Programming languages

slidesPRL2021-part2.pdf

Bogdan Wiszniewski
rm. EA423
ph. 58-347-1089
email: bowisz@eti.pg.gda.pl

Course organization

- **Office hours:**

  Tuesdays, 17:15-19:00

- **Project:**

Dr. Magdalena Godlewska,
rm. EA405, Magdalena.Godlewska@eti.pg.gda.pl

Dr. Jerzy Dembski,
rm. EA422, jerzy.dembski@eti.pg.gda.pl

Course organization

- **Objectives:**

1. **Introduce the concept of a programming model supported by the class of programming languages**
2. **Characterize various programming models from the perspective of evolution of programming languages.**
3. **Point out the theoretical and practical limitations affecting usability of various models in the design and coding of IT products**

Course content

1. Procedural programming.
2. Linear syntax (FORTRAN); block syntax (ALGOL, PASCAL).
3. Identifier-object bindings; binding scope.
4. Types of parameter passing techniques; static and dynamic procedure calls.
5. Data abstraction; data access control. Modularization (MODULA, ADA).
6. Exception handling models.
7. Co-routines. Rendezvous.
8. Object oriented programming. Inheritance. Polymorphism. SMALLTALK, C++.
9. Mid-term exam #1

Course content

10. Symbolic computation. Tail-end recursion: LISP.
11. Functional programming: Haskell, XSL.
12. Programming in logic: PROLOG.
13. Facts, rules, goals. Matching and unification.
14. Constrained logic programming (CLP)
15. Syntax definition methods.
16. Denotational semantics.
17. Mid-term exam #2

Literature

Ada Programming,
http://en.wikibooks.org/wiki/Ada

ADA Core, the GNAT Pro Company,
http://www.adacore.com/home
https://libre.adacore.com/

Cincom Smalltalk,
http://www.cincomsmalltalk.com/

D. S. Touretzky: Common Lisp: A Gentle Introduction to Symbolic Computation,
http://www.cs.cmu.edu/~dst/LispBook/

Literature

Altova XMLSpy (JSON and XML editor for modeling, editing, transforming, and debugging)
https://www.altova.com/xmlspy-xml-editor

SWI-Prolog,
www.swi-prolog.org

A. Niederliński: A Gentle Guide to Constraint Logic Programming via ECLiPSe
http://www.anclp.pl/

The ECLiPSe Constraint Programming System
https://eclipseclp.org/

Grading

A#	week	%	comment
A1	4	15	Haskell: function composition
A2	7	15	Prolog: relations, reasoning
T1	4/5	20	lectures 1-7
A3	10	15	ADA: package, threads, rendezvous
A4	13	15	SmallTalk: objects, methods
T2	8	20	lectures 9-14

= 100 final score

Pass/fail criteria

1. Total score of 50% minimum
2. Attending both midterm exams (any non-zero score accepted)
3. All assignments will have to be submitted in the due time. No late assignments accepted, except of a valid medical excuse.

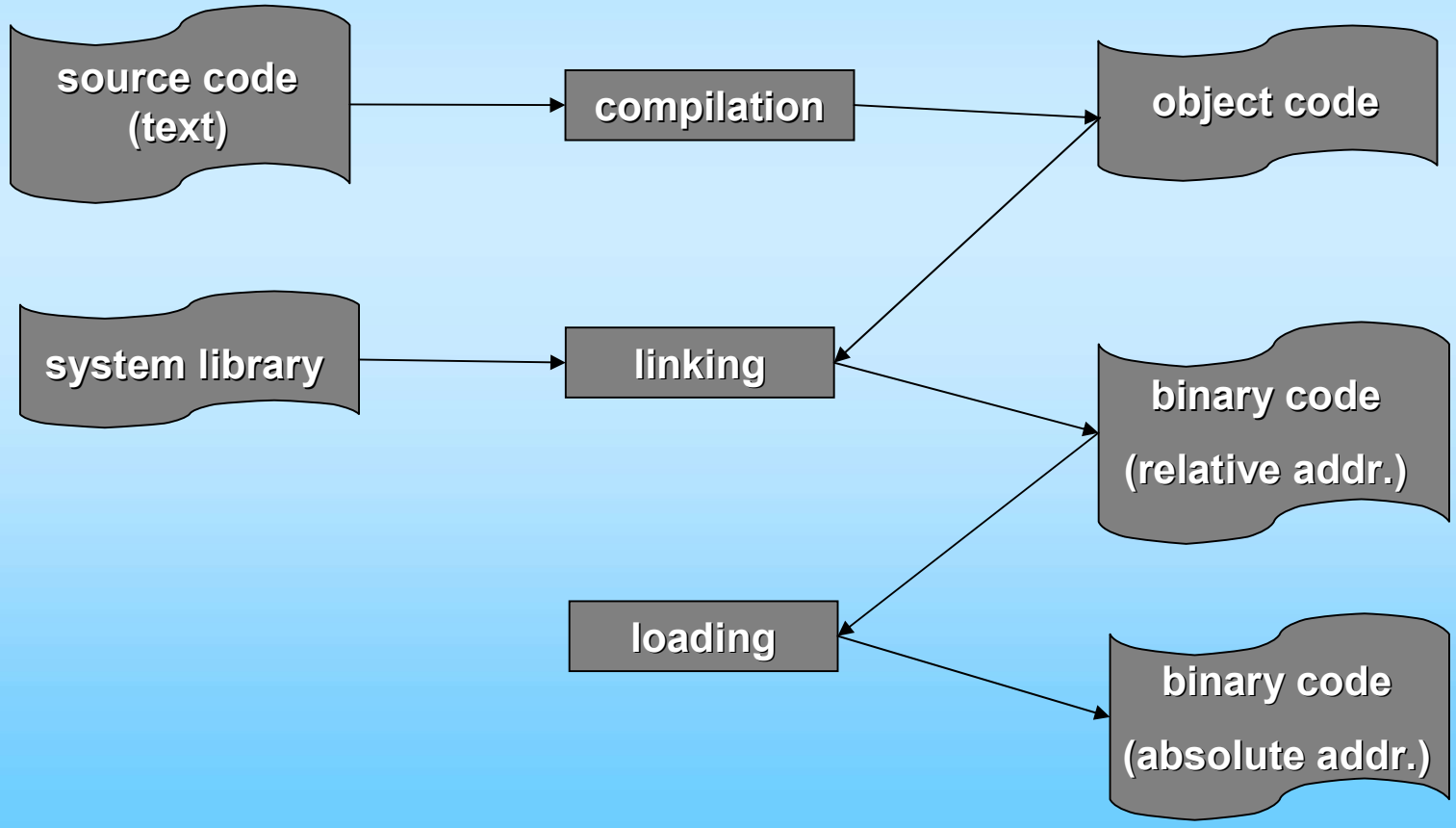
Programming language

- Independence from the structure of machine code, computer architecture, memory organization, etc.
  - translation into machine language (compilation)
  - interpretation (virtual machine)
- Solution modeling
  - procedural
  - object oriented

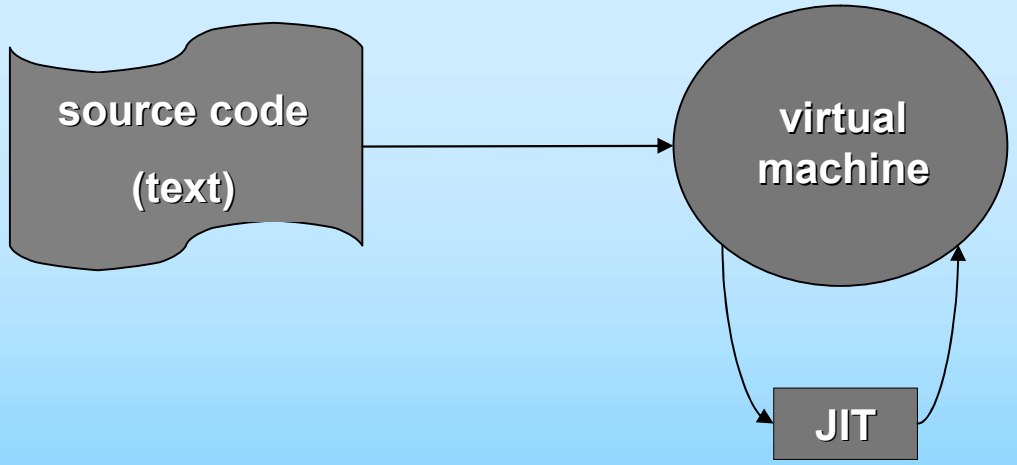
  imperative languages
  - functional
  - relational

  declarative languages

Compilation



Interpretation



Procedural model

- Data delivered to the function
- Global system state shared by all functions
- Local function state reset when function is called
- von Neumann architecture

Quindi condivide dati e isturzioni nello stesso spazio di memoria

Object oriented model

- A set of interacting objects (mutual provision of services)
- Continuity of the private state of the object (state history)
- Messages instead of data sharing
- Physical distribution of objects (going beyond the von Neumann architecture)

(saltato)

Functional model

- Functions delivered to data
- No data flow
- Evaluation instead of execution

(saltato)

Relational model

- Description of the result, not the solution
- First order predicate calculus (quantifies only variables that range over individual elements of some domain).

Characteristics of the imperative language

- Program organization
- Control structures
- Data structures
- Lexical structures (identifiers)
- Syntax
- Semantics

Program org

- **Linear organization**

  main program + subprograms

FORTRAN	C

c function 1
subroutine F_1 (arg)
integer arg
end
	/* function 1 */
	int F_1 (int arg)
	{
	}

c function 2
subroutine F_2
end
	/* function 2 */
	int F_2 (void)
	{
	}

c ...	/* ... */

c function N
subroutine F_N (x)
character x
end
	/* function N */
	int F_N (char x)
	{
	}

c main program program EXAMPLE integer a call F_1(5) call F_2 c ... call F_N ('A') end	/* main program */ void main(void) {int a=0; F_1(5); F_2(); /* … */ F_N('A'); }
End	

Program organization

- **Linear organization**

access to variables

Fortran

```
program main
      real alpha, beta
      common /coeff/ alpha, beta
      …
      end

 subroutine sub1 (…)
      real alpha, beta
      common /coeff/ alpha, beta
      …
      return
      end

…

 subroutine sub2 (…)
      real alpha, beta
      common /coeff/ alpha, beta
      …
      return
      end
```

C

```
int A; /* data object */

/* function 1 */
int F_1 (int arg)
{A++; /* access to object A */
}

int B; /* data object */

/* function 2 */
int F_2 (void)
{B=2*A; /* access to objects A and B */
}

/* ... */

int Z; /* data object */

/* function N */
int F_N (char x)
{B=1-Z; /* access to objects B and Z */
}

/* main program */
void main(void)
{int a;
a=F_1(5)+F_2()+F_N('A');
}
```

The concept of instructions

- **Declarations (performed by the compiler)**
  - **static allocation of memory bytes for the object**
  - **binding the object name (identifier) with the object representation in the computer memory**
  - **object initialization**
- **Statements (performed by the processor)**
  - **computing expressions**
  - **control flow**
  - **data input**
  - **data output**

→ von Neumann's model (the register transfer level, RTL)

Control structures

abstraction rule → independence from the internal organization of the computer

1. **Unconditional branching statements:**
   - **goto** *label*;
   - **continue;**
   - **break;**
   - **return** *expression*;
2. **Conditional statements**
   - **if** (*expression*) *statement*
   - **if** (*expression*) *statement* **else** *statement*
   - **switch** (*expression*) *statement*
3. **Iteration statements:**
   - **while** (*expression*) *statement*
   - **do** *statement* **while** (*expression*);
   - **for** (*expression*; *expression*; *expression*) *statement*
4. **Function/procedure call statements**

Data structures

- **The concept of types**
  {problem domain} → {language domain}
- **Type identifier**
  its name directly identifies the type,
  int, long, float, double, char, ...
  and the binary representation
  The number and arrangement of bytes in the computer memory
- **Different types require different operations**
- **Type equivalence**
  Name equivalence
  different names indicate different types (→ *strong typing*)
  Structural equivalence
  identical number and arrangement of bytes
  (→ *modern languages do not provide in general, a few exceptions exist)*

Type equivalence

- **Type synonyms**

  Declaration ‘typedef’

C

```
typedef char letter;

char A = 'a';
letter B = 'b';
int main(void)
{ letter C = A;
  char D = B;
}
```

C

```
enum Boolean {false, true};

Boolean YES = false;
Boolean NO = true;
int main(void)
{
   if (YES)
      NO = false;
   else
      NO = YES+1;
}
```

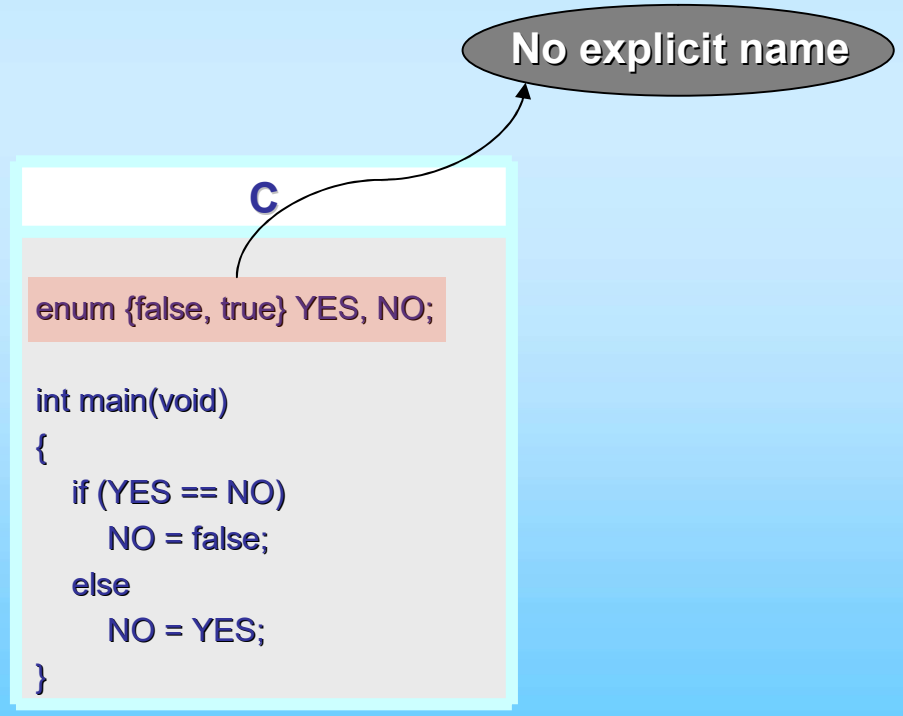
- **Structural equivalence**

  Enumeration data type in C

Compiler considers ‘YES’ to be ‘int’

Type equivalence

- **Anonymous type**



Redefinition not allowed

C

```
enum {false, true} YES;
enum {false, true} NO;

int main(void)
{
    if (YES == NO)
        NO = false;
    else
        NO = YES;
}
```

Type equivalence

- **Anonymous type**

Ada

```
...
tab1: array(1..3, character range 'a'..'d') of integer;
tab2: array(1..3, character range 'a'..'d') of integer;

tab1:=tab2 – operation not allowed
...
```

Type equivalence

- **Subtypes**

Ada

```
...
N: INTEGER
type INDEX is range 1..100;
I: INDEX
...
N := I + 1;
...
```

other operator

different types

Ada

```
...
N: INTEGER
subtype INDEX is INTEGER range 1..100;
I: INDEX
...
...
N := I + 1;
...
```

subtypes inherit operators

Pascal

```
...
type
 index = 1..100;
var
  N: integer;
   I: index;
...
```

Type equivalence

- **Derived types**

Ada

```
...
type KELVIN is new FLOAT;
type CELSIUS is new FLOAT;
...
type PERCENT is new INTEGER range 0..100;
...
I := PERCENT(N);
...
type Integer_1 is range 1 .. 10;
type Integer_2 is new Integer_1 range 2 .. 8;
A : Integer_1 := 8;
B : Integer_2 := A; -- operation not allowed
```

→ *only explicit call of the type conversion function*

Type conversion

- **Implicit type conversion**
  - **substituting the value for an object transforms its type into the type of the result object**
  - **narrowing the result type domain**
  - **loss of accuracy**

– sostituendo il valore con un oggetto sostituendo il valore con un oggetto trasforma il suo tipo nel tipo di trasforma il suo tipo nel tipo dell'oggetto risultato l'oggetto risultato
– restringimento del dominio del tipo di risultato restringimento del dominio del tipo di risultato
– perdita di precisione perdita di precisione

passiamo valore float a un funzione int, quindi perdiamo info

C

```
…
void ff(int);
…
int a = 1.0;
…
ff(3.0);
…
int a=3.14159;
ff(3.14159);
…
```

Type conversion

- **Type promotion**
  - **Performing operations on the expanded type**

C

```
float b = 3;
…
int val = 5;
…
printf("suma=%d",val+3.14159);
…
```

Promotion to '*double*'

Type conversion

- **Explicit type conversion (casting)**
  - **enforced by the programmer**

Ada

```
i, j : Integer;
k : Float := 15.0;
...
i:= Integer(k);
...
str: String := "13";
...
j := Integer'Value(str);
```

C

```
…
a= int(1.2);
b = (float)3;
…
float x=3.1415;
...
i=i+(int)x;
...
```

Type conversion

- **Automatic**
  - **defined by the programmer but**
  - **implicit for the compiler**

a programmer implements conversion of 'int' to 'huge'

type 'huge' may be used wherever type 'int' is used.

C++

```
class huge {
public:
   operator int();
private:
   int D[100]; // table of digits
};
...
huge::operator int() {
...
return ...;
};

huge L;
int j=1+L; // int() called automatically
...
```

Primitive data types

- **Direct interpretation of the machine word**

e.g. in C

type	Size in bytes
char	1
short	2
int	2 or 4
long	4 or 8
float	4
double	8
long double	12 or 16

→ *basic, built-in, standard…*

Composite data types

- **structure (record)**
  - **a sequence of named objects of different types**
  - **selective access**
- **array**
  - **homogeneous collection of elements of same data types**
  - **Indexed access**

→ *built-in constructors*

Records

C

```
struct node {              /* node of a genealogy tree */
   struct node *father;    /* points to father */
   struct node *mother;    /* points to mother */
};
…
node T;
T.father = new node;
…
```

Ada

```
type NODE; -- node of a genealogy tree
type ptr is access NODE;
type NODE is record
    father:ptr; -- points to father
    mother:ptr; --  points to mother
end record;
…
T: node;
T.father := new NODE;
…
```

selector of the record's field

Arrays

- Optimal allocation of data objects (soluzione non ottimizzata)

C

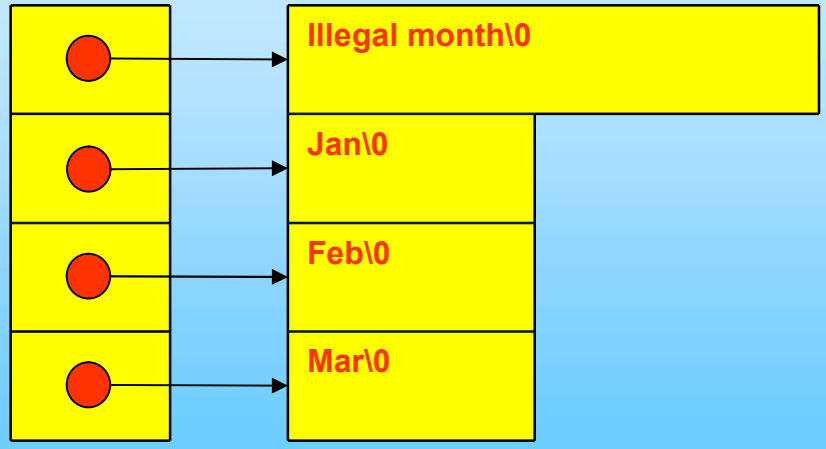
```
...
char aname[][15] = {"Illegal month", "Jan", "Feb", "Mar"};
...
```

Arrays

- **Optimal allocation of data objects** (soluzione ottimizzata)

C

```
…
char *name[] = {"Illegal month", "Jan", "Feb", "Mar"};
…
```

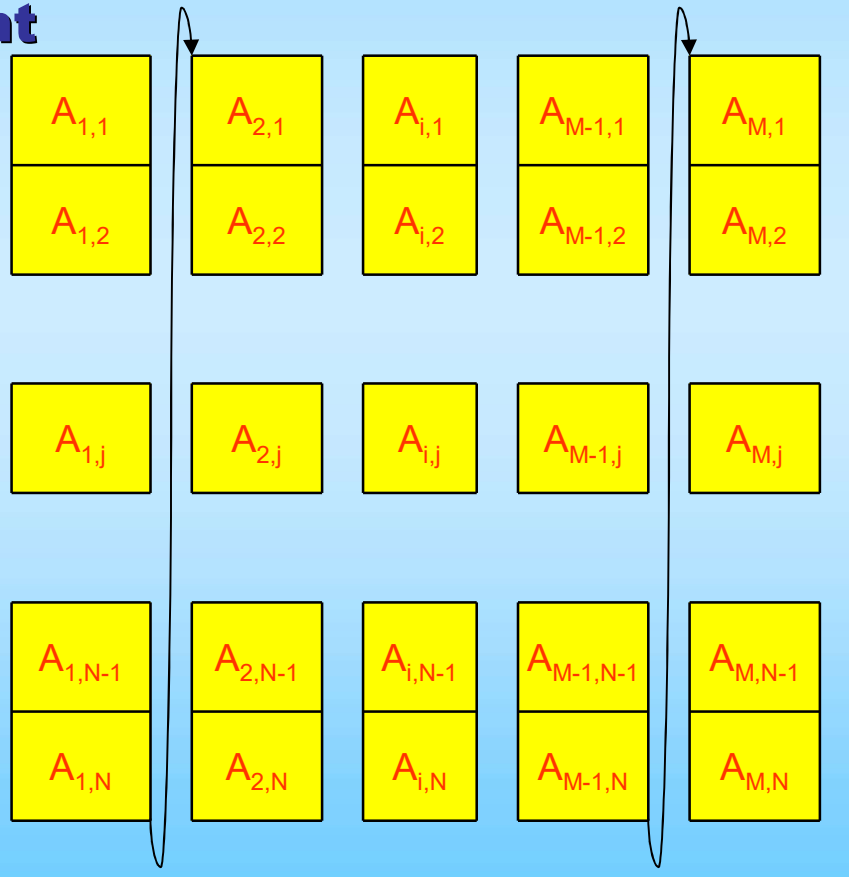


Arrays

- **Access to the array element**

C

```
const int M = 10;
const int N = 20;
...
int A[M][N];
...
print("%d", A[3][9]); /* where is the element? */
...
```

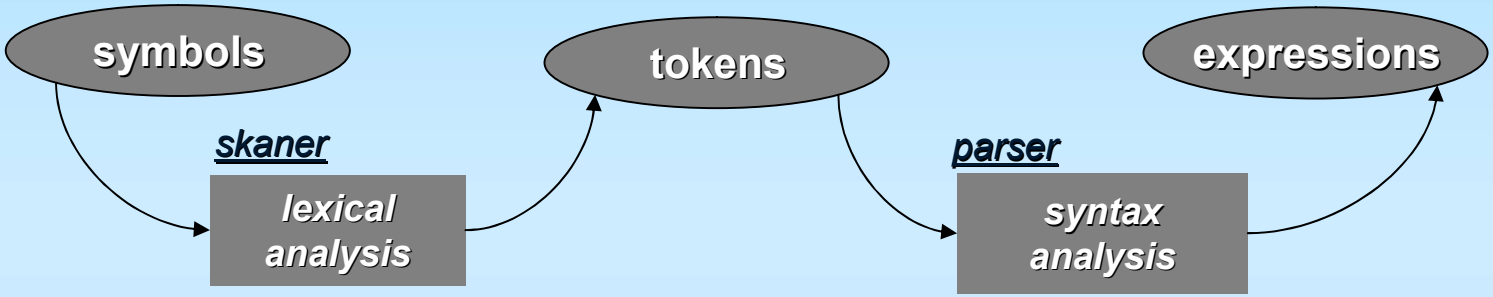


$$address\{A_{i,j}\}=address\{A_{1,1}\}+sizeof(int)\cdot(N\cdot(i-1)+(j-1))$$

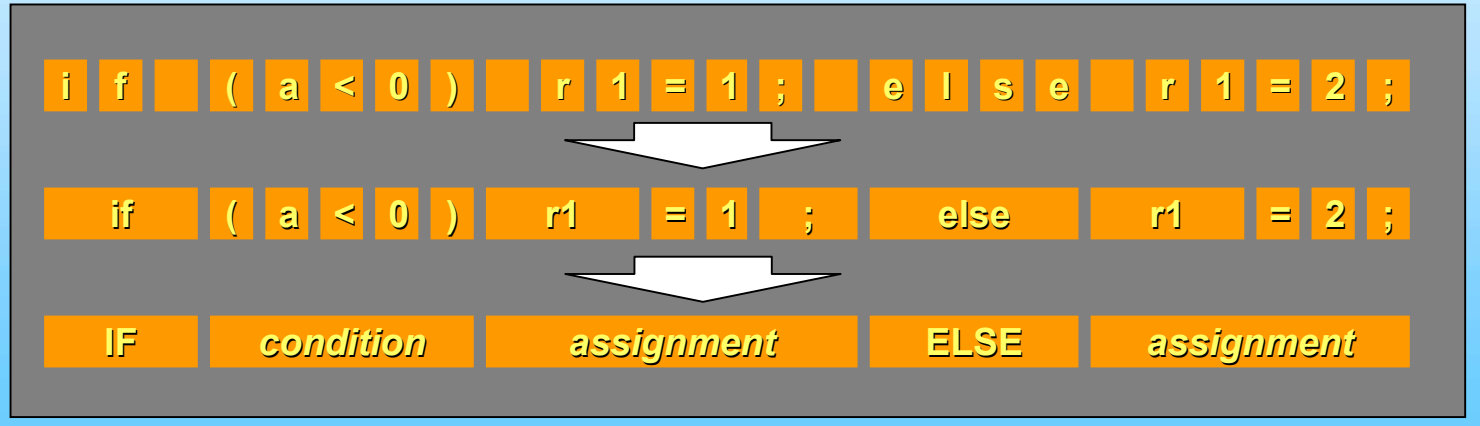
Lexical structures

- **Organization (!)**
  - **primitive data types → composite data types (data structures)**
  - **unconditional/conditional control statements → program control flow structures**
- **Limited namespace**
  - **scoping rules (identifiers visible only to selected statements)**
  - **variables local and global**
- **Optimal memory use:**
  - **variables static and automatic**
  - **name synonyms**
  - **shared data objects**

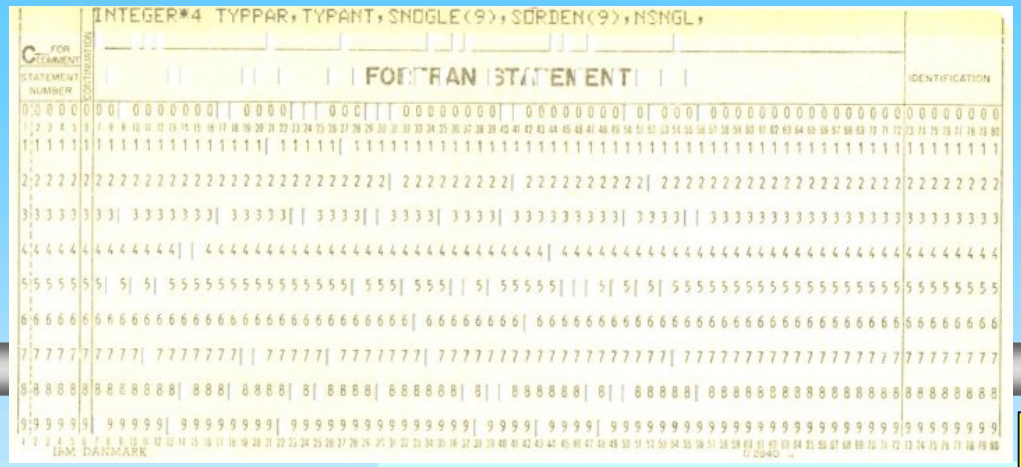
Syntax



if (a>0) r1=0; else r1=2;



Syntax



1. **Format**
   - **fixed (→ punch card)**
   - **free (→ separators, reserved words**
2. **Operator precedence**
3. **Compound statements**

1.

C

```
…
if (a<1) x=a+1; else if (a==1) x=a; else x=a-1;
…
if (a<1) x=a+1; else if (a==1) x=a;
else x=a-1;
…
if (a<1) x=a+1;
else if (a==1) x=a;
else x=a-1;
…
```

2.

C

```
…
x=a+2*x-1; x=a+2*(x-1); x=(a+2)*x-1;
…
a[i]=i++;
…
printf("%d%d\n",++n,power(2,n));
…
```

3.

C

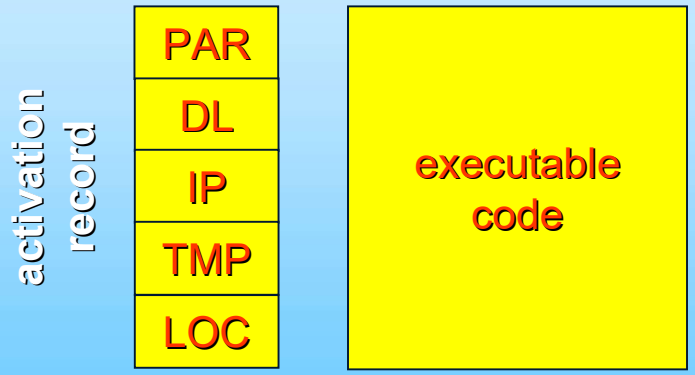
```
…
if (a<1) {x=a+1; a--;}
…
```

Subprogram

- **Procedure, function**
  → procedural abstraction (control flow abstraction)
- **Caller – callee**
  - various parameter values,
  - from many places of the caller,
  - by multiple callees
- **Call/return**
  - the caller must pass arguments to the callee,
  - the callee must know the place where to resume the caller,
  - the callee must restore the caller's state

Call/return

- **Run-time support**:
  - **activation record (AR)**
    - **call parameters (PAR)**
    - **dynamic link (DL)**
    - **return address - instruction pointer (IP)**
    - **procesor/system status register settings (TMP)**
    - **local memory (LOC)**



Calling a subprogram

C

```
void F(int p,q) /* callee*/
      {
       /* ... callee's code ... */
       return ;
      }

void S() /* caller */
      {
       /* ... caller's code ... */
       F(5,102); /* function F call*/
       /* ... caller's code ... */
      }
```

- **Two activation records:**
  S_{ar} for subprogram S,
  F_{ar} for subprogram F

1. Caller S copies machine registers to S_{ar}.tmp
2. S_{ar}.IP ← caller's S return address
3. F_{ar}.PAR[1] ← 5
4. F_{ar}.PAR[2] ← 102
5. F_{ar}.DL ← address S_{ar}
6. Caller S jumps (goto) to the address of the first instruction of the callee's executable code

Returning from a subprogram

C

```
void F(int p,q) /* callee*/
      {
       /* ... callee's code ... */
       return ;
      }

void S() /* caller */
      {
       /* ... caller's code ... */
       F(5,102); /* function F call*/
       /* ... caller's code ... */
      }
```

- **Two activation records:**
  S_{ar} for subprogram S,
  F_{ar} for subprogram F

1. Callee F loads machine registers with F_{ar}.DL → TMP
2. Callee F jumps (goto) to the instruction at the address F_{ar}.DL → IP of the caller's executable code

Structured programming languages

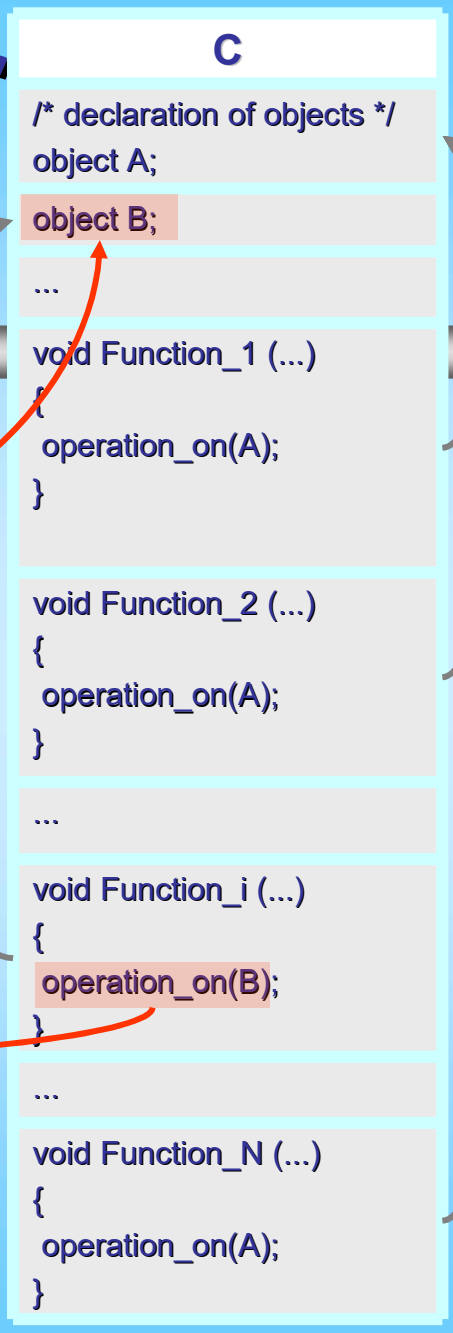
- **The problem of „large programs” discovered in the late fifties for Fortran:**
  - **insufficient control of the use of identifiers in the namespace**
  - **only global scoping rule for the *name* ↔ *object* bindings**
- **How to limit the program namespace without limiting expression power of the language used to implement it?**

→ use the structural syntax rules!

Linear or structur syntax?

- **object visibility**

Should this object be global?



Linear or structur syntax?

- **object visibility**

Should this object be global?

Visibility of object B should be limited!

Local declaration of object B

Object B is not accessible outside of Function_i()

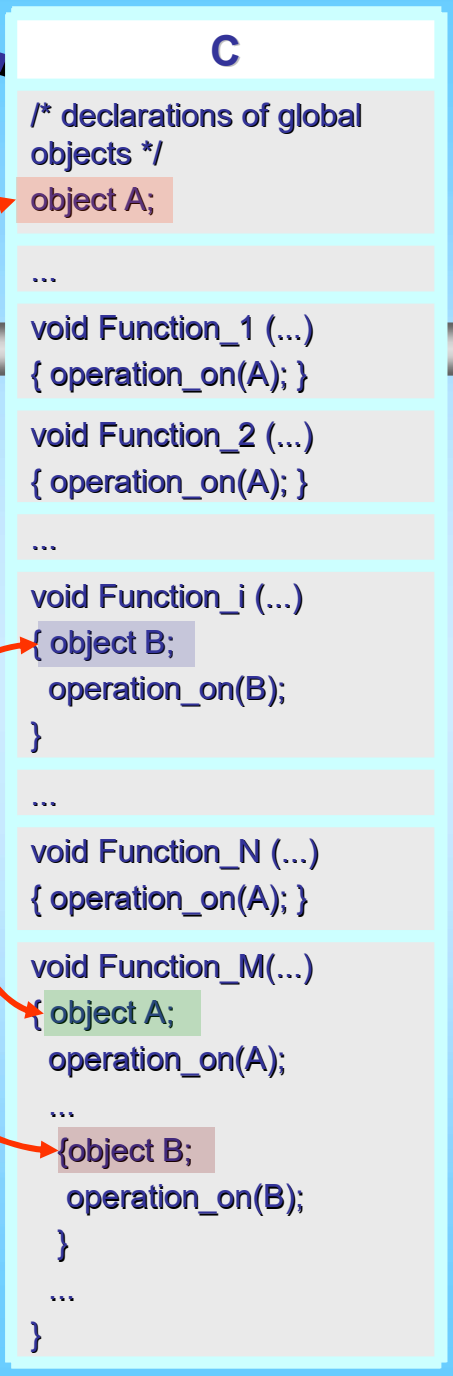
```
C
/* deklaracje objectów */
object A;
object B;
...
void Function_1 (...)
{
 operation_on(A);
}
void Function_2 (...)
{
 operation_on(A);
}
...
void Function_i (...)
{
 object B;
 operation_on(B);
}
...
void Function_N (...)
{
 operation_on(A);
}
```

Linear or structur syntax?

- **object visibility**

different objects of name A

different objects of name B



Linear or structur syntax?

- object visibility
- object lifetime

C

```
/* declarations of global
objects */
object A;

...
void Function_1 (...)
{ operation_on(A); }
void Function_2 (...)
{ operation_on(A); }
...
void Function_i (...)
{ object B;
  operation_on(B);
}
...
void Function_N (...)
{ operation_on(A); }
void Function_M(...)
{ object A;
  operation_on(A);
  ...
   {object B;
    operation_on(B);
   }
  ...
}
```

← the block for A

← ← the blok for B

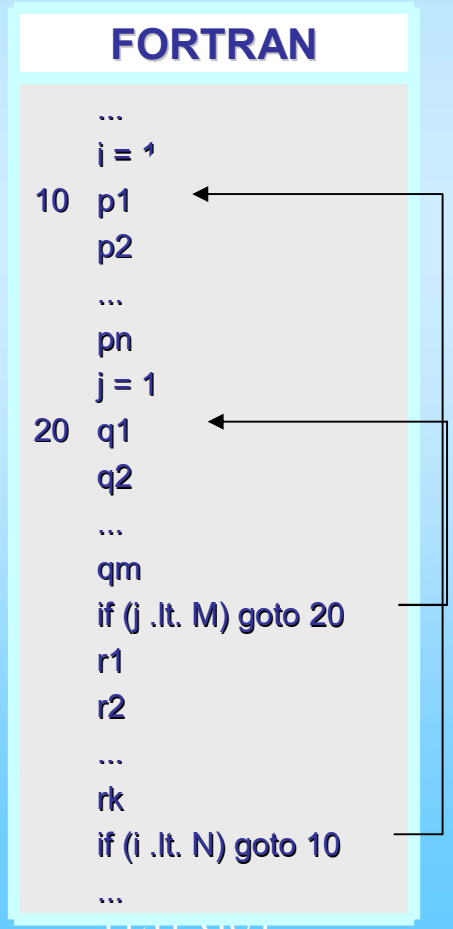
Block-structured programming languages

- **Compound statements**
  - hierarchy of operations
  - free format
- **Nested scopes**
  - full control of object visibility
  - multiple use identifiers
  - optimal memory use
- **Structural programming**
  - single-entry-single-exit structures
  - linear control flow

Compound statements

- **Linear organization**

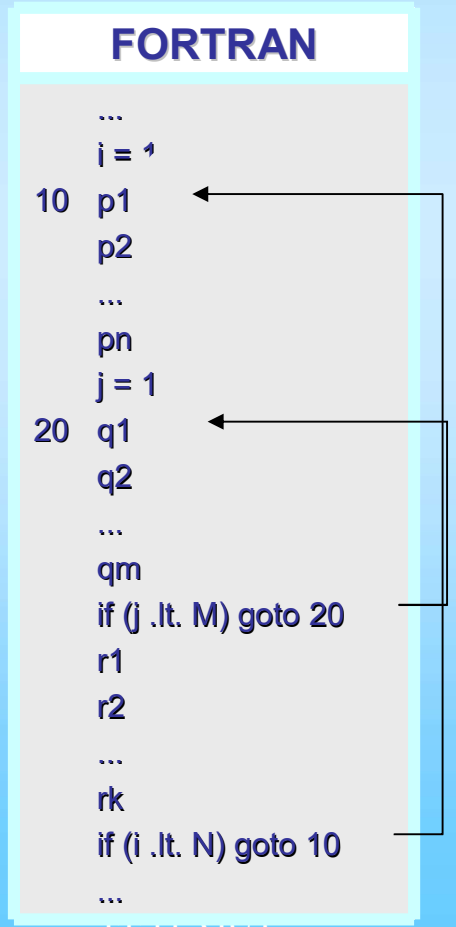
Penso che l'organizzazione lineare sia tipo quella che ho fatto in assembly

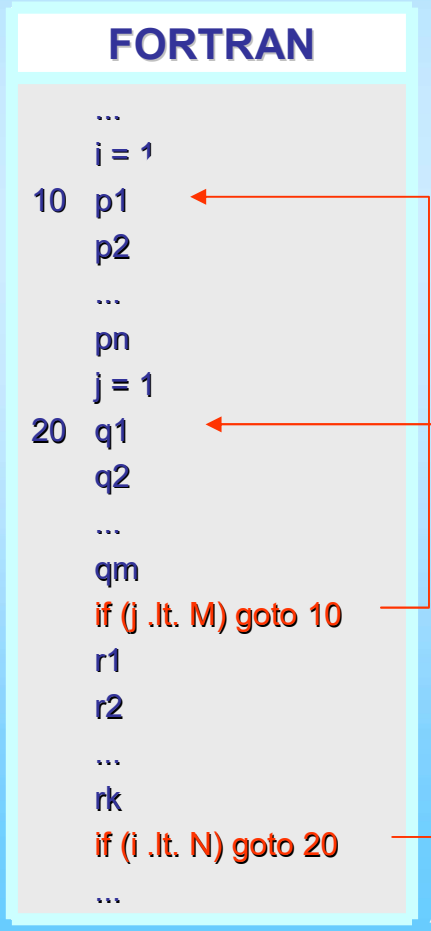


Compound statements

- **Linear organization**

?





Compound statements

- **Block organization**

C

```
...
i=1;
do
{ p1; p2; ... pn;
  j=1; do {q1; q2; ... qm;} while ( j < M) ;
  pn+1; pn+2; ... pk;
} while ( i < N );
...
```

PASCAL

```
...
i := 1;
repeat
  p1; p2; ... pn;
  j := 1; repat q1; q2; ... qm until ( j >= M ) ;
  pn+1; pn+2; ... pk
until ( i >= N );
...
```

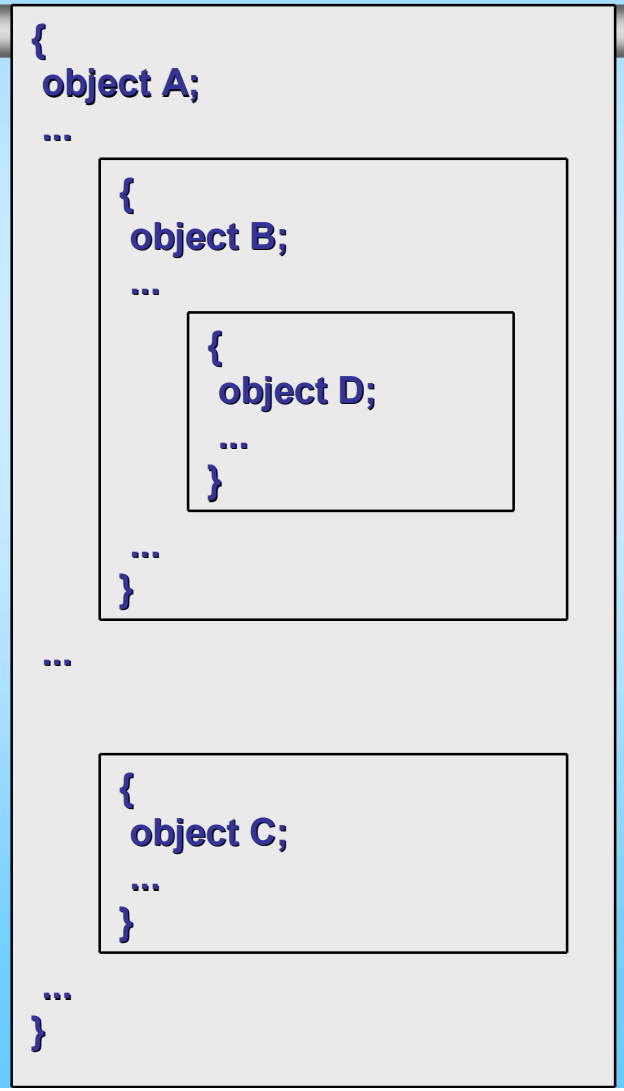
Nested scopes

- **Linear syntax**
  - **Only two visibility scopes (global and local)**
- **Block structured syntax**
  - **Arbitrarily many visibility scopes**
- **The block:**

  { declarations; statements; }

Nested scopes

- scoping rules



← visible A

← visible A and B

← visible A, B and D

← visible A and B

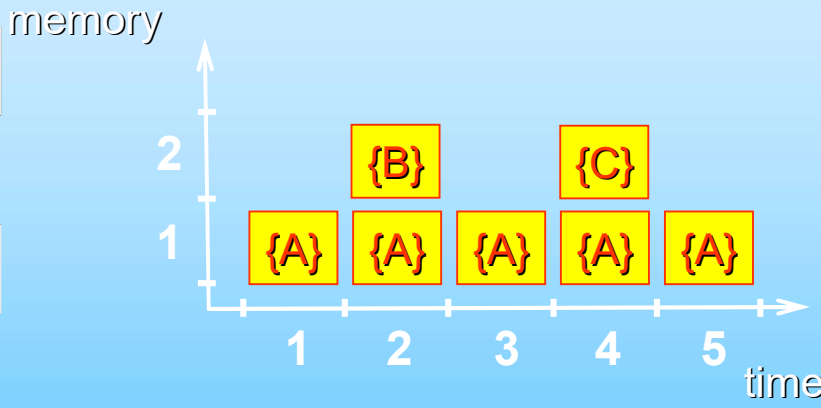
← What is visible here?

← visible A and C

←what is visible here?

Nested scopes

- **Object's lifetime**

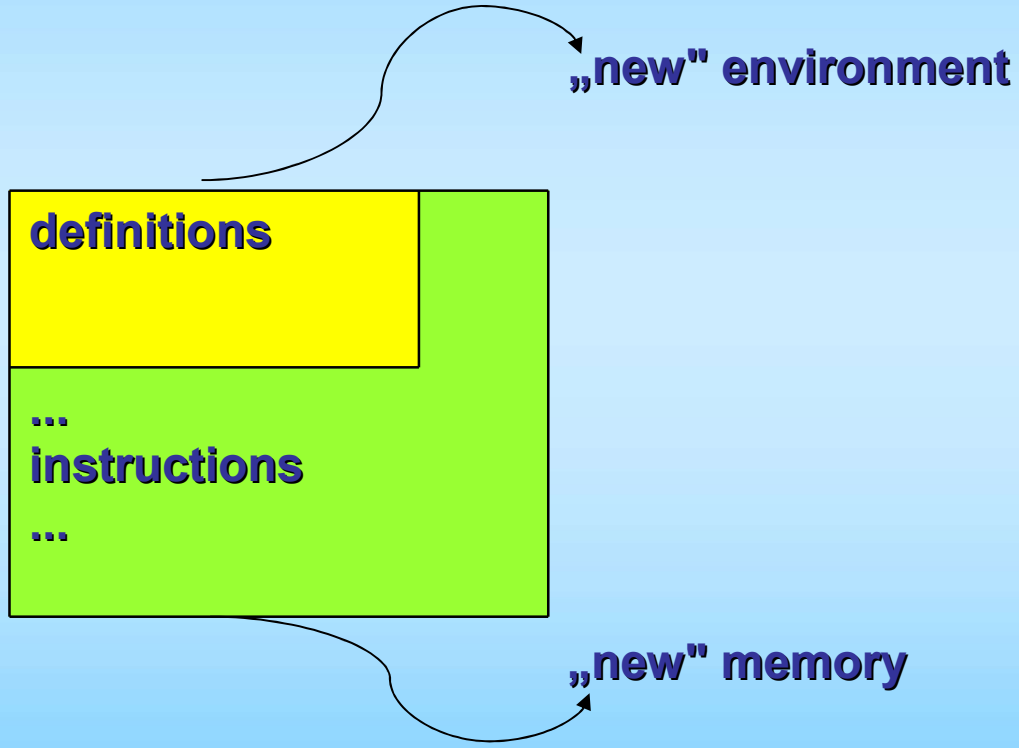



```
{
 object A;
 ...
    {
     object B;
     ...
    }
 ...
    {
     object C;
     ...
    }
 ...
}
```

1. exists A
2. exist A and B
3. exists A
4. exist A and C
5. exists A

→ *the LIFO (stack) principle for structural languages*

Block



Block

- **Basic terminology**

syntax	*run-time*
binding:	**memory allocation:**
assigning an identifier to the data object	**requesting memory for the data object**
(visibility) scope:	**lifetime:**
portion of the program text where the particular binding takes place	**period of time (program execution) from allocation to reallocation of the same memory fragment**

Procedure call environment

Question:

- Who decides what bindings to see?

Answer:

1. callee → static model (bindings of the callee's definition environment apply)
2. caller → dynamic model (bindings of the caller's definition environment apply)

Procedure call environment

- **Static model**

C

```
int A, B, C;    /* global objects */
...
int INC(void)
{
  return (A+1);
}
...
main()
{ A=1;
  { int A; /* local object */
    A=0;
    B=INC();
  }
 C=INC();
 }
```

B = 2

C = 2

Procedure call environment

- **Dynamic model**

?

```
int A, B, C;    /* global objects */
...
int INC(void)
{
  return (A+1);
}
...
main()
{ A=1;
  { int A; /* local object */
    A=0;
    B=INC();
  }
 C=INC();
 }
```

B = 1

C = 2

→ *The static model is a standard for structural programming languages*

Parameter passing techniques

- **Pass-by-value:**
  - formal parameter name is bound to the value of the actual parameter
  - each occurrence of the name of a formal parameter refers to a local copy of the actual parameter

The caller and callee have two **independent** variables with the same value. If the callee modifies the parameter variable, the effect is not visible to the caller.

C

```
#include <stdio.h>
int A=0;
void F (int B,char C)
{
  B=A-1;
  printf("%c=%d\n",C,B);
}
void E (int *B, char C)
{
  *B=A+1;
  printf("%c=%d\n",C,*B);
}
void main(void)
{ F(A,'B');
  E(&A,'B');
}
```

PASCAL

```
program MAIN (output);
  var A: integer;
  procedure F (B: integer; C: char);
  begin
    B:=A-1;
    writeln(C,B)
  end
  procedure E(B:^integer; C: char);
  begin
    B^:=A+1;
    writeln(C,B^)
  end
begin
  F(A,'B');
  E(address(A),'B')
end
```

Parameter passing techniques

- **Pass-by-reference (variable, address):**
  - the name of the formal parameter is bound to the address of the actual parameter
  - each occurrence of the name of a formal parameter is a reference to where the current parameter is stored

The caller and the callee use **the same variable** for the parameter. If the callee modifies the parameter variable, the effect is visible to the caller's variable.

C++	PASCAL

```
#include <stdio.h>
int A=0;
void H (int &B, char C)
{
 B = A-1;
 printf("%c=%d\n",C,B)
}
void E (int *B, char C)
{
 *B=A+1;
 printf("%c=%d\n",C,*B
}
void main(void)
{ H(A,'B');
 E(&A,'B');
}
```

```
program MAIN (output);
  var A: integer;
  procedure H (var B: integer; C: char);
  begin
    B:=A-1;
    writeln(C,B)
  end
  procedure E(B:^integer; C: char);
  begin
    B^:=A+1;
    writeln(C,B^)
  end
begin
  H(A,'B');
  E(address(A),'B')
end
```

Parameter passing techniques

- **Pass-by-reference (variable, address):**
  - the name of the formal parameter is bound to the address of the parameter argument
  - each occurrence of the name of a formal parameter is a reference to where the actual parameter is stored

→ Many modern languages prefer the actual object data to be stored separately (usually, on the heap), and only "references" to it are ever held in variables and passed as parameters

C++

```
#include <stdio.h>
int A=0;
void H (int &B, char C)
{
 B = A-1;
 printf("%c=%d\n",C,B)
}
void E (int *B, char C)
{
 *B=A+1;
 printf("%c=%d\n",C,*B
}
void main(void)
{ H(A,'B');
 E(&A,'B');
}
```

PASCAL

```
program MAIN (output);
  var A: integer;
  procedure H (var B: integer; C: char);
  begin
    B:=A-1;
    writeln(C,B)
  end
  procedure E(B:^integer; C: char);
  begin
    B^:=A+1;
    writeln(C,B^)
  end
begin
  H(A,'B');
  E(address(A),'B')
end
```

Parameter passing techniques

- **Pass-by-name**
  - the name of the formal parameter is associated with a special system procedure for analyzing the textual structure of the actual parameter
  - each occurrence of the formal parameter name is textually substituted with the name of the actual parameter

ALGOL-60

```
real procedure Sum (k, l, u, ak);
value l, u:
integer k, l, u; real ak;
begin
   real S; S := 0;
   for k := 1 step 1 until n do
      S ;= S + ak;
Sum := S;
end;
…
x := Sum (i, 1, n, V[i]);
…
```

$$x = \sum_{i=1}^{n} V_i$$

```
begin
   real S; S := 0;
   for i := 1 step 1 until n do
      S := S + V[i];
   x := S;
end;
```

the evaluation method similar to that of C preprocessor macros

Parameter passing techniques

- **Pass-by-name**

  …is a dangerous solution!

ALGOL-60

```
procedure Swap (x, y);
integer x, y;
begin
  integer t;
  t := x;
  x := y;
  y := t;
end;
…
Swap (A[i],i) ;
Swap (i,A[i]) ;
…
```

```
begin
  integer t;
  t := A[i];
  A[i] := i;
  i := t;
end;
```

```
begin
  integer t;
  t := i;
  i := A[i];
  A[i] := t;
end;
```

i=1

A[1]=27

```
t := A[1]=27;
A[1] := 1;
i := 27;
```

```
t := 1;
i := A[1]=27;
A[27] := 1;
```

≠ !?

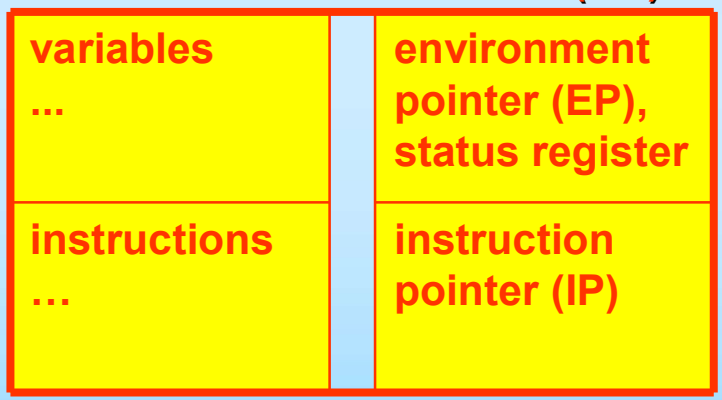
→ *superseded by other, safer techniques (pass-by-reference and lambda functions)*

Implementation of block-structured languages

program text (static representation)

procedure:

program code (run-time representation)



context of use

execution environment, machine status

activation place

resumption point

local context:

- local variables
- actual parameters

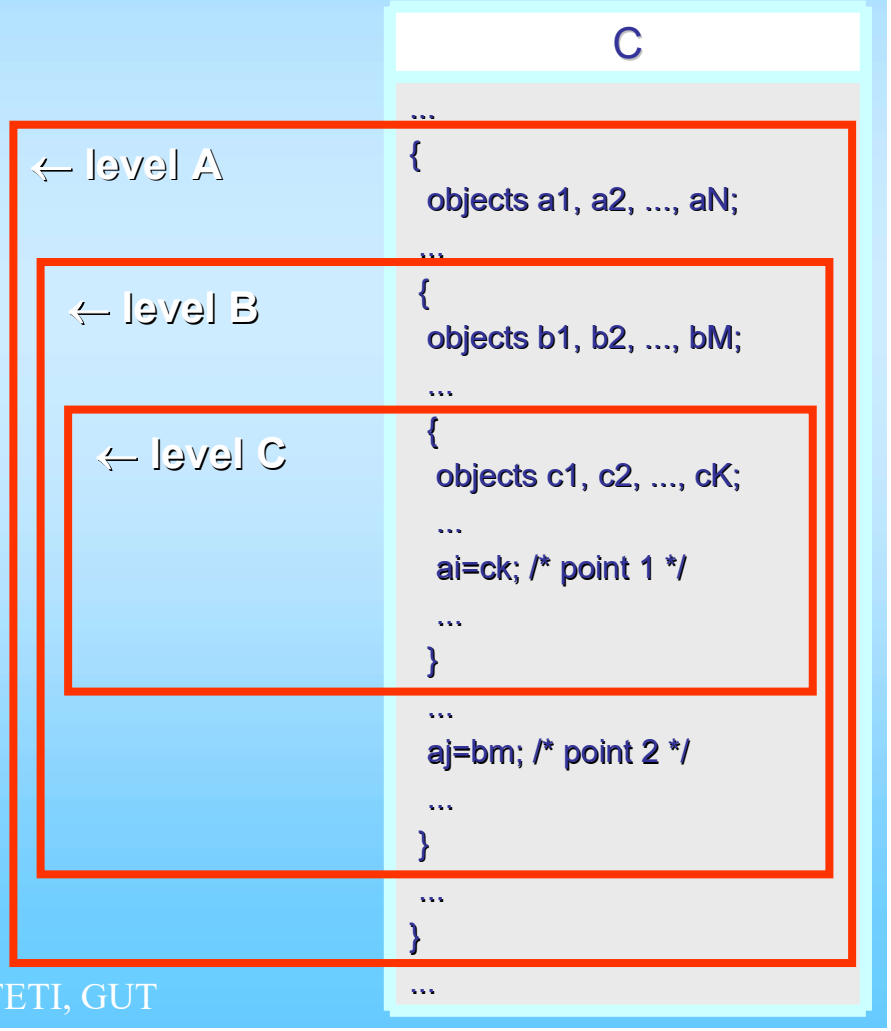
external contexts:

- global variables
- variables of outer blocks

→ *direct access in AR via pointers*

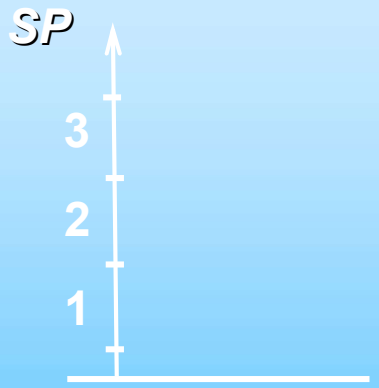
Implementation of block-structured languages

- **Program stack (run-time)**



Implementation of block-structured languages

- **Program stack (run-time)**



C

```
...
{
 objects a1, a2, ..., aN;
 ...
 {
  objects b1, b2, ..., bM;
  ...
  {
   objects c1, c2, ..., cK;
   ...
   ai=ck; /* point 1 */
   ...
  }
  ...
  aj=bm; /* point 2 */
  ...
 }
 ...
}
...
```

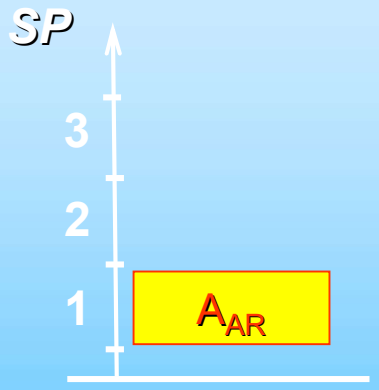
← level A

← level B

← level C

Implementation of block-structured languages

- **Program stack (run-time)**



C

```
...
{
 objects a1, a2, ..., aN;
 ...
 {
  objects b1, b2, ..., bM;
  ...
  {
   objects c1, c2, ..., cK;
   ...
   ai=ck; /* point 1 */
   ...
  }
  ...
  aj=bm; /* point 2 */
  ...
 }
 ...
}
...
```

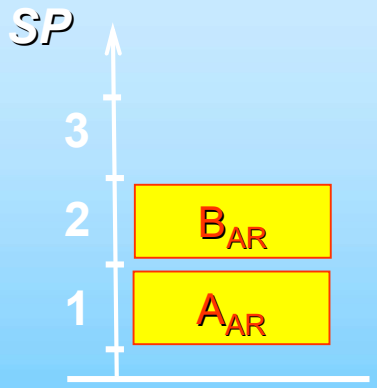
← level A

← level B

← level C

Implementation of block-structured languages

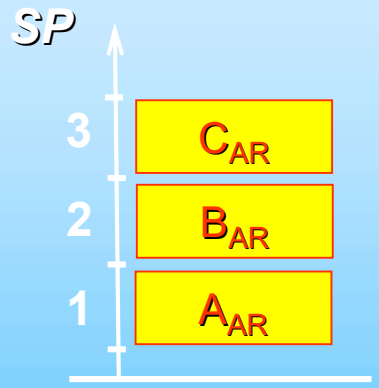
- **Program stack (run-time)**




```
C
...
{                          ← level A
  objects a1, a2, ..., aN;
  ...
  {                        ← level B
    objects b1, b2, ..., bM;
    ...
    {                      ← level C
      objects c1, c2, ..., cK;
      ...
      ai=ck; /* point 1 */
      ...
    }
    ...
    aj=bm; /* point 2 */
    ...
  }
  ...
}
...
```

Implementation of block-structured languages

- **Program stack (run-time)**



C

```
...
{
  objects a1, a2, ..., aN;
  ...
  {
  objects b1, b2, ..., bM;
  ...
  {
  objects c1, c2, ..., cK;
  ...
  ai=ck; /* point 1 */
  ...
  }
  ...
  aj=bm; /* point 2 */
  ...
  }
  ...
}
...
```

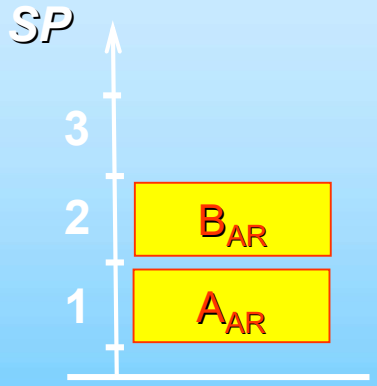
← level A

← level B

← level C

Implementation of block-structured languages

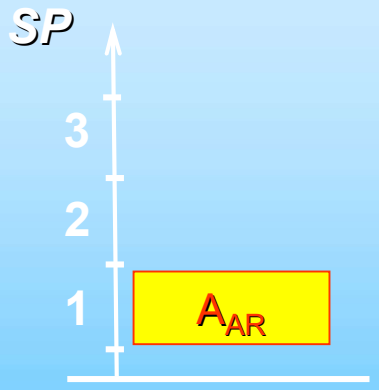
- **Program stack (run-time)**




```
C
...
{                          ← level A
  objects a1, a2, ..., aN;
  ...
  {                        ← level B
    objects b1, b2, ..., bM;
    ...
    {                      ← level C
      objects c1, c2, ..., cK;
      ...
      ai=ck; /* point 1 */
      ...
    }
    ...
    aj=bm; /* point 2 */
    ...
  }
  ...
}
...
```

Implementation of block-structured languages

- **Program stack (run-time)**



C

```
...
{
 objects a1, a2, ..., aN;
 ...
 {
 objects b1, b2, ..., bM;
 ...
 {
  objects c1, c2, ..., cK;
  ...
  ai=ck; /* point 1 */
  ...
 }
 ...
 aj=bm; /* point 2 */
 ...
 }
 ...
}
...
```

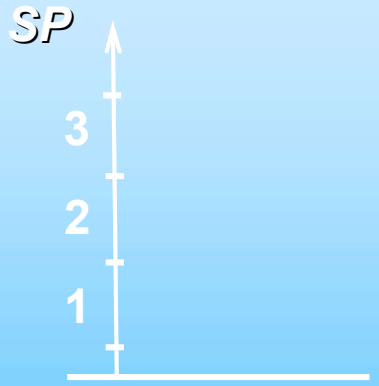
← level A

← level B

← level C

Implementation of block-structured languages

- **Program stack (run-time)**



C

```
...
{
 objects a1, a2, ..., aN;
 ...
 {
  objects b1, b2, ..., bM;
  ...
  {
   objects c1, c2, ..., cK;
   ...
   ai=ck; /* point 1 */
   ...
  }
  ...
  aj=bm; /* point 2 */
  ...
 }
 ...
}
...
```

← level A

← level B

← level C

Implementation of block-structured languages

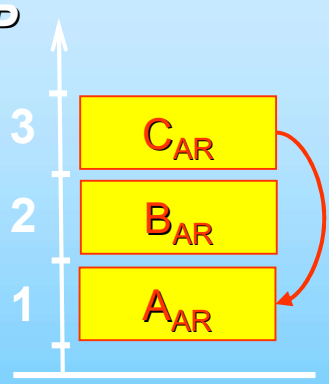
- **Program stack**
  - static links

level A = 1

level B = 2

level C = 3

static distance between nesting levels



point 1: 3-1 = 2

```
C
...
{
 objects a1, a2, ..., aN;
 ...
 {
  objects b1, b2, ..., bM;
  ...
  {
   objects c1, c2, ..., cK;
   ...
   ai=ck; /* point 1 */
   ...
  }
  ...
  aj=bm; /* point 2 */
  ...
 }
 ...
}
...
```

← level A

← level B

← level C

Implementation of block-structured languages

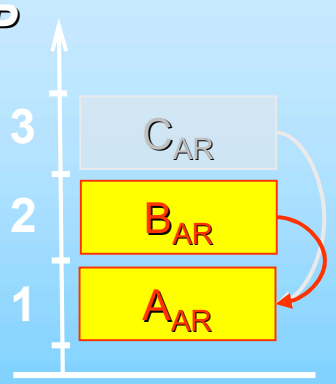
- **Program stack**
  - static links

level A = 1

level B = 2

level C = 3

static distance between nesting levels



point 2: 2-1 = 1

C

```
...
{
  objects a1, a2, ..., aN;
  ...
  {
   objects b1, b2, ..., bM;
   ...
   {
    objects c1, c2, ..., cK;
    ...
    ai=ck; /* point 1 */
    ...
   }
   ...
   aj=bm; /* point 2 */
   ...
  }
  ...
}
...
```

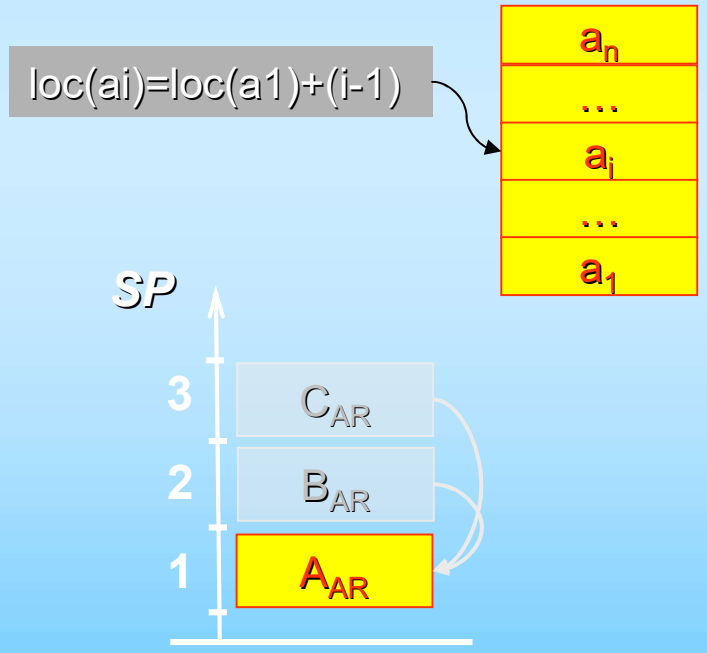
← level A

← level B

← level C

Implementation of block-structured languages

- **Program stack**
  - static links



C

```
...
{
 objects a1, a2, ..., aN;
 ...
 {
  objects b1, b2, ..., bM;
  ...
  {
   objects c1, c2, ..., cK;
   ...
   ai=ck; /* point 1 */
   ...
  }
  ...
  aj=bm; /* point 2 */
  ...
 }
 ...
}
...
```

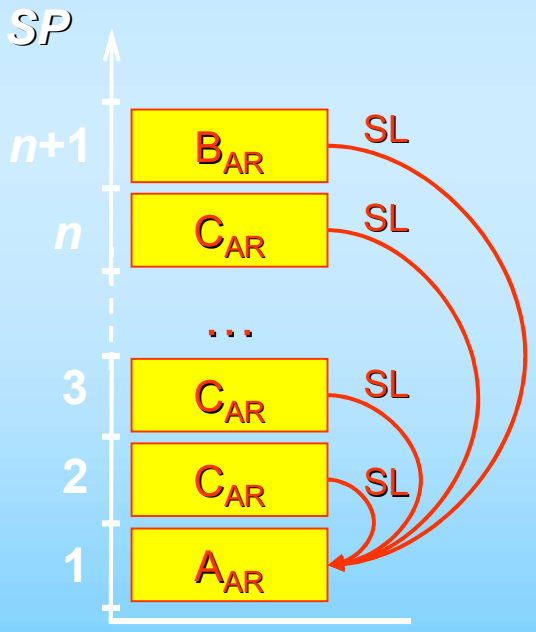
← level A

← level B

← level C

Implementation of block-structured languages

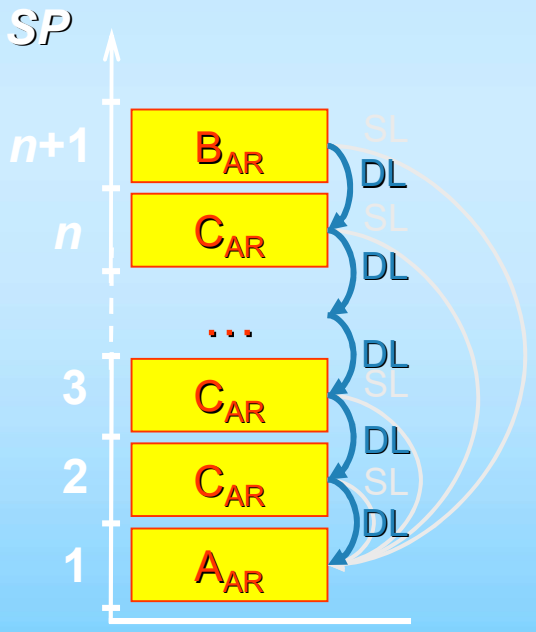
- **Program stack**
  - dynamic links




```
                                        C
                                        ...
← level A (1)                           objects a1, a2, ..., aN;
                                        void B(void)
  ← level B (2)                         { objects b1, b2, ..., bM;
                                          ...
                                          ai=...; /* access to ai */
                                          ...
                                        }
                                        void C(void)
  ← level C (2)                         { objects c1, c2, ..., cK;
                                          ...
                                          if (...) C();
                                          ...
                                          B();
                                          ...
                                        }

                                        void main(void)
                                        { C();
                                        }
```

Implementation of block-structured languages

- **Program stack**
  - dynamic links



SL ≠ DL

C

```
...
objects a1, a2, ..., aN;
void B(void)
{ objects b1, b2, ..., bM;
  ...
  ai=...; /* access to ai */
  ...
}
void C(void)
 { objects c1, c2, ..., cK;
   ...
   if (...) C();
   ...
   B();
   ...
}

void main(void)
{ C();
}
```

← level A (1)

← level B (2)

← level C (2)

Run-time procedure management

SP: points to the top of the runtime stack (of activation records)

EP: points to the activation record of the currently active environment (local variables)

IP: points to the resumption address of the procedure

SL: points to the activation record of the surrounding environment (visible variables)

DL: points to the caller's activation record

Procedure call (activation)

caller (A) → callee (B)

1. Save the caller's state in its activation record (A_{ra}):
   A_{ra}.IP ← resumption address upon return
   A_{ra}.tmp ← machine registers to be restored

2. Create the callee's activation record (B_{ra}) on top of the run-time stack and load it with the relevant information:
   B_{ra}.loc ← actual parameters of the call
   B_{ra}.SL ← static distance
   B_{ra}.DL ← address of caller's activation record rekordu A_{ra} (the current EP value

3. Set the currently active environment (B_{ra}):
   EP ← SP (the bottom byte address of B_{ra})

4. Set the new top of the run-time stack:
   SP ← size_of(B_{ra})

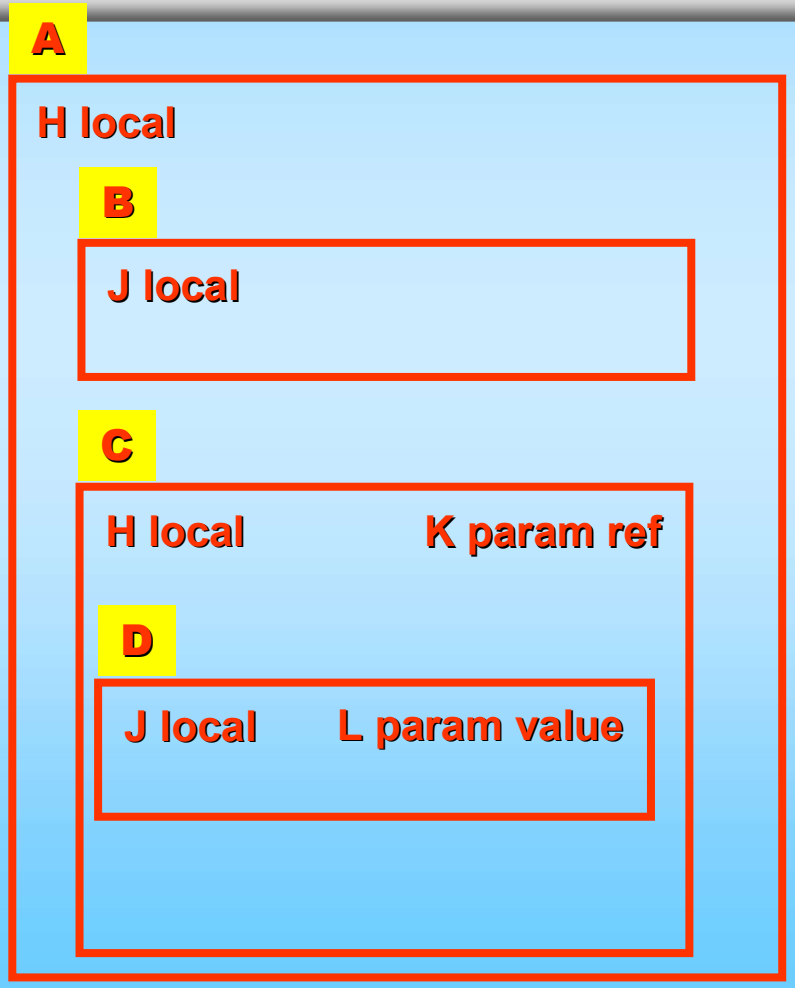
5. goto entry(B)

Return from a procedure call (resumption)

callee (B) → caller (A)

1. Set the previously active environment as the current one:
   $EP \leftarrow B_{ra}.DL$
2. Delete the top activation record (B_{ra}) friom the run-time stack:
   $SP \leftarrow SP - size\_of(B_{ra})$
3. Restore machine registers from before the call:
   $registers \leftarrow A_{ra}.tmp$
4. goto $A_{ra}.IP$ 

Big example



$A \rightarrow C(H_A)$

$C \rightarrow D(H_D)$

$D \rightarrow D(J_D)$

Big example

A → C

1. M[EP].IP ← return address
2. M[SP].PAR[1] ← address(H_A)

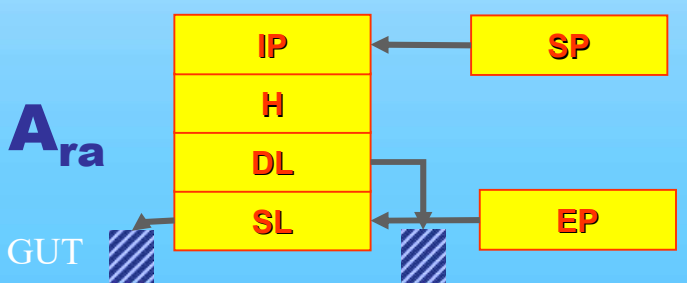
   M[SP].DL ← EP

   sd ← snl(use C)-snl(decl C) = 1-1 = 0

   M[SP].SL ← EP
3. EP ← SP

   SP ← SP+size(C_{ra})

   goto entry(C)



Big example

A → C

1. M[EP].IP ← return address
2. M[SP].PAR[1] ← address(H_A)

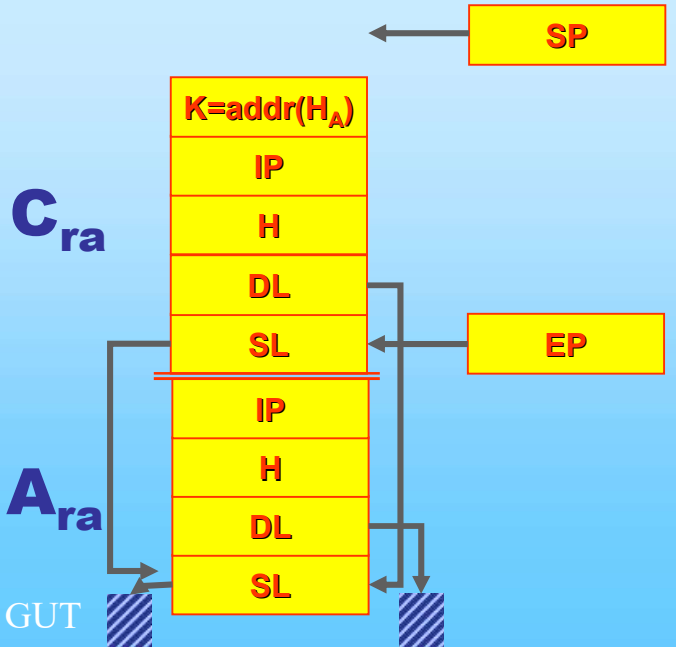
   M[SP].DL ← EP

   sd ← snl(use C)-snl(decl C) = 1-1 = 0

   M[SP].SL ← EP
3. EP ← SP

   SP ← SP+size(C_{ra})

   goto entry(C)



Big example

C → D

1. M[EP].IP ← return address
2. M[SP].PAR[1] ← address(H_C)

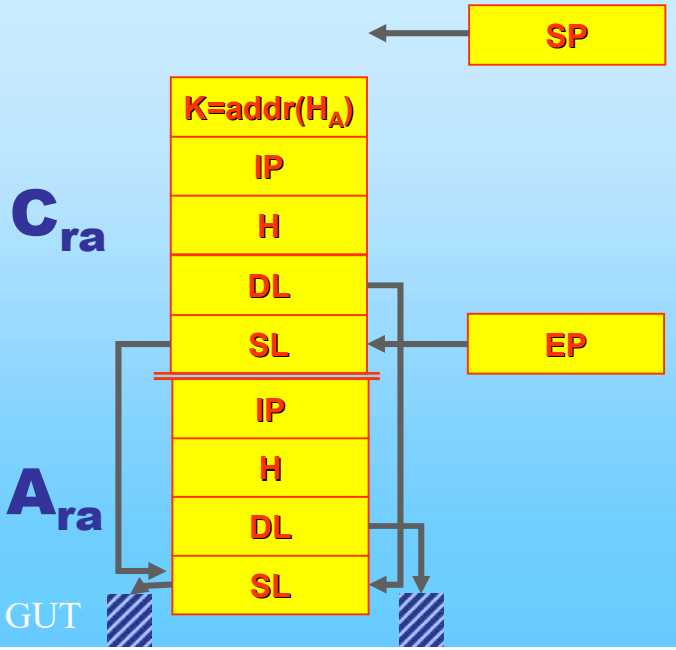
   M[SP].DL ← EP

   sd ← snl(use D)-snl(decl D) = 2-2 = 0

   M[SP].SL ← EP
3. EP ← SP

   SP ← SP+size(D_{ra})

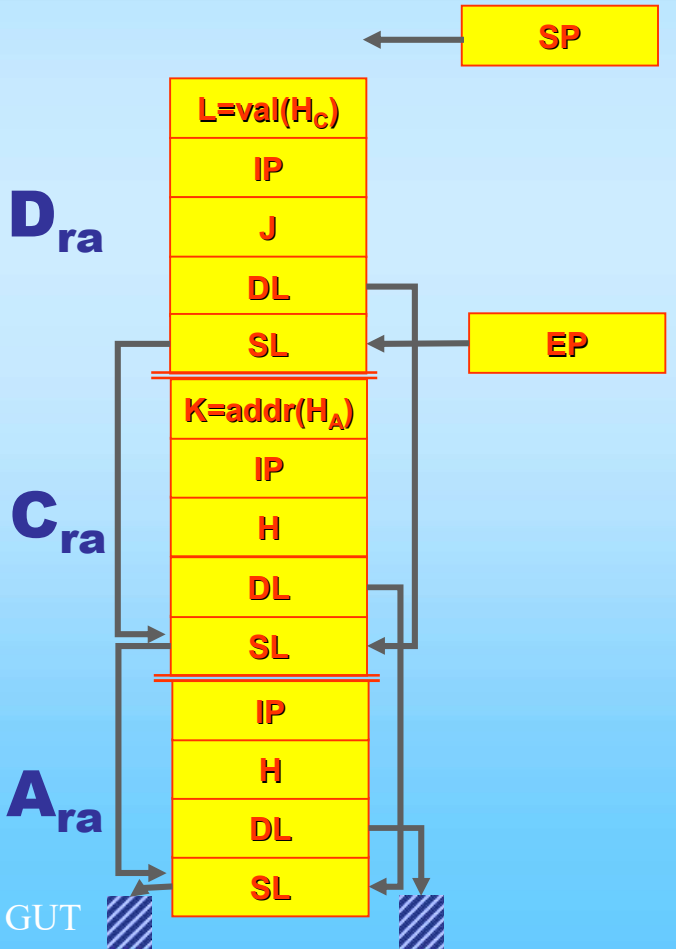
   goto entry(D)



Big example

C → D

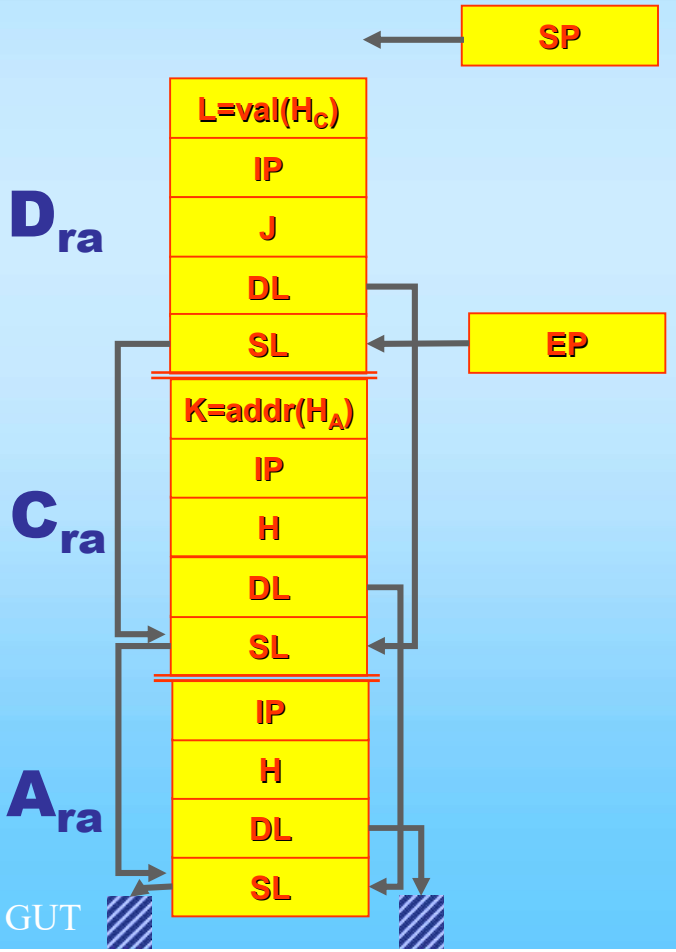
1. M[EP].IP ← return address
2. M[SP].PAR[1] ← address(H_C)
   M[SP].DL ← EP
   sd ← snl(use D)-snl(decl D) = 2-2 = 0
   M[SP].SL ← EP
3. EP ← SP
   SP ← SP+size(D_{ra})
   goto entry(D)



Big example

D → D

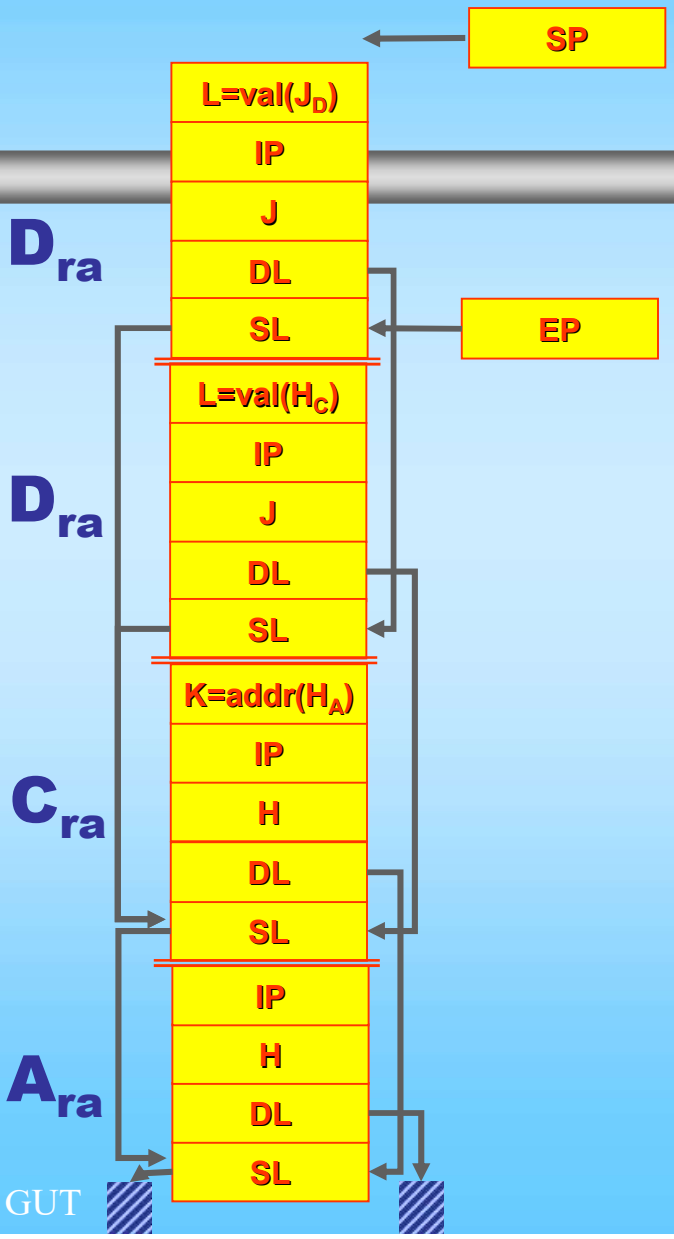
1. M[EP].IP ← return address
2. M[SP].PAR[1] ← address(J_D)
   M[SP].DL ← EP
   sd ← snl(use D)-snl(deck D) = 3-2 = 1
   EP ← M[EP].SL
   M[SP].SL ← EP
3. EP ← SP
   SP ← SP+size(D_{ra})
   goto entry(D)



Big example

D → D

1. M[EP].IP ← return address
2. M[SP].PAR[1] ← address(J_D)
   M[SP].DL ← EP
   sd ← snl(use D)-snl(decl D) = 3-2 = 1
   EP ← M[EP].SL
   M[SP].SL ← EP
3. EP ← SP
   SP ← SP+size(D_{ra})
   goto entry(D)



Big example

D → B

1. M[EP].IP ← return address
2. M[SP].DL ← EP

   sd ← snl(use B)-snl(decl B) = 3-1 = 2

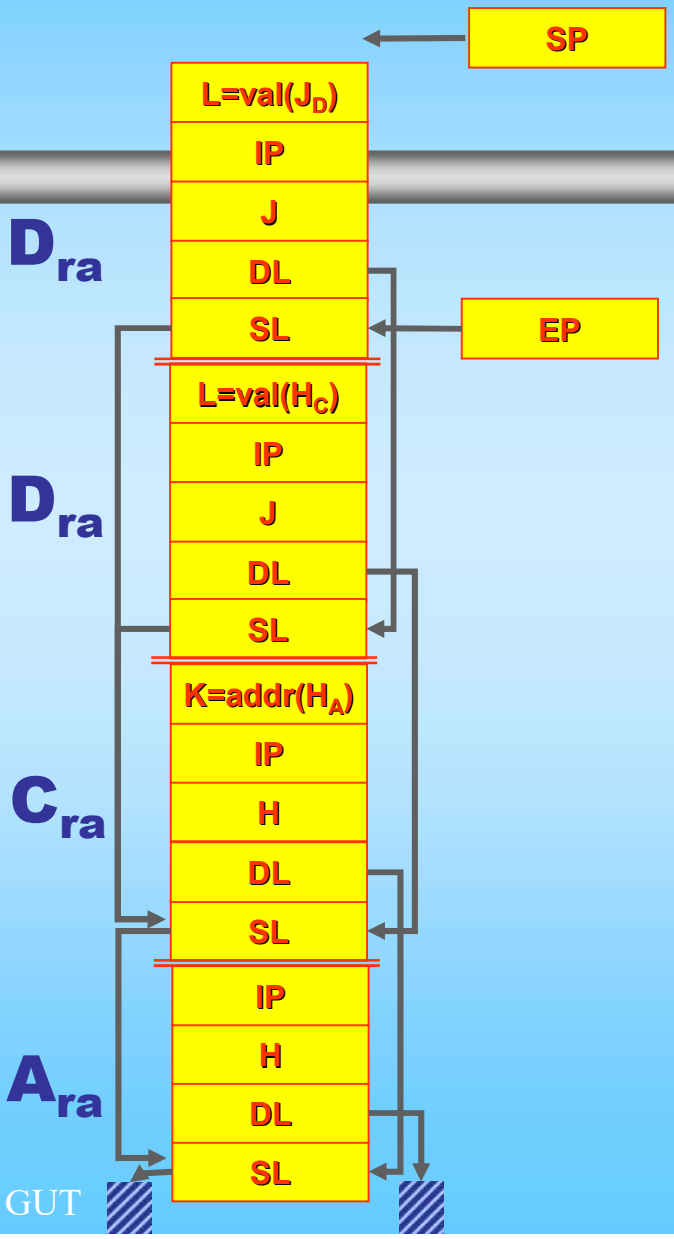
   EP ← M[EP].SL

   EP ← M[EP].SL

   M[SP].SL ← EP
3. EP ← SP

   SP ← SP+size(B_{ra})

   goto entry(B)



Big example

D → B

1. M[EP].IP ← return address
2. M[SP].DL ← EP

   sd ← snl(use B)-snl(decl B) = 3-1 = 2

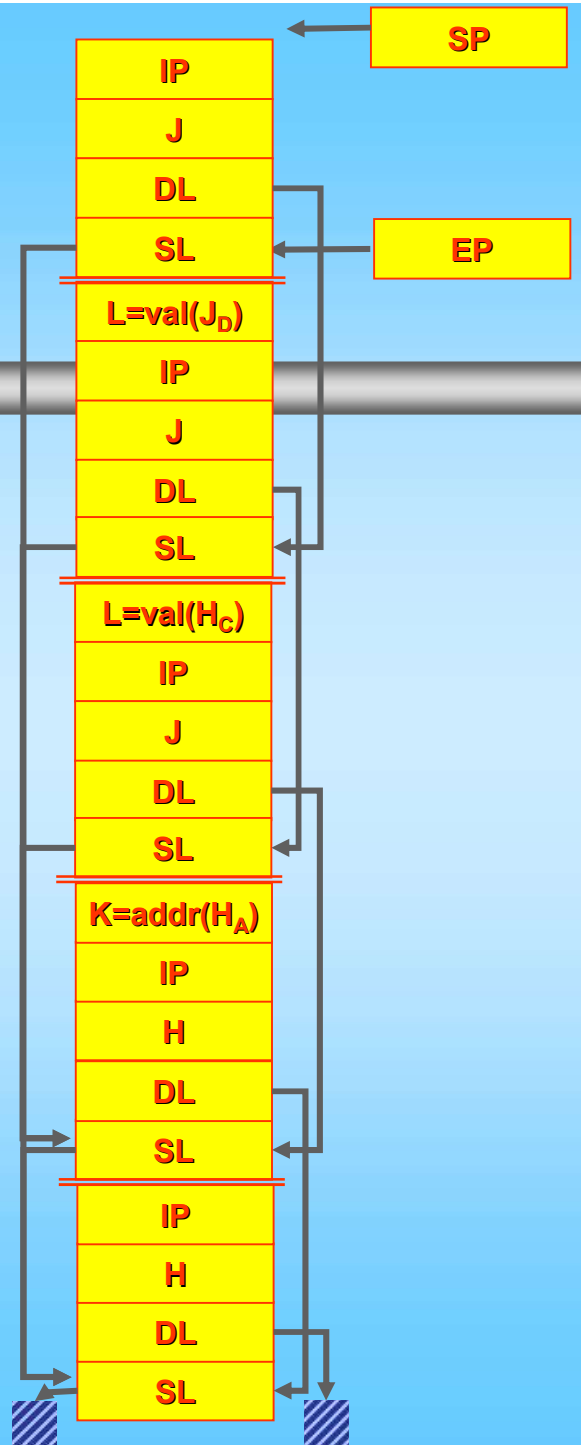
   EP ← M[EP].SL

   EP ← M[EP].SL

   M[SP].SL ← EP
3. EP ← SP

   SP ← SP+size(B_{ra})

   goto entry(B)



Functional parameter

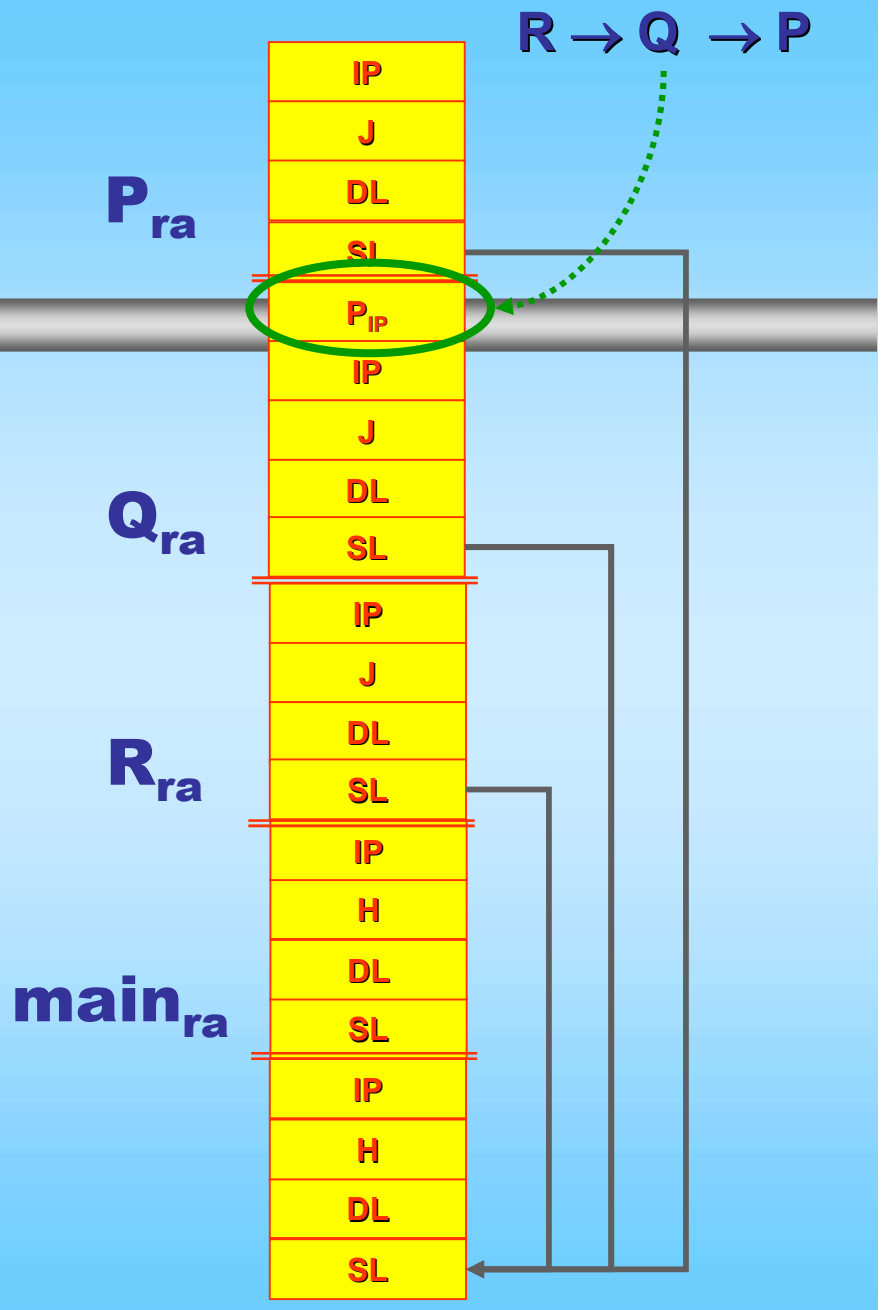
Linear syntax:

function pointer type

execute the pointed function

C

```
...
object A;
...
void P(void)
{ operation_on(A); }
void T(void)
{ operation_on(A); }
typedef void (*proc)(void);
void Q(proc f)
{
 (*f)();
}
void R(void)
{
 Q(P);
 Q(T);
}
main()
{
 R();
}
```

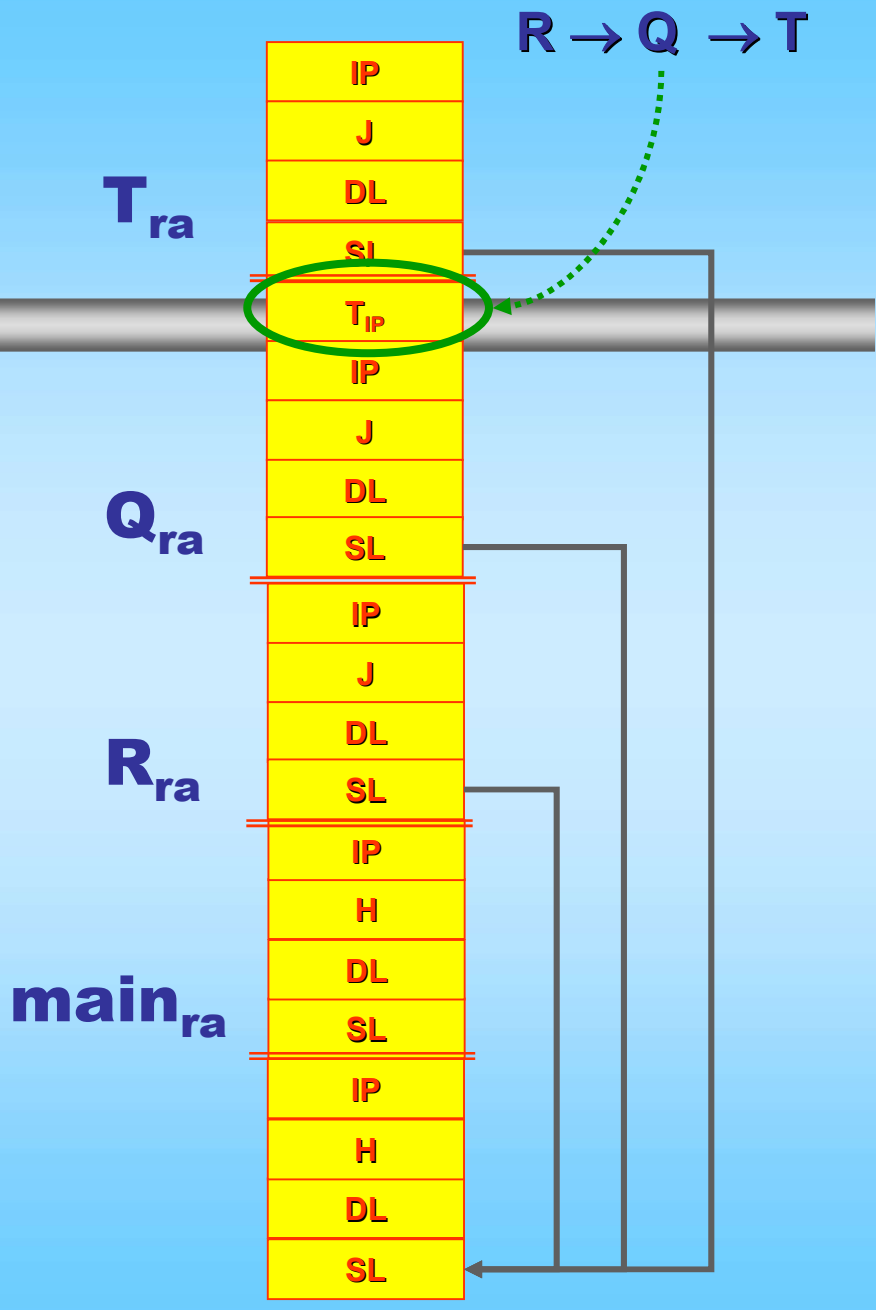


Functional parameter

Linear syntax:

C

```
...
object A;
...
void P(void)
{ operation_on(A); }
void T(void)
{ operation_on(A); }
typedef void (*proc)(void);
void Q(proc f)
{
 (*f)();
}
void R(void)
{
 Q(P);
 Q(T);
}
main()
{
 R();
}
```



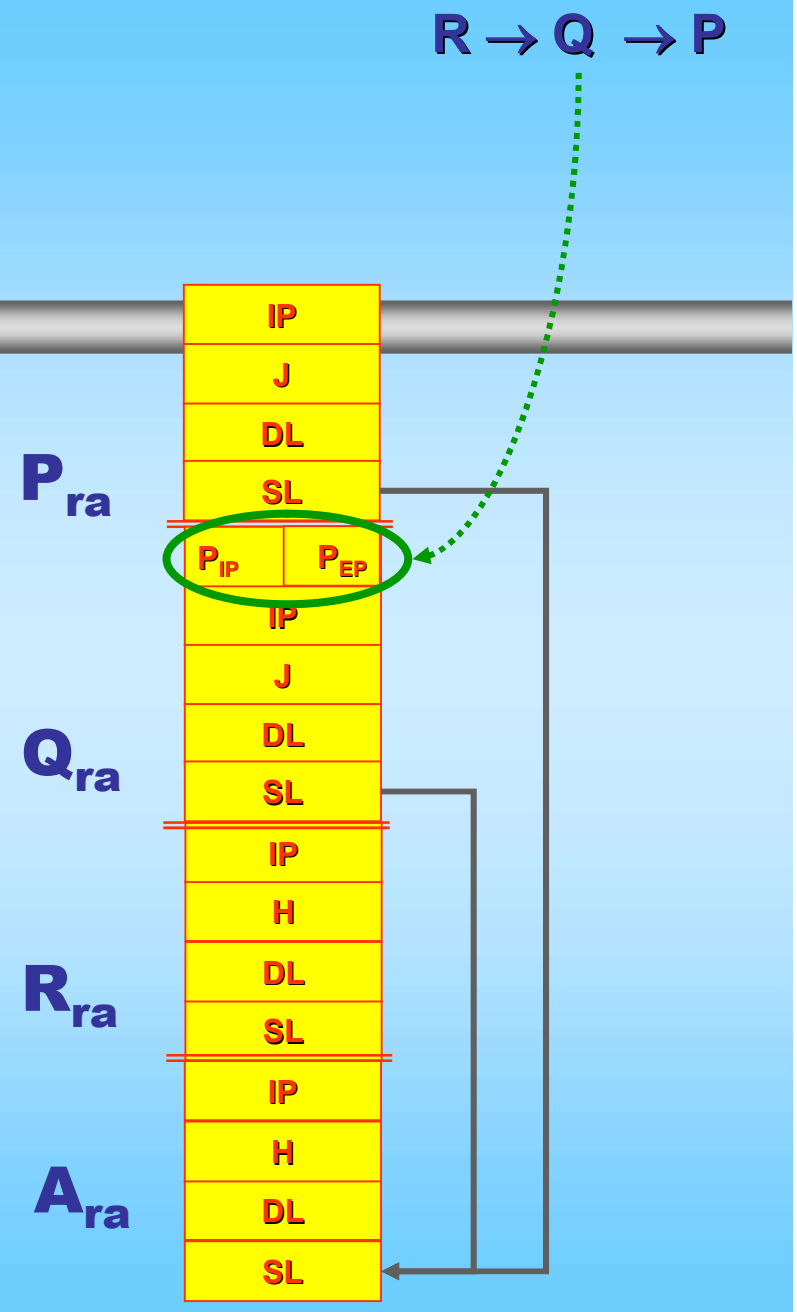
Functional parameter

Block syntax:

Pascal

```
program A;
  var A:object ;

  procedure P;
  begin
    operation_on(A);
  end;

  procedure Q(procedure f)
  begin
    f;
  end

  procedure R(void)
      procedure T;
      begin
        operation_on(A);
      end;
  begin
      Q(P);
      Q(T);
  end;

begin
      R;
end;
```

Execute the pointed function

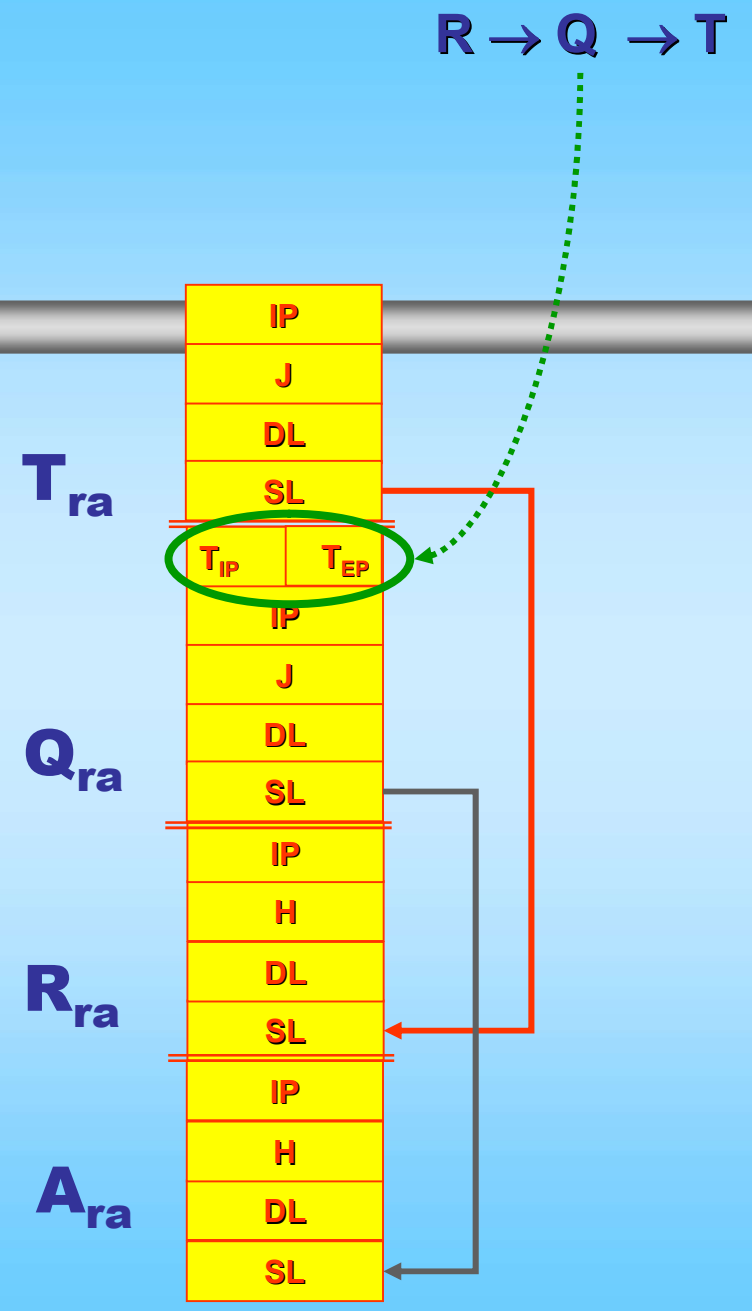


Functional parameter

Block syntax:

Pascal

```
program A;
  var A:object ;

  procedure P;
  begin
    operation_on(A);
  end;

  procedure Q(procedure f)
  begin
    f;
   end

   procedure R(void)
       procedure T;
       begin
          operation_on(A);
       end;
    begin
       Q(P);
       Q(T);
    end;

begin
      R;
end;
```



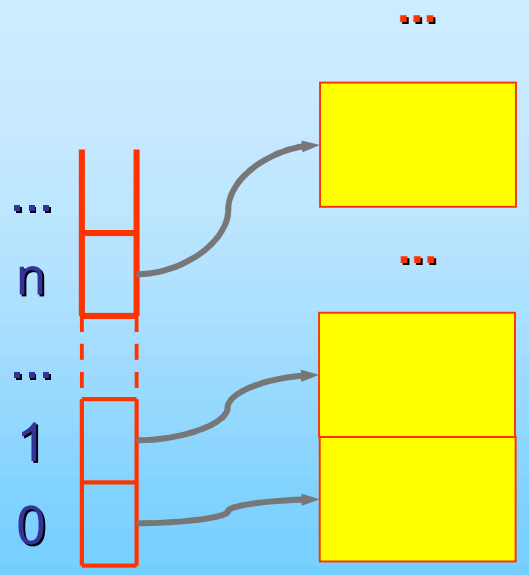
Is the SL pointer needed?

- **Global variables in a program with deeply nested blocks**

Instead of searching a singly linked list of ARs use indexing (principle of locality):

- **no EP**
- **direct access to the given nesting level:**

  loc(a_i)=cont(D[n])+i (only two operations!)
- **fast machine registers**

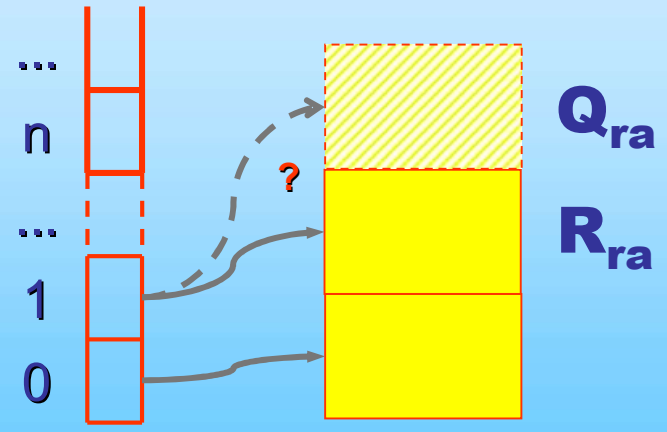


Is the SL pointer needed?

C

```
...
object A;
...
void P(void)
{ operation_on(A); }
void T(void)
{ operation_on(A); }
typedef void (*proc)(void);
void Q(proc f)
{
 (*f)();
}
void R(void)
{
 Q(P);
 Q(T);
}
main()
{
 R();
}
```

$R \rightarrow Q$

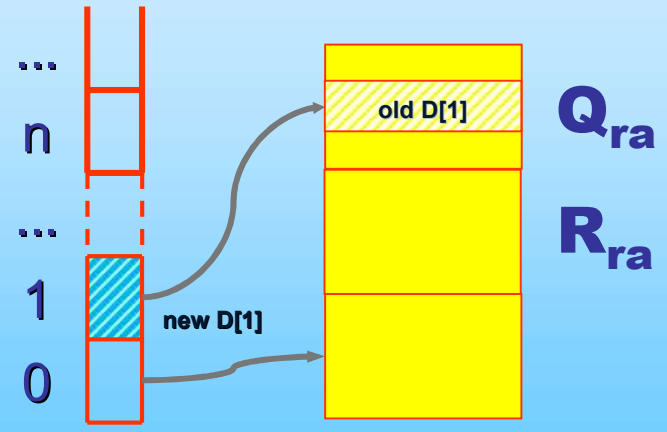


Is the SL pointer needed?

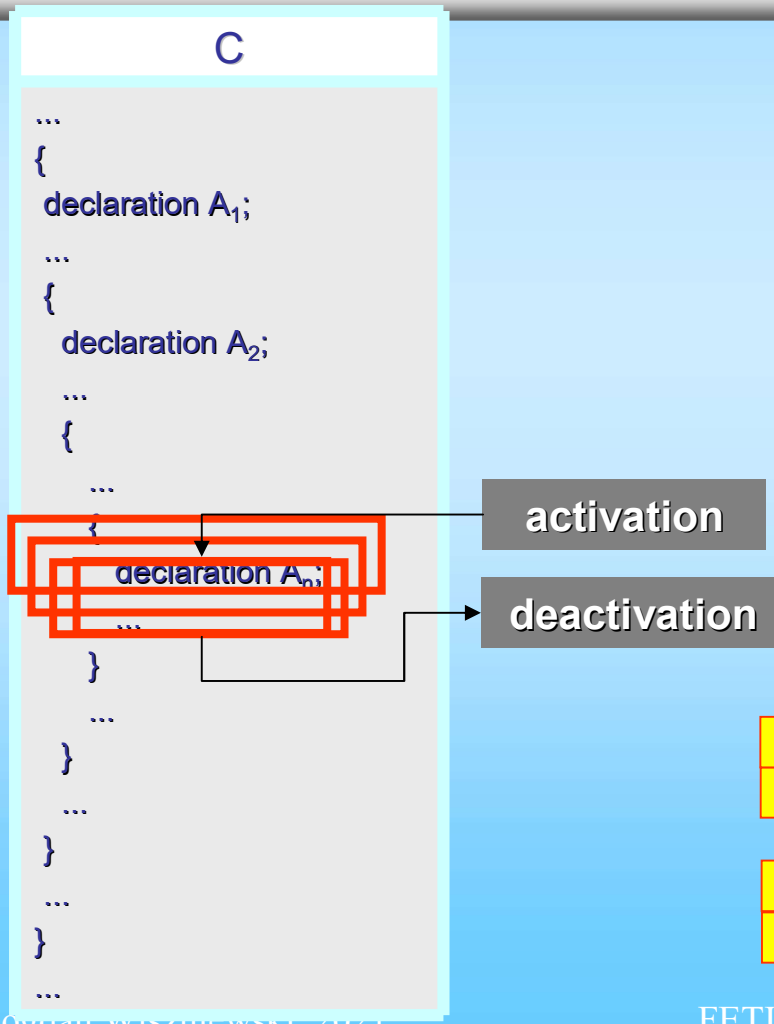
C

```
...
object A;
...
void P(void)
{ operation_on(A); }
void T(void)
{ operation_on(A); }
typedef void (*proc)(void);
void Q(proc f)
{
 (*f)();
}
void R(void)
{
 Q(P);
 Q(T);
}
main()
{
 R();
}
```

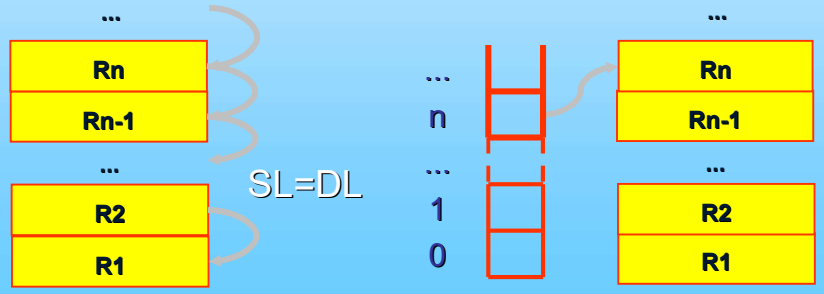
$R \rightarrow Q$



Block vs. procedure



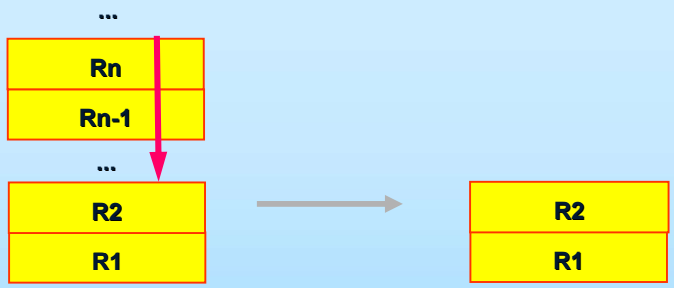
1. Anonymous procedure always called from one place:
   - no parameters,
   - SL always points to the enclosing block
   - DL is not needed with only one activation place
2. Static execution context (determined by the program text), whereas procedures have dynamic execution contexts



Jumping out of a block (goto)

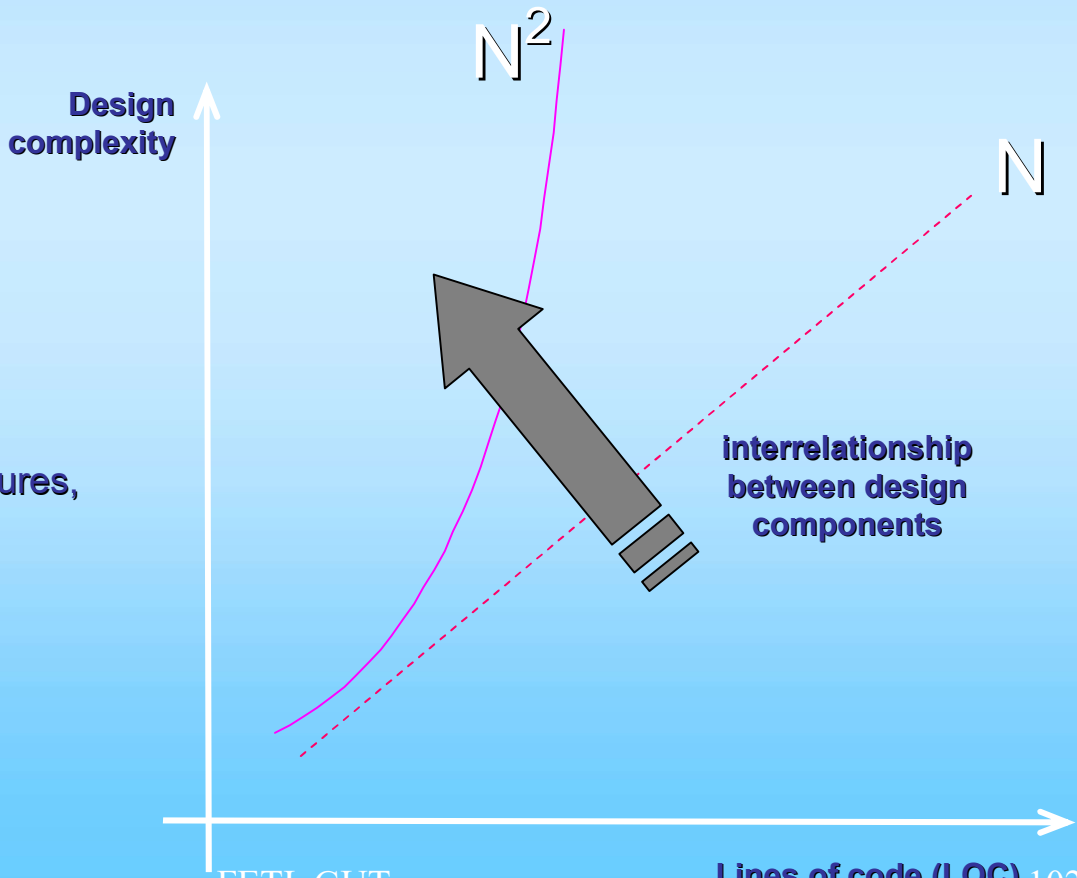
C

```
...
{
 declaration A1;
 ...
 {
  declaration A2;
  ...
  {
   ...
   {
    declaration An;
    ...
    goto L;
   }
   ...
  }
  ...
 }
 L: ...
}
...
```



Program design complexity

- **Block syntax → structural programming**

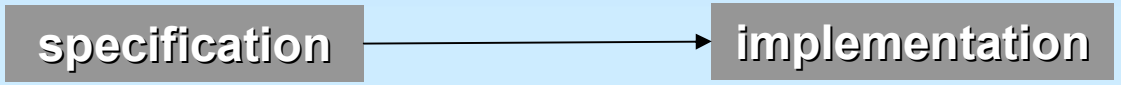


calling patterns:

- number of variables, procedures, blocks, nesting levels, etc.
- references to variables, procedures, etc.

Modularization

- **Functional decomposition**

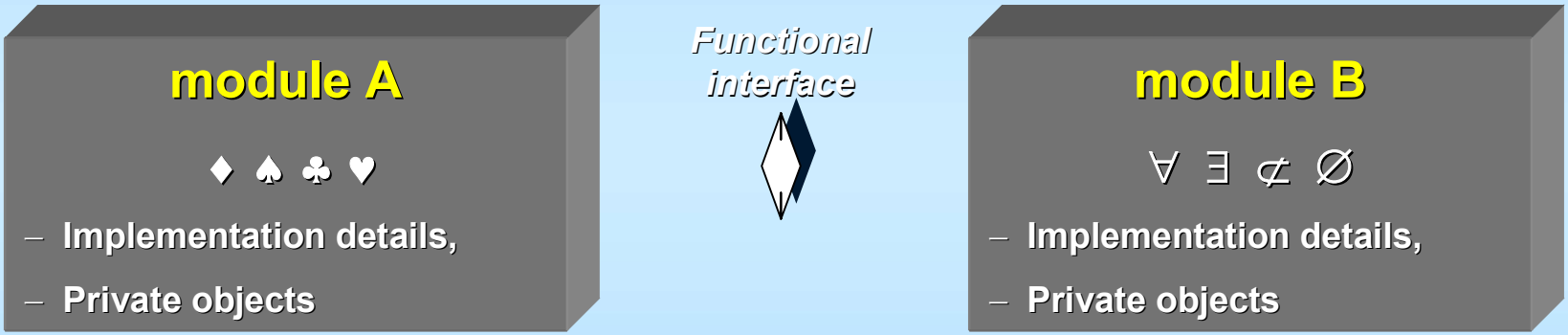


- Independent units (interface),
- Clarity of structure (architecture).

- Collaborative development,
- Testing, debugging,
- Maintenance

Modularization

- **Modularization principle**



neither A nor B can assume anything about each other

otherwise

what is contained in their specifications

DParnas, D.L: On the Criteria To Be Used in Decomposing Systems into Modules, Comm. ACM, Dec. 1972

Modularization rules

type identifier available for I_Stack users, but type details are hidden

Operation not permitted

access only through authorized functions

Ada

```
…
S: Int_Stack;
…
Item: Integer;
…
-- S.Size := 5;
…
Push(S, 5);
Pop(S, Item);
…
```

Ada

```
package I_Stack is
  type Int_Stack is private;
  procedure Push(S: Int_Stack; I: Integer);
  procedure Pop(S: Int_Stack; I: Integer);
  private
  type Stack_Data_Type is array(1..10) of Integer;
  type Int_Stack is record
     SP: Integer := 0;
     Data: Stack_Data_Type;
  end record;
end I_Stack;
```

```
package body I_Stack is
   procedure Push(S: Int_Stack; I: Integer) is
   begin
        S.SP := S.SP + 1;
        S.Data(S.SP) := I;
   end Push;
   procedure Pop(S: Int_Stack; I: Integer) is
   begin
        I := S.Data(S.SP);
        S.SP := S.SP - 1;
   end Pop;
end I_Stack;
```

Modularization rules

- **abstraction**

  operations on module data structures are carried out only through the functions of this module (eg. `Push`, `Pop`)

- **encapsulation**

  automatic control by the compiler (eg. `SP` and `Int_Stack` type `Data` pbjects)

- **diagnostics**

  exceptions (eg. overflow of the `Data` array)

- **uniformity**

  Compilable module definitions syntax (eg. `package`, `private`, `body`)

- **localization**

  details of the hardware hidden in the module (eg. positions of specific flags in a device status register)

Limitations of block structures

- **Side effects**
  - **in some situations the program performs undesirable actions on variables,**
  - **the circumstances of these activities as well as the identity of the variables are not clear for a programmer from the structure of the program alone.**

Effetti collaterali
- in alcune situazioni il programma esegue azioni indesiderate su variabili,
- le circostanze di tali attività nonché l'identità delle variabili non sono chiare per un programmatore dalla struttura diprogramma da solo.

implicit operations on a global variable

C

```
...
int count=10; /* global variable */
...
int MAX(int x, int y)
{
 count++;
 return ((x>y)? x : y);
}
...
main()
{
  printf("number=%d",MAX(count,10));
}
```

Limitations of block structures

- **unrestricted access (to data objects)**
  - **there is the potential for the program to operate on certain variables that the programmer cannot prevent**

• accesso illimitato (a oggetti di dati)
- c'è il potenziale per il programma operare su determinate variabili che il programmatore non può prevenire

The ability of main to call Push () and Pop () simultaneously means its ability to access SP and Data variables

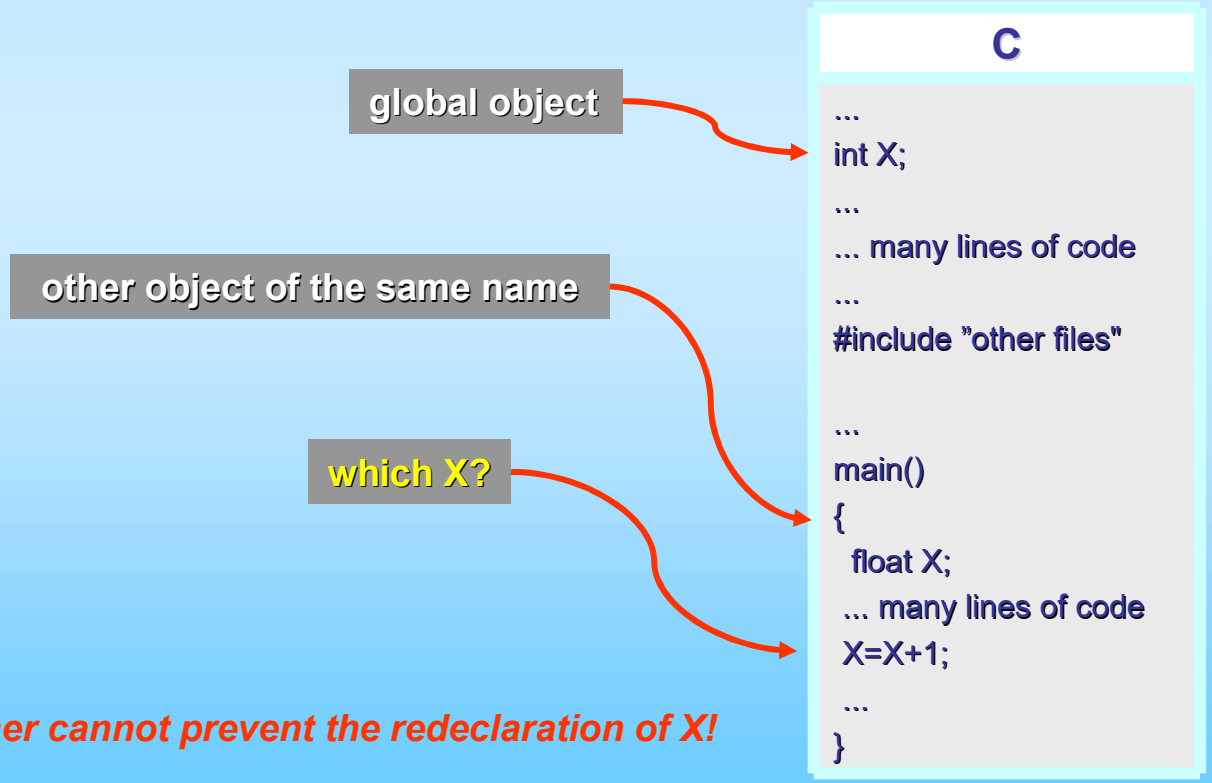
La capacità del principale di chiamare Push () ePop() contemporaneamente significa la sua capacità di abilitàper accedere alle variabili SP e Data

C

```
...
int Data[10];   /* stack memory */
int SP=0;   /* stack pointer */
void Push(int x)
{
 Data[SP++]=x;
}
int Pop()
{
 return(Data[--SP]);
}
...
main()
{
Push(5);
Data[3] = 5;
Data[3]++;
}
```

Limitations of block structures

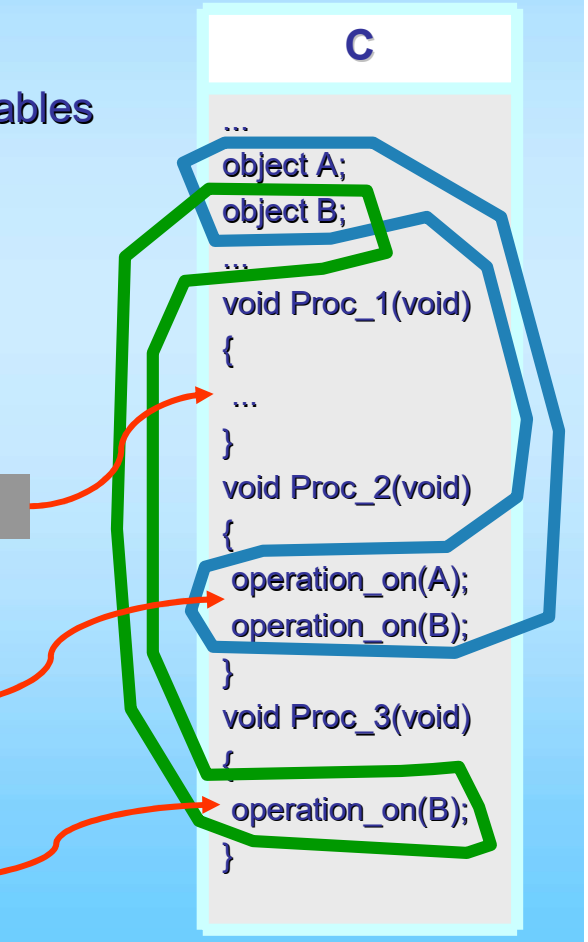
- **threatened access (to data objects)**
  - a given fragment of the program cannot guarantee access to specific variables



The programmer cannot prevent the redeclaration of X!

Limitations of block structures

- **selective access**
  - no possibility of selective access to particular variables from various parts of the program



access to A and B should be forbidden

access to both A and B should be guaranteed

access to B should be guaranteed and forbidden to A

Languages supporting modularization

- **Extending language syntax by the notion of the ”module”:**

  Ada (1979) → *package*
  Modula-2 (1980) → *module*
  C++ (1984) → *(static) class*

- **New features for the programmes to use:**
  - **”private” and ”public” information,**
  - **functional interface of the module**
  - **selective access to data objects**
  - **encapsulation of objects in modules**

Interface specification:
- declarations
- definitions

Body:
- functions
- data objects

Initialization of local data objects

Module representation

- **External representation**
  - operators from one package to operate on all objects of a given type

Ada

```
...
use Stack_Type;
stack1 : Stack;
stack2 : Stack;
...
Item: Integer;
Push(stack1,5);
Pop(stack2,Item);
...
```

Ada

```
package Stack_Type is
  type Stack is private;
  procedure Push(S: Stack; I: Integer);
  procedure Pop(S: Stack; I: Integer);
  private
  type Stack is record
     SP: Integer := 0;
     Data: array(1..10) of Integer;
   end record;
end Stack_Type;
```

```
package body Stack_Type is
   procedure Push(S: Stack; I: Integer) is
   begin
        S.SP := S.SP + 1;
        S.Data(S.SP) := I;
   end Push;
   procedure Pop(S: Stack; I: Integer) is
   begin
        I := S.Data(S.SP);
        S.SP := S.SP - 1;
   end Pop;
end Stack;
```

Module representation

- **Internal representation**
  - operators in one package with the data on which they operate

Ada

```
...
package stack1 is new Stack;
package stack2 is new Stack;
...
Item: Integer;
stack1.Push(5);
stack2.Pop(Item);
...
```

Ada

```
generic
package Stack is
  procedure Push(I: Integer);
  procedure Pop(I: Integer);
end Stack;
```

```
package body Stack is
  Data: array(1..10) of Integer;
  SP: Integer range 0..10 :=0;
   procedure Push(I: Integer) is
   begin
        SP := SP + 1;
        Data(SP) := I;
   end Push;
   procedure Pop(I: Integer) is
   begin
        I := Data(SP);
        SP := SP - 1;
   end Pop;
end Stack;
```

Procedure call

- **New features**
  - default parameters
  - rearrangement of the parameters in the call (named parameters)
  - variable number of parameters (variadic function)
  - (function) operator overloading

Default parameters

- some arguments are used more often and others less often

C++

```
…
char *scrInit(int height=24, int width=80, char background=' ')
{
 …
}
```

```
…
char *cursor;
…
```

```
cursor=scrInit(); /* equivalent to scrInit(24,80,' ') */
…
cursor=scrInit(66); /* equivalent to scrInit(66,80,' ') */
…
cursor=scrInit(66,256); /* equivalent to scrInit(66,256,' ') */
…
cursor=scrInit(66,256,'#');
```

Named parameters

- **A long list of parameters may be rearranged**

Ada

```
void Draw_Axes(int X0, int Y0,
        float X_scale, float Y_scale,
        unsigned int X_spacing, unsigned int Y_spacing,
        boolean X_logarithmic, boolean Y_logarithmic,
        boolean X_label, boolean Y_label,
        boolean GRID)
{
 ...
}
```

```
-- ugh!
Draw_Axes(500,500,1.0,0.5,10,10,False,True,False,False,Tr
ue);
```

```
...
-- much better!
Draw_Axes(  X0=500,Y0=500,GRID=True,
                X_label=false,X_scale=1.0,
                X_logarithmic=False,X_spacing=10,
                Y_label=false,Y_scale=0.5,
                Y_logarithmic=true,Y_spacing=10
);
```

Variable number of parameters

- **Procedures with an unknown number and type of parameters**

  eg., the ellipsis operator informs the C/C++ compiler that a given function may be called with different types and numbers of parameters

C

```
...
void f();      /* void f(void); */
void f(...);      /* different function! */
...
f(5);
f(a,b,'x');
...
f();
...
```

C

```
...
int printf(const char *format, ...);
...
```

```
...
printf("Hello, world");
...
printf("Hello, %s", userName);
...
printf("Hello, %s, at terminal #%d", userName, 531);
...
```

Operator overloading

- **Reducing the space of function operators**

C

```
/* return max */
int i_max(int,int);
…
int ia_max(const *int,int);
…
int list_max(const List&);
```

C++

```
/* better! */
int max(int,int);
…
int max(const *int,int);
…
int max(const List&);
```

C++

```
…
// Overloading arithmetic operators…
complex complex::operator + (complex y);
complex complex::operator + (float y);
complex complex::operator   (complex y);
complex complex::operator   (float y);
complex complex::operator * (complex y);
complex complex::operator * (float y);
complex complex::operator / (complex y);
complex complex::operator / (float y);
…
```

→ in C ++, function overload occurs when functions with the same name and the same return type have different signatures

Context of use of function operators

- **Feature clashes**

  eg., named parameters + default parameters + operator overloading

C++

```
...
void P(int X, boolean Y=False);
void P(int X, int Y=0);
...
P(9,True);
P(5,8);
P(X=1); /* ?! */
P(20); /* ?! */
```

the compiler must reject ambiguous contexts!

Mathematical linguistics (Chomsky hierarchy):

$$L_{reg} \subset L_{CF} \subset L_{CS} \subset L_{RE}$$

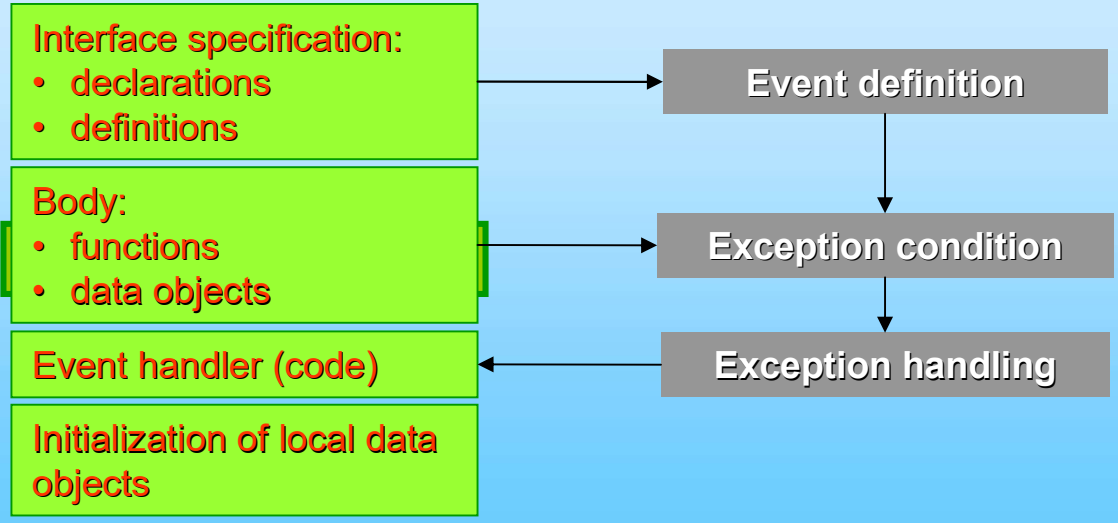
Exception handling

- **when the normal program operation must be suspended, eg.**

  a variable exceeds its legitimate value, operations performed outside of the data structure, illegal memory access, reading values of non-existing objects, corrupted data, etc.

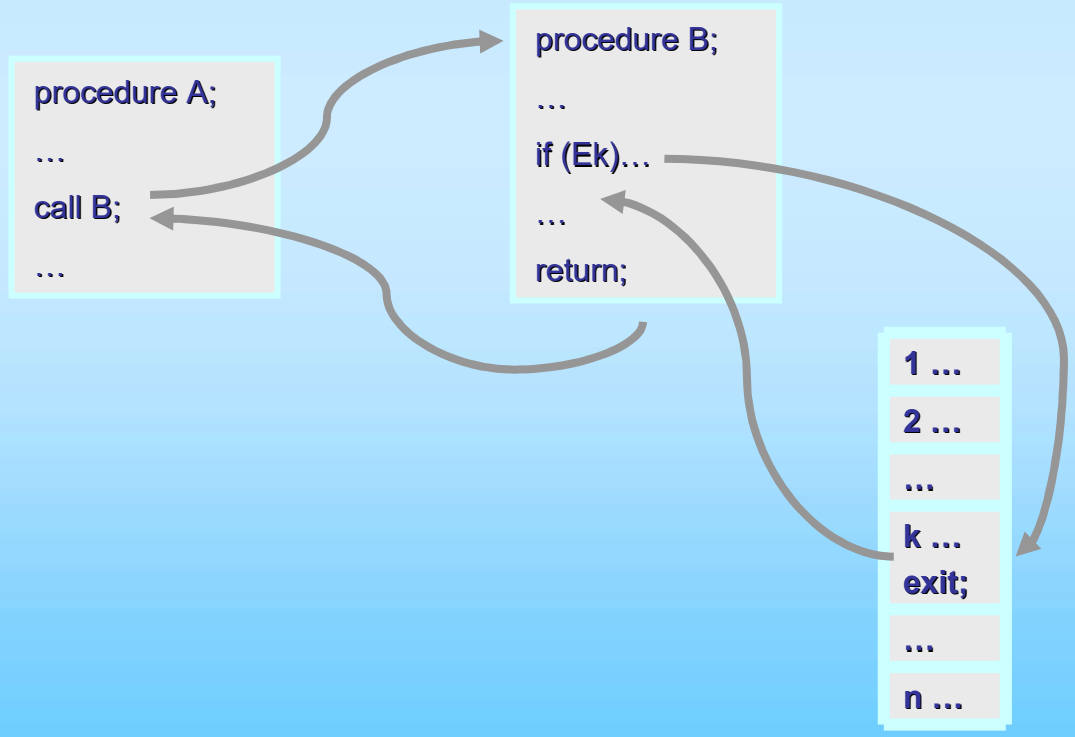
- **language syntax allows programmers to define the code structure with exceptions**

  exception event → exception condition → exception handling



Exception handling models

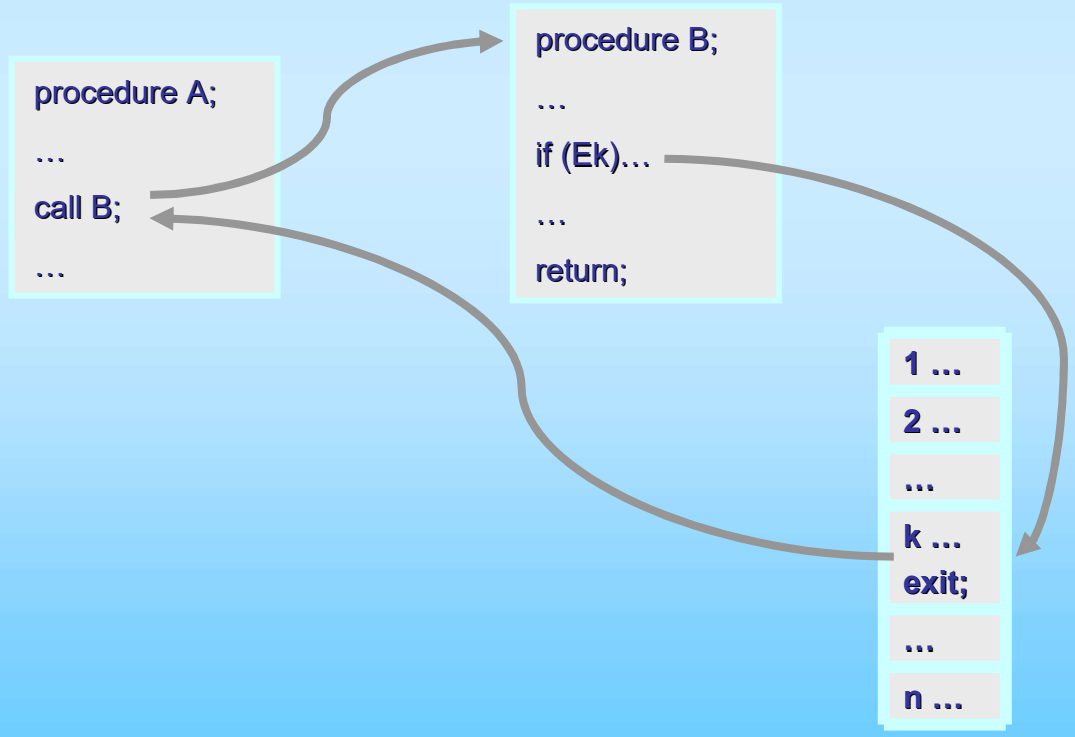
- **resumption model**



eg., PL/I

Exception handling models

- **termination model**



eg., Ada83, C++

Event syntax

- **eg., C++ (Release 3, ANSI 1990) a block may ”throw” (rise) exceptions**

C++

```
...
try {
 int x, y, z;
 ...
 x=y+z;
 if (x-z!=y) throw ”out of scope!";
 ...
 if (z==0) throw ”division by zero!";
 y=x/z;
 ...
}
catch (const char* p)
{
 printf("Panic: %s, p);
}
...
```

the block throws an object…

…and the handler catches it

Event syntax

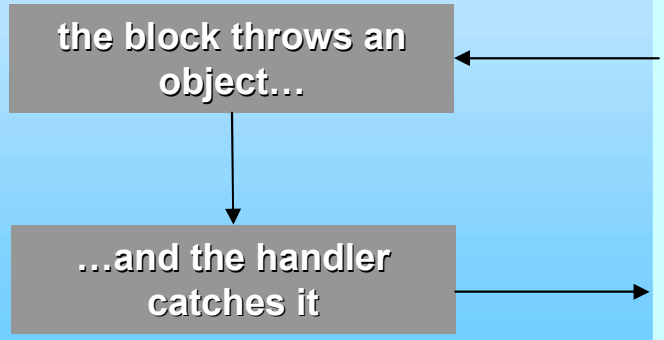
- **Functions may ”throw” (rise) exceptions**

C++

```
...
typedef int ErrNo;
...
void do_it() throw (const char, ErrNo)
{
 ...
 if ... throw 'A';
 if ... throw 15;
 ...
 return;
}
```

```
void foo()
{
 ...
 try
 {
  do_it();
 }
```

```
catch (char c)
 { ... }
 catch (ErrNo n)
 { ... }
}
```



Event syntax

- **Functions may rise (throw) exceptions**

Ada

```
...
use Stack_Type;
stack1 : Stack;
stack2 : Stack;
...
Item: Integer;
begin
  Push(stack1,5);
  Pop(stack2,Item);
 exception
    when Overflow =>
        Put_Line("Overflow");
    when Underflow =>
        Put_Line(„Underflow");
 end;
...
```

Ada

```
package Stack_Type is
  type Stack is private;
  Underflow, Overflow: exception;
  procedure Push(S: Stack; I: Integer);
  procedure Pop(S: Stack; I: Integer);
  private
  type Stack is record
    ...
  end record;
end Stack_Type;
```

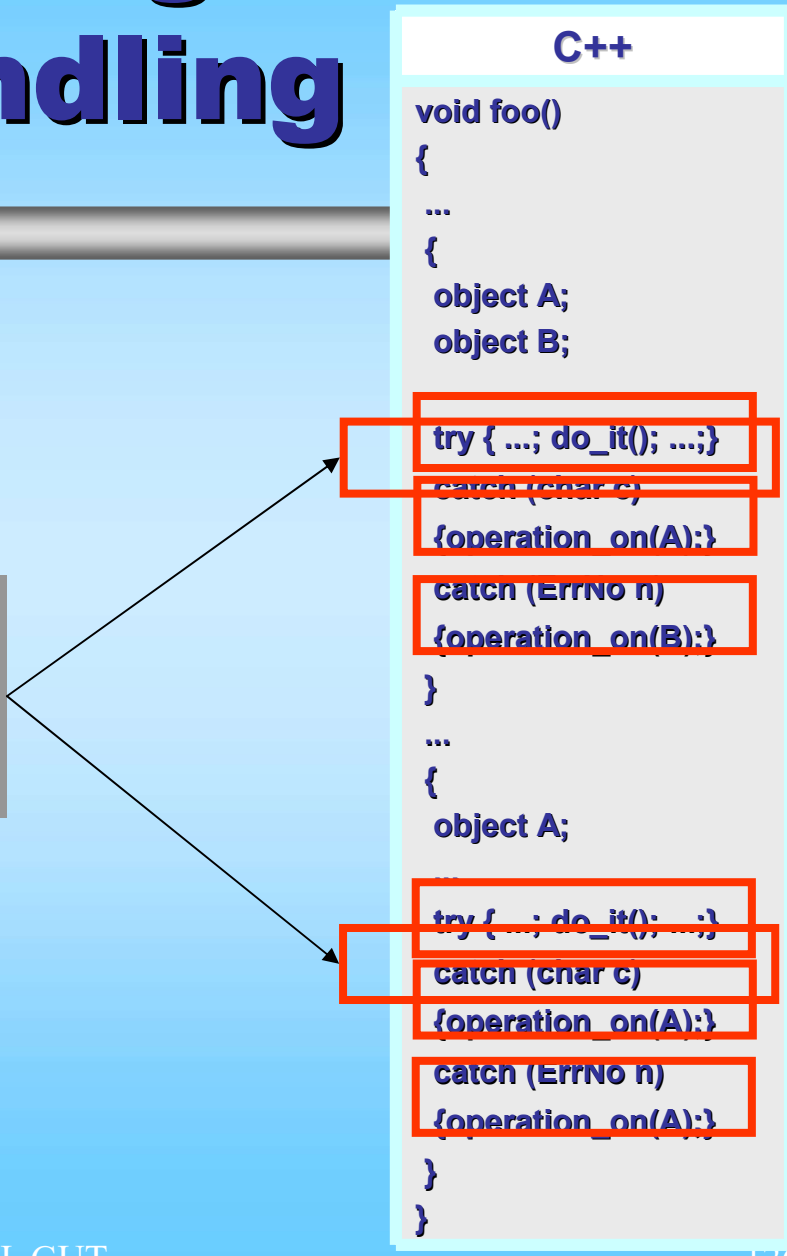
```
package body Stack_Type is
   procedure Push(S: Stack; I: Integer) is
   begin
      if S.SP < 10 then
        S.SP := S.SP + 1;
        S.Data(S.SP) := I;
      else raise Overflow;
      end if;
   end Push;
   procedure Pop(S: Stack; I: Integer) is
   begin
      if S.SP > 0 then
        I := S.Data(S.SP);
        S.SP := S.SP - 1;
      else raise Underflow;
      end if;
   end Pop;
end Stack;
```

Dynamic scoping rules in exception handling

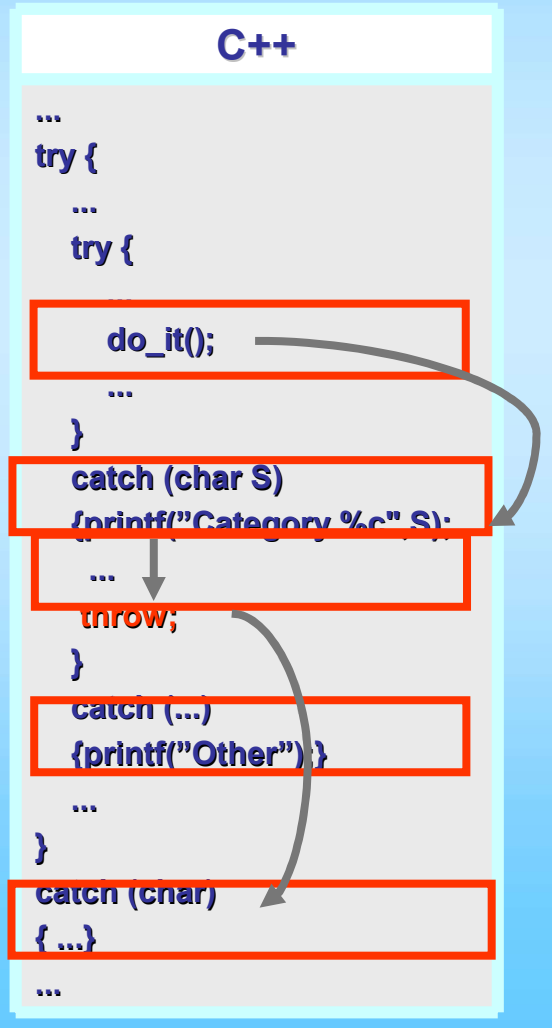
handlers of exceptions thrown by 'do_it' are called in

different

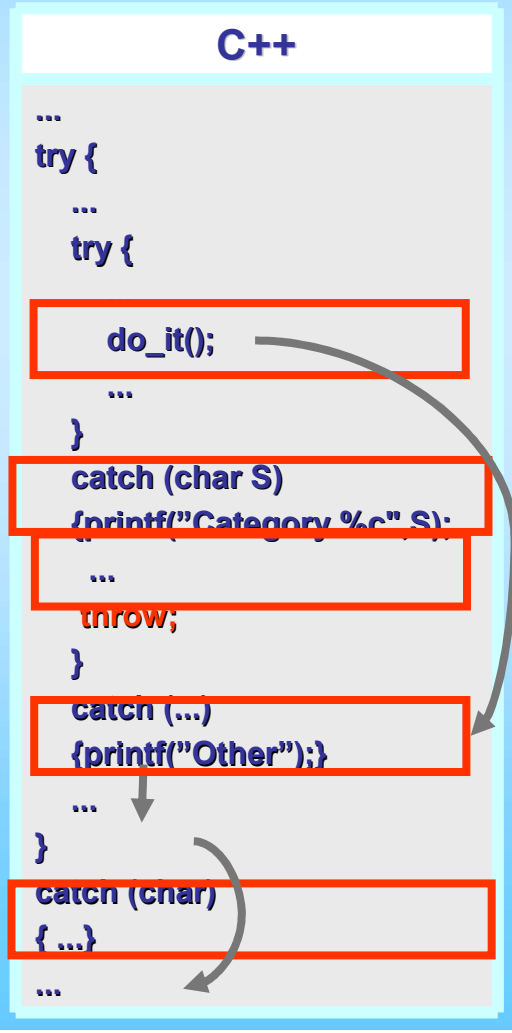
environments (contexts)



Propagation of exceptions

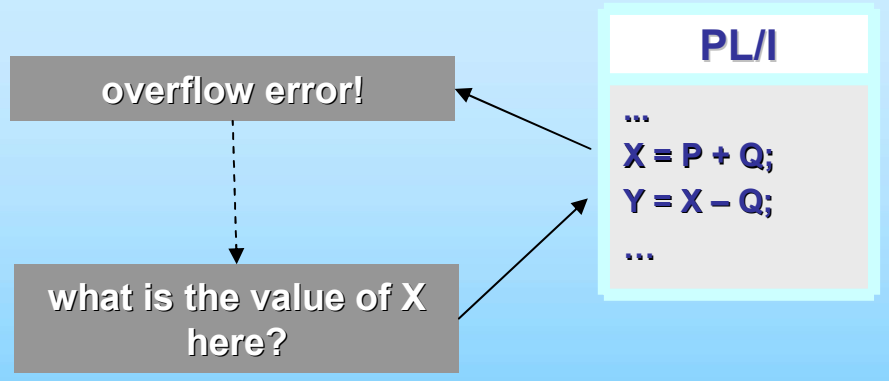


Propagation of exceptions



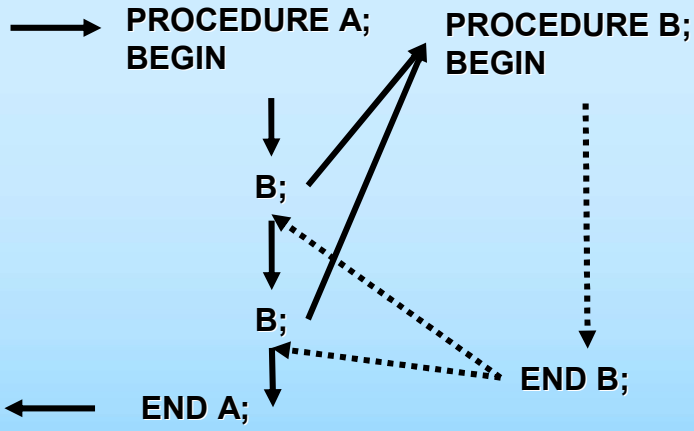
Why the termination model?

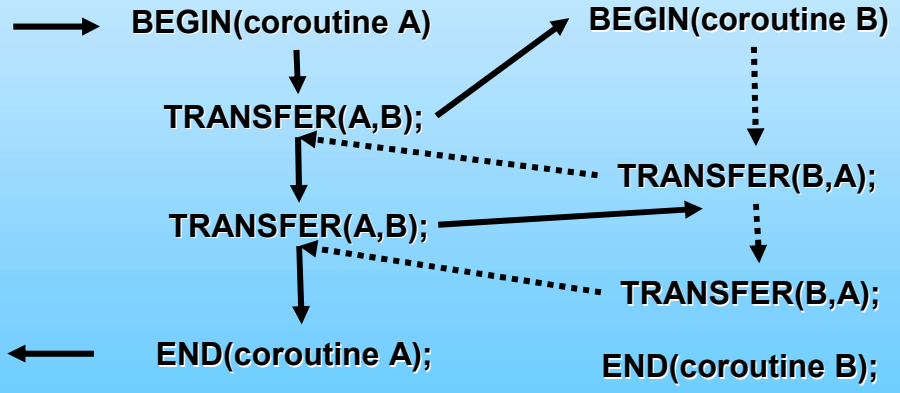
- **Automatic resumption:**
  - explicit diagnostic information not available
  - code optimization problems
  - formal code verification (proof of correctness) more difficult



Procedural concurrency

- **co-routines**





procedures (functions) ⊂ coroutines

single stack → multiple stacks

Procedural c

- **Co-routines:**
  - Simula-67
  - Modula-2
  - Extensions in Python and Ruby

Modula-2

```
PROCEDURE Produce(VAR newtoken: Token;
                  VAR producer, consumer: Coroutine);
  VAR
    token: Token;
BEGIN
LOOP (* produce token *)
    newtoken := token;
    TRANSFER(producer,consumer);
  END;
END Produce;
```

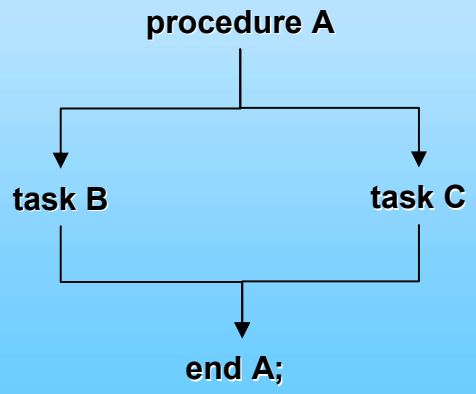
```
PROCEDURE Consume(VAR newtoken: Token;
                  VAR producer, consumer: Coroutine);
  VAR
    token: Token;
BEGIN
LOOP (* consume token *)
    token := newtoken;
    TRANSFER(consumer,producer);
  END;
END Consume;
```

```
PROCEDURE Setup;
  VAR
    token: Token;
    producer, consumer: Coroutine;
BEGIN
  Produce(token, producer, consumer);
  Consume(token, producer, consumer);
END Setup;
```

Procedural concurrency

- **threads**

  parallel block (threads without internal communication)



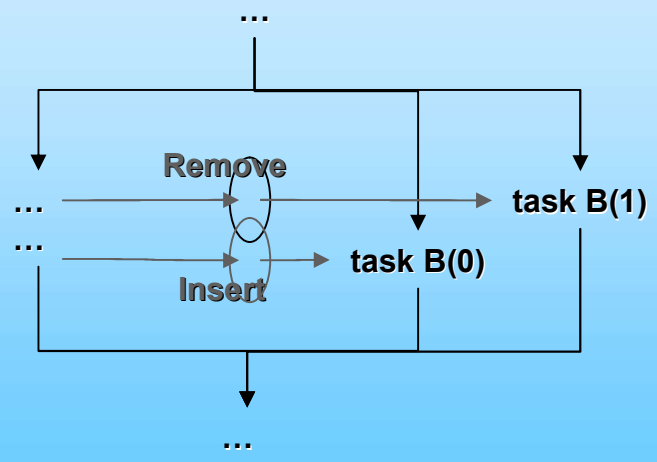
ADA95

```
procedure A is
   task B;
   task C;
   task body B is
      ...
   end B;
   task body C is
      ...
   end C;
   -- threads B i C created and started
begin -- thread A (main)
   null;
   -- thread A waits for termination
end A;
```

Procedural concurrency

- **threads**

  Rendezvous, eg. blocking synchronous and symmetric (1-1) event



ADA95

```
...
task type Encapsulated is
   entry Insert (An_Item : in  Item);
   entry Remove (An_Item : out Item);
end Encapsulated;
...
B: array (0 .. 1) of Encapsulated;
This_Item   : Item;
...
task body Encapsulated is
   Datum : Item;
begin
   loop
      accept Insert (An_Item : in  Item) do
         Datum := An_Item;
      end Insert;
      accept Remove (An_Item : out Item) do
         An_Item := Datum;
      end Remove;
   end loop;
end Encapsulated_Buffer_Task_Type;
...
B(0).Remove (This_Item);
B(1).Insert (This_Item);
```

Procedural concurrency

- selective wait (non-deterministic selection)

ADA95

```
task type Encapsulated is
  entry Store (An_Item : in  Item);
  entry Fetch (An_Item : out Item);
end Encapsulated;
...
task body Encapsulated is
  Datum : Item;
begin
  accept Store (An_Item : in Item) do
    Datum := An_Item;
  end Store;
  loop
    select
      accept Store (An_Item : in Item) do
        Datum := An_Item;
      end Store;
    or
      accept Fetch (An_Item : out Item) do
        An_Item := Datum;
      end Fetch;
    end select;
  end loop;
end Encapsulated;
```

Procedural concurrency

- guards

ADA95

```
task Cyclic is
   entry Insert (An_Item : in  Item);
   entry Remove (An_Item : out Item);
end Cyclic;
...
task body Cyclic is
   Q_Size : constant := 100;
   subtype Q_Range is Positive range 1 .. Q_Size;
   Length : Natural range 0 .. Q_Size := 0;
   Head, Tail : Q_Range := 1;
   Data : array (Q_Range) of Item;
begin
   loop
      select
         when Length < Q_Size =>
            accept Insert (An_Item : in  Item) do
               Data(Tail) := An_Item;
            end Insert;
            Tail := Tail mod Q_Size + 1;
            Length := Length + 1;
      or
         when Length > 0 =>
            accept Remove (An_Item : out Item) do
               An_Item := Data(Head);
            end Remove;
            Head := Head mod Q_Size + 1;
            Length := Length - 1;
      end select;
   end loop;
end Cyclic;
```

Object oriented approach

- **Abstract data types**
  - private representation (structure),
  - public set of operations

C++

```
template <class item>
class uniSTOS
{
 private:
   int SP;
   item A[10];
 public:
   void PUSH(item x)
   {
     A[++SP]=x;
   }
   item POP(void)
   {
     return A[SP--];
   }
};
```

C++

```
template <class item, int size>
class genSTOS
{
 private:
   int SP;
   item A[size];
 public:
   void PUSH(item x)
   {
     A[++SP]=x;
   }
   item POP(void)
   {
     return A[SP--];
   }
};
```

C++

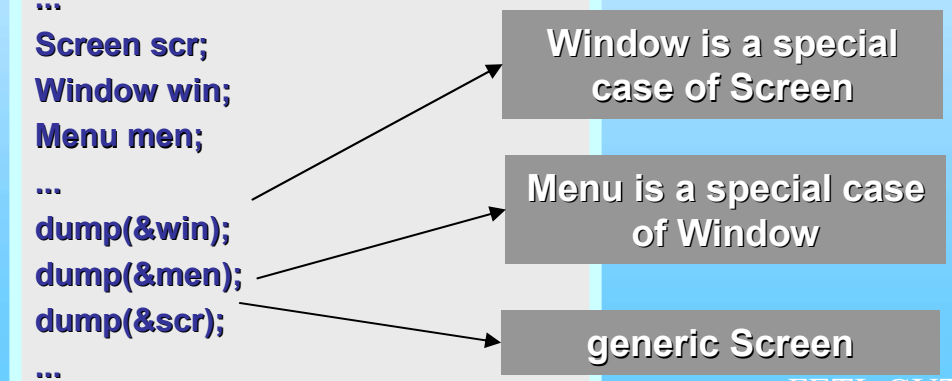
```
uniSTOS<char> s1;
genSTOS<float,12> s2;
...
s1.PUSH('a');
s2.PUSH(3.14);
print("top=%d",s2.POP());
...
```

Object oriented approach

- **Abstract data types extended by adding the inheritance relation**
  - class, SIMULA-67
  - object-oriented, SMALLTALK-80

C++

```
extern void dump(Screen *s);
...
Screen scr;
Window win;
Menu men;
...
dump(&win);
dump(&men);
dump(&scr);
...
```



C++

```
class Screen {
public:
  void home();
protected:
  char *S; // field of characters
  int H,W; // height, width
};
```

```
class Window : public Screen {
public:
  void open_at(int x, int y);
private:
  int x0, y0; // base vertex
};
```

```
class Menu : public Window {
public:
  void entry(int field_no);
private:
   int E[]; // indexes of menu entries
};
```

Object oriented approach

- **Non-object oriented solution**

C

```
enum {SCREEN,WINDOW,MENU};
void dump(Screen *s)
{
  switch(what_is(s))
  { case SCREEN:
      s->dumpImage(); break;
    case WINDOW:
      ((Window*)s)->dumpImage(); break;
    case MENU:
      ((Menu*)s)->dumpImage(); break;
    default:
      printf("panic: unknown object type");
  }
}
```

- **Object-oriented solution**

C++

```
void dump(Screen *s)
{
  s->dumpImage();
}
```

Basic object-oriented terms

Smalltalk-80 (Alan Kay et. al., 1980, Xerox)

- object: private memory + operations
- sender/receiver: An object (sender) asks another object (receiver) to perform an action by sending it a message
- method: description of operations the object can perform (understands the message)
- interface: a set of messages understood by the object
- class: decription of the group of similar objects
- instance: a single object described by its class
- instance variable: private memory of the object
- class variable: a variable shared by all objects of the same class
- primitive method: operation performed directly by the Smalltalk VM

→ *In the beginning there was MetaClass and the MetaClass was Object. Through it all things were made; without it nothing was made that has been made.*

Smalltalk messages

- message selector: name of the operation requested from the receiver by the sender, eg.

```
theta sin
list removeLast
```

- unary message: sent to an object without any other information (no message arguments)

- binary message: the receiver and the argument object; its selector consists of a sequence of one or more characters +, -, *, /, &, =, >, |, <, ~,@, eg.

```
base + 12
index > limit
```

- keyword message: one or more arguments; its selector consists of one or more keywords each ending in :

```
list addFirst: item
ages at: 'Franek' put: 31
```

Message syntax

1. Unary message - left to right

```
1.5 tan rounded
(1.5 tan) rounded
```

2. Binary message - left to right

```
index + offset * 2
(index + offset) * 2
index + (offset * 2)
```

3. Binary messages have a higher priority than messages with keywords

```
bigFrame width: smallFrame width * 2
bigFrame width: ((smallFrame width) * 2)
```

4. Unary messages have a higher priority than binary messages

5. Expressions in parentheses have a higher priority than unary messages

```
(2 * theta) sin
2 * theta sin
```

Performing of actions

- **Action block**
  - class ‘BlockClosure’

Smalltalk

```
...
index := 0.
A := [index := index + 1].
A value.
```

```
...
actions
  at: 'monthly payments'
  put: [HouseHoldFinances spend: 650 on: 'rent‘
        HouseHoldFinances spend: 7.25 on: 'newspaper'.
        HouseHoldFinances spend: 225.50 on: 'car payment'].
...
(actions at: 'monthly payments') value.
```

Performing of actions

- **Iterations, conditions**

Smalltalk

```
4 timesRepeat: [amount := amount + amount]

(number \\ 2) = 0
    ifTrue: [parity := 0]
    ifFalse: [parity := 1]

index <= limit
    ifTrue: [total := total + (list at: index)
    ifFalse: [ ]

index := 1.
[index <= list size]
     whileTrue: [list at: index put: 0. index := index + 1]
```

Performing of actions

- **Blocks with parameters**

Smalltalk

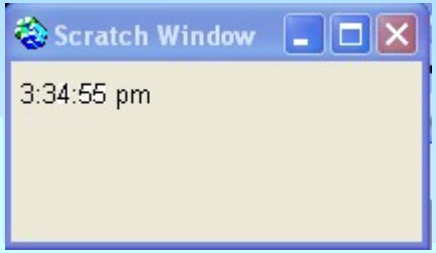
```
sum := 0.
#(2 3 5 7 11) do: [:prime | sum := sum + (prime * prime)].

sizeAdder := [:array | total := total + array size].
total := 0.
sizeAdder value: #(a b c).
sizeAdder value: #(1 2).
sizeAdder value: #(e f)
```

Performing of actions

- **Threads**

Smalltalk

```
| gc |
gc := (Examples.ExamplesBrowser prepareScratchWindow)
graphicsContext.
[
 [true] whileTrue:
       [ | clock |
         clock := Time now printString asComposedText.
         gc paint: ColorValue black.
         clock displayOn: gc at: 5 @ 5.
         (Delay forSeconds:1) wait.
         gc paint: ColorValue white.
         clock displayOn: gc at: 5 @ 5.
       ]
] fork
```

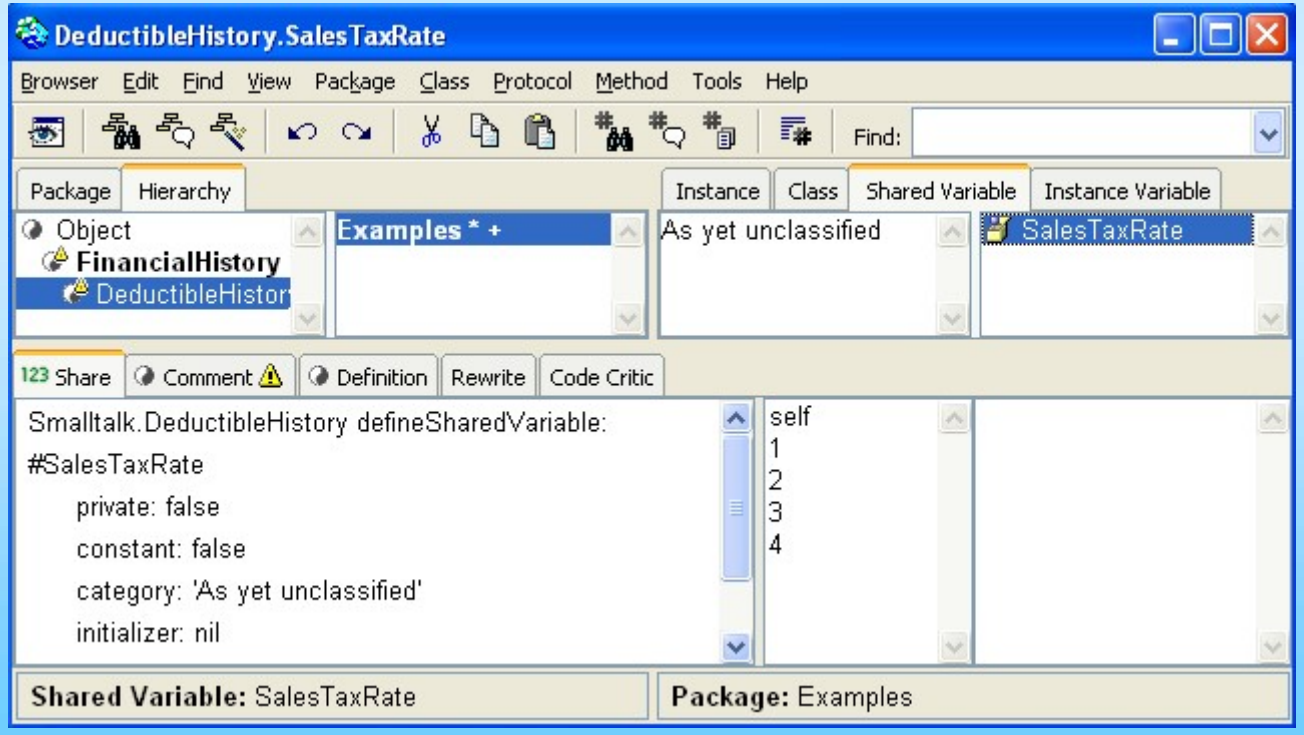


Smalltalk

```
les.ExamplesBrowser prepareScratchWindow)
text.

rue:
 |
= Time now printString asComposedText.
t: ColorValue black.
lisplayOn: gc at: 5 @ 5.
forSeconds:1) wait.
t: ColorValue white.
clock displayOn: gc at: 5 @ 5.
      ]
] value
```

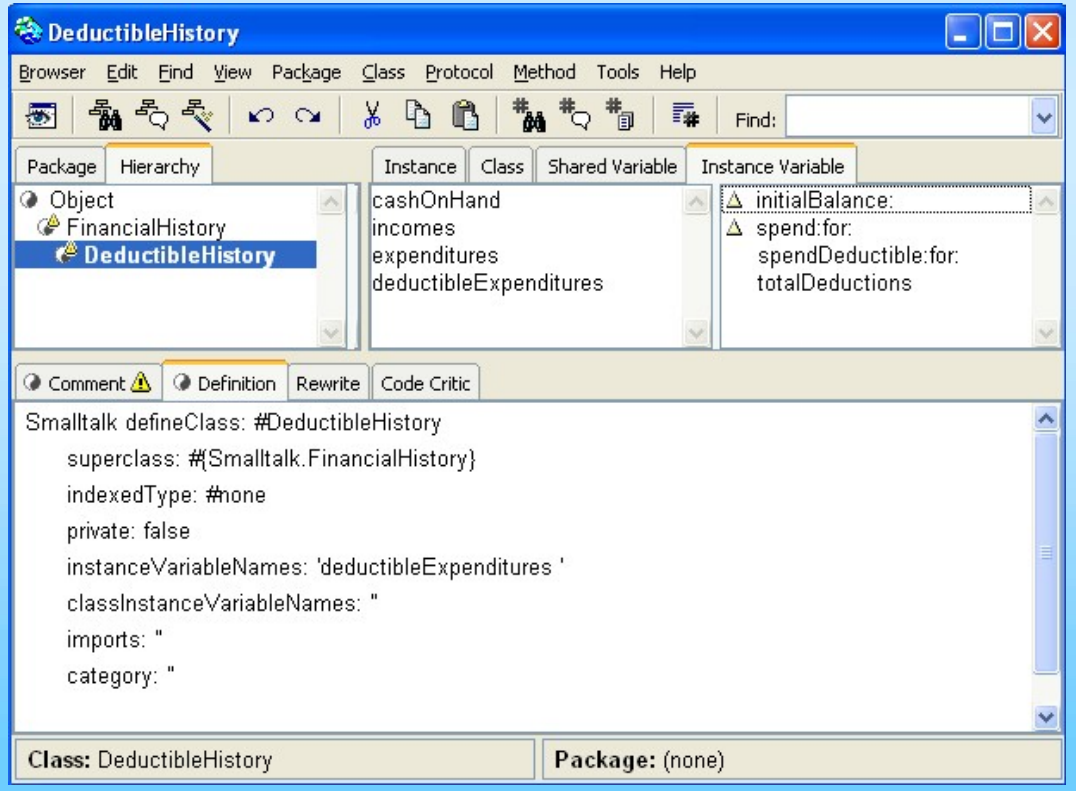
Class implementation

- **Shared variables (previously class variables):**
  - One copy for all objects of the same class



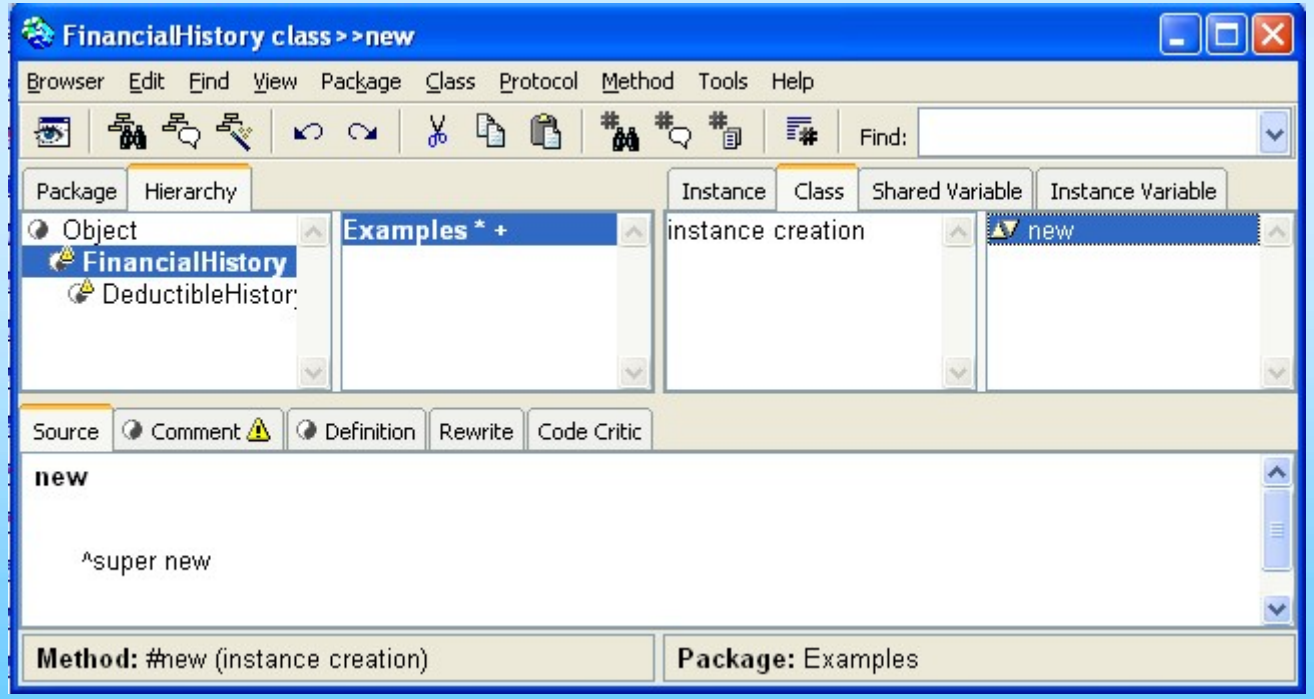
Class implementation

- **Instance variables:**
  - each object has its own copies



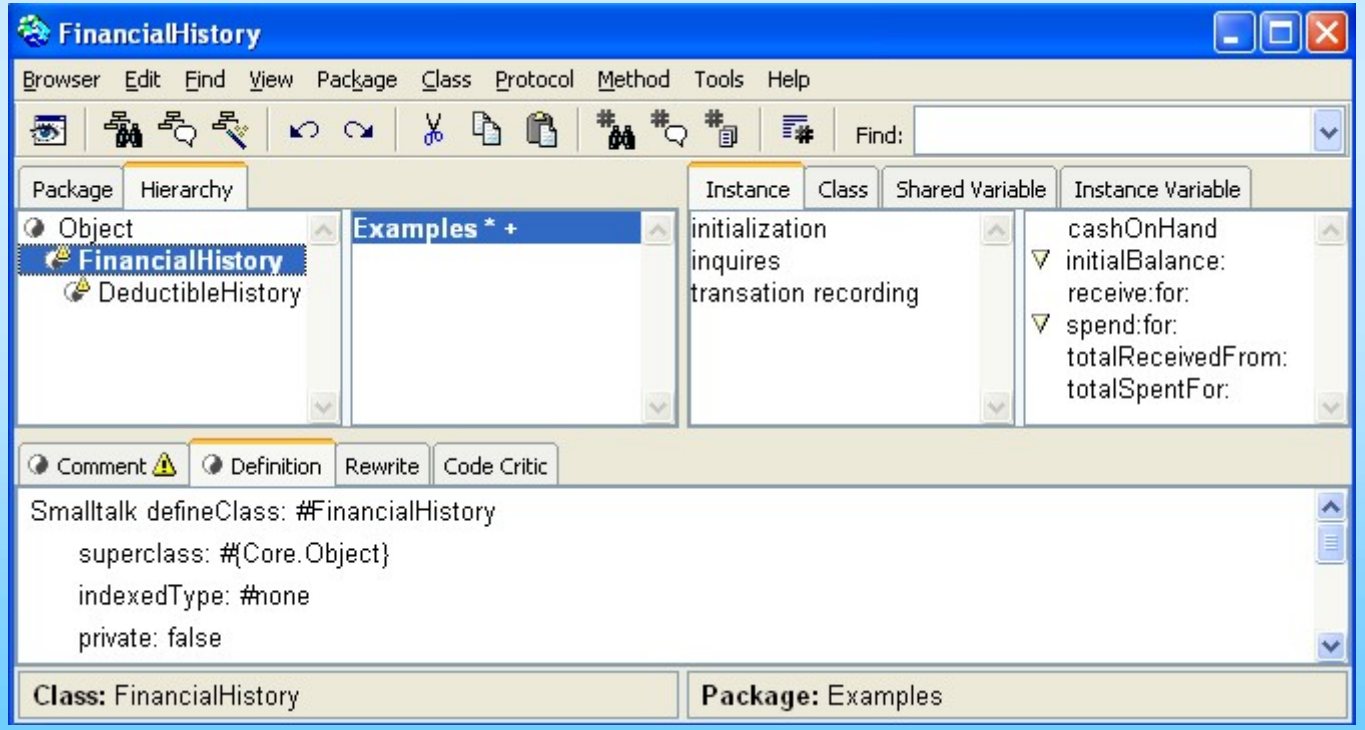
Class implementation

- **Class methods**
  - Object (class instance) creation

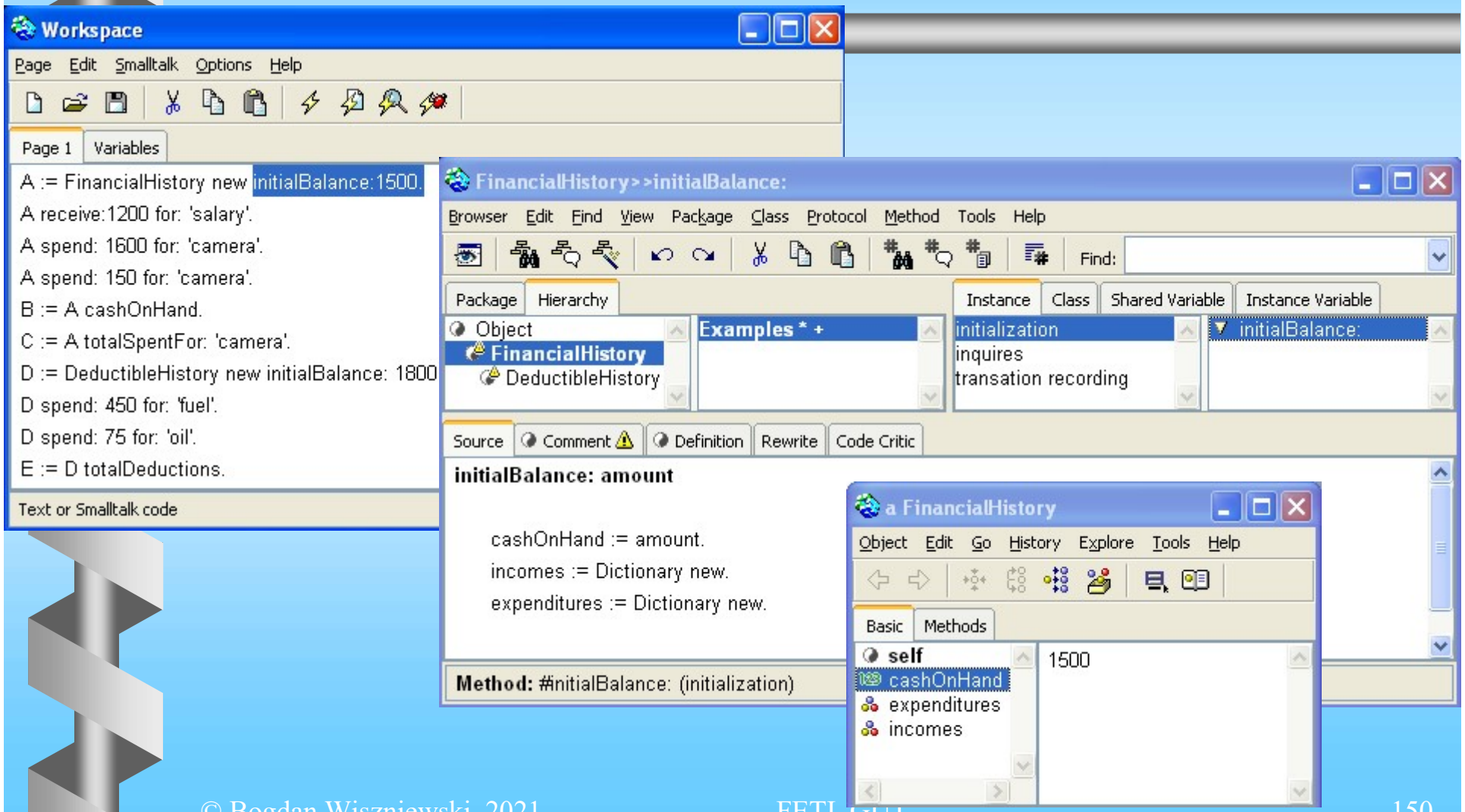


Class implementation

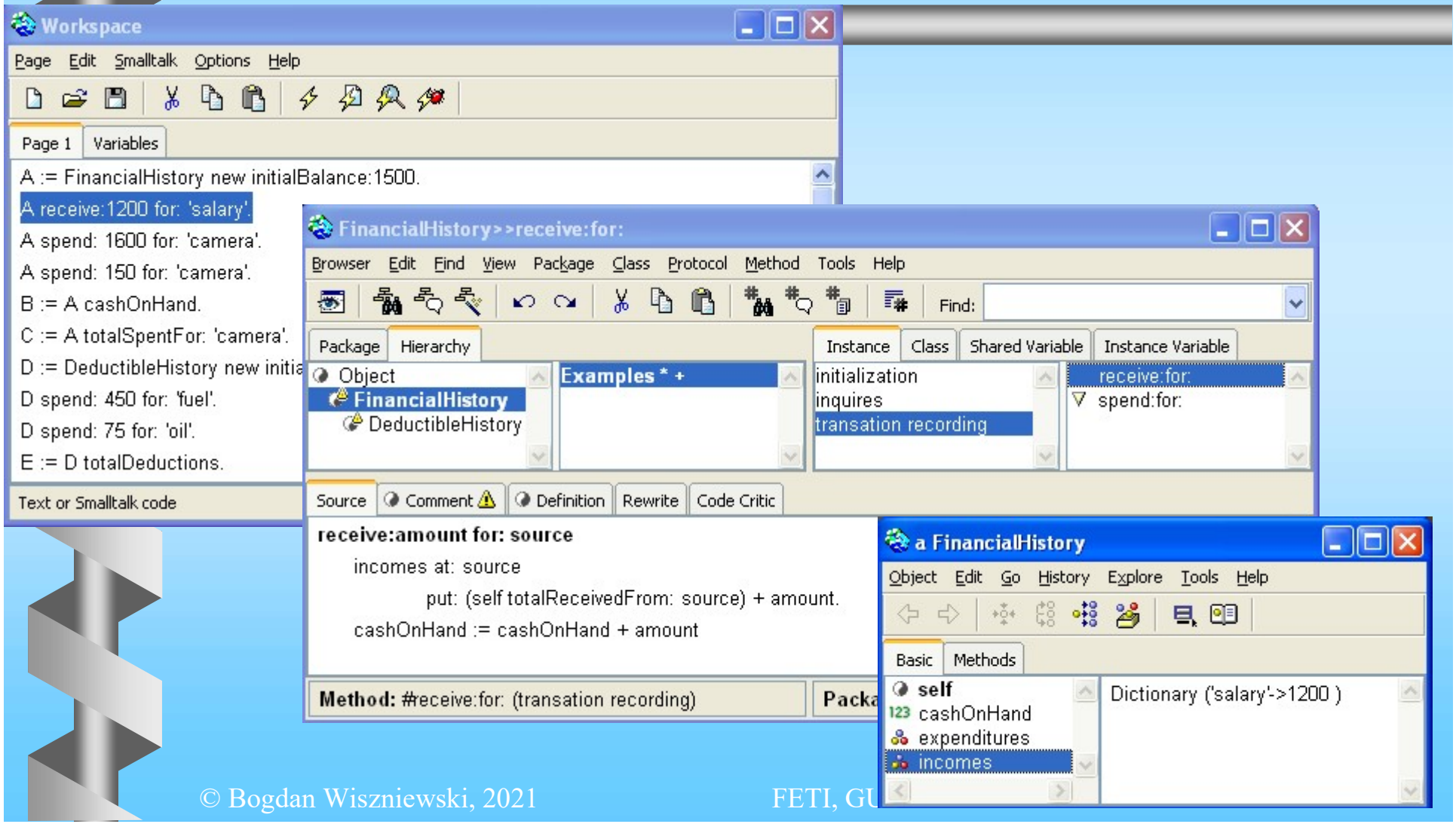
- **Class instance (object) methods:**
  - Object operations



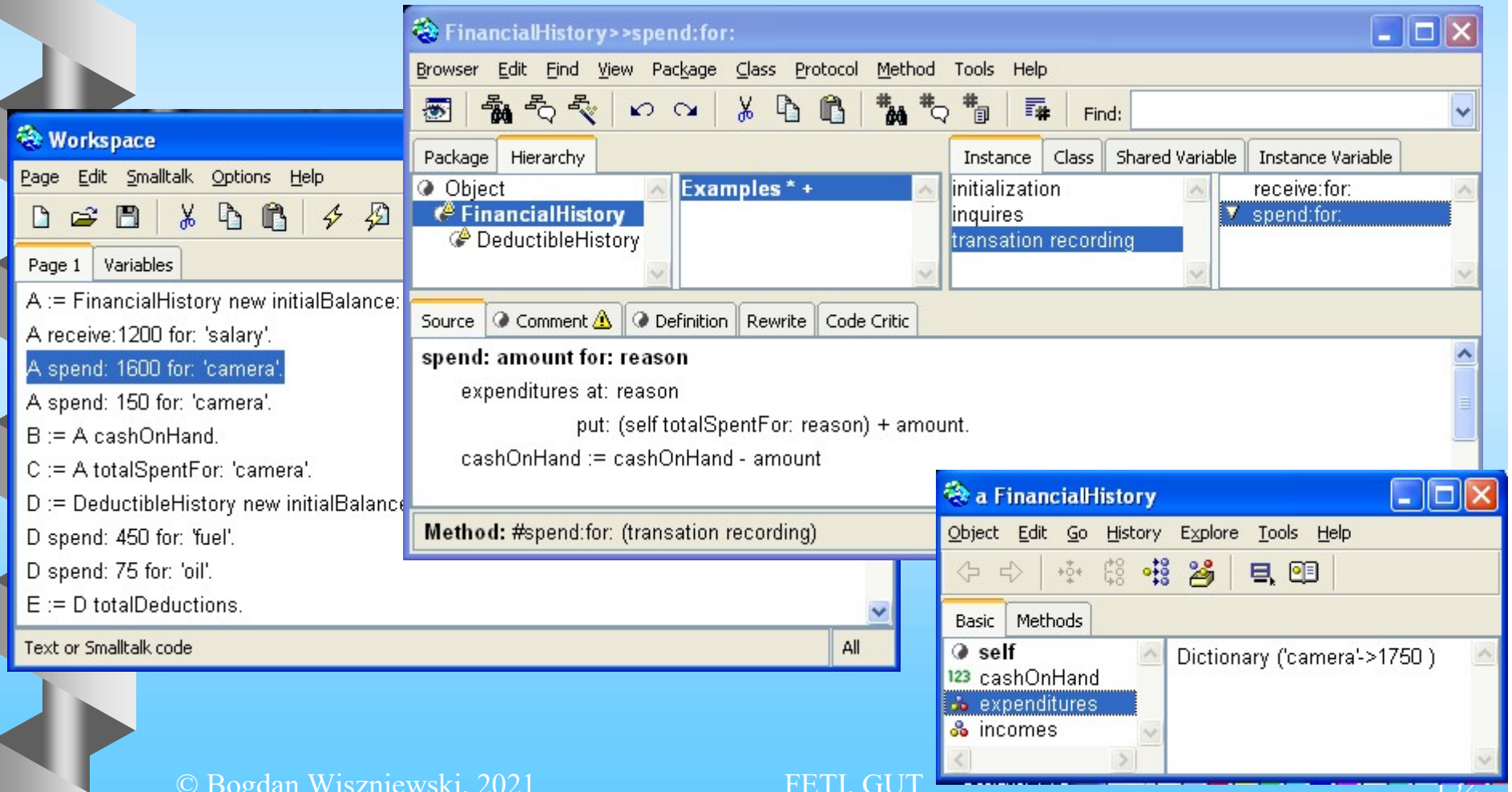
Example: ‘FinancialHistory’



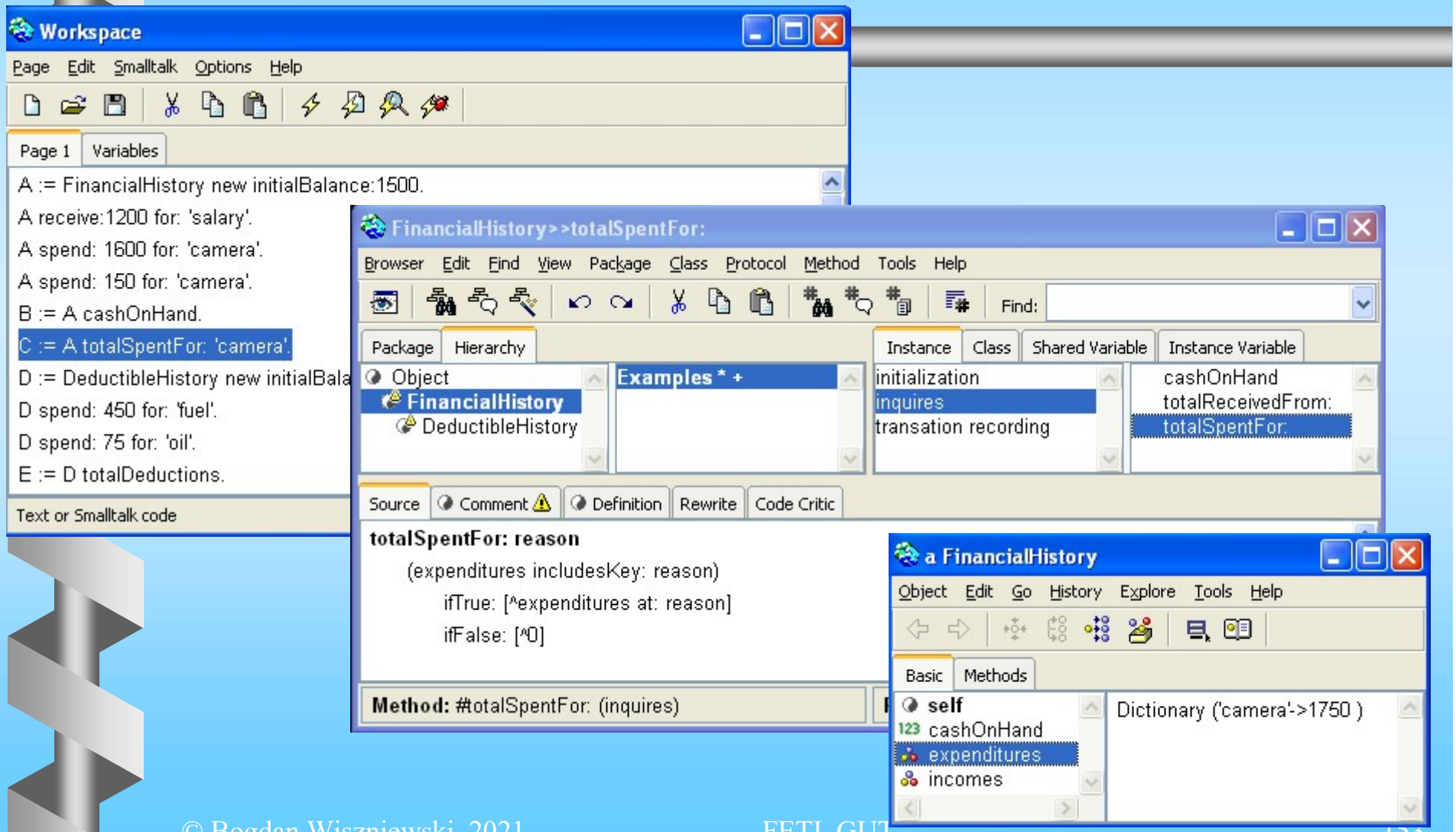
Example: ‘FinancialHistory’



Example: ‘FinancialHistory’

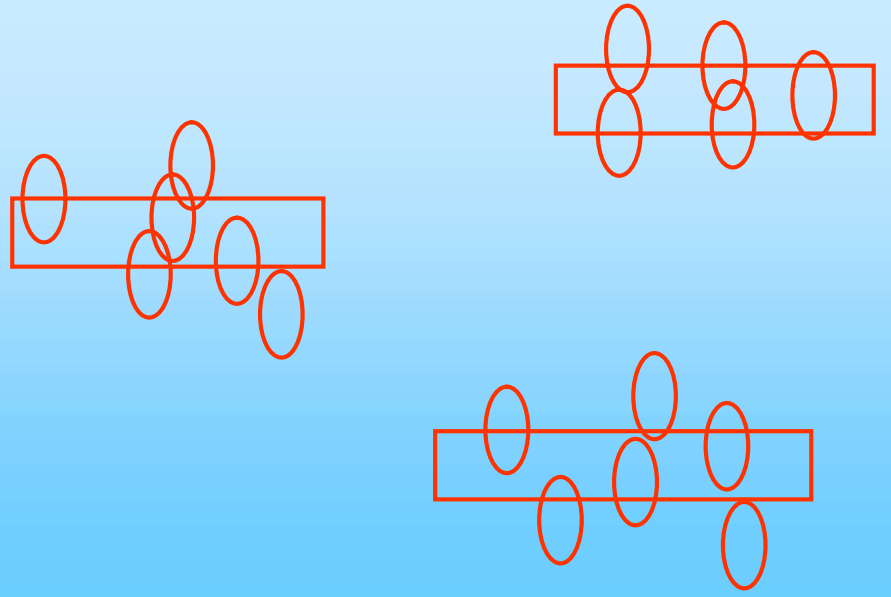


Example: ‘FinancialHistory’



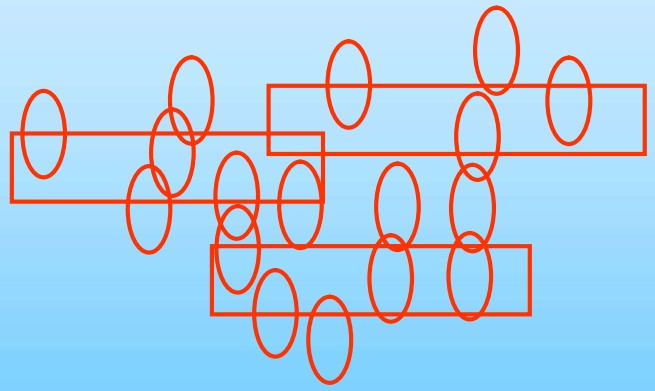
The concept of a class

- **Clusters, classes without inheritance:**
  - Ada83, Modula-2



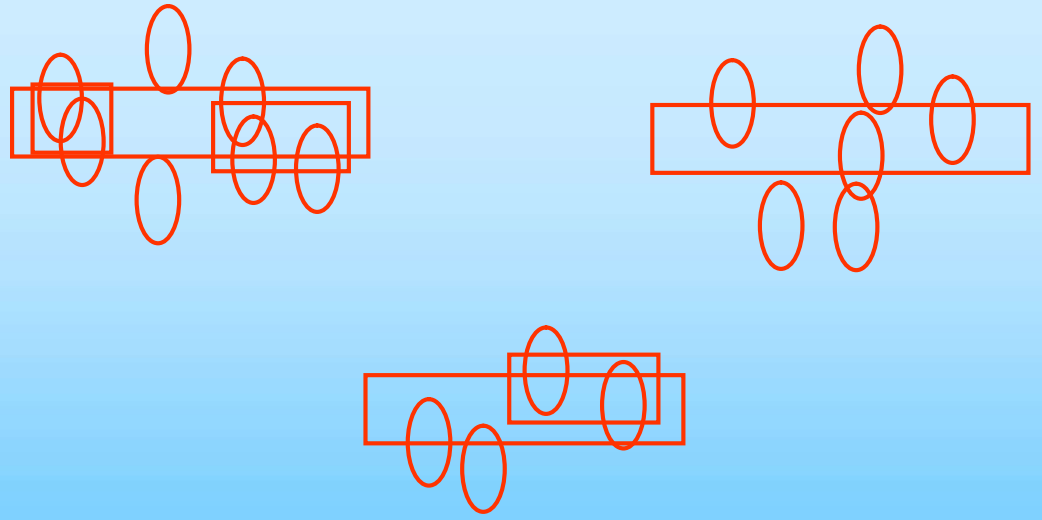
The concept of a class

- **Classes with multiple inheritance**
  - C++



The concept of a class

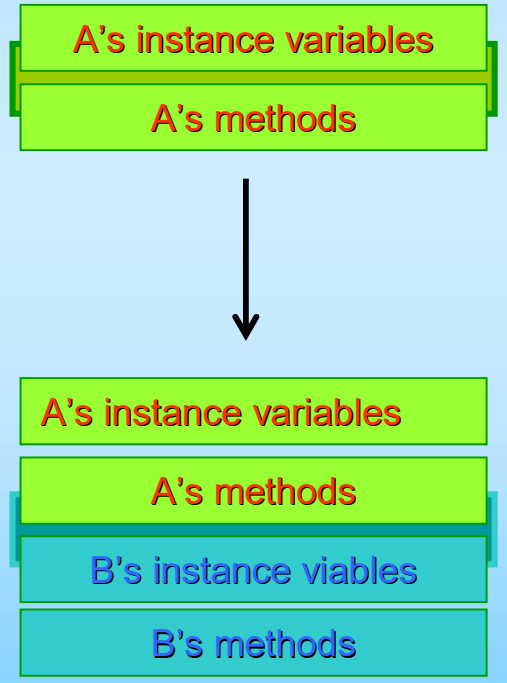
- **Classes with single inheritance (subclassing):**
  - Simula-67, Smalltalk-80



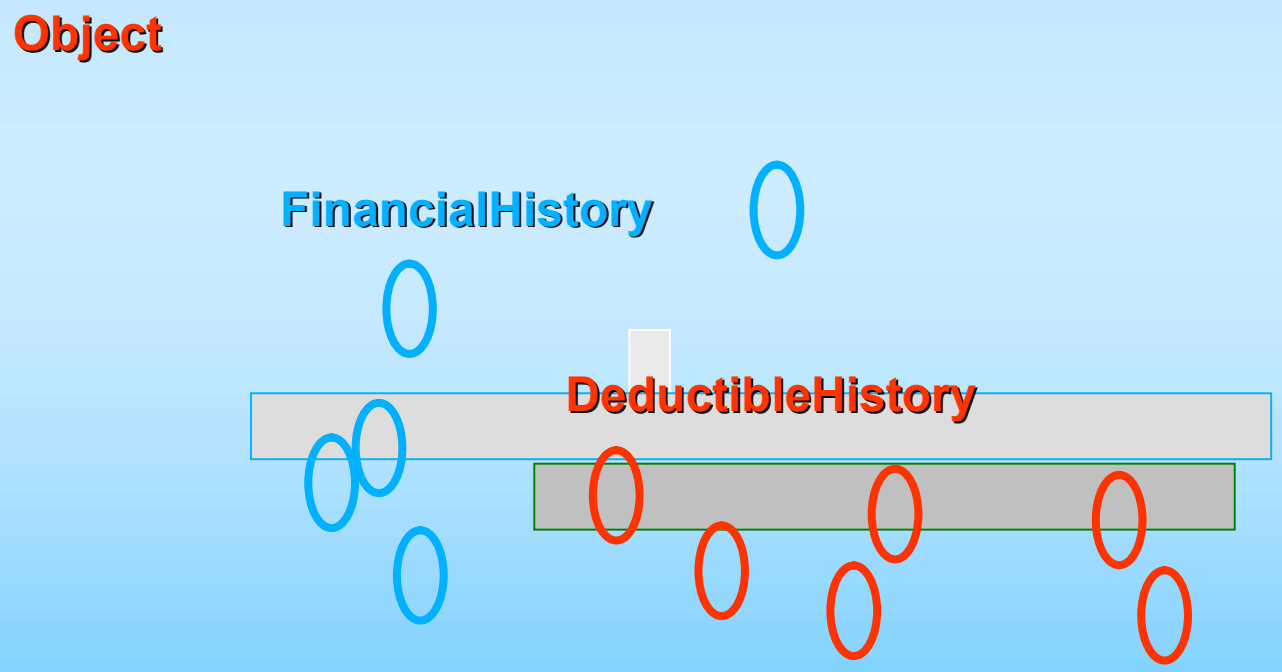
Inheritance

```
class A;
```

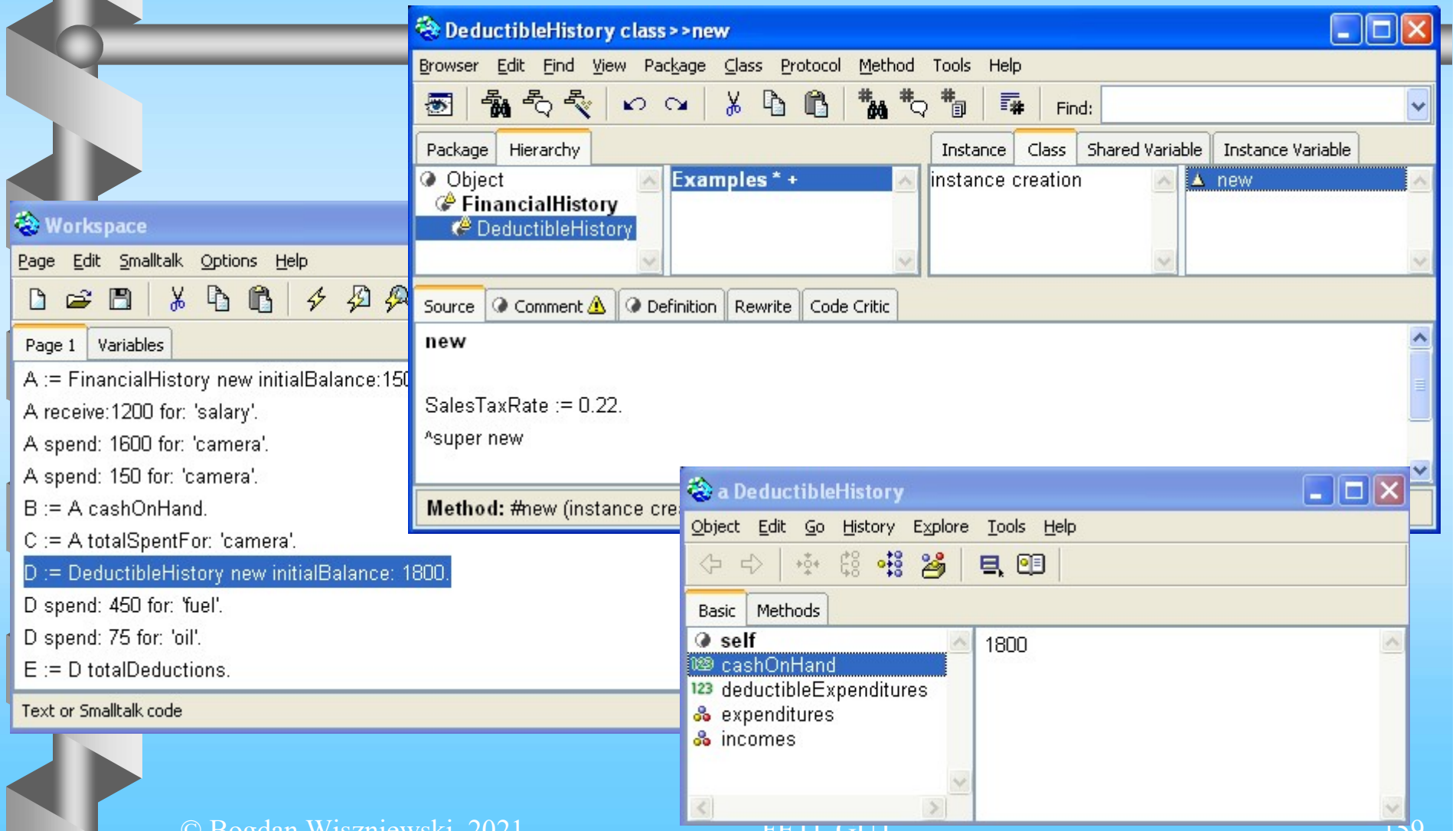
```
class B: public A;
class B: protected A;
class B: private A;
```



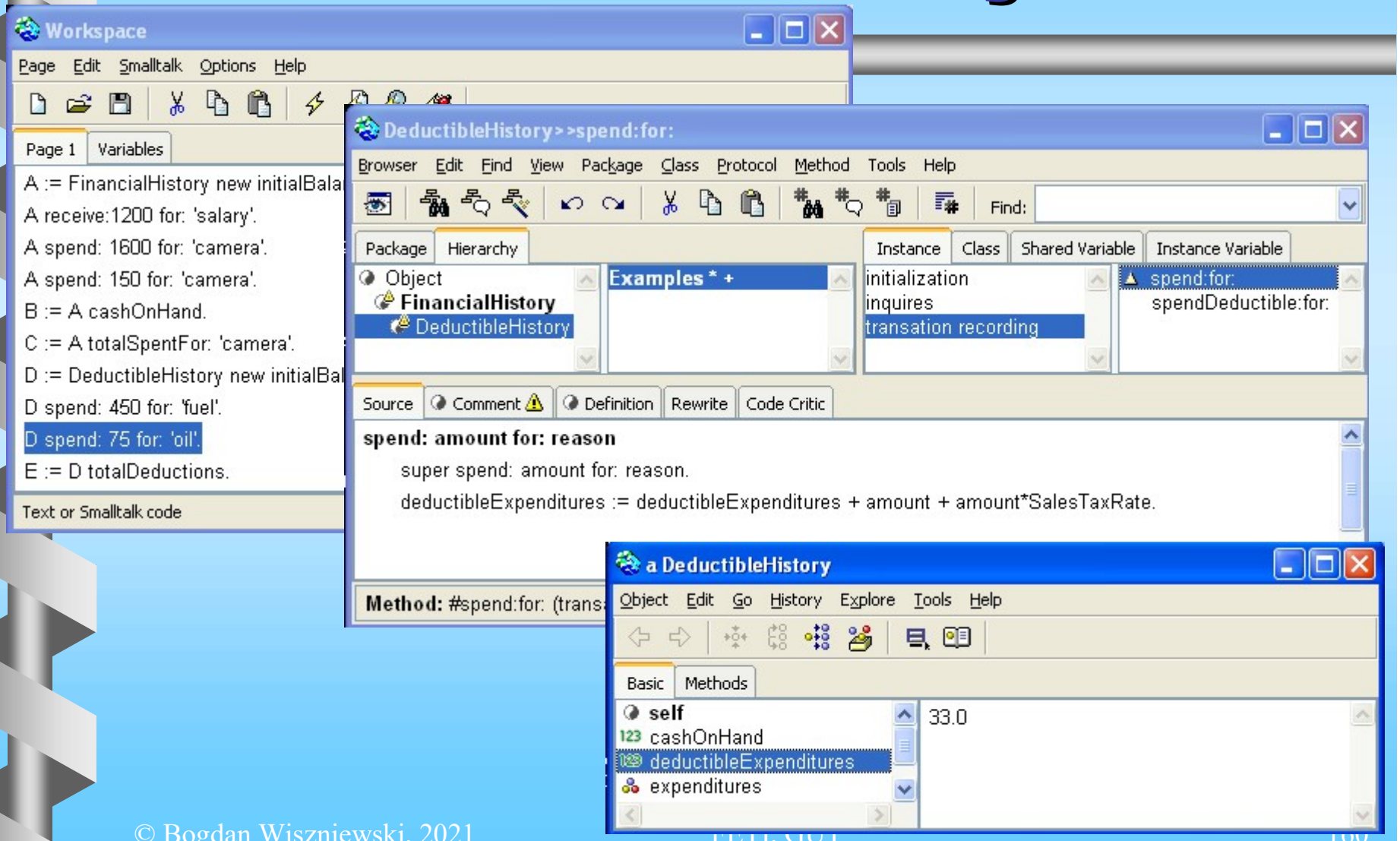
Example: ‘DeductibleHistory’



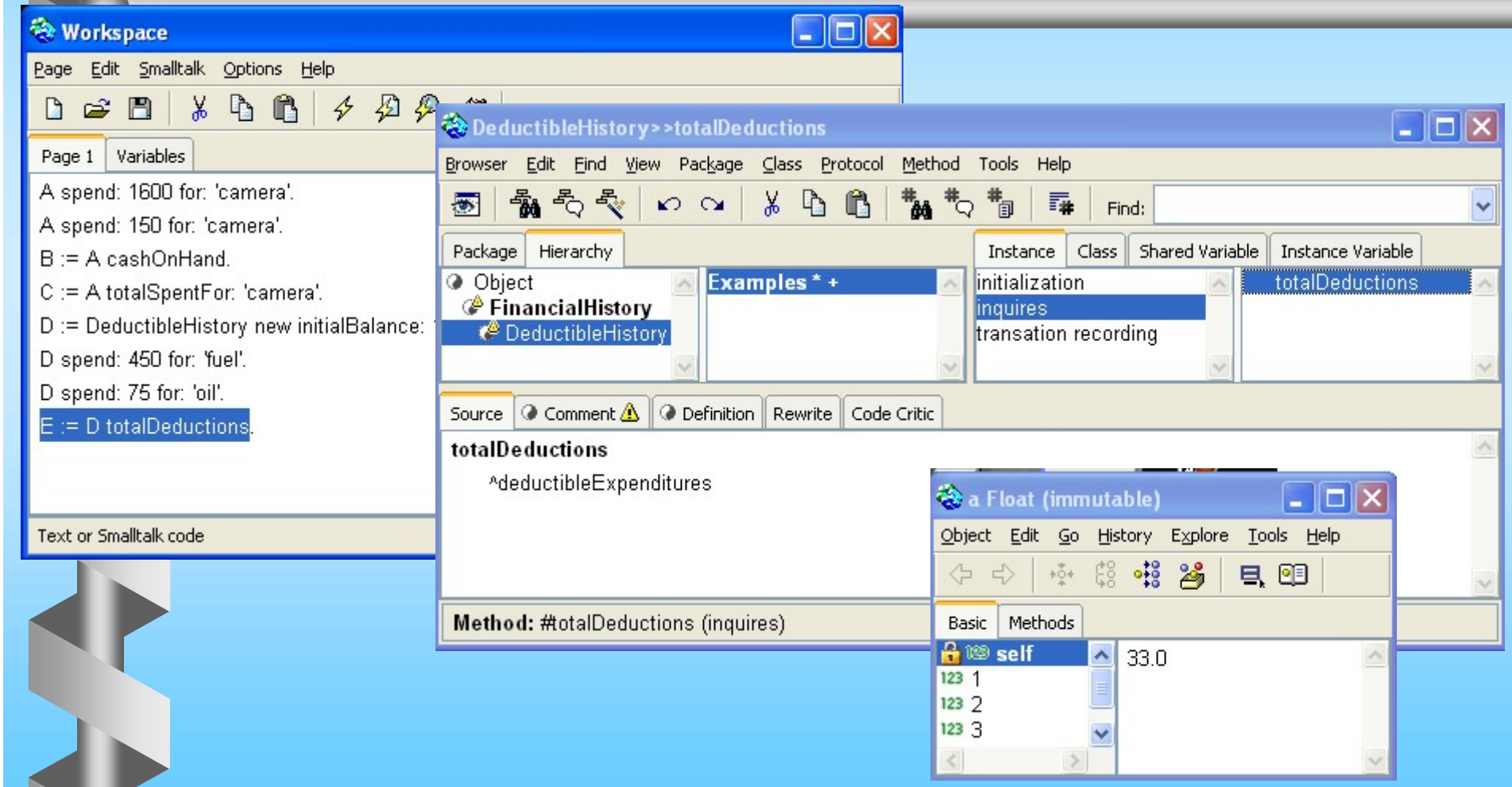
Example: ‘DeductibleHistory’



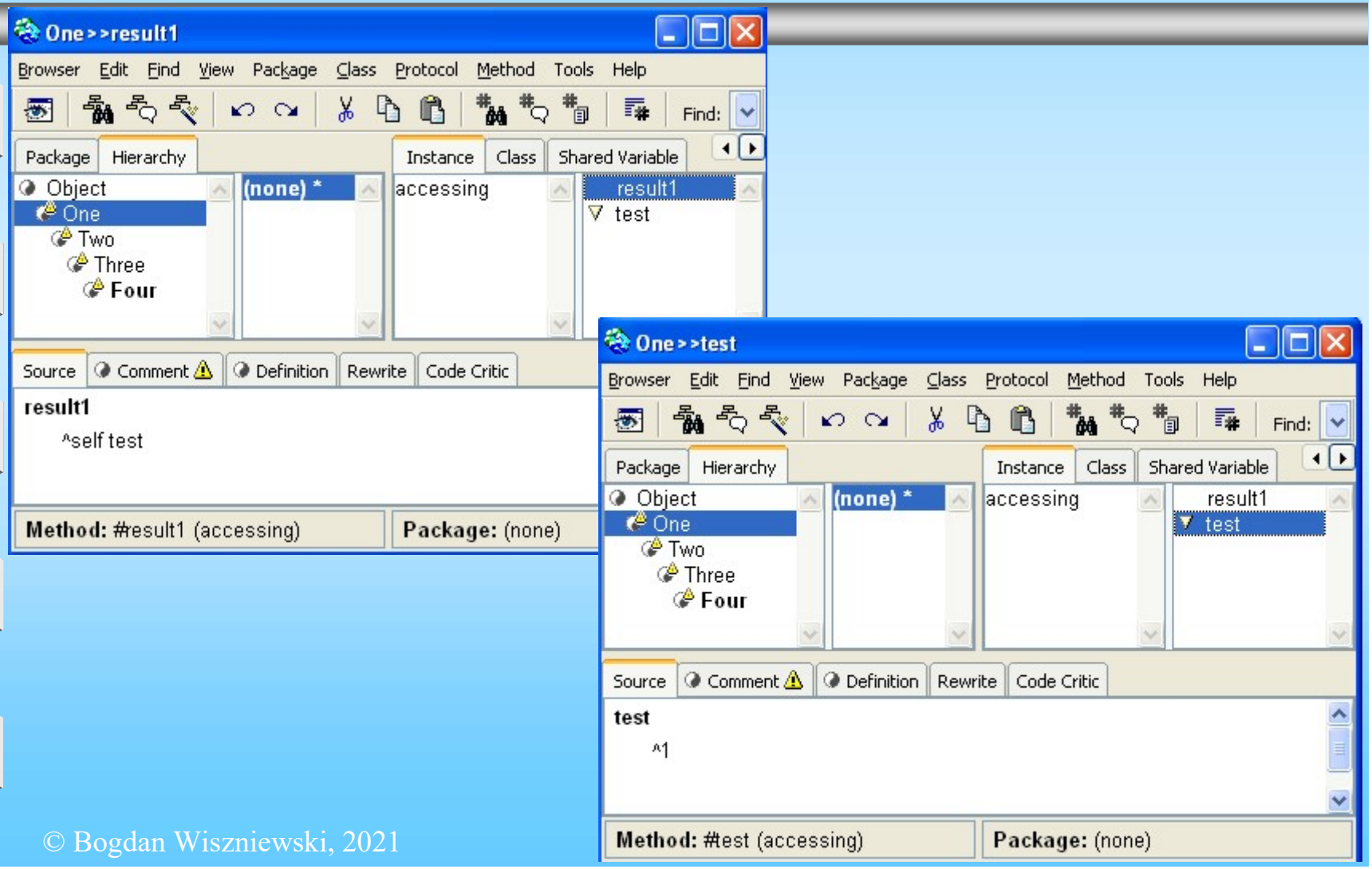
Example: ‘DeductibleHistory’



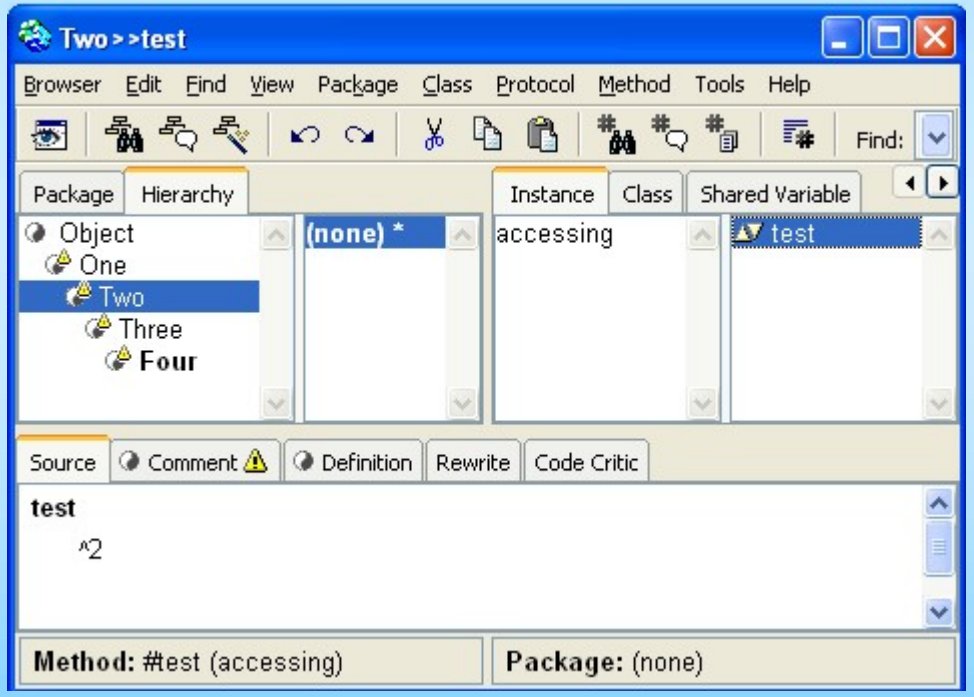
Example: ‘DeductibleHistory’



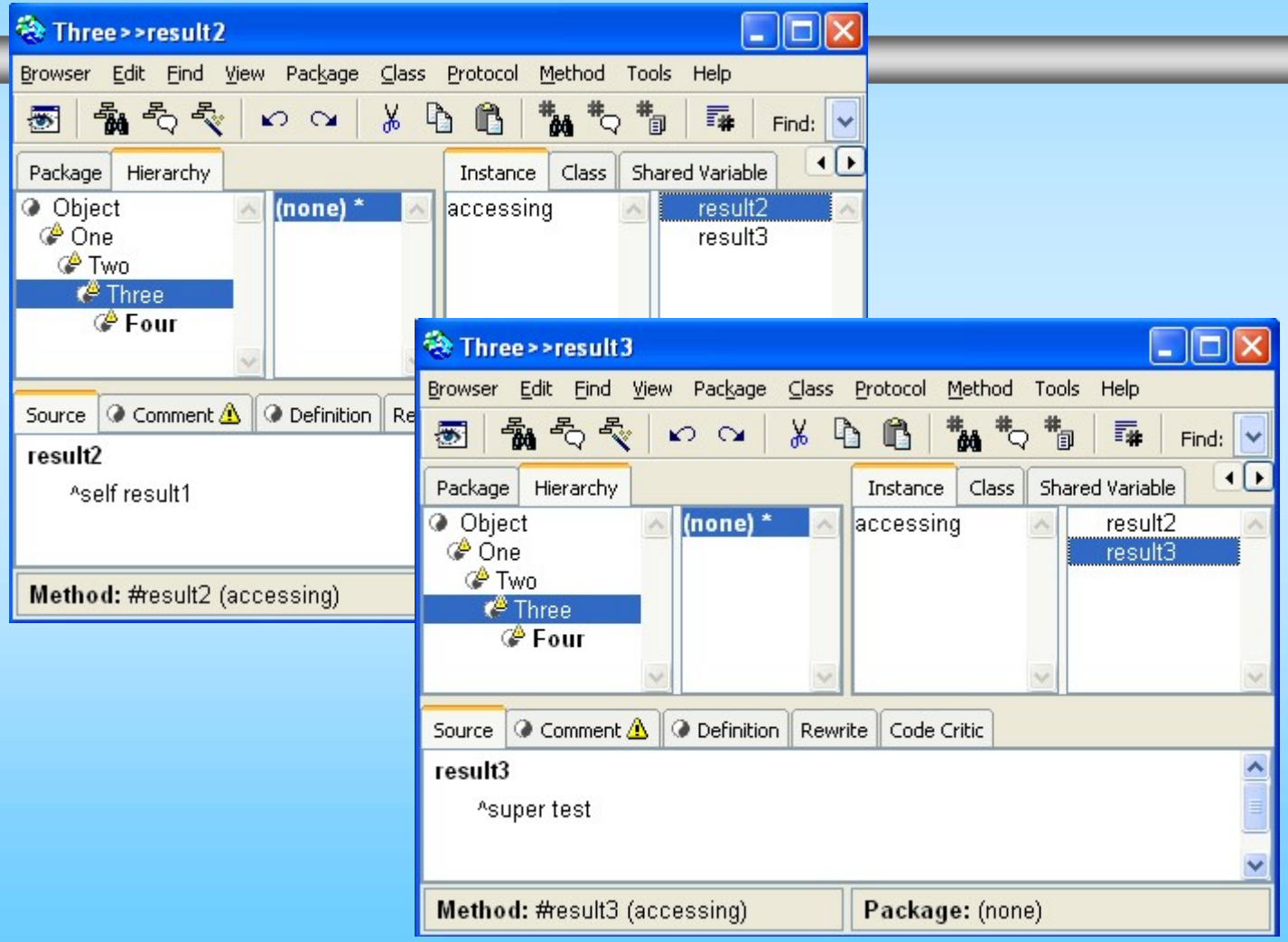
Information protection in the inheritance hierarchy



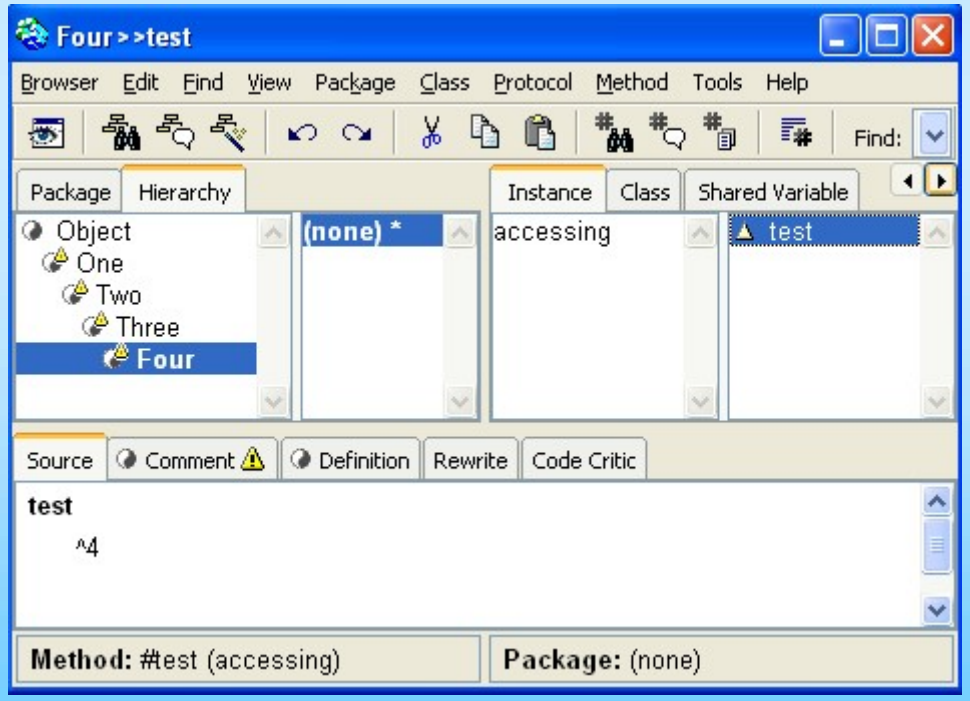
Information protection in the inheritance hierarchy



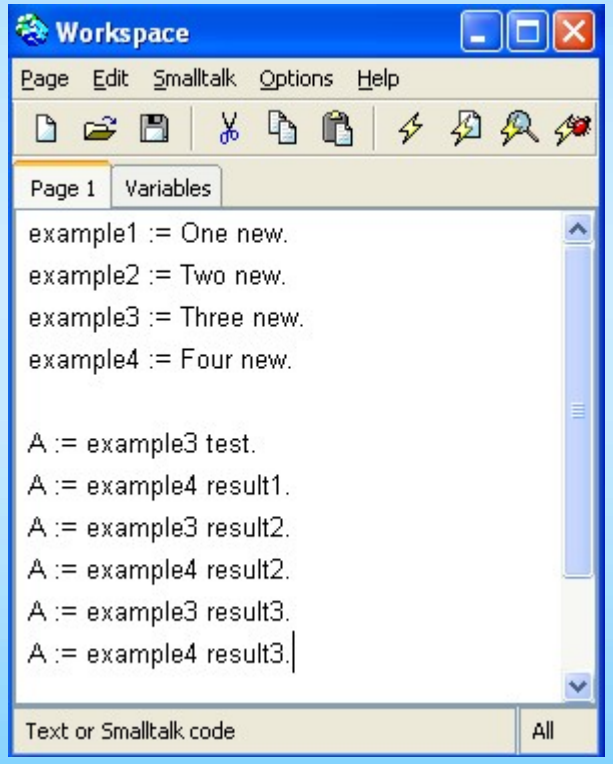
Information protection in the inheritance hierarchy



Information protection in the inheritance hierarchy



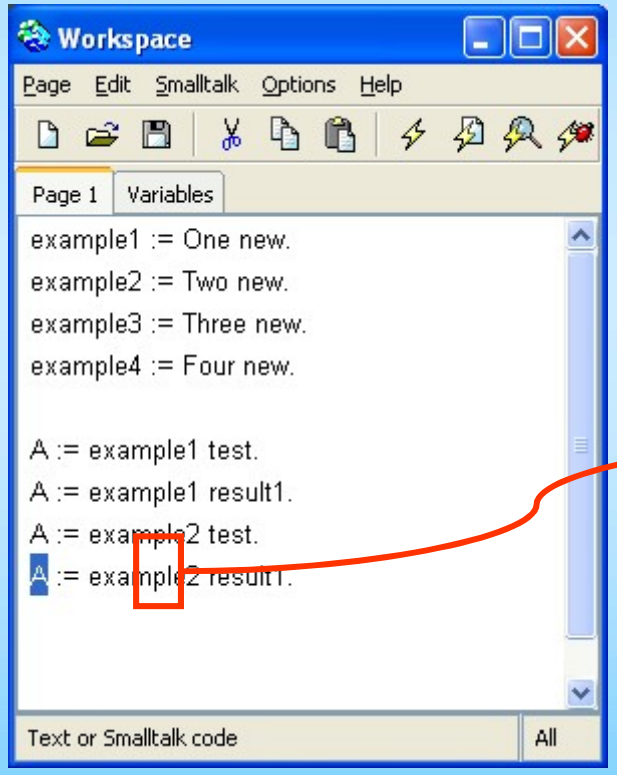
Information protection in the inheritance hierarchy



Smalltalk

```
...
!One methodsFor: 'accessing'!
result1
   ^self test!
test
   ^1! !

!Two methodsFor: 'accessing'!
test
   ^2! !

!Three methodsFor: 'accessing'!
result2
   ^self result1!
result3
   ^super test! !

!Four methodsFor: 'accessing'!
test
   ^4! !
```

Information protection in the inheritance hierarchy

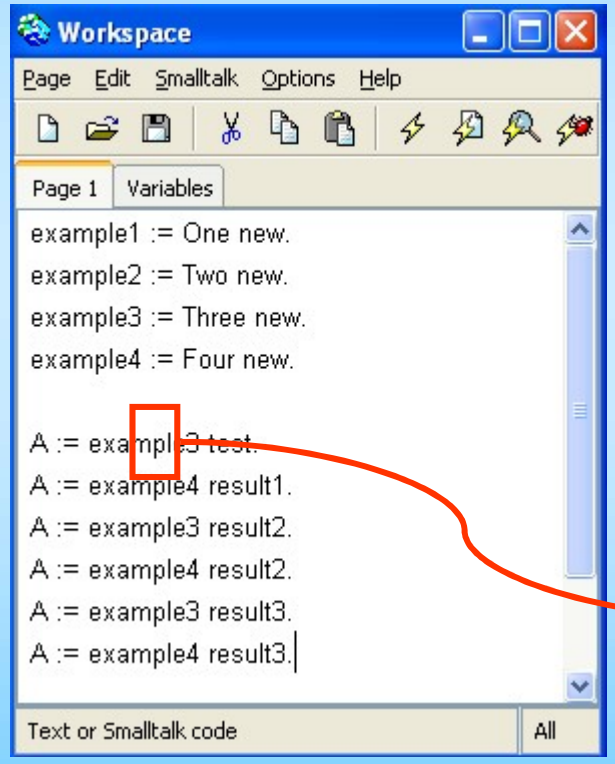


Smalltalk

```
...
!One methodsFor: 'accessing'!
result1
  ^self test!
test
  ^1! !
!Two methodsFor: 'accessing'!
test
  ^2! !
!Three methodsFor: 'accessing'!
result2
  ^self result1!
result3
  ^super test! !
!Four methodsFor: 'accessing'!
test
  ^4! !
```

2

Information protection in the inheritance hierarchy

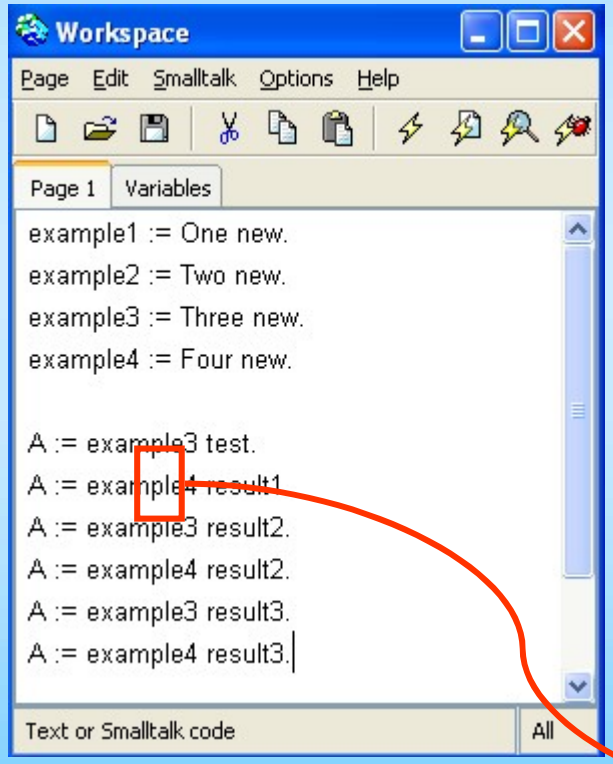


Smalltalk

```
...
!One methodsFor: 'accessing'!
result1
   ^self test!
test
   ^1! !
!Two methodsFor: 'accessing'!
test
   ^2! !
!Three methodsFor: 'accessing'!
result2
   ^self result1!
result3
   ^super test! !
!Four methodsFor: 'accessing'!
test
   ^4! !
```

2

Information protection in the inheritance hierarchy

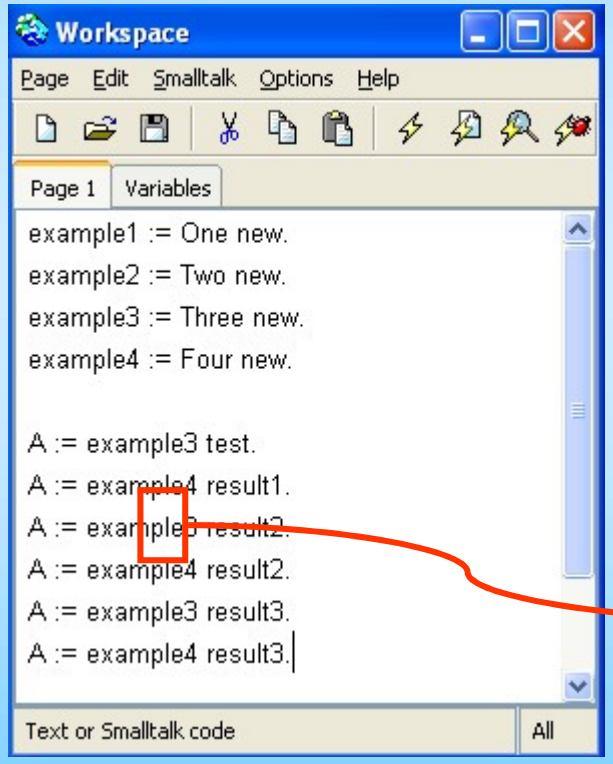


Smalltalk

```
...
!One methodsFor: 'accessing'!
result1
   ^self test!
test
   ^1! !

!Two methodsFor: 'accessing'!
test
   ^2! !

!Three methodsFor: 'accessing'!
result2
   ^self result1!
result3
   ^super test! !

!Four methodsFor: 'accessing'!
test
   ^4! !
```

4

Information protection in the inheritance hierarchy

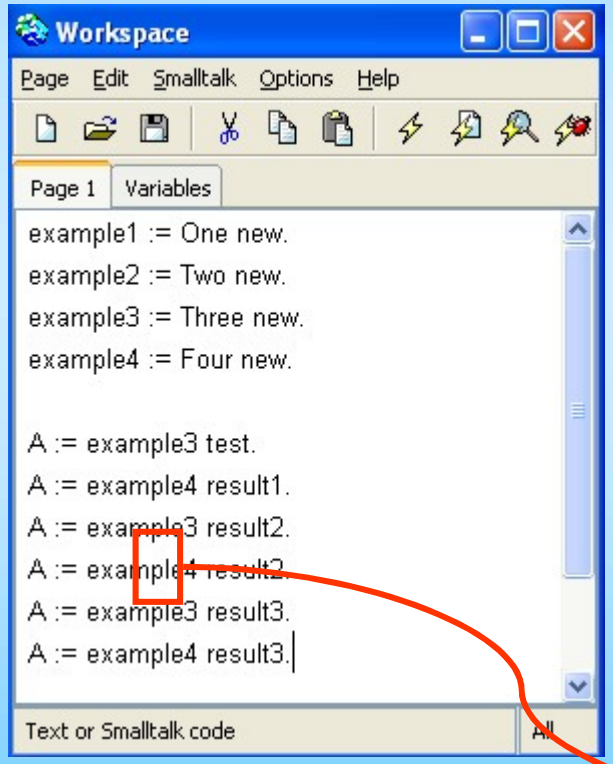


Smalltalk

```
...
!One methodsFor: 'accessing'!
result1
  ^self test!
test
  ^1! !
!Two methodsFor: 'accessing'!
test
  ^2! !
!Three methodsFor: 'accessing'!
result2
  ^self result1!
result3
  ^super test! !
!Four methodsFor: 'accessing'!
test
  ^4! !
```

2

Information protection in the inheritance hierarchy

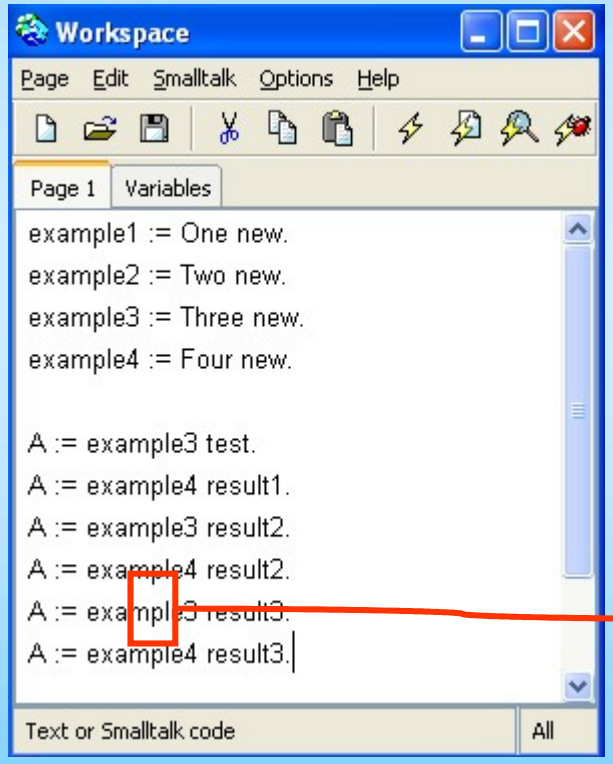


Smalltalk

```
...
!One methodsFor: 'accessing'!
result1
  ^self test!
test
  ^1! !

!Two methodsFor: 'accessing'!
test
  ^2! !

!Three methodsFor: 'accessing'!
result2
  ^self result1!
result3
  ^super test! !

!Four methodsFor: 'accessing'!
test
  ^4! !
```

4

Information protection in the inheritance hierarchy

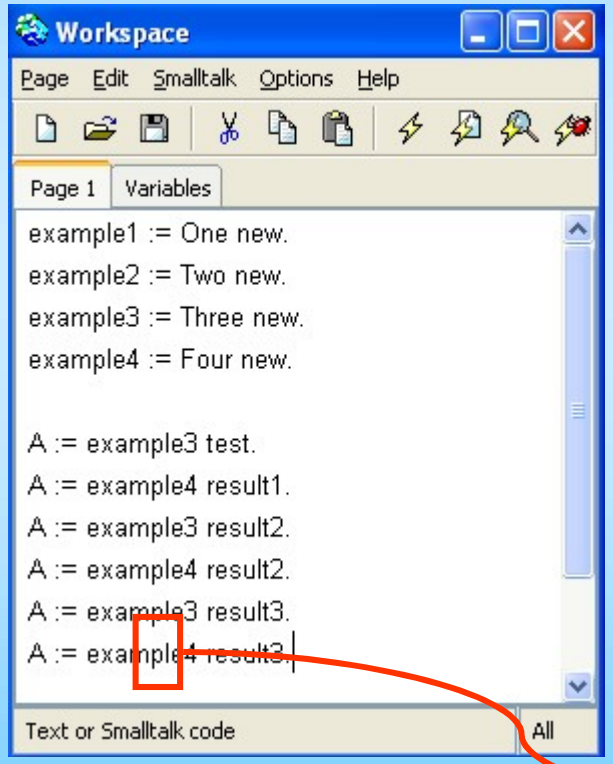


Smalltalk

```
...
!One methodsFor: 'accessing'!
result1
   ^self test!
test
   ^1! !

!Two methodsFor: 'accessing'!
test
   ^2! !

!Three methodsFor: 'accessing'!
result2
   ^self result1!
result3
   ^super test! !

!Four methodsFor: 'accessing'!
test
   ^4! !
```

2

Information protection in the inheritance hierarchy



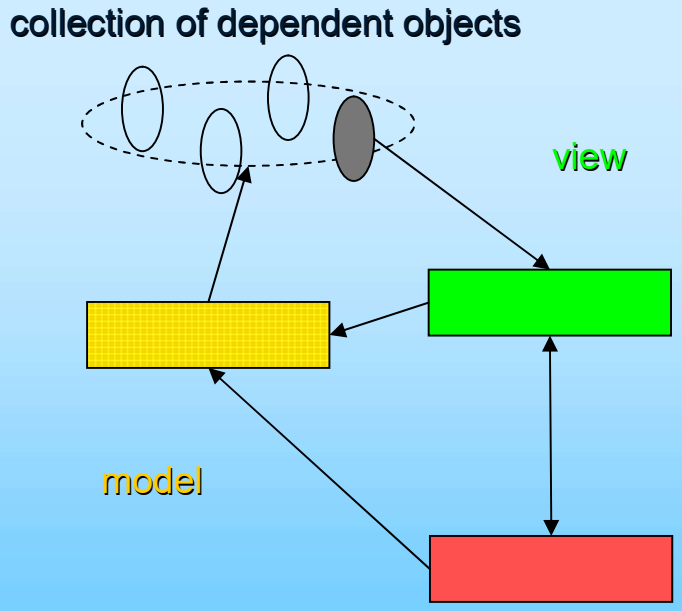
Smalltalk

```
...
!One methodsFor: 'accessing'!
result1
   ^self test!
test
   ^1! !

!Two methodsFor: 'accessing'!
test
   ^2! !

!Three methodsFor: 'accessing'!
result2
   ^self result1!
result3
   ^super test! !

!Four methodsFor: 'accessing'!
test
   ^4! !
```

2

Programming in graphics?

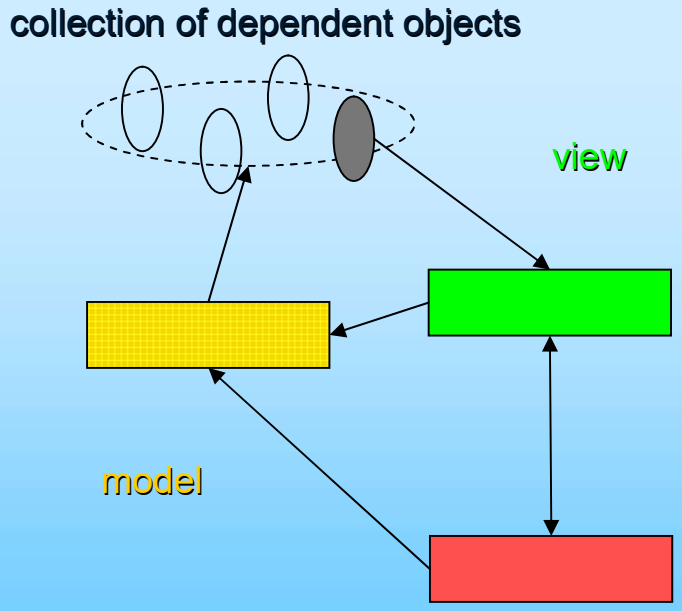
- **Eg. the model-view-controller (MVC) programming pattern**
  - Semantics of the application + I/O



- **the model does not depend on any implementation detail of the interface (the view or the driver)**
- **one model can have multiple dependent views**

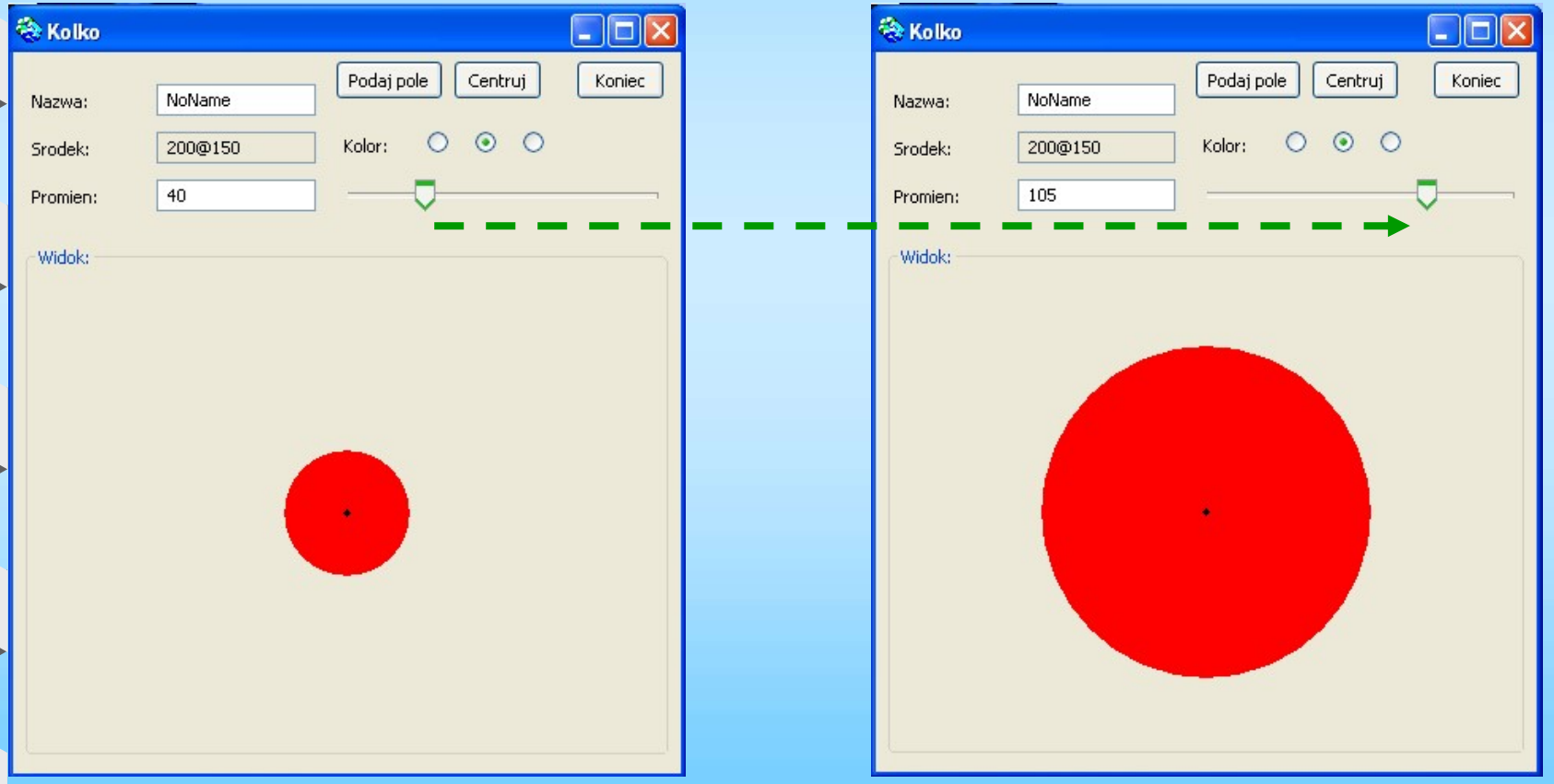
Programming in graphics?

- **Eg. the model-view-controller (MVC) programming pattern**
  - Semantics of the application + I/O

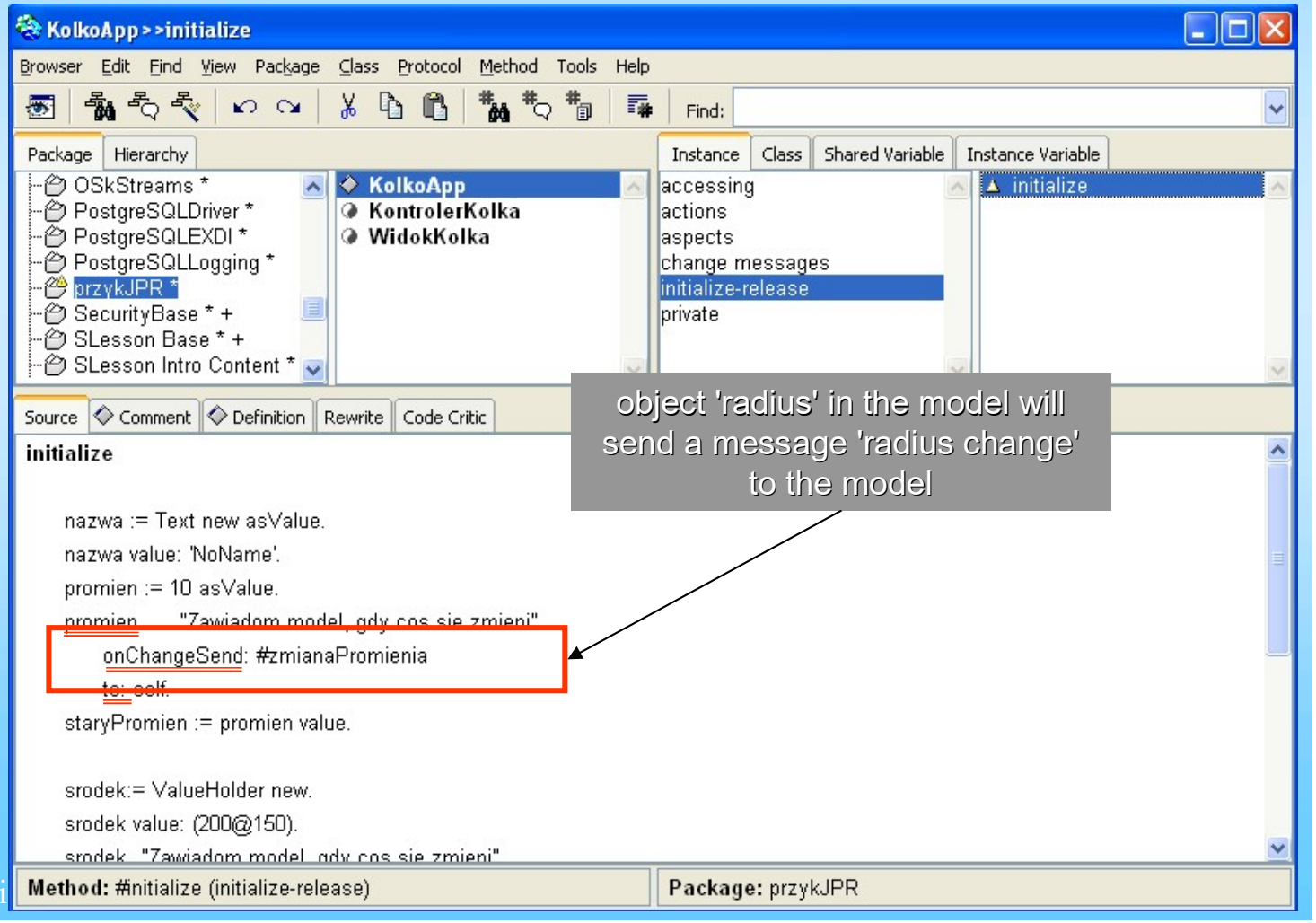


- **Changing any aspect of the model (object state) causes a message to be sent to the view**
- **The view refreshes itself for new values of aspects that it is interested in**
- **The controller detects events, changes model aspect values and notifies the view of model changes**

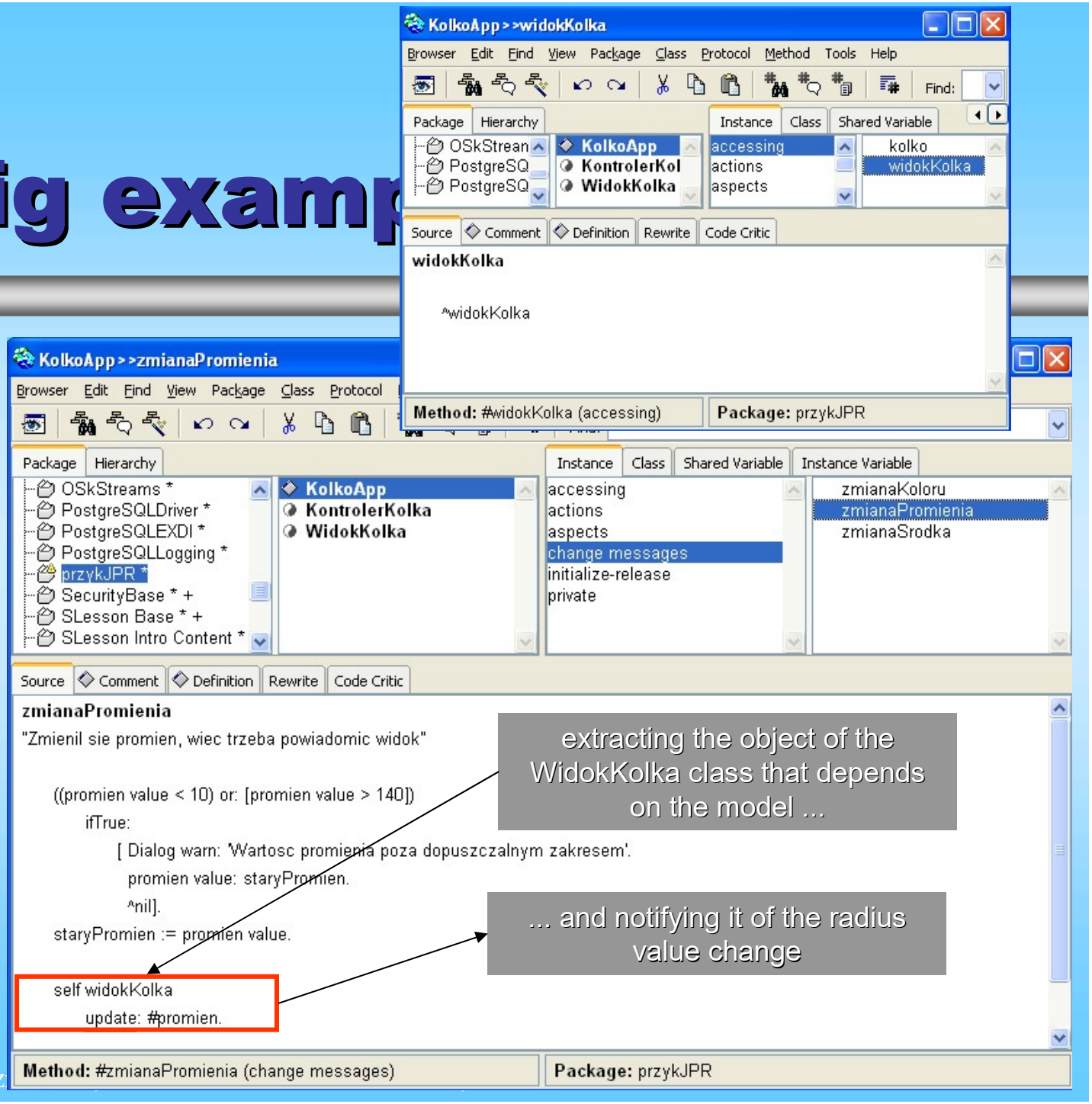
A big example of MVC



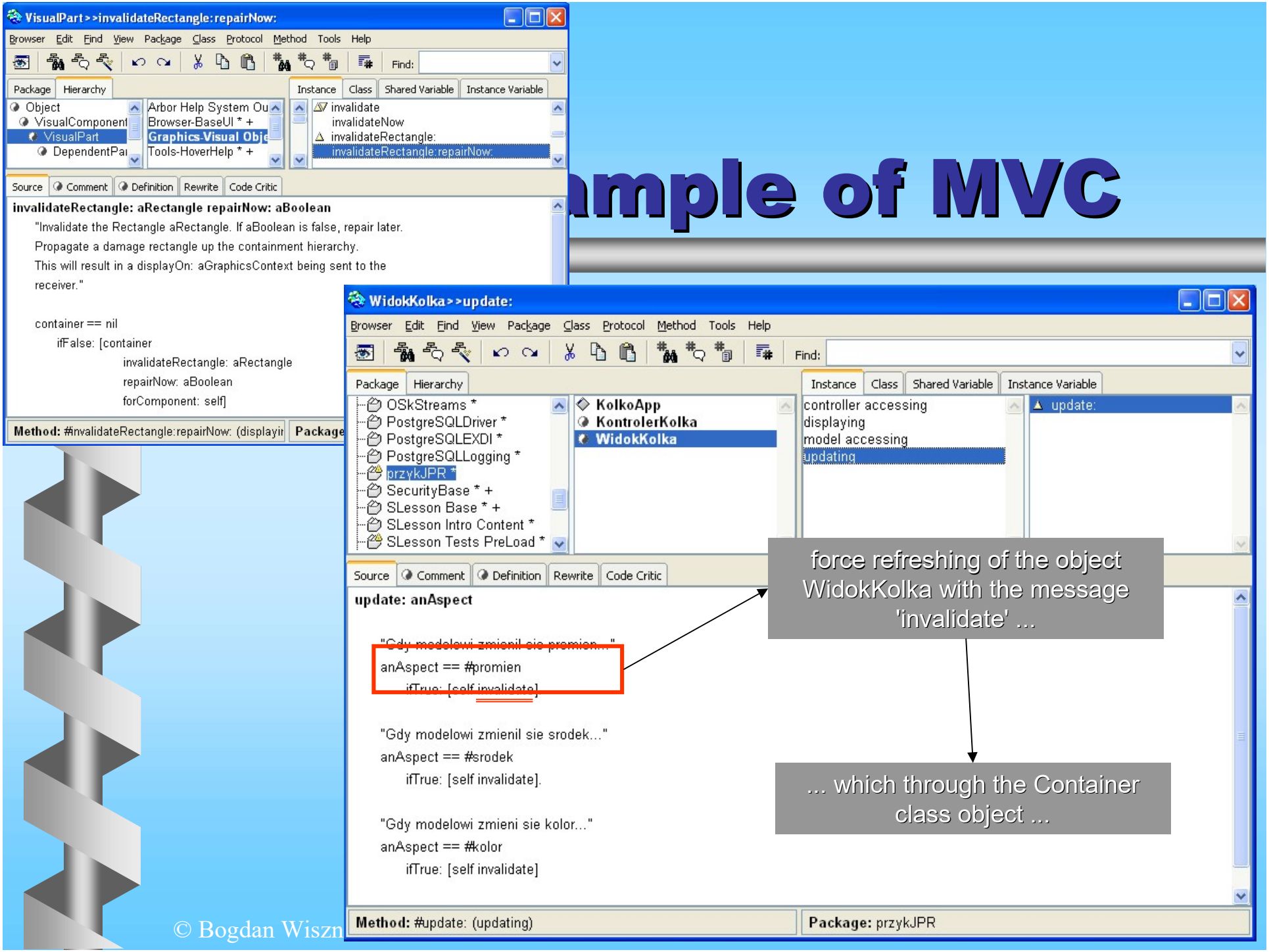
A big example of MVC



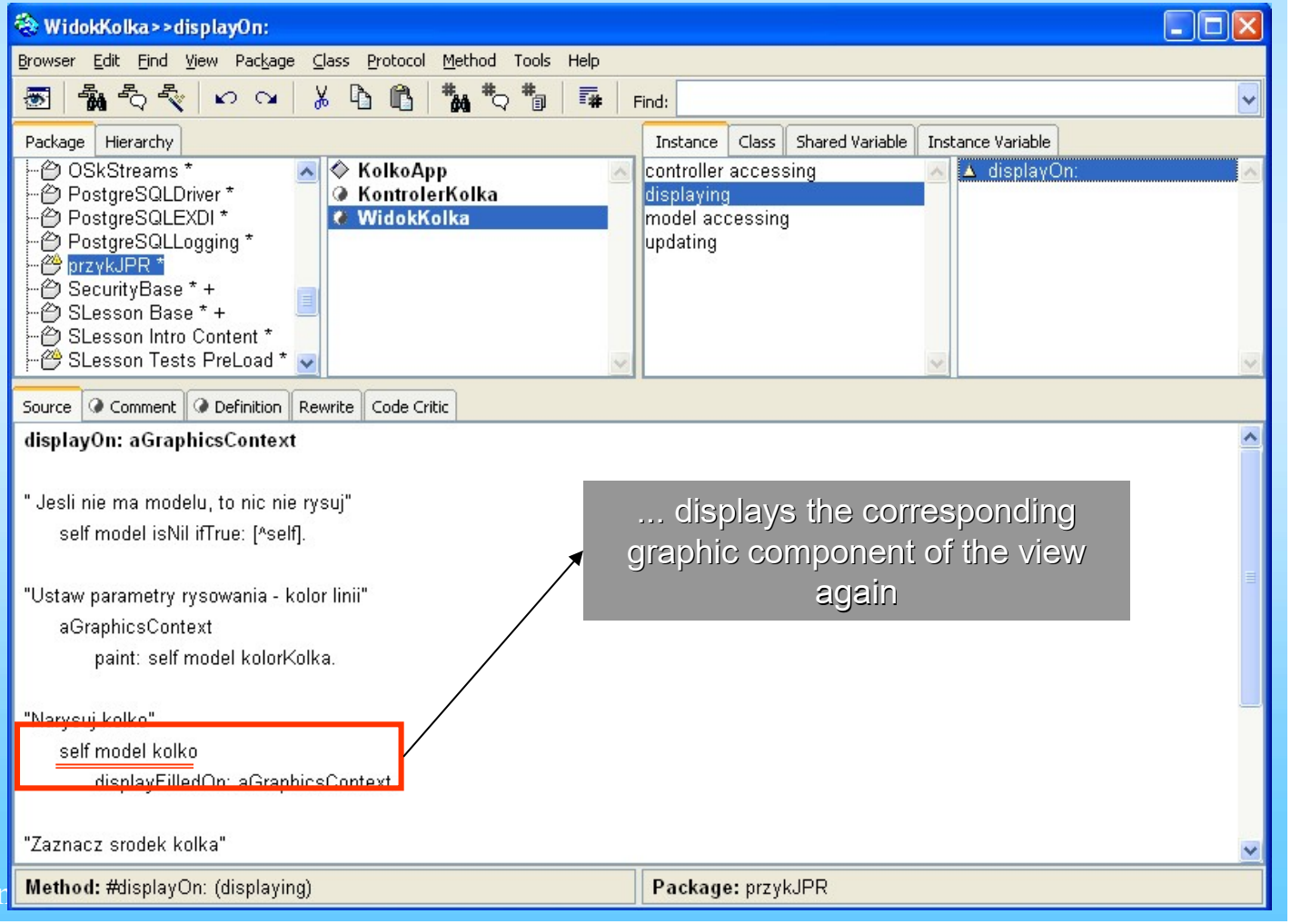
A big examp



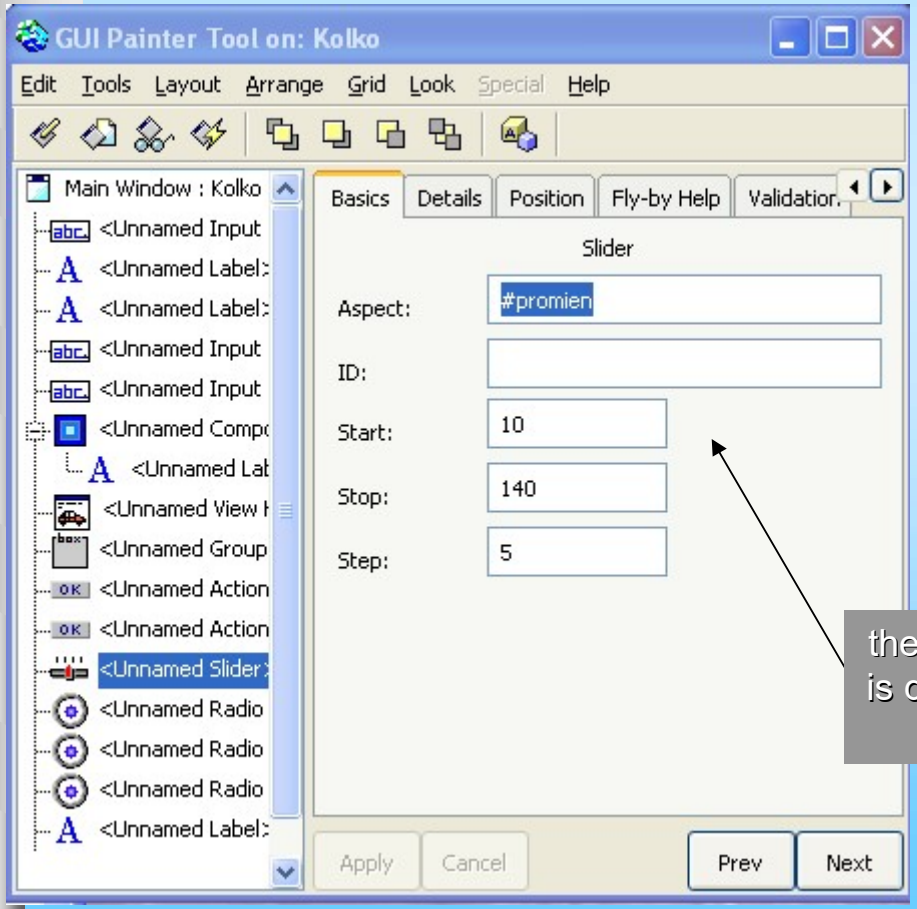
ample of MVC

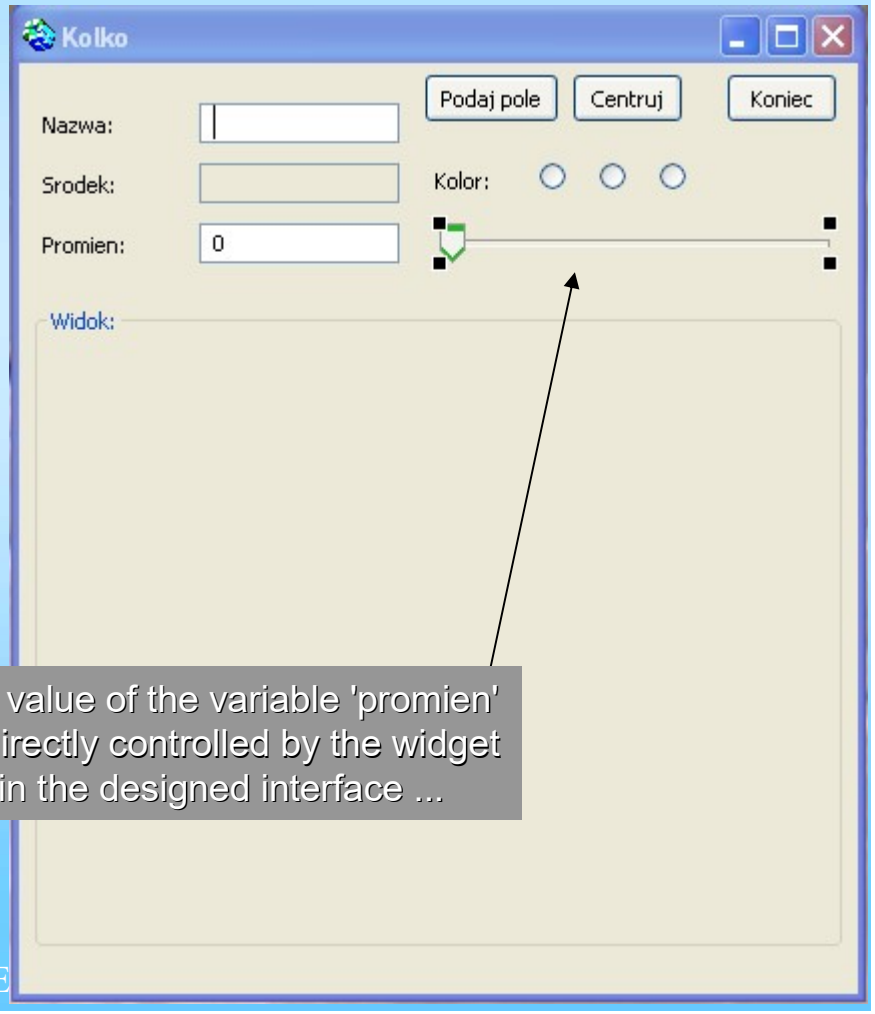


A big example of MVC



A big examp





the value of the variable 'promien' is directly controlled by the widget in the designed interface ...