Consider the following example code:

```c
int addThemUp(int x, int y)
{ int z;
  z = x+y;
  return z;
}

main()
{ int c,d,e;
  c = 5;
  d = 6;
  e = addThemUp(c,d);
  e = addThemUp(c,7);
  e = addthemUp(4,c+8);
}
```

**Pass-By-Value**

*mechanism* -

- Formal parameters are *local variables* in the function.
- Their declarations are in the formal parameter list.
- They are initialized to the *actual parameter* values during the call.

**implications** -

- Can pass variables, values, and expressions.
- Can be modified, but modifications don’t affect actual parameters.
- This is the only mechanism in C, Java. It is an option in Pascal, C++.
- Can achieve a call-by-reference effect by passing a pointer.
- In Java, when passing an object, the effect is like pass-by-reference. That’s because an object name is like a pointer.
- In Ada, pass-by-value parameters are *constants*. Thus in Ada, it is not possible to modify the formal parameters.
**Pass-By-Reference**

*mechanism* -
- Formal parameters are *temporary aliases* to the actual parameters.

*implications* -
- Can only pass variables.
- Changing a formal parameter directly affects the actual parameter.
- This is the only mechanism in Fortran. It is an option in Pascal, C++.

**Pass-By-Value-Result**

*mechanism* -
- Formal parameters are *local variables* (same as pass-by-value).
- Initialized to the *actual parameter* values (same as pass-by-value).
- At the time of return, formal parameter values are copied back out to the corresponding actual parameters.

*implications* -
- Can only pass variables.
- Also called “copy-in-copy-out”, “pass-by-result”, and “in-out”.
- Not available in C, C++, Java, Pascal. It is an option in Ada.
- Order of copy-out can be important if formal parameters are modified:

```c
int Test(int x, int y)
{
    x = x+1;
}
main()
{
    int c;
    c = 5;
    Test(c,c);
}  
```

*Now, what is the value of c??*
**Example:**

```c
int c; // global

int Test(int x)
{
    x = x+3;
    c = c+1;
}

main()
{
    c = 5;
    Test(c);
}
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>X 5 8</td>
<td>X 5 8 C</td>
<td>X 5 8</td>
</tr>
<tr>
<td>C 5 6</td>
<td></td>
<td>C 5 6</td>
</tr>
</tbody>
</table>

Result: C=6   Result: C=9   Result: C=8

**Pass-By-Name**

*mechanism* -

- Text of actual parameter replaces text of formal parameter.

*implications* -

- Used in Scala, Panacea, C macros, early versions of Algol, Simula.
- Can result in strange behavior, especially if passing array elements:

```c
Swap(x,y)
{
    int t = x;
    x = y;
    y = t;
}

Swap(i,A[i]) ← trace it!
```
**Jensen’s Device**

A trick that can be used in a **pass-by-name** environment to simulate matrix math:

- Pass an array element subscripted by a variable.
- Also pass in the subscript variable.
- Incrementing the variable allows program to walk through array without subscripts, as if it were a single Matrix operation.

**Example:**

```plaintext
procedure Sum(A,B,C,index)
{
  for index = 1 to max;
  C = A + B;
}
...
(* call *)
Sum(a[i],b[i],c[i],i)
```

Consider the arrays:

a:  [1,2,5]
b:  [6,7,12]

After the call to “Sum”, the array “c” will become: [7,19,17]