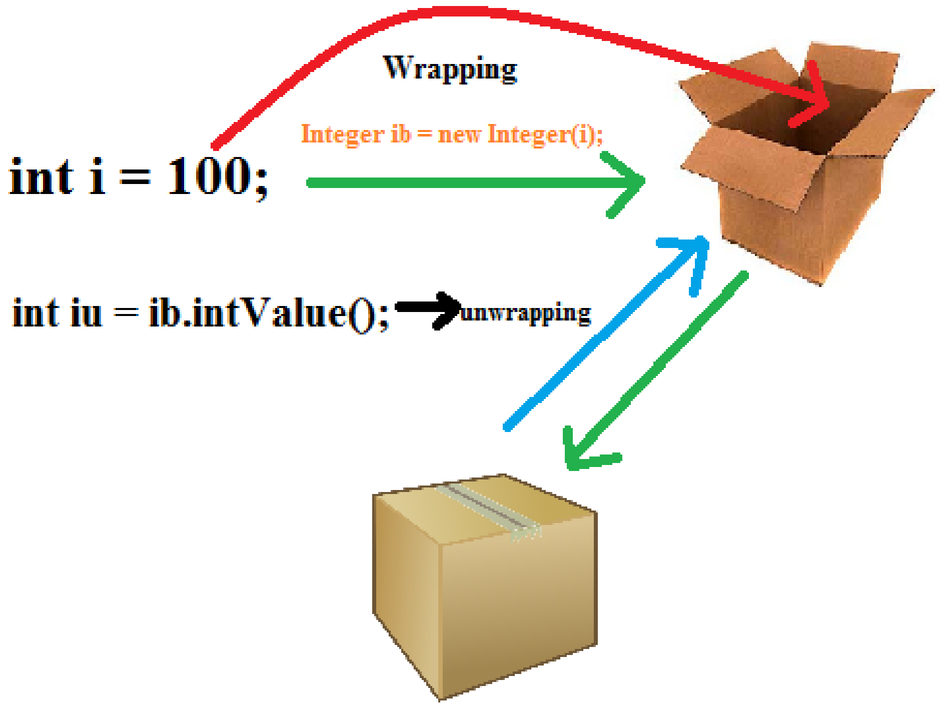
- value-oriented
- reference-oriented

Immutable Data

Expression Evaluation

Reference Model (Boxing/Unboxing)





Expression Evaluation

Expression-oriented vs. statement-oriented languages

- expression-oriented:
  - functional languages (Lisp, Scheme, ML)
  - Algol-68
- statement-oriented:
  - most imperative languages
- C is kind of halfway in-between (distinguishes)
  - allows expression to appear instead of statement

Expression Evaluation II

Orthogonality

SECTION 4

Expression Evaluation

Orthogonality

- Features that can be used in any combination
  - Meaning is consistent

```
if (if b != 0 then a/b == c else false) then ...
if (if f then true else messy()) then ...
```

Algol 68:

```
begin
  a:= if b<c then d else e;
  b:= begin f(b); g(c) end;
  g(d);
  2+3
end
```

C:

```
if (a == b) { /* do something if a == b */ }
if (a = b) { /* do if the assigned value is true (none-zero) */}
```

Expression Evaluation

Special Cases

Combination Assignment Operators:

```
a = a + 1;
Or,
b.c[3].d = b.c[3].d * e;
```

Equivalent

Assignment Operators:

```
a += 1;
Or,
b.c[3].d *= e;
```

Side Effect of Function as parameter:

```
void update(int a[], int index_fn(int n)){
  int i, j;
  /* calculate i */
  …
  j = index_fn(i);
 A[j] = A[j] + 1;
}
```

Here, we cannot safely write: (Orthogonality Violation)

```
A[index_fn(i)] = A[index_fn(i)] + 1;
```

```
A[index_fn(i)] += 1;  /* safe */
```

Equivalent

Post/Pre Increment/Decrement:

```
A[index_fn(i)]++;  /* safe */
++A[index_fn(i)];  /* safe */
```

Post increment or decrement are used for stack index operations

Multiway Assignment:

```
a, b = c, d;   or a, b, c = foo(d, e, f);
```

Expression Evaluation

Initialization

- Initialization
  - Pascal has no initialization facility (assign)
- Aggregates
  - Compile-time constant values of user-defined composite types (C, C++, Ada, Fortran)(Anonymous Objects)

Expression Evaluation

Assignment

- statement (or expression) executed for its side effect
- assignment operators (+=, -=, etc)
  - handy
  - avoid redundant work (or need for optimization)
  - perform side effects exactly once
- C --, ++
  - postfix form

Expression Evaluation

Side Effects

- often discussed in the context of functions
- a side effect is some permanent state change caused by execution of function
  - some noticeable effect of call other than return value
  - in a more general sense, **assignment** statements provide the ultimate example of side effects
    - they change the value of a variable

Expression Evaluation

Side Effects

- SIDE EFFECTS ARE FUNDAMENTAL TO THE WHOLE VON NEUMANN MODEL OF COMPUTING

- In (pure) functional, logic, and dataflow languages, there are no such changes
  - These languages are called SINGLE-ASSIGNMENT languages

Expression Evaluation

Side Effects

- Several languages outlaw side effects for functions
  - easier to prove things about programs
  - closer to mathematical intuition
  - easier to optimize
  - (often) easier to understand
- But side effects can be nice
  - consider **`rand()`**

```
x = 0;
def xSetter(n):
    global x
    x = n
xSetter(5)
xSetter(5)
```

Expression Evaluation

Side Effects

- Side effects are a particular problem if they affect state used in other parts of the expression in which a function call appears
  - It's nice not to specify an order, because it makes it easier to optimize
  - Fortran says it's OK to have side effects
    - they aren't allowed to change other parts of the expression containing the function call
    - Unfortunately, compilers can't check this completely, and most don't at all

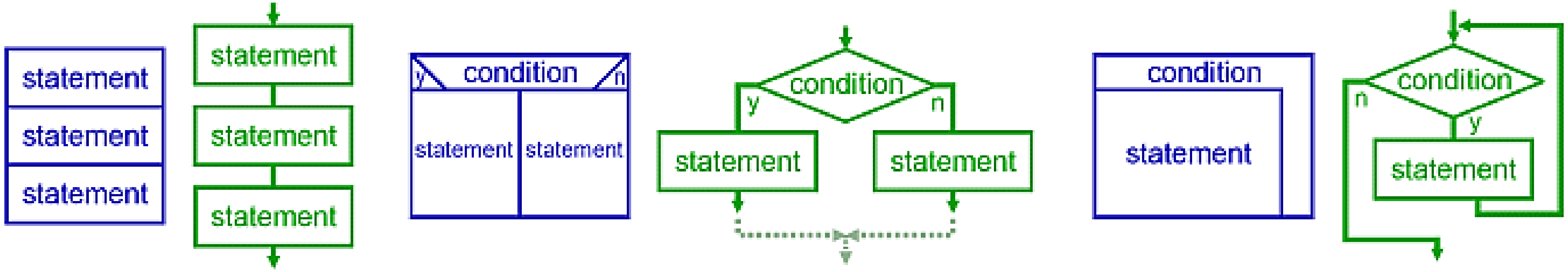
Control Structures I

Selection

SECTION 5

Structured Programming

Structured programming is a programming paradigm aimed at improving the clarity, quality, and development time of a computer program by making extensive use of **subroutines**, **block structures**, **for and while loops**—in contrast to using simple tests and jumps such as the go to statement which could lead to "spaghetti code" causing difficulty to both follow and maintain.



Structured control flow

- Statement sequencing
- Selection with “if-then-else” statements and “switch” statements
- Iteration with “for” and “while” loop statements
- Subroutine (function/method) calls (including recursion)
- All of which promotes “structured programming”
- Break levels (pass, continue, break, return, exit(0))

Code Blocks

- Statements;
- Compound Statements;
- Program Structure (loops);
- Procedure or Functions;

Selection

Condition, Switch, and if-elif-else

- sequential if statements

```
if ... then ... else
if ... then ... elsif ... else
```

Selection

Condition, Switch, and if-elif-else

LISP cond:

```
(cond
    (C1)  (E1)
    (C2)  (E2)
    ...
    (Cn)  (En)
    (T)   (Et)
  )
```

Selection
Condition, Switch, and if-elif-else

Algo-60 if-then-else:

```
if condition then statement
 else if condition then statement
 else if condition then statement
 else statement
```

Selection

Condition, Switch, and if-elif-else

C Case-Switch:

```
switch(expression){
  case value1 : body1; break;
  case value2 : body2; break;
  case value3 : body3; break;
  default: default-body; break;
}
```

Selection

Condition, Switch, and if-elif-else

Ruby if-elsif-else-end:

```
if condition then statement
 elsif condition then statement
 elsif condition then statement
 else statement
 end
```

Selection

- Fortran computed gotos
- jump code
  - for selection and logically-controlled loops
  - no point in computing a Boolean value into a register, then testing it
  - instead of passing register containing Boolean out of expression as a synthesized attribute, pass inherited attributes INTO expression indicating where to jump to if true, and where to jump to if false

Selection

- Jump is especially useful in the presence of short-circuiting
- **Example** (section 6.4.1 of book):

```
if ((A > B) and (C > D)) or (E <> F) then
   then_clause
  else
   else_clause
```

Selection

Code generated w/o short-circuiting (Pascal)

```
if ((A > B) and (C > D)) or (E <> F)
   then
    then_clause
   else
    else_clause
```

```
        r1 := A            -- load
        r2 := B
        r1 := r1 > r2
        r2 := C
        r3 := D
        r2 := r2 > r3
        r1 := r1 & r2
        r2 := E
        r3 := F
        r2 := r2 $<>$ r3
        r1 := r1 $|$ r2
        if r1 = 0 goto L2
L1:     then_clause    -- label not actually used
        goto L3
L2:     else_clause
L3:
```

Selection

Code generated w/ short-circuiting (C)

```
if ((A > B) and (C > D)) or (E <> F)
   then
    then_clause
   else
    else_clause
```

```
      r1 := A
      r2 := B
      if r1 <= r2 goto L4
      r1 := C
      r2 := D
      if r1 > r2 goto L1
L4:   r1 := E
      r2 := F
      if r1 = r2 goto L2
L1:   then_clause
      goto L3
L2:   else_clause
L3:
```

Jump Table

Implementation of Switch

C Case-Switch:

```
switch(expression){
 case value1 : clause_A; break;
 case value2 : clause_B; break;
 case value3 : clause_C; break;
 …
 default: clause_E; break;
}
```

```
T:  &L1               -- controlling expression = 1
    &L2
    &L3
    &L3
    &L3
    &L5
    &L2
    &L5
    &L5
    &L4               -- controlling expression = 10
L6: r1 := . . .       -- calculate controlling expression
    if r1 < 1 goto L5
    if r1 > 10 goto L5     -- L5 is the "else" arm
    r1 -:= 1               -- subtract off lower bound
    r1 := T[r1]
    goto *r1
L7:
```

L1 Handler

```
    goto L6           -- jump to code to compute address
L1: clause_A
    goto L7
L2: clause_B
    goto L7
L3: clause_C
    goto L7
    . . .
L4: clause_D
    goto L7
L5: clause_E
    goto L7

L6: r1 := . . .       -- computed target of branch
    goto *r1
L7:
```

Control Structures II

Iteration

SECTION 6

Loops

- while-loop
- do-while-loop (repeat-until)
- for-loop
- for-each-loop

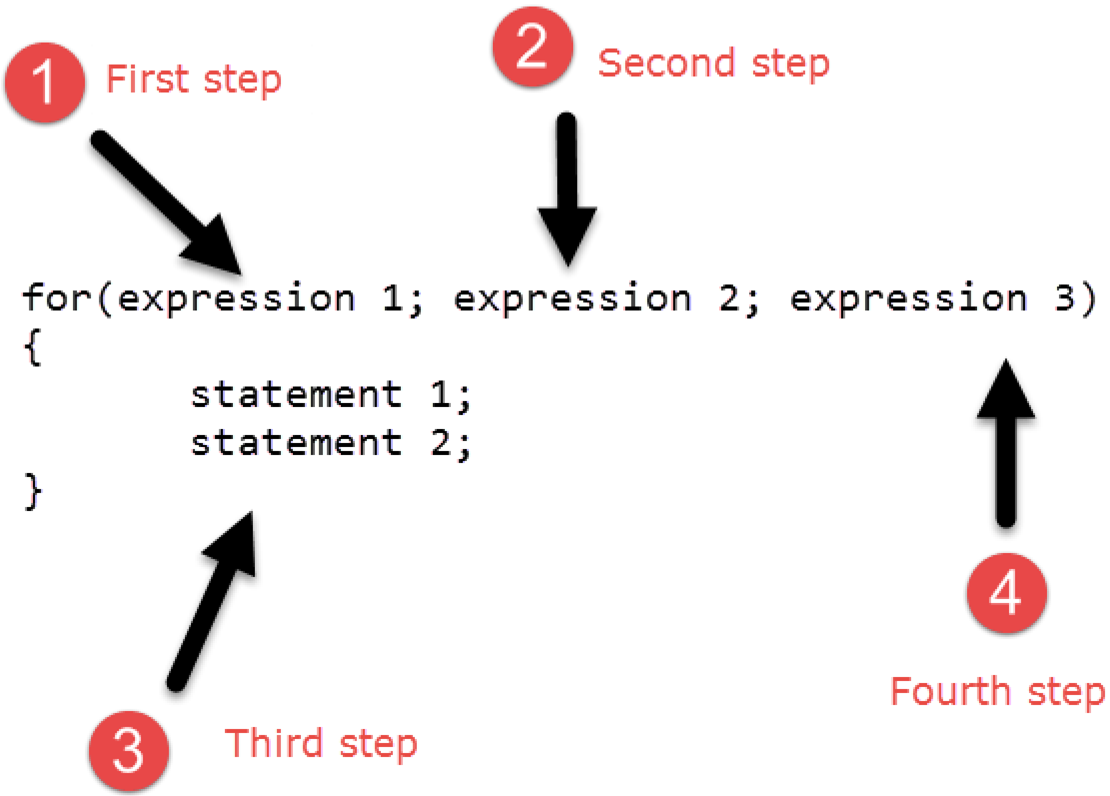
Iteration

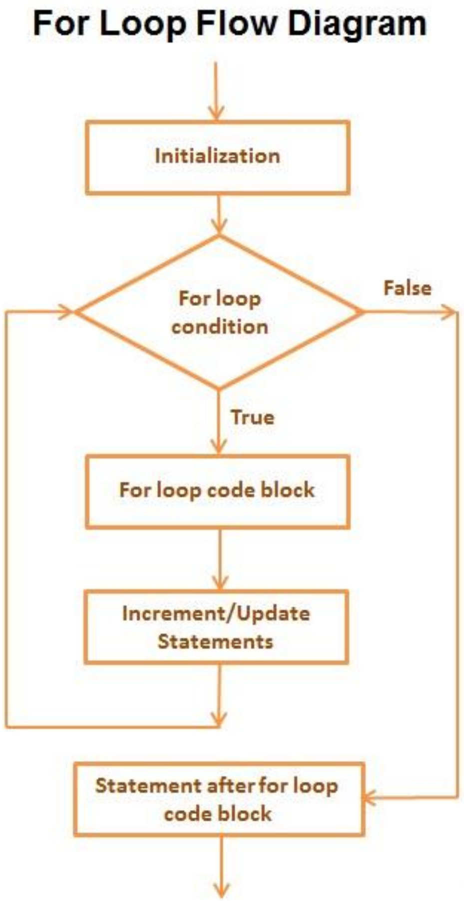
Enumeration-controlled (indexed loop)

- Pascal or Fortran-style **for** loops
  - scope of control variable
  - changes to bounds within loop
  - changes to loop variable within loop
  - value after the loop
- Can iterate over elements of any well-defined set
- repeat a collection of statements a number of times, where in each iteration a loop index variable takes the next value of a set of values specified at the beginning of the loop

for-loop

```
for( a = 10; a < 20; a = a + 1 ){
    printf("value of a: %d\n", a);
}
```





for-loop

```
for( a = 10; a < 20; a = a + 1 ){
      printf("value of a: %d\n", a);
   }
```

for-each-loop

Algorithm

{Sum first *n* integers}
begin
1. **input** *n*;
2. *sum* := 0;
3. **for** *i* := 1 to *n* **do**
4. *sum* := *sum* + *i*;
5. **output** *sum*;
end

Python

```
# sum first n integers

n = int(argv[1])
sum = 0
for i in range(1,n):
    sum = sum + i
print(sum)
```

Logically-Controlled Loop

Pre-Test Loops(P):

```
readln(line)
while line[1] <> '$' do
      readln(line);
```

Post-Test Loops(P):

```
repeat
   readln(line)
until line[1]='$';
```

Post-Test Loops(C):

```
do{
   line = read_line(stdin);
} while (line[0] != '$');
```

Mid-Test Loops(C):

```
for (;;){
   line = read_line(stdin);
   if (all_blanks(line)) break;
   consum_line(lin);
}
```

Iterator

Python Iterator for preorder enumeration of the nodes of a binary tree

Clu, **Python**, **Ruby**, and **C#** allow any container abstraction to provide an iterator that enumerates its items. The iterator resembles a subroutine that is permitted to contain yield statements, each of which produces a loop index value. For loops are then designed to incorporate a call to an iterator. To Modula-2 fragment

```
FOR i:= first TO last by step DO
   …
END
```

With Iterator, a non-linear data structure can be operated in a loop.

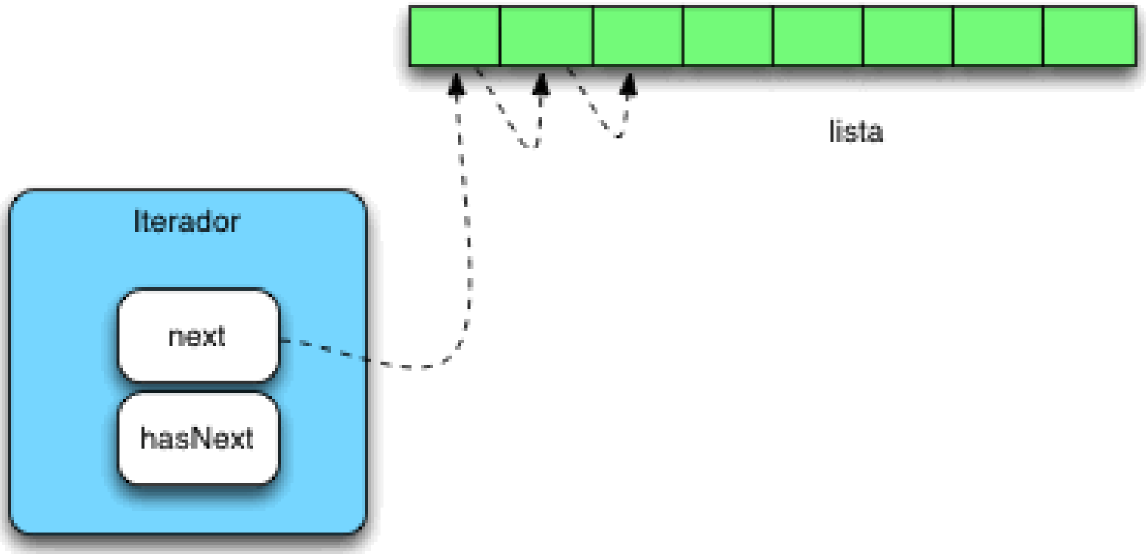
```python
class BinTree:
    def __init__(self):      # constructor
        self.data = self.lchild = self.rchild = None
    ...
    # other methods: insert, delete, lookup, ...

    def preorder(self):
        if self.data != None:
            yield self.data
        if self.lchild != None:
            for d in self.lchild.preorder():
                yield d
        if self.rchild != None:
            for d in self.rchild.preorder():
                yield d
```

Iterator

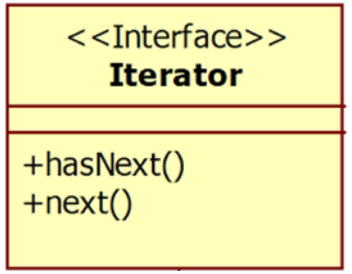
Java Iterable Interface

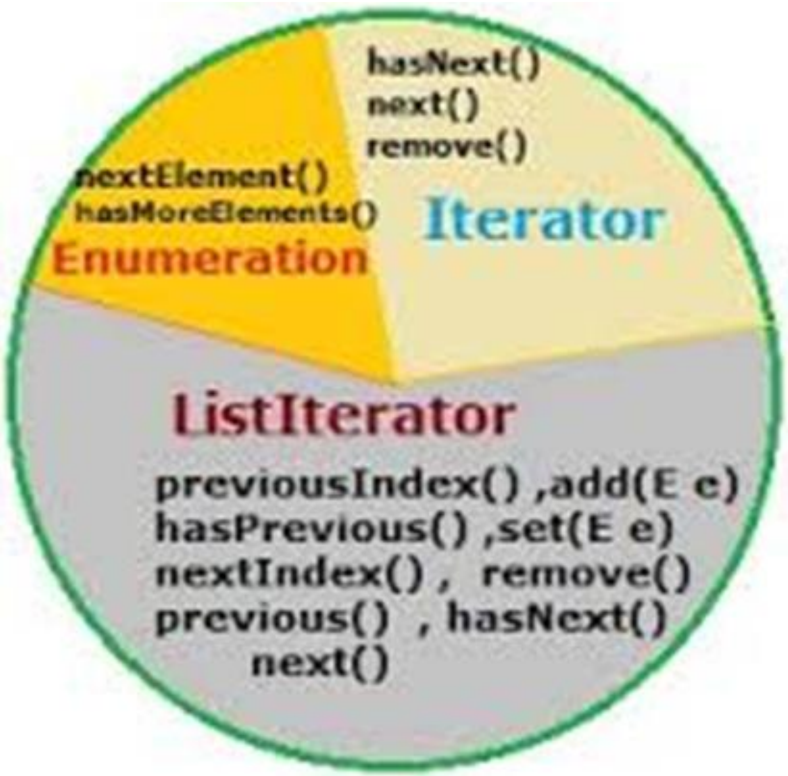
- Iterator has three methods: next(), hasNext(), and remove().
- Iterable has one methods: Iterator()
- While-loop can be used to traverse through this BinTree Data structure.



```java
class BinTree<T> implements Iterable<T> {
    BinTree<T> left;
    BinTree<T> right;
    T val;
    ...
    // other methods: insert, delete, lookup, ...

    public Iterator<T> iterator() {
        return new TreeIterator(this);
    }
    private class TreeIterator implements Iterator<T> {
        private Stack<BinTree<T>> s = new Stack<BinTree<T>>();
        TreeIterator(BinTree<T> n) {
            if (n.val != null) s.push(n);
        }
        public boolean hasNext() {
            return !s.empty();
        }
        public T next() {
            if (!hasNext()) throw new NoSuchElementException();
            BinTree<T> n = s.pop();
            if (n.right != null) s.push(n.right);
            if (n.left != null) s.push(n.left);
            return n.val;
        }
        public void remove() {
            throw new UnsupportedOperationException();
        }
    }
}
```

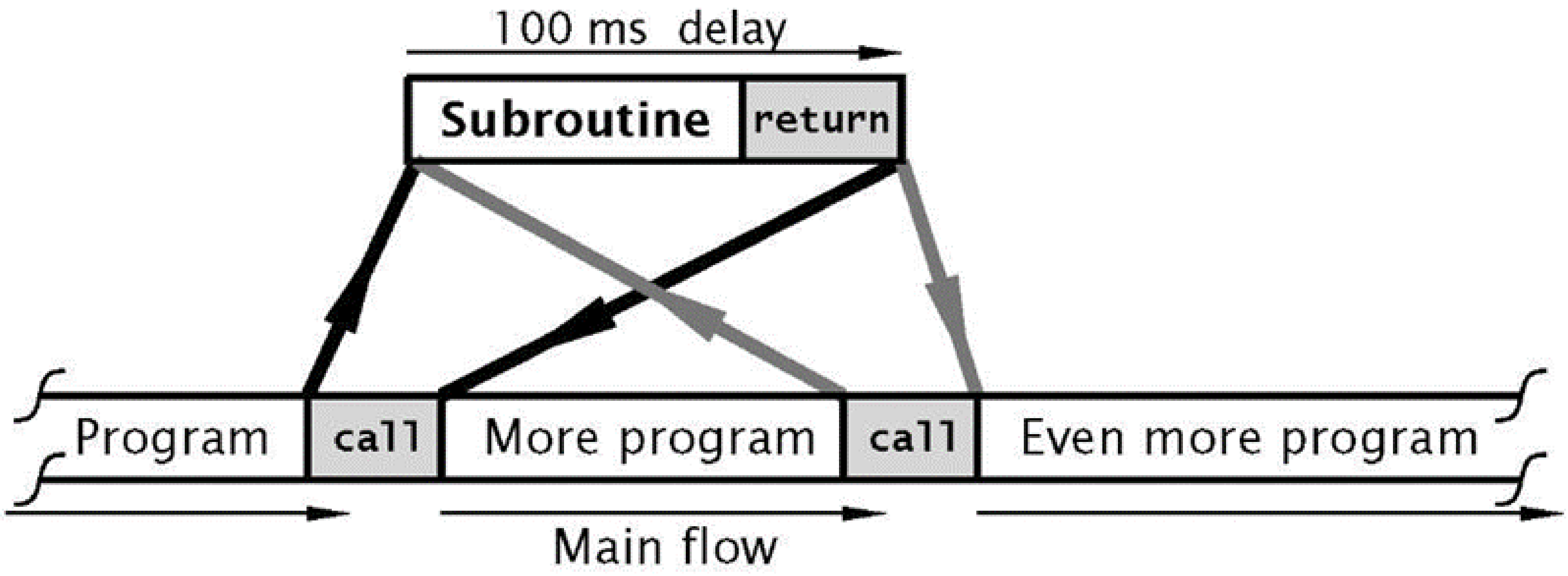




Control Structures III

Function

SECTION 7

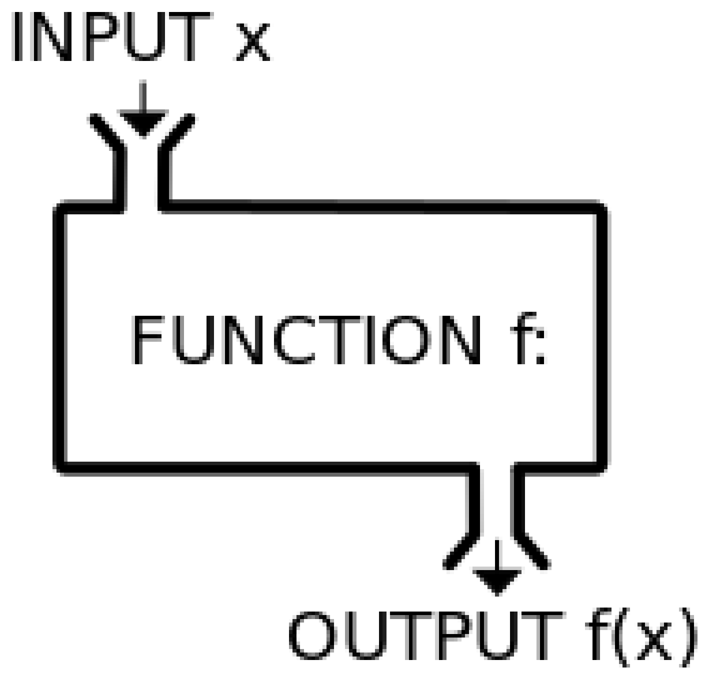

100 ms delay
Subroutine
return
Program
call
More program
call
Even more program
Main flow

Control Structure III

- **Procedural abstraction**: subroutines encapsulate collections of statements and subroutine calls can be treated as single statements
- **Recursion**: subroutines which call themselves directly or indirectly to solve a problem, where the problem is typically defined in terms of simpler versions of itself
- **Concurrency**: two or more program fragments executed in parallel, either on separate processors or interleaved on a single processor
- **Non-determinacy**: the execution order among alternative constructs is deliberately left unspecified, indicating that any alternative will lead to a correct result. (function as pointer, randomized functional calls)

Sub-Routine, Function, and Methods

- Abstraction
- Reusability
- Library



Method Signature

Java

signature

return type

method name

argument type

argument variable

```
public static double sqrt ( double c )
{
    if (c < 0) return Double.NaN;
    double err = 1e-15;
    double t = c;
    while (Math.abs(t - c/t) > err * t)
       t = (c/t + t) / 2.0;
    return t;
}
```

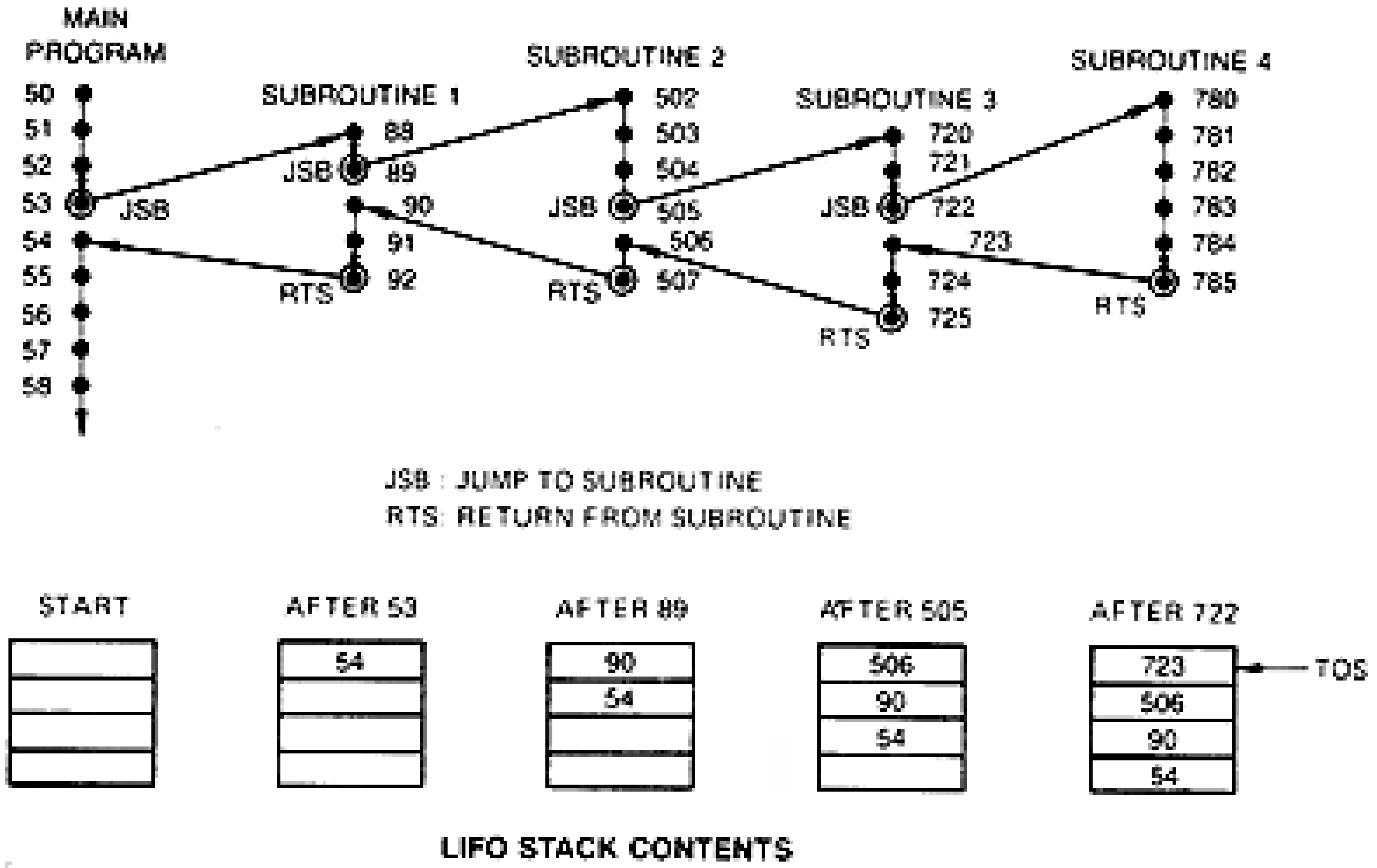
local variables

method body

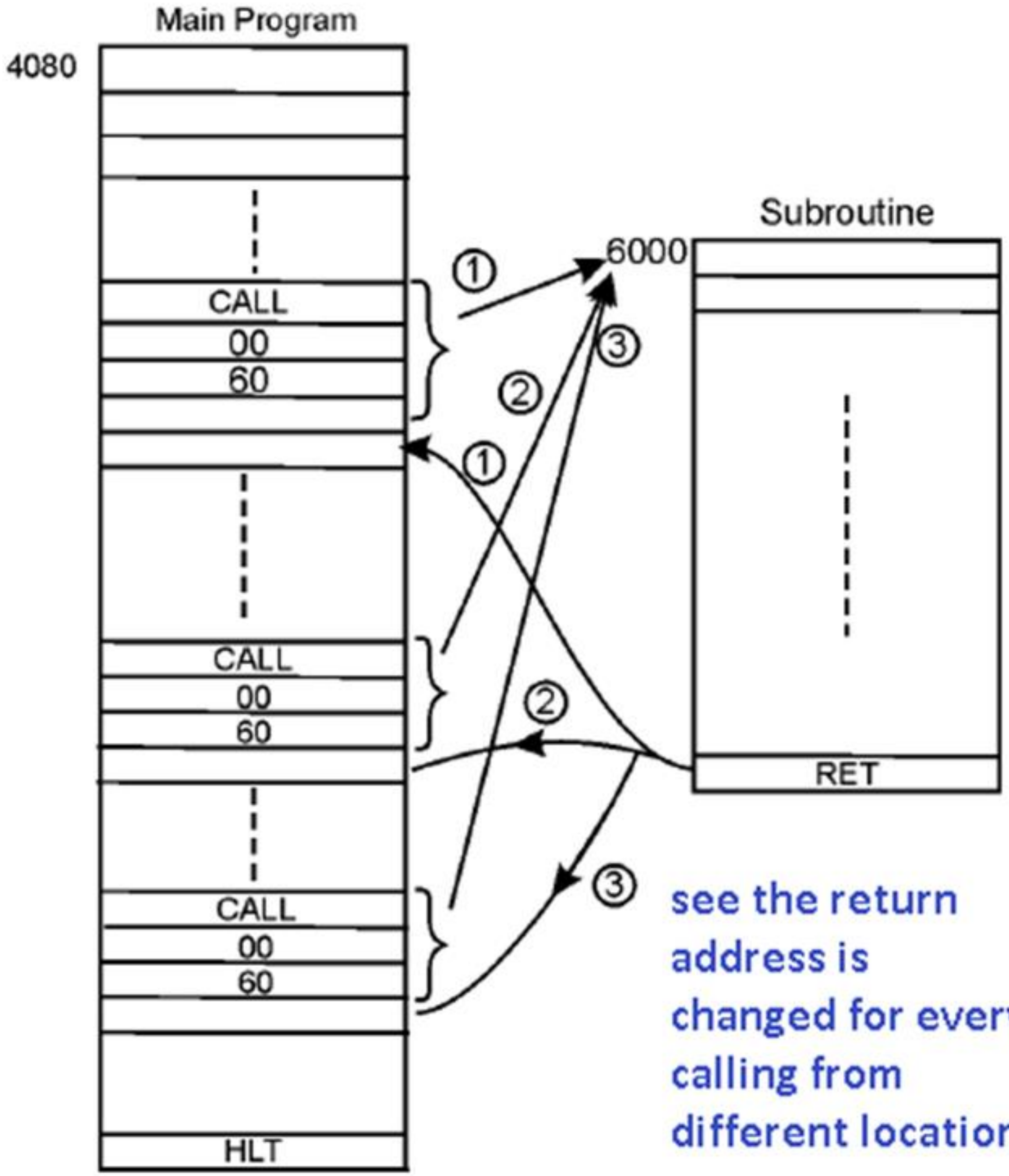
return statement

call on another method

Call Stack



MAIN PROGRAM
SUBROUTINE 1
SUBROUTINE 2
SUBROUTINE 3
SUBROUTINE 4
50
51
52
53
54
55
56
57
58
JSB
RTS
88
89
90
91
92
502
503
504
505
506
507
720
721
722
723
724
725
780
781
782
783
784
785
JSB : JUMP TO SUBROUTINE
RTS: RETURN FROM SUBROUTINE
START
AFTER 53
54
AFTER 89
90
54
AFTER 505
506
90
54
AFTER 722
723
506
90
54
TOS
LIFO STACK CONTENTS

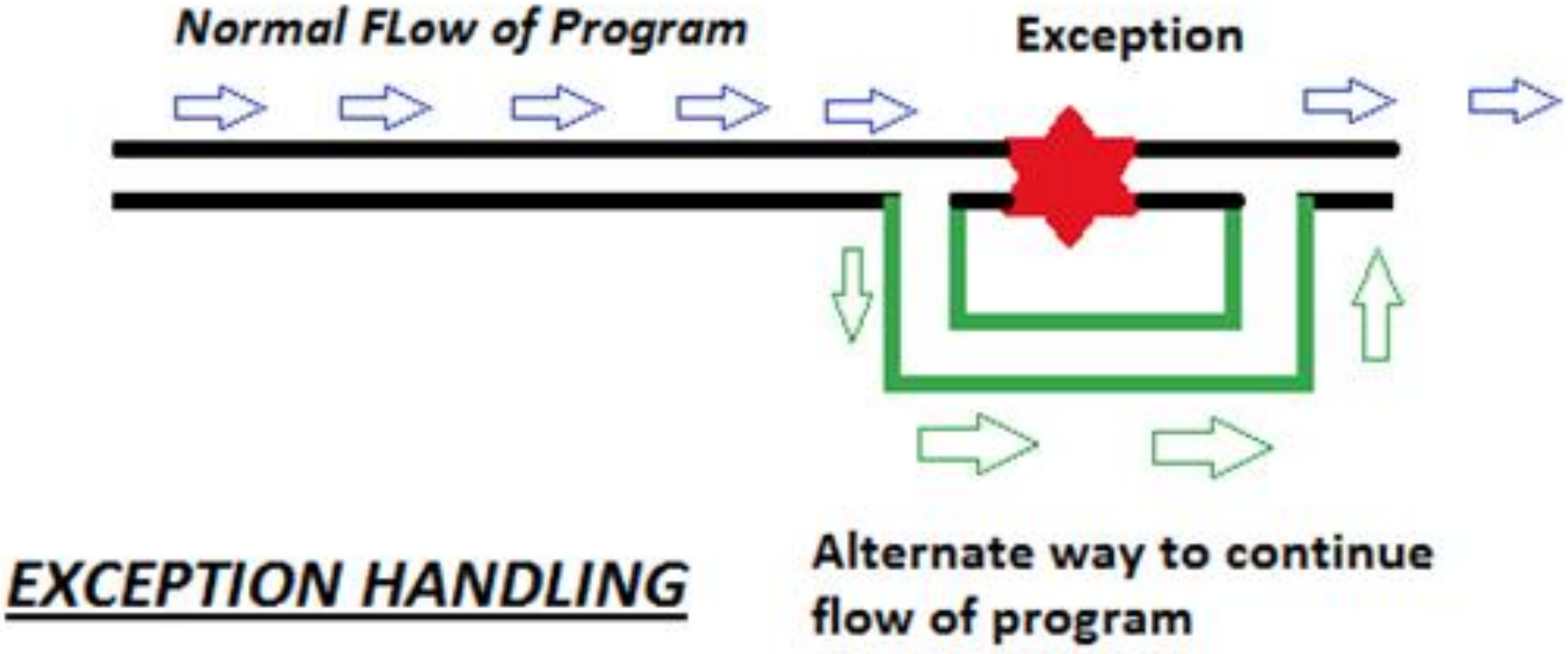


see the return address is changed for every calling from different location

Control Structures IV

Exception

SECTION 8

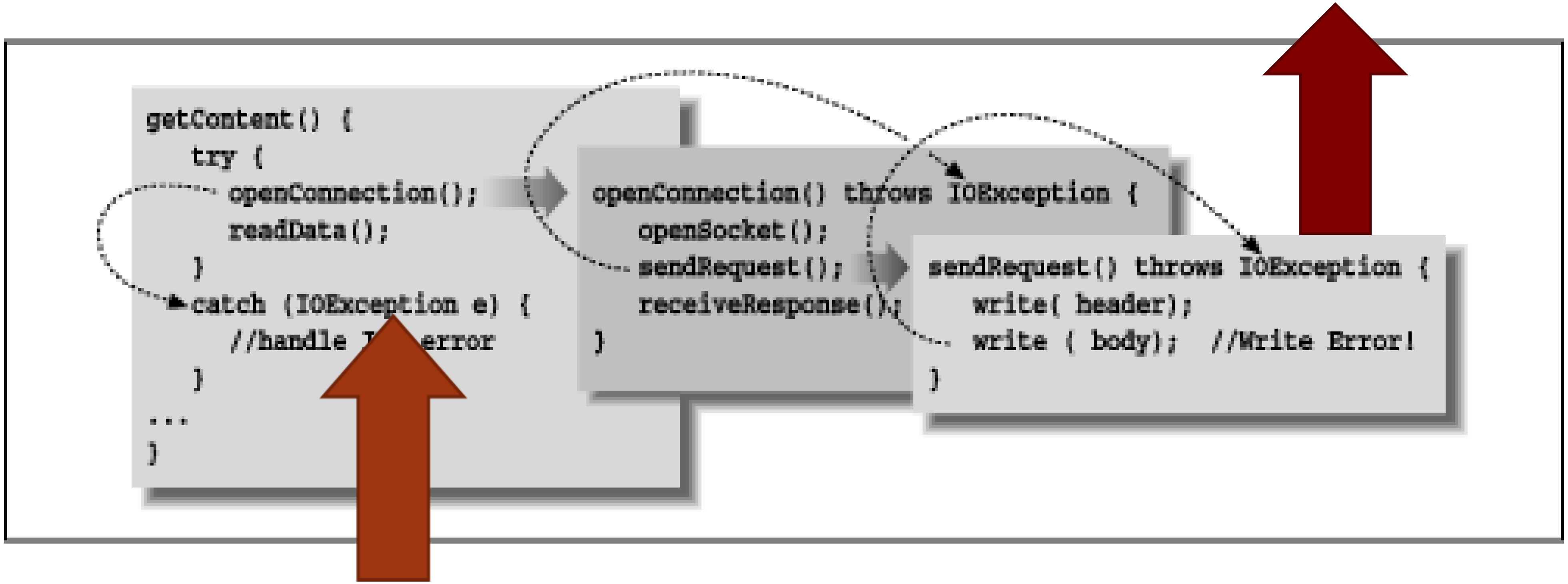

Normal FLow of Program
Exception
EXCEPTION HANDLING
Alternate way to continue
flow of program


Problem
Occurs
Create
Exception
Throw
Exception
Handle
Exception

Exception Handling

Propagate to the Level with Handler

Outgoing Exception Object (like returned object)



Incoming Exception Object as Parameter

Control Structures III

Recursion

SECTION 9

Recursion

- equally powerful to iteration
- mechanical transformations back and forth
- often more intuitive (sometimes less)
- *naïve* implementation less efficient
    - no special syntax required
    - fundamental to functional languages like Scheme

Recursion

- **Recursion:** subroutines that call themselves directly or indirectly (mutual recursion)
- Typically used to solve a problem that is defined in terms of simpler versions, for example:
  - To compute the length of a list, remove the first element, calculate the length of the remaining list in *n*, and return *n*+1
  - Termination condition: if the list is empty, return 0
- Iteration and recursion are equally powerful in theoretical sense
  - Iteration can be expressed by recursion and vice versa
- Recursion is more elegant to use to solve a problem that is naturally recursively defined, such as a tree traversal algorithm
- Recursion can be less efficient, but most compilers for functional languages are often able to replace it with iterations

Tail recursion

- No computation follows recursive call

```
int gcd (int a, int b) {
   /* assume a, b > 0 */
         if (a == b) return a;
   else if (a > b) return gcd (a - b,b);
   else return gcd (a, b - a);
}
```

Tail-Recursive Functions

- *Tail-recursive functions* are functions in which no operations follow the recursive call(s) in the function, thus the function returns immediately after the recursive call:

tail-recursive

```
int trfun()
{ …
  return trfun();
}
```

not tail-recursive

```
int rfun()
{ …
  return rfun()+1;
}
```

- A tail-recursive call could ***reuse*** the subroutine's frame on the run-time stack, since the current subroutine state is no longer needed
  - Simply eliminating the push (and pop) of the next frame will do
- In addition, we can do more for *tail-recursion optimization*: the compiler replaces tail-recursive calls by jumps to the beginning of the function

Tail-Recursion Optimization

Consider the GCD function:

```
int gcd(int a, int b)
{ if (a==b) return a;
  else if (a>b) return gcd(a-b, b);
  else return gcd(a, b-a);
}
```

a good compiler will optimize the function into:

```
int gcd(int a, int b)
{ start:
    if (a==b) return a;
    else if (a>b) { a = a-b; goto start; }
    else { b = b-a; goto start; }
}
```

which is just as efficient as the iterative version:

```
int gcd(int a, int b)
{ while (a!=b)
    if (a>b) a = a-b;
    else b = b-a;
  return a;
}
```

When Recursion is inefficient

The Fibonacci function implemented as a recursive function is **very inefficient** as it takes exponential time to compute:

```
int fib(n) {
  if (n=1) return 1;
  if (n=2) return 1;
  return fib(n-1) + fib(n-2);
}
```