Independent Compilation

- independent subprograms
- separate compilation
- independent scopes
  - communication via parameters
  - no type checking between subprograms
- `COMMON` blocks
  - common storage, not shared scope
  - overlaying
- subprogram libraries
- independent modification

Independent Compilation

- FORTRAN

```
PROGRAM MAIN
COMMON X, Y, Z
INTEGER R
…
STOP
END
```

```
SUBROUTINE A …
COMMON U, V, W
INTEGER S
…
RETURN
END
```

```
SUBROUTINE B …
COMMON E, F, G
INTEGER T
…
RETURN
END
```

Block-Structured Languages

- ALGOL 60, Pascal
- monolithic program
- nested procedures
  - inheritance of declarations
- blocks
  - local scope without being procedure
- no independent compilation
- full type checking
- nested procedures may access & modify inherited non-local variables
  - side-effects
  - hard to reason about programs
- closed scope
  - block must declare what is used from outside

Block-Structured Languages

```
program main …
   var v, w : Real;
  procedure a …
     var x, y : Real;
    procedure b …
      var x, y : Real;

    begin … end;

  begin … end;

  procedure d …
     var w, x : Integer

  begin … end;

begin … end.
```

Block-Structured Languages

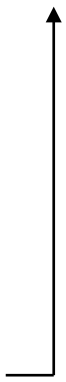
```
declare
   a, b : Integer;
begin
   …

   declare
      b, d : Real;
   begin
      …
   end;

   …
end;
```

Block-Structured Languages

Swapping values of two variables:

Pascal vs Ada

```
procedure something …
  var a, b, h : Integer;
begin
  …
  …
  …
  …
  …
  h := a;
  a := b;
  b := h;
  …
end;
```



```
procedure something …
  a, b : Integer;
begin
  …
  declare
    h : Integer;
  begin
    h := a;
    a := b;
    b := h;
  end;
  …
end;
```

Modules

- Ada, Modula-2
- collect types, procedures, variables & constants
- control scope
- package specification gives interface
- package body gives implementation
- separate compilation of units
- context clause

```
with BankAccounts;
use BankAccounts;
…
am1 := getBalance();
```

or

```
with BankAccounts;
…
…
am1 := BankAccounts.getBalance();
```

- `BankAccounts` package
  - representation in private part
  - separate compilation needs to know size
  - change of representation requires recompilation of client
  - can use access types

```
package BankAccounts is
  type BankAccount is private;
  procedure makeBankAccount ( b: out BankAccount );
  procedure deposit ( b: in out BankAccount;
                      amount: in Integer );
  function getBalance ( b: BankAccount) return Integer;
private
  type ActualBankAccount;
  type BankAccount is access ActualBankAccount;
end BankAccounts;

package body BankAccounts is
  -- definitions of ActualBankAccount,
  -- makeBankAccount, deposit and getBalance
end BankAccounts;
```

Hybrid Languages

- object-oriented & procedural
- Delphi, C++
- procedures can still be separate from class (procedural)
- C++ - program is a set of declarations
  - function prototypes
  - classes
    - data members
    - function members
- `Cost` example
  - data members and methods
  - constructor
  - scope resolution operator
  - selectors vs modifiers
  - object declaration and reference
  - reference to object declaration, creation and reference

```
int checkCents(int c); // function prototype

class Cost {
private:
   int cents, dollars;
public:
   Cost(int d, int c); // constructor
   void add(int d, int c);
   int getDollars() const;
   int getCents() const;
};

// main program
void main() {
   Cost dress(45, 95);
   Cost *book = new Cost(15, 50);
   …
   dress.add(5, 0);
   book ->add(3,15);
   …
}
```

```
// definition of function body
int checkCents(int c) {
   if (c < 100)
        return 1;
   else
        return 0;
}  //checkCents

// definition of method bodies
void Cost::Cost(int d, int c) {
   dollars = d; cents = c;
}  // constructor

void Cost::add(int d, int c) {
   cents += c;
   if (cents > 100) {
      dollars += d + cents/100;
      cents = cents % 100;
   }
   else
      dollars += d;
}  // add
```

```
int Cost::getDollars() const {
   return dollars;
}  // getDollars

int Cost::getCents() const {
   return cents;
}  // getCents
```

Object-Oriented Languages

- Java
- `Cost` example
  - starting point - `main` method
  - `Cost` class
  - objects always accessed by reference

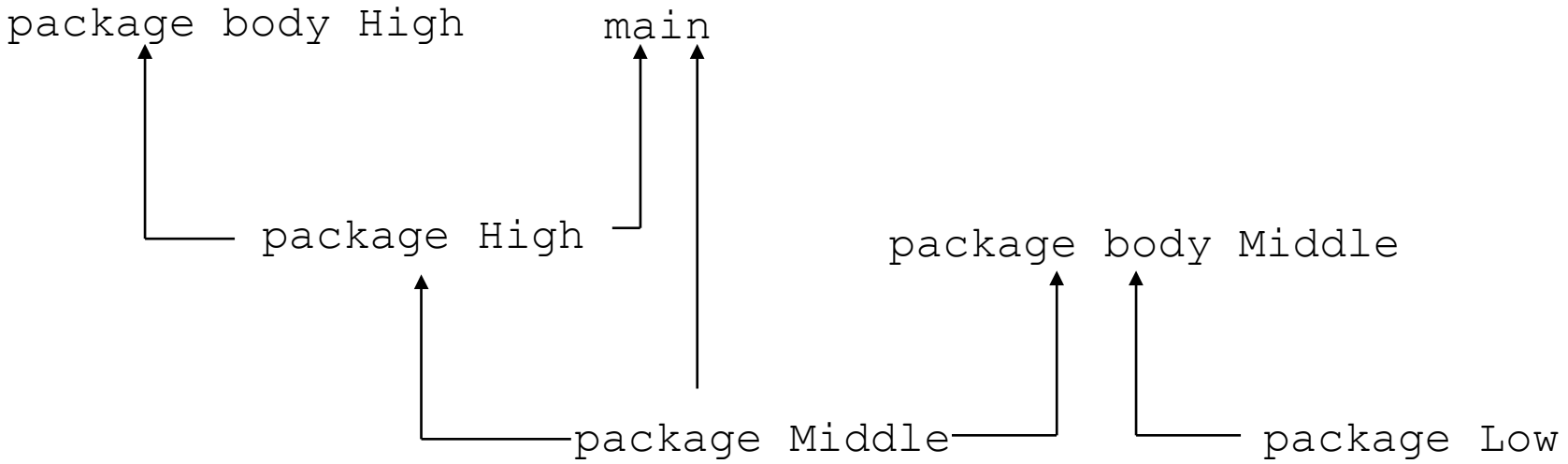
```
import java.awt.*;

public class Example extends Frame {
   private Cost dress = new Cost(45, 95);
   private Cost book = new Cost(15, 50);
   …
   public static void main (Sting [] args) {
        Example ex = new Example();
        ex.setSize(400,200);
        ex.setVisible(true);
   } // main
   public Example() {
        super(“Example”);
        …
        dress.add(5, 0);
        book.add(3, 15);
   }  // constructor
   …
} // Example
```

```
class Cost {
  private int cents, dollars;
  public Cost(int d, int c) {
       dollars = d; cents = c;
  } // constructor
  public void add(int d, int c) {
       cents += c;
       if (cents > 100) {
            dollars += d + cents / 100;
            cents = cents % 100;
       }
       else
            dollars += d;
  }  // add
  public int getDollars() {
       return dollarts;
  }  // getDollars
  public int getCents() {
       return cents;
  } // getCents
} // Cost
```

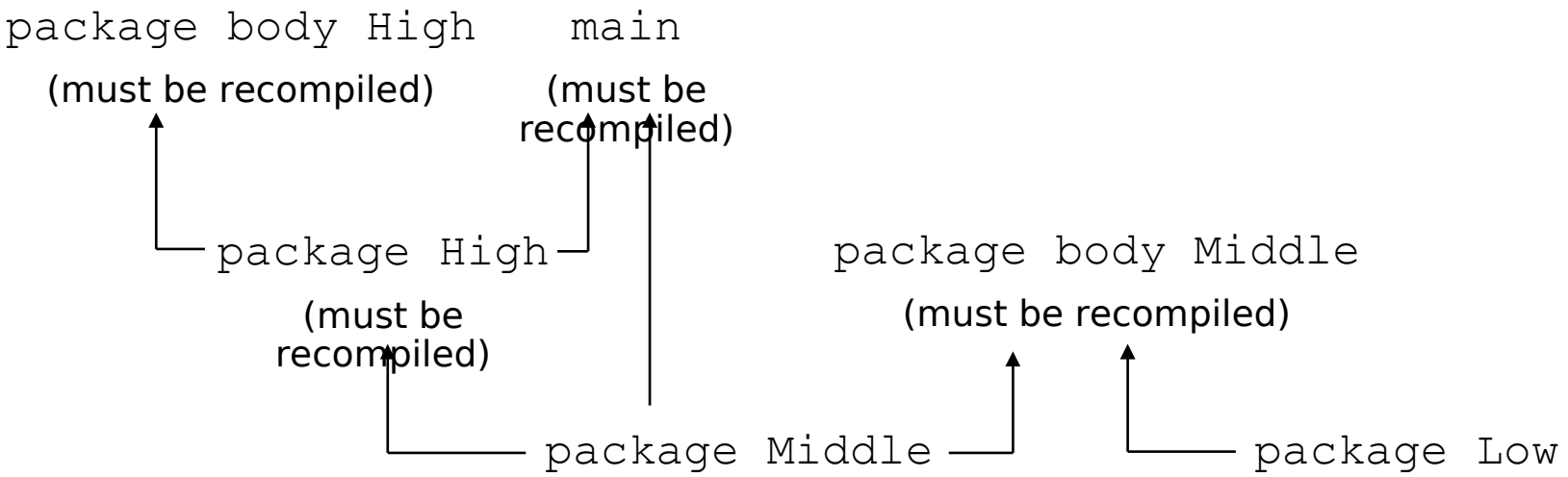
Separate Compilation

- independent compilation vs separate compilation
- compilation units
- dependencies
- order of (re)compilation
- modification
  - of specification
  - of implementation
- C++
  - no automatic facility
  - header files
  - make utility
  - disadvantages
- IDEs
  - can handle dependencies and recompilation
- Java
  - specification & implementation in same file
  - interfaces

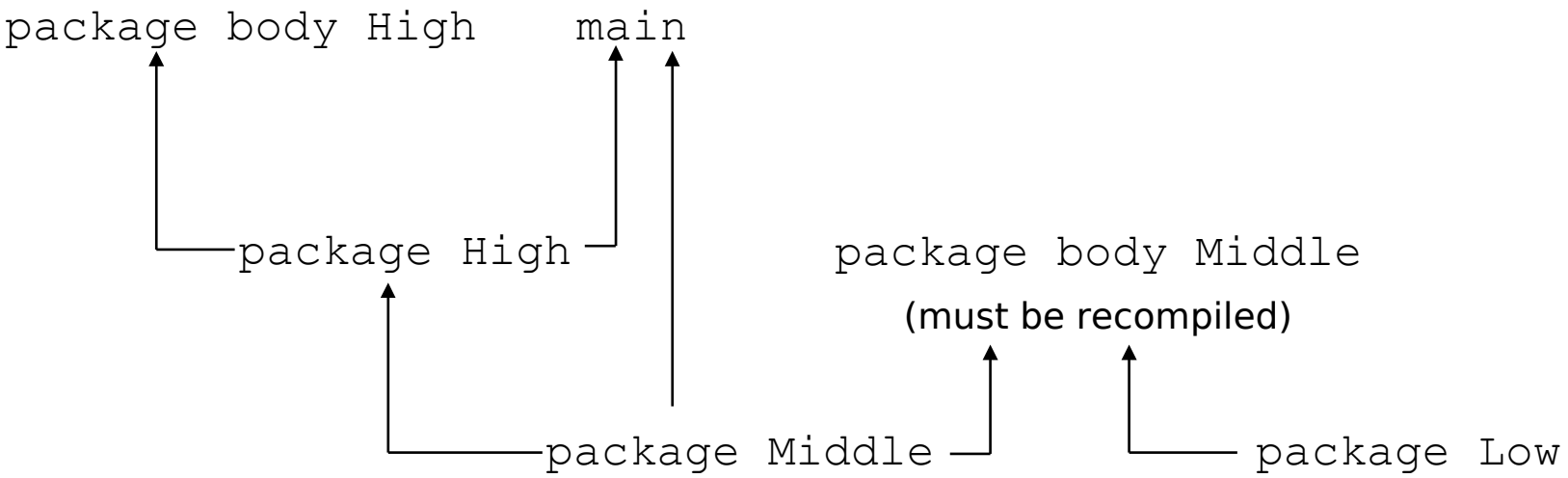
```
package Low is … end Low;

package Middle is … end Middle;

with Low;
package body Middle is … end Middle;

with Middle;
package High is … end High;

package body High is … end High;

with High, Middle
procedure main is begin … end main;
```



Package dependencies



Effect of recompiling package `Middle`



Effect of recompiling package `Low`

Larger Units

- larger units of structure
- system may contain 1000s of classes
- Java
  - package
  - `import` clause
  - collection of related classes
    - stronger coupling
    - non-public classes still visible within package
    - `friend` in C++

Procedures, Functions & Methods

- formal parameter vs actual parameter
- parameter passing
- C++ swap

```
void swap(int& first, int& second) {
     int intermediate;
     intermediate = first;
     first = second;
     second = intermediate;
} // swap
```

- procedural abstraction
  - new operations

Object-Oriented Languages

- methods
- target object
  - implicit parameter

```
bk1.deposit(6);      vs      deposit(bk1,6);
```

- access to non-local data
  - target object
- self reference (`this`, `self`)

```
privmethod();        vs      this.privmethod();
```

- class methods

Parameter Passing

- modes
  - in
  - out
  - update (in/out)
- implementations
  - in – call-by-value, call-by-constant-value, call-by-reference-constant
  - out – call-by-result
  - in/out – call-by-value-result, call-by-reference, call-by-name

In Parameters

- call-by-value
  - Pascal, C++, C, Java
  - formal is copy of actual

```
int total(int val) {
   int sum = 0;
   while (val > 0) {
      sum += val;
      val--;
   }
   return sum;
}// total
```

In Parameters

- call-by-constant-value
  - Ada, FORTRAN 90
  - call-by-value but formal is a constant

```
function total(val : in Integer) return Integer is
      sum : Integer  := 0;
      count : Integer  := val;
begin
      while count > 0 loop
         sum := sum + count;
         count := count + 1;
      end loop;
      return sum;
end total;
```

In Parameters

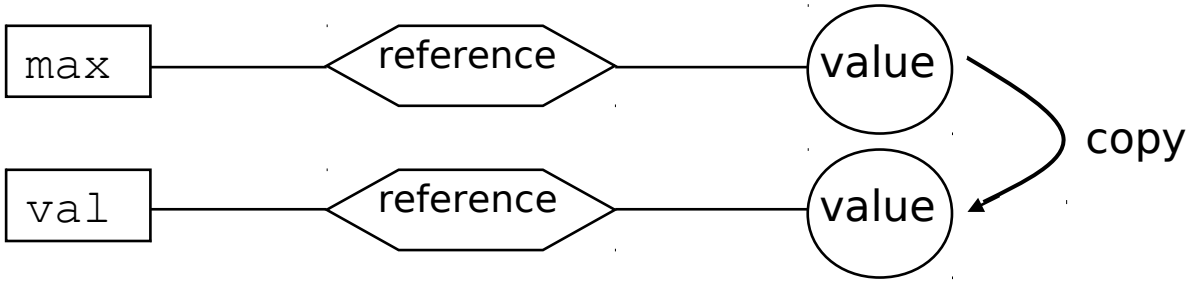
- call-by-reference
  - cost of copying structures (e.g. large arrays)
  - Pascal, C++
  - formal is reference to actual
  - no protection
- call-by-reference-constant
  - Ada, FORTRAN 90, C++
  - call-by-reference but formal is constant
  - C++

```
float sum(const Matrix& m) { … }
```

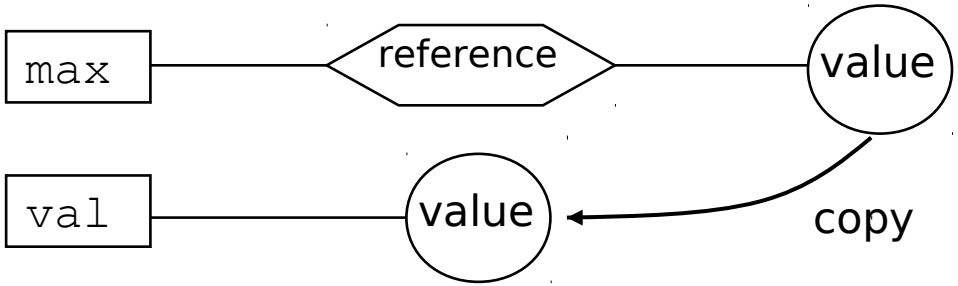
In Parameters

- value vs constant-value vs reference-constant

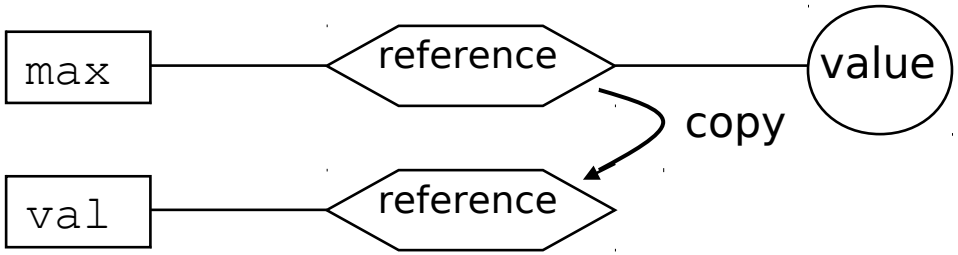
call-by-value



call-by-constant-value



call-by-reference-constant



Out Parameters

- call-by-result
  - Ada
  - formal is uninitialized local
  - value copied to actual at exit

```
procedure read_negative(neg_number : out Integer) is
    number : Integer;
begin
     get(number);
     while number >= 0 loop
        put_line("number not negative, try again");
        get(number);
     end loop;
     neg_number  := number;
end read_negative;
```

  - Algol W
    - address of actual computed at exit

Update Parameters

- call-by-value-result
  - Ada
  - formal is copy with copy back to actual at exit

```
procedure update(balance : in out Integer) is
     transaction : Integer;
begin
     for j in 1 .. 10 loop
        get(transaction);
        balance := balance + transaction;
     end loop;
end update;
```

Update Parameters

- call-by-reference
  - Pascal

```
procedure update(var balance : Integer);
```

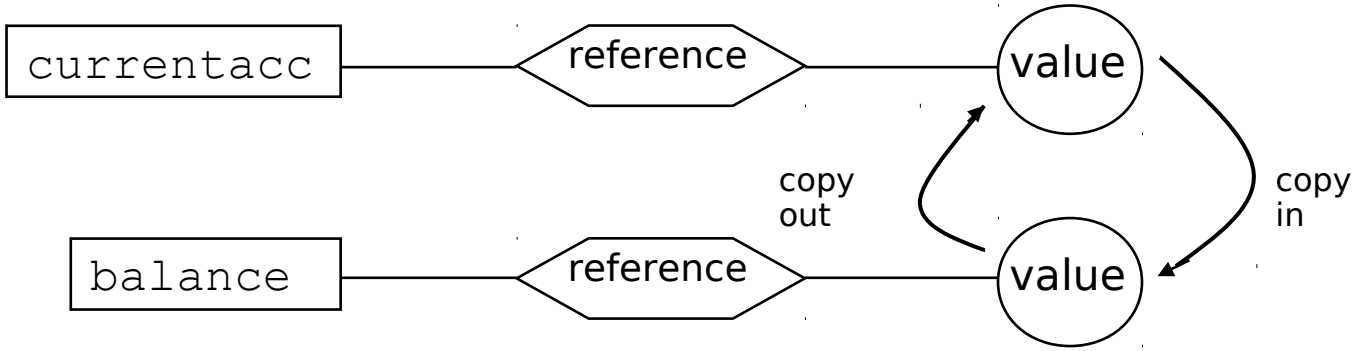
  - C++

```
void update(int& balance);
```

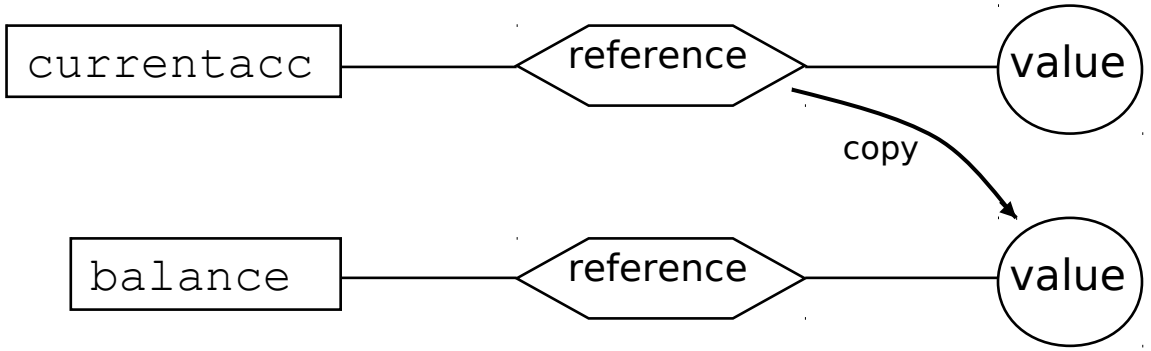
  - value-result vs reference
  - aliasing
    - actual in inherited scope and parameter
    - same actual as multiple parameters
  - expression parameters?

Update Parameters

call-by-value-result



call-by-reference



Java

- call-by-value only
- cannot modify primitive type parameters
- object variables are references
  - reference passed
  - object can be modified, but cannot change to different object

C

- call-by-value only
- can pass addresses (`&` operator)
  - procedure treats as pointer

```
void swap(int *first, int *second) {
     int intermidiate;
     intermidiate =  *first;
     *first = *second;
     *second = intermidiate;
}

swap(&higher, &lower);
```

  - disadvantages

Call-by-name

- textual substitution
- ALGOL 60
- actual parameter is unevaluated at call, but evaluated upon each reference
- seldom used in imperative languages
- used in functional languages (lazy evaluation)

Languages & Mechanisms

Language	Mechanism						
	value	constant-value	reference-constant	result	reference	value-result	name
Algol 60	*						*
Fortran 90		*		?	?	?	
Pascal	*				*		
C	*						
C++	*		*		*		
Ada (scalars)		*		*		*	
Ada		?	?	?	?	?	

Comparison

```
…
var element : Integer;
     a : array [1 .. 2] of Integer;

procedure whichmode(x : ? mode Integer);
begin
   a[1] := 6;
   element := 2;
   x := x + 3;
end;

begin
   a[1] := 1; a[2] := 2;
   element := 1;
   whichmode(a[element]);
   …
```

Comparison

Mechanism	Result		
	a[1]	a[2]	element
call-by-value	6	2	2
call-by-value-result (Algol W)	6	4	2
call-by-value-result	4	2	2
call-by-reference	9	2	2
call-by-name	6	5	2