SYSTEM MODELLING AND SYNTHESIS WITH HDL

DTEK0078

2022 Lecture 4

UNIVERSITY OF TURKU

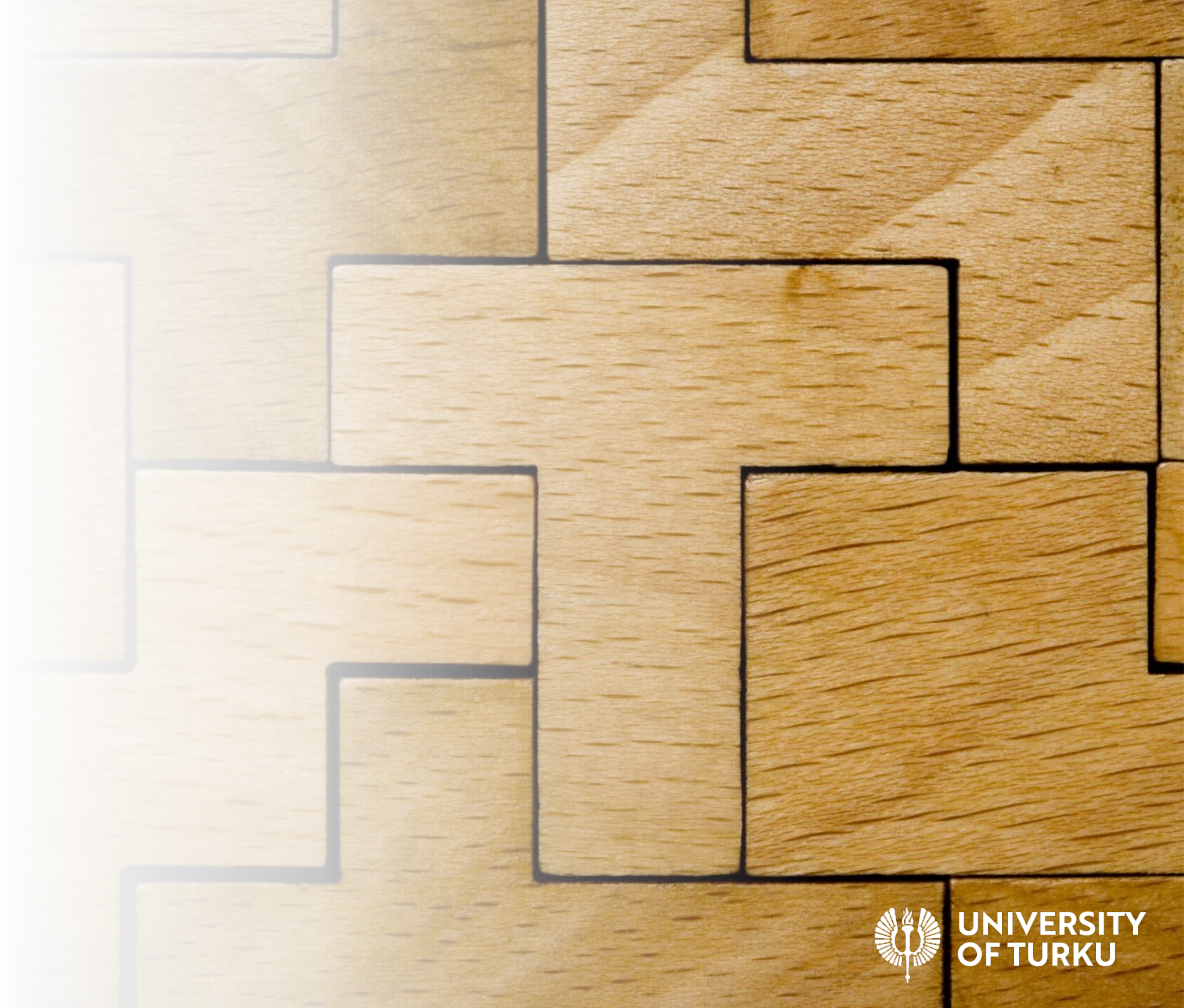
Objects and Data Types

VHDL is a strongly typed language

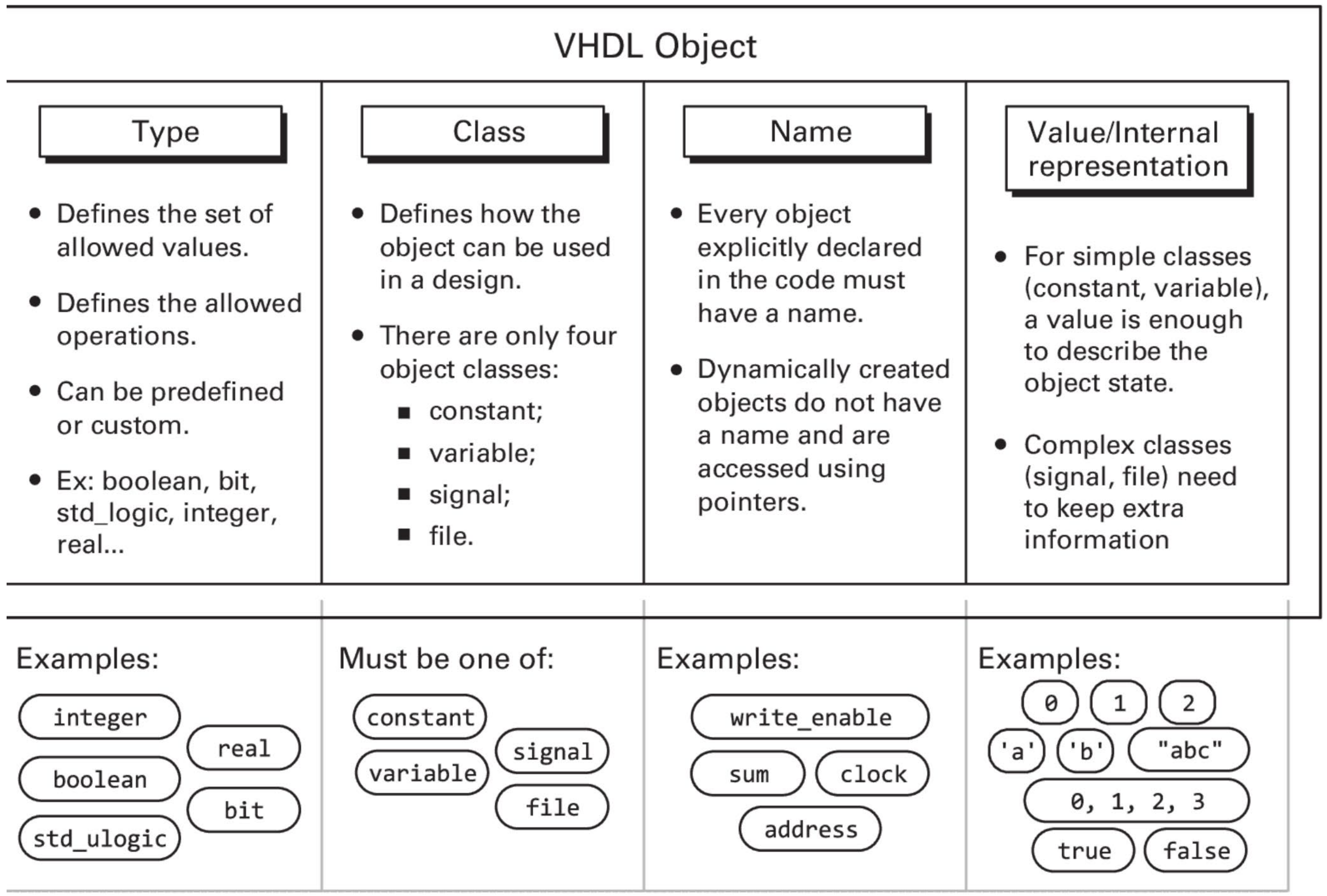
- Every object may only assume values of its nominated type
- The definition of each operation includes the types of values to which the operation may be applied

Take it as a good thing

- Compiler and other static analysis tools can perform many checks for us
- The compiler can also guarantee that an operation will never be invoked with the wrong type of argument



Objects in VHDL



VHDL Object

Type	Class	Name	Value/Internal representation
• Defines the set of allowed values. • Defines the allowed operations. • Can be predefined or custom. • Ex: boolean, bit, std_logic, integer, real...	• Defines how the object can be used in a design. • There are only four object classes: ■ constant; ■ variable; ■ signal; ■ file.	• Every object explicitly declared in the code must have a name. • Dynamically created objects do not have a name and are accessed using pointers.	• For simple classes (constant, variable), a value is enough to describe the object state. • Complex classes (signal, file) need to keep extra information
Examples: integer real boolean bit std_ulogic	Must be one of: constant signal variable file	Examples: write_enable sum clock address	Examples: 0 1 2 'a' 'b' "abc" 0, 1, 2, 3 true false

Four Classes of Objects

- **Signal**
  - An object with *current* and *future* values
  - Can be changed as many times as neccessary
- **Constant**
  - The value cannot be changed after initialisation
- **Variable**
  - An object with *current* value
  - Can be changed as many times as neccessary
- File

- An object is a named item in a VHDL model that has a value of a specified type

The terms class and object in VHDL and object-oriented programming (OOP)

Class, OOP language

A single construct that defines a type, specifies its data structure, and defines the operations for instances of the class.

Similar to a VHDL protected type, or to a record type if it could include routines and hide its data fields.

Class, VHDL

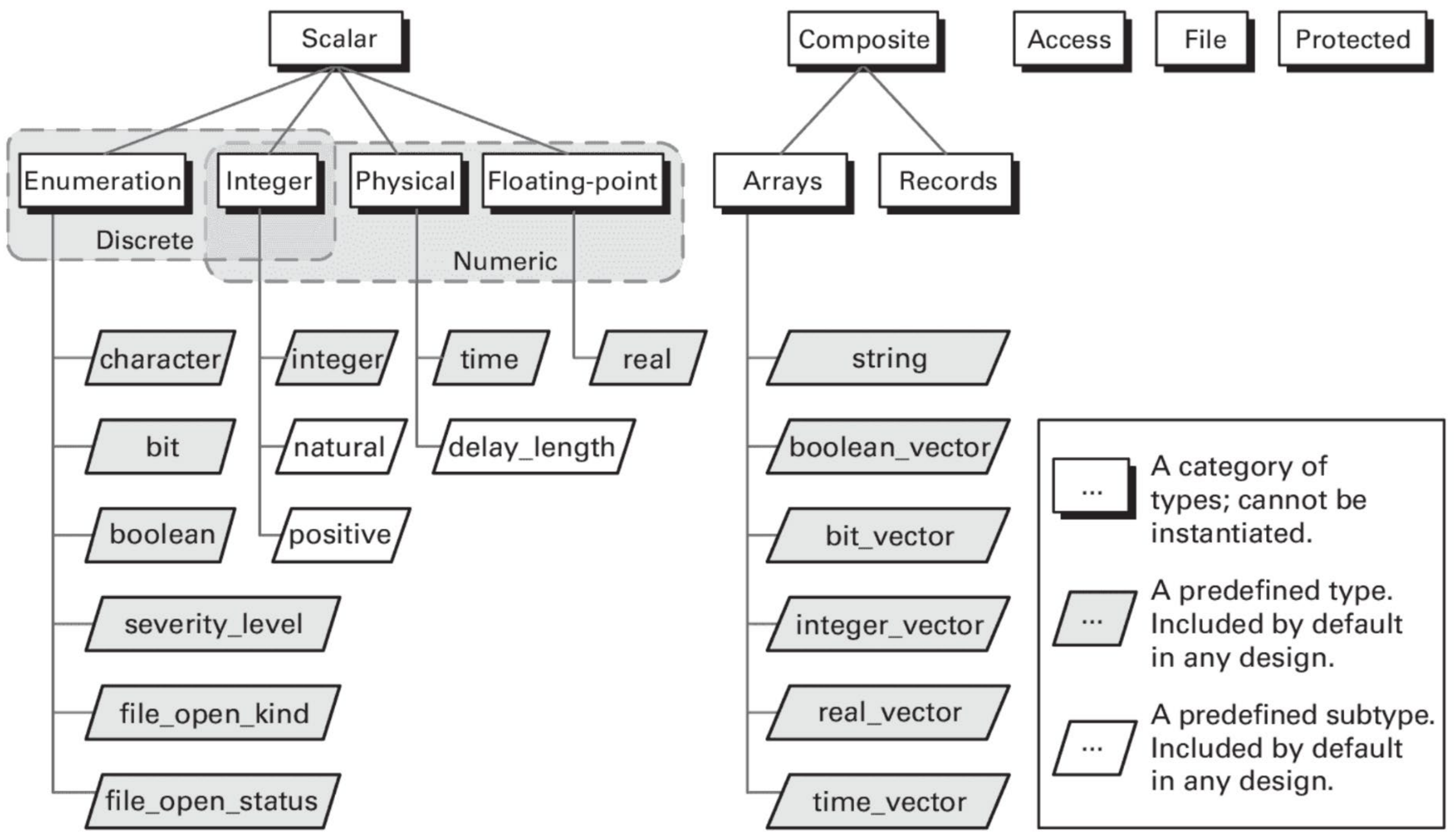
One of the four object classes: signal, variable, constant, or file. Determines how an object can be used, but does not define its data structure or operations.

Object, OOP language

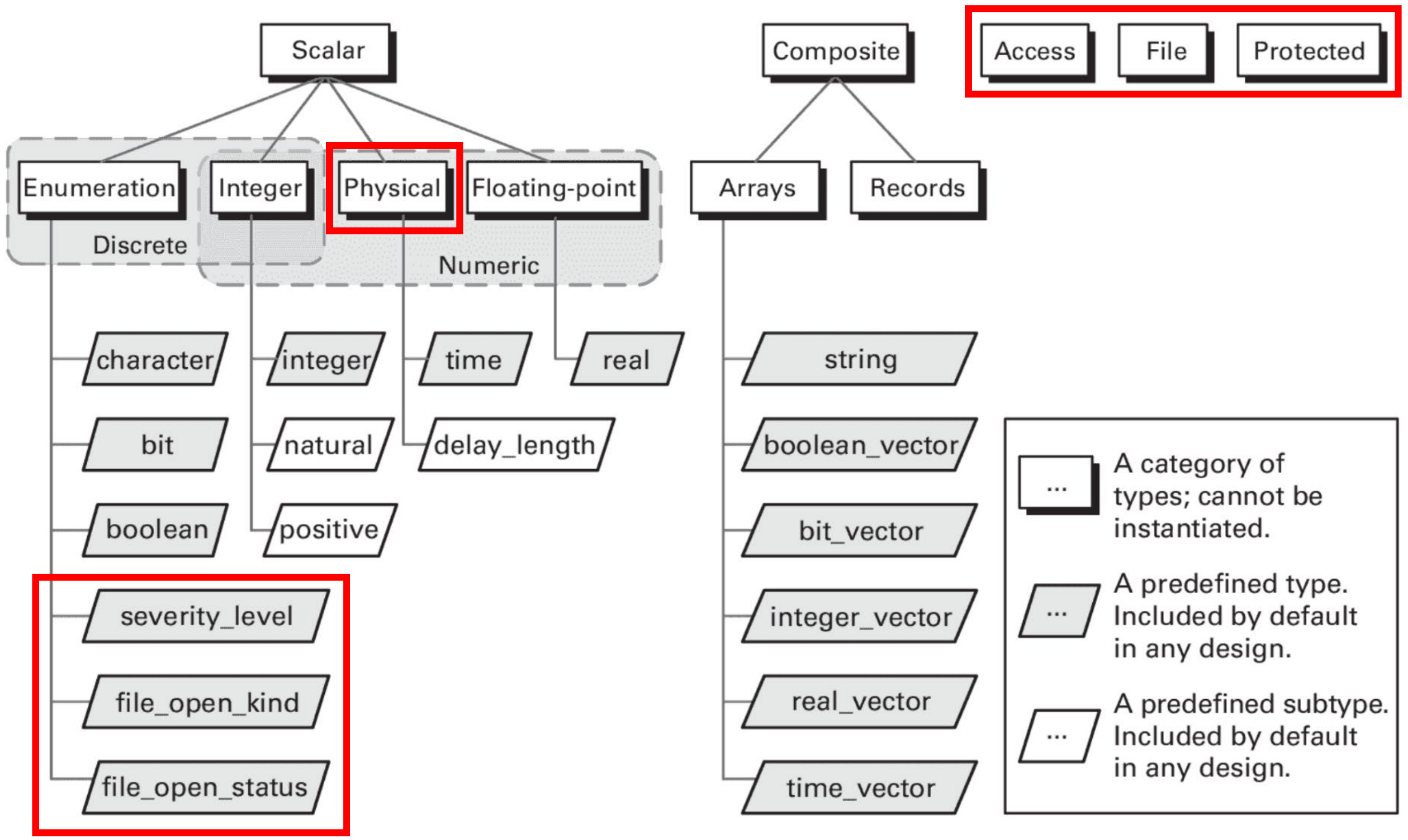
An instance of a class.

Object, VHDL

An instance of a type. Also has a class, a name, and a value.



VHDL type hierarchy and predefined data types



VHDL type hierarchy and predefined data types

NOT synthesisable

UNIVERSITY OF TURKU

Some Notes

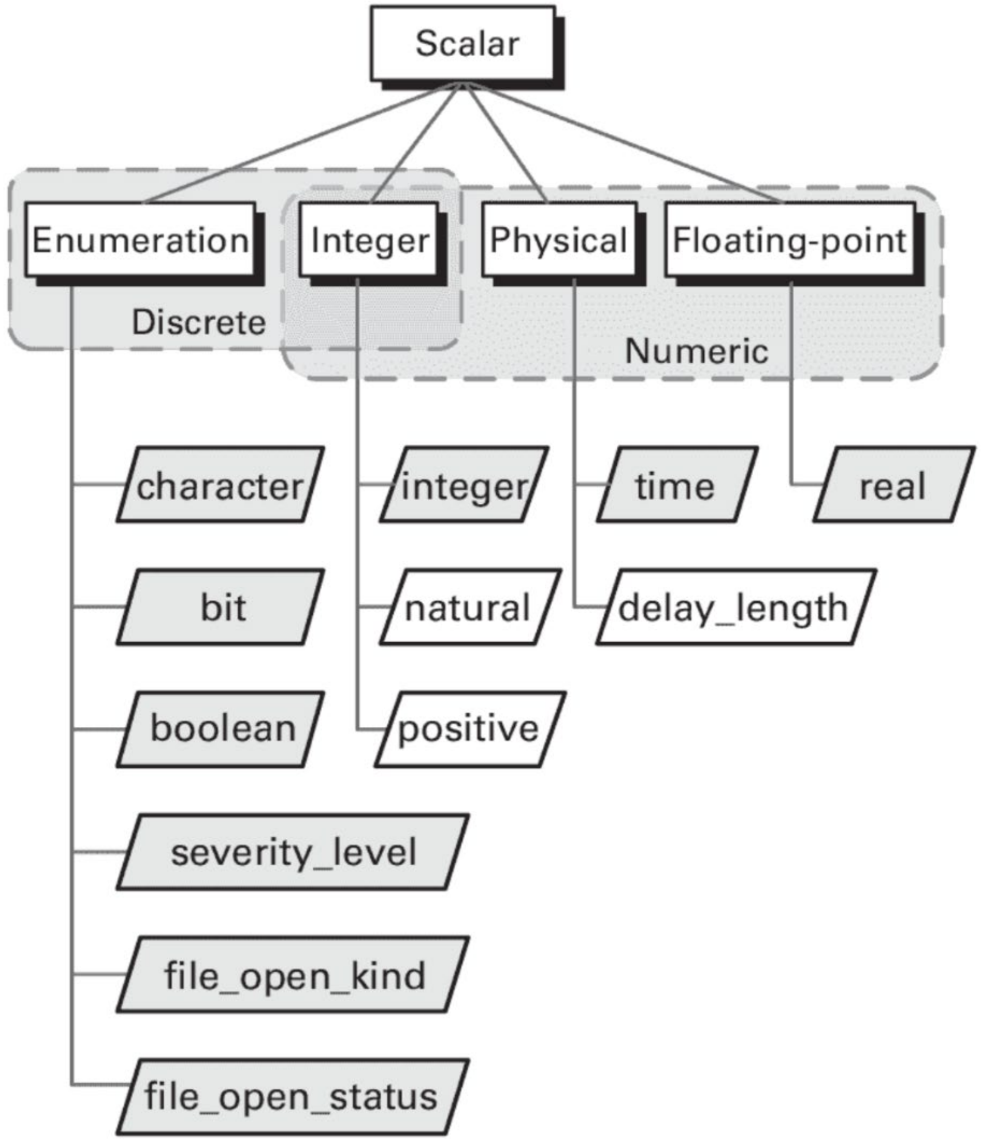
- VHDL, each object must have a type
- The language forces the designer to be explicit about it, and the compiler checks this type every time the object is used in the code
- In VHDL, two types are treated as different even if they are declared exactly the same way and have the exact same underlying data structure
- Objects of one type cannot be assigned to (or used as) objects of a different type

UNIVERSITY OF TURKU

Scalar Types

Scalar Types

- Scalar types are objects can hold only a single value at a time
- The type definition of a scalar type specifies all the allowed values



Type Declarations

- The declaration names a type and specifies which values may be stored into it

```
type own_int1 is range 0 to 100;
type own_int2 is range 0 to 100;

variable v1: own_int1;
variable v2: own_int2;

v1:= 10;
v2:= 12;
```

UNIVERSITY OF TURKU

Type Declarations

- The declaration names a type and specifies which values may be stored into it

- NOTE that if two types are declared separately with identical type definitions, they are nevertheless **distinct** and **incompatible** types

```
type own_int1 is range 0 to 100;
type own_int2 is range 0 to 100;

variable v1: own_int1;
variable v2: own_int2;

v1:= 10;
v2:= 12;
```

```
-- strongly typed
v1 := v2 -- NOT ALLOWED
```

UNIVERSITY OF TURKU

Data Types

Type	Range of values	Example
bit	‘0’, ‘1’	signal x: bit :=1;
bit_vector	an array with each element of type bit	signal INBUS: bit_vector(7 downto 0);
std_logic, std_ulogic	'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-'	signal x: std_logic;
std_logic_vector, std_ulogic_vector	an array with each element of type bit	signal x: std_logic_vector(0 to 7);
boolean	FALSE, TRUE	variable TEST: Boolean :=FALSE’
character	any legal VHDL character (see package standard); printable characters must be placed between single quotes (e.g. ‘#’)	variable VAL: character :=’$’;
integer	range is implementation dependent but includes at least $-(2^{31}-1)$ to $+(2^{31}-1)$	constant CONST1: integer :=129;
natural	integer starting with 0 up to the max specified in the implementation	variable VAR1: natural :=2;
positive	integer starting from 1 up the max specified in the implementation	variable VAR2: positive :=2;

UNIVERSITY OF TURKU

BIT vs STD_ULOGIC

Value	Meaning
‘0’	Forcing (Strong driven) 0
‘1’	Forcing (Strong driven) 1

Value	Meaning
‘U’	Uninitialized
‘X’	Forcing (Strong driven) Unknown
‘0’	Forcing (Strong driven) 0
‘1’	Forcing (Strong driven) 1
‘Z’	High Impedance
‘W’	Weak (Weakly driven) Unknown
‘L’	Weak (Weakly driven) 0. Models a pull down.
‘H’	Weak (Weakly driven) 1. Models a pull up.
‘-’	Don't Care

synthesisable

Resolved Types

- Resolved types are declared with a resolution function
- A resolution function defines the resulting value of a signal if there are multiple driving sources

```
subtype std_logic is resolved std_ulogic;
```

- *As a subtype, all operations and functions defined for std_ulogic apply to std_logic*

Resolution Table for std_logic

	U	X	0	1	Z	W	L	H	-
U	U	U	U	U	U	U	U	U	U
X	U	X	X	X	X	X	X	X	X
0	U	X	0	X	0	0	0	0	X
1	U	X	X	1	1	1	1	1	X
Z	U	X	0	1	Z	W	L	H	X
W	U	X	0	1	W	W	W	W	X
L	U	X	0	1	L	W	W	W	X
H	U	X	0	1	H	W	H	H	X
-	U	X	X	X	X	X	X	X	X

Enumerated Types

- Enumeration types are defined by giving a name to each of the values that compose the type

```
type transmission_mode_type is (park, reverse, neutral, drive, second);
```

- Good candidates for enumerations include the states of a finite state machine, the operations of an ALU

```
-- An enumeration type describing the states of an FSM:
type garage_door_state is (opened, closing, closed, opening);
```

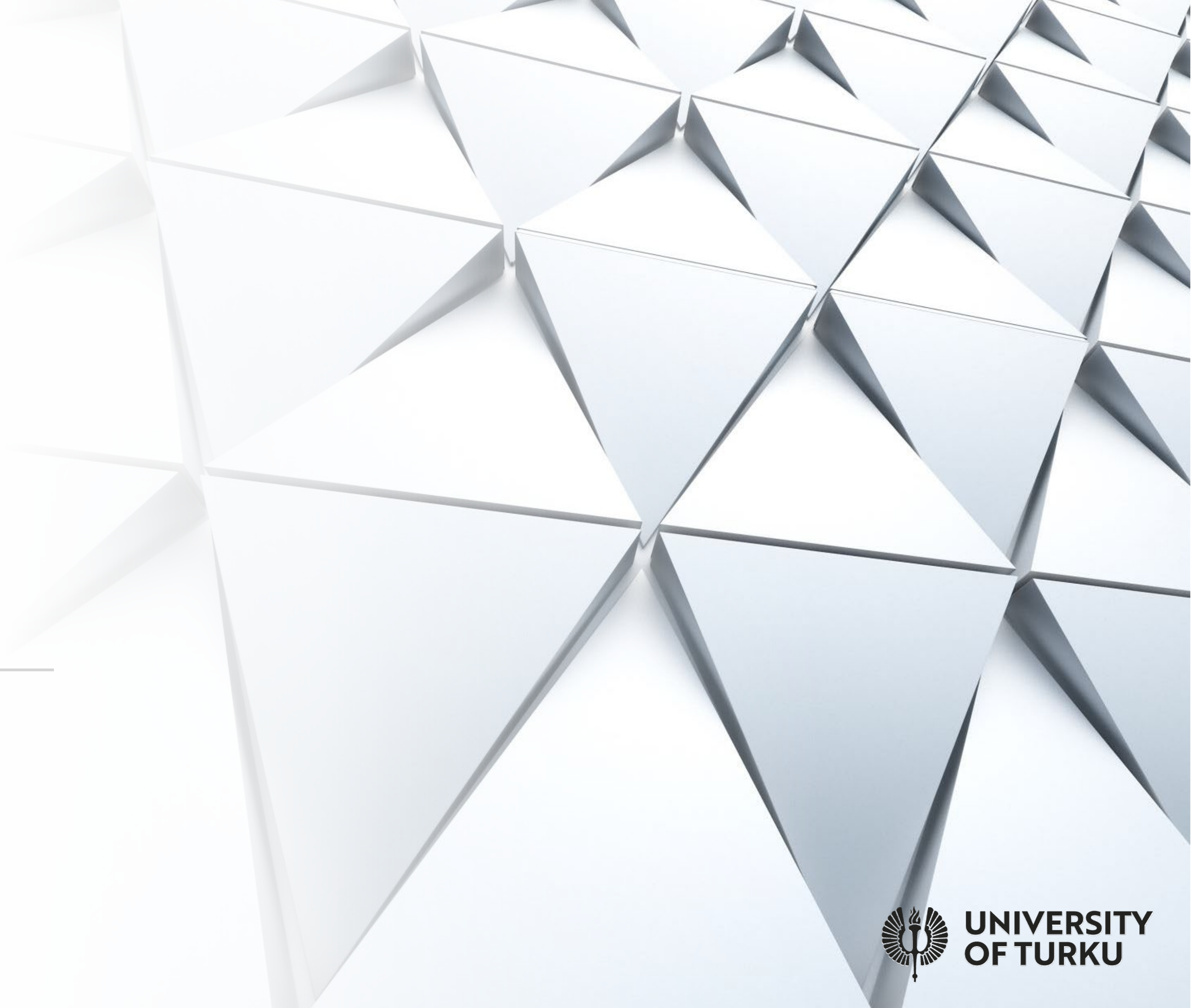
UNIVERSITY OF TURKU

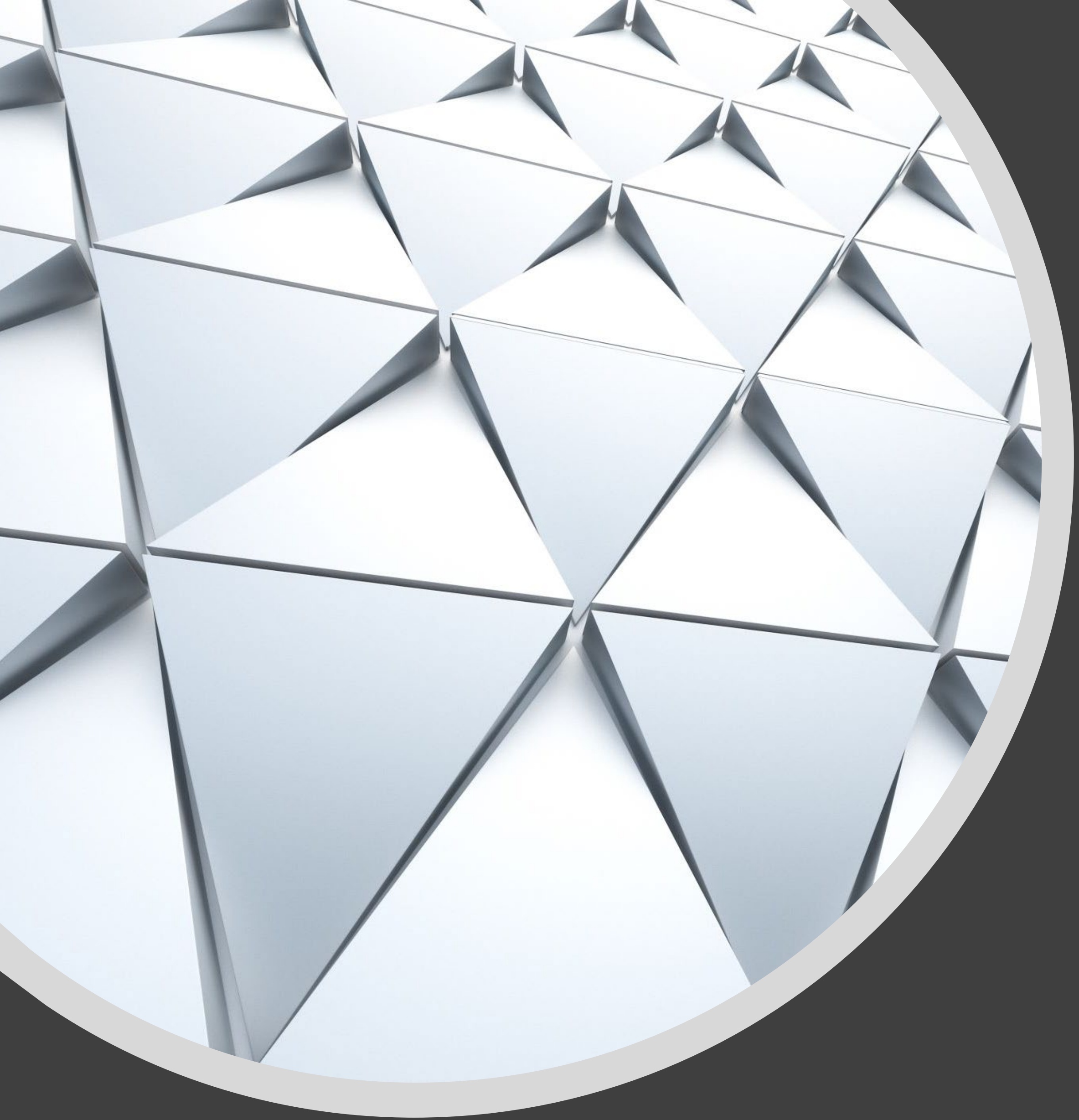
Enumerated Types

- Enumeration types are great for improving readability, especially when they are used to replace enigmatic literal values:

```
if error_condition = 3 then ... -- Bad example: what is error condition 3?
if error = overflow_error then ... -- Good example, error is overflow

if alu_mode = 5 then ... -- Bad example: what is alu_mode 5?
if alu_operation = op_subtract then -- Good example, operation is subtraction
```

Composite Types





Composite Types

- Composite types are objects that can hold multiple values at a time
- Composite types are arrays and records

Array

- An array is a composite object whose elements are all of the same type
- A one-dimensional array is typically called a vector
- A two-dimensional array is called a matrix

```
-- A one-dimensional (1D) array of bits:
type memory_word_type is array (natural range <>) of std_logic;
-- A one-dimensional (1D) array of memory_word_type:
type memory_type is array (natural range <>) of memory_word_type;
-- A two-dimensional (2D) array of bits:
type bit_matrix_type is array(natural range <>, natural range <>) of std_logic;
```

Array

```
-- A one-dimensional (1D) array of bits:
type memory_word_type is array (natural range <>) of bit;
-- A one-dimensional (1D) array of memory_word_type:
type memory_type is array (natural range <>) of memory_word_type;
-- A two-dimensional (2D) array of bits:
type bit_matrix_type is array(natural range <>, natural range <>) of bit;
```

- When we create objects of these types, we need to constrain them

```
variable memory_word: memory_word_type(127 downto 0);
variable memory_contents: memory_type(0 to 65535)(127 downto 0);
```

Array

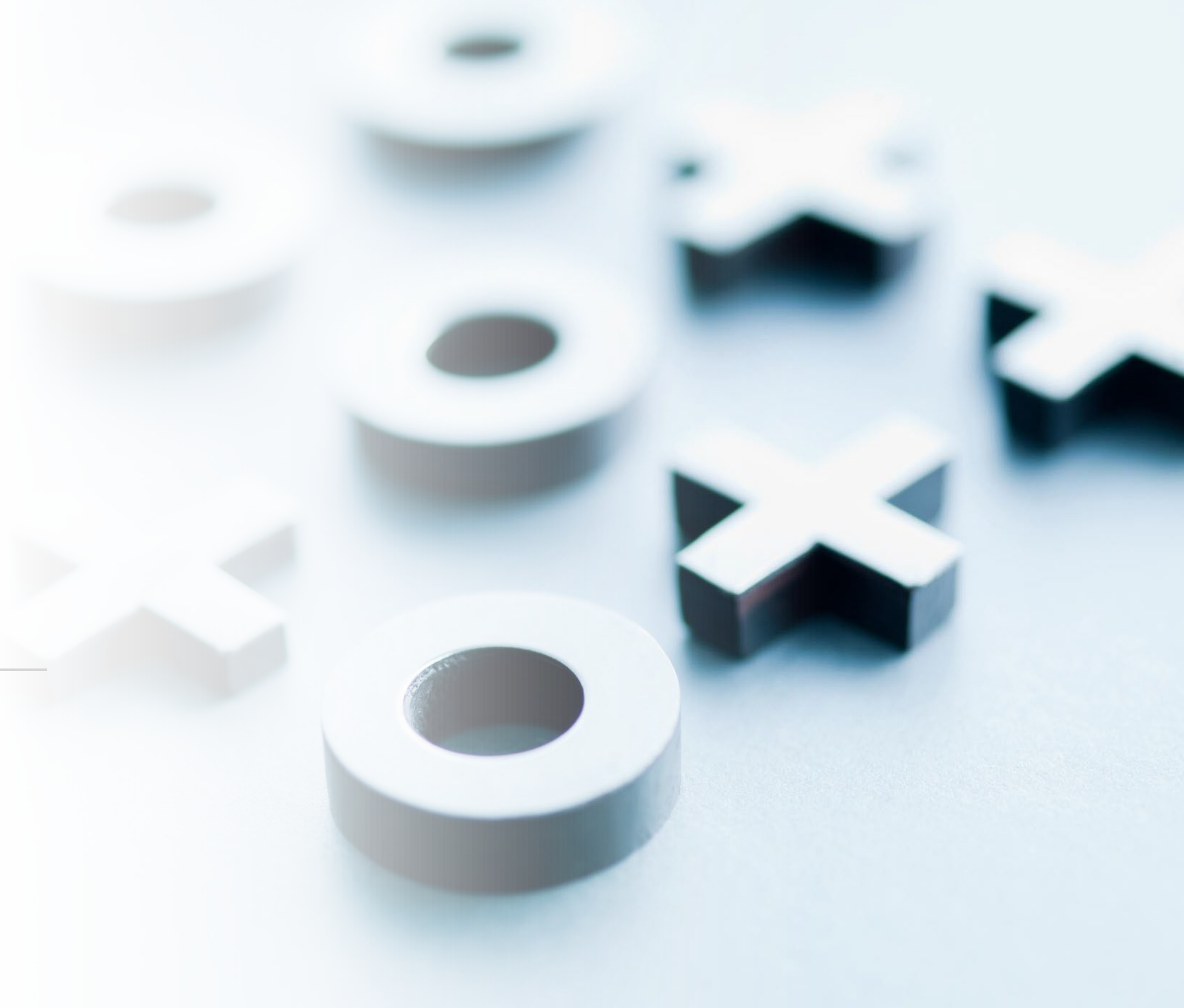
```
variable memory_word: memory_word_type(127 downto 0);
variable memory_contents: memory_type(0 to 65535)(127 downto 0);
```

OR

```
type memory_word_type is array (127 downto 0) of bit;
type memory_type is array (0 to 65535) of memory_word_type;

variable memory_word: memory_word_type;
variable memory_contents: memory_type;
```

- The disadvantage of this approach is that it forces all instances of the type to have the same dimensions

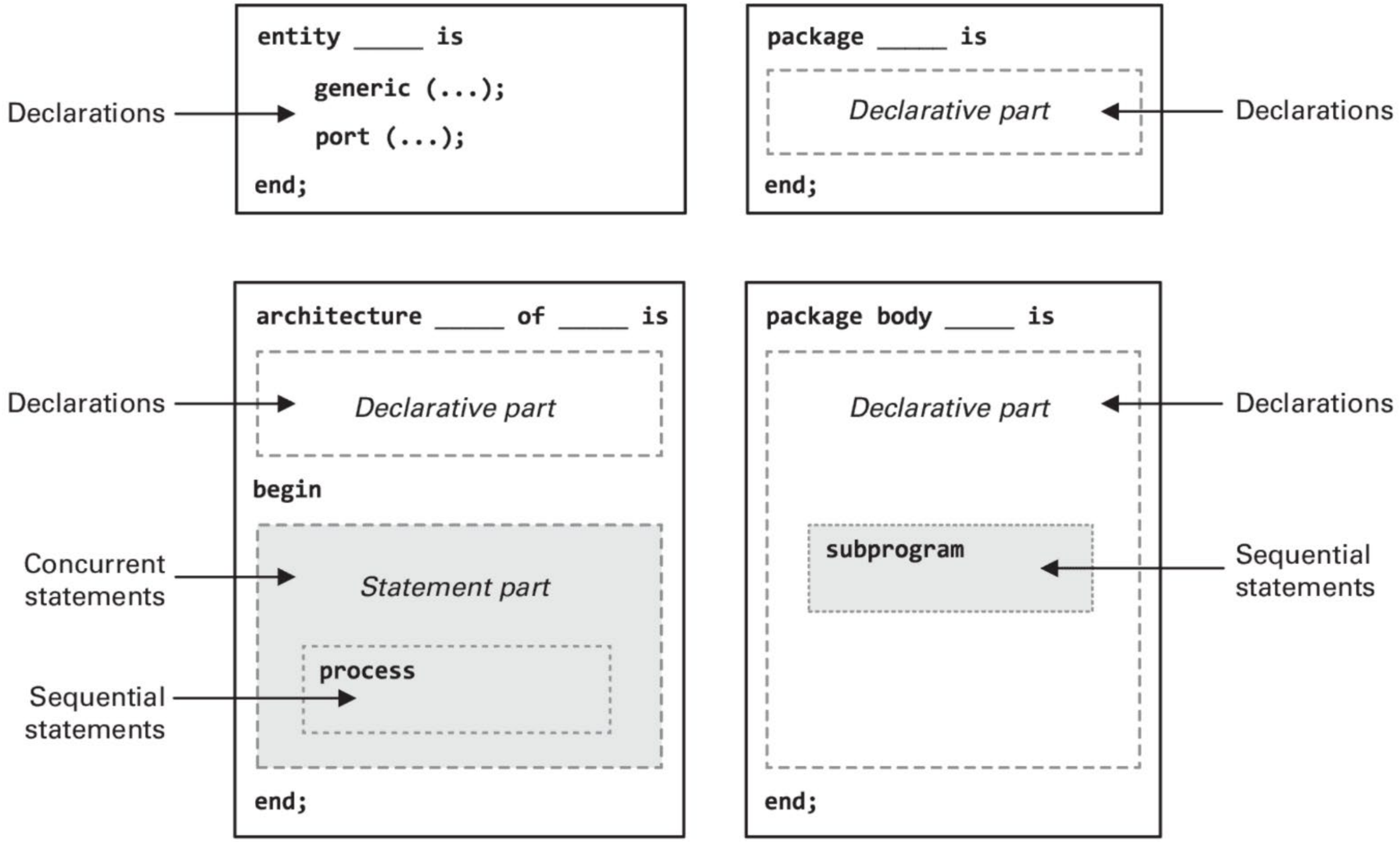
Statements



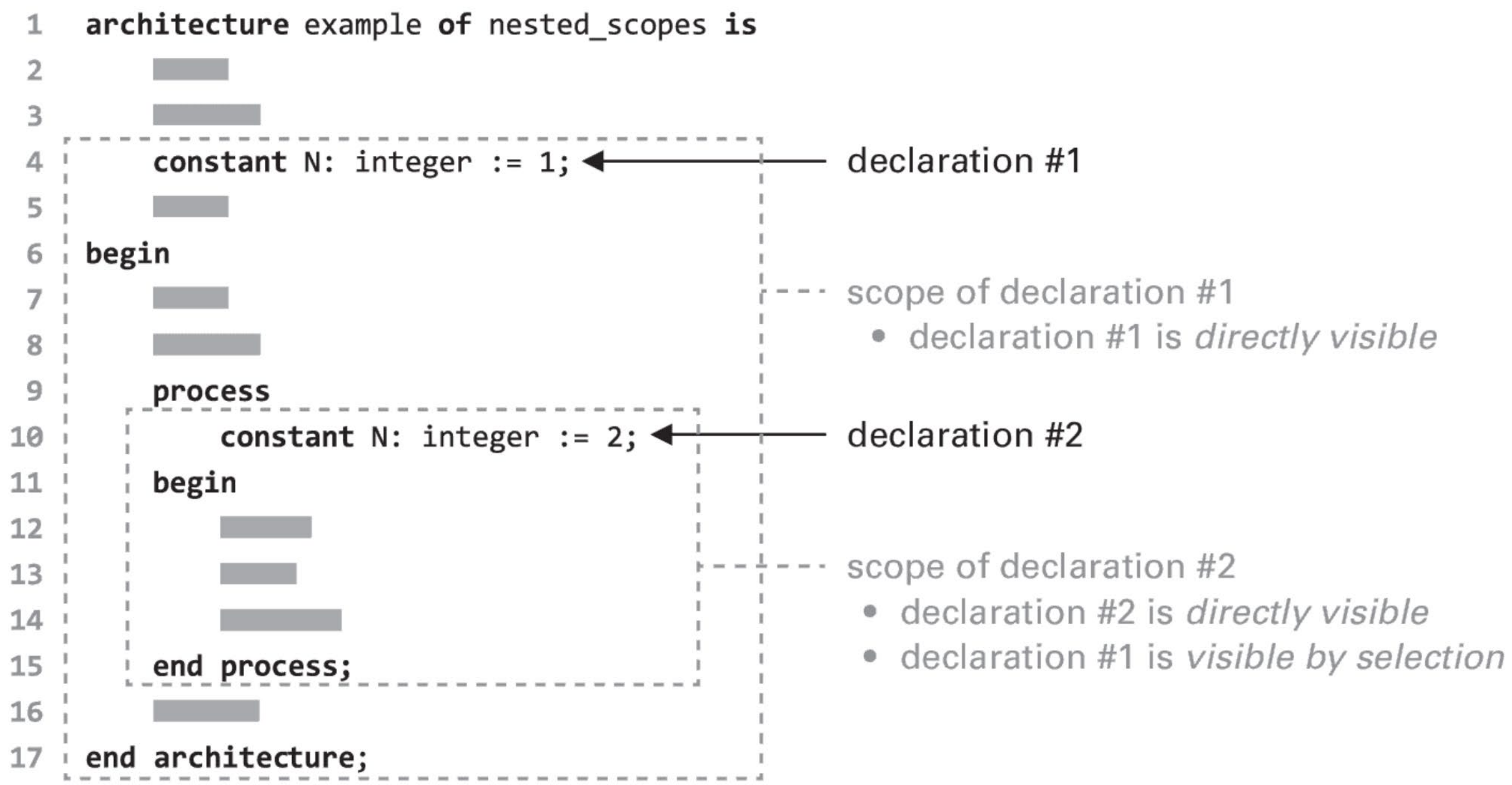
Statements

Statements are the basic units of behavior in a program; they specify basic actions to be performed when the code is executed and control its execution flow.

Statements


entity _____ is
generic (...);
port (...);
end;
Declarations
package _____ is
Declarative part
end;
Declarations
architecture _____ of _____ is
Declarative part
begin
Statement part
process
end;
Declarations
Concurrent statements
Sequential statements
package body _____ is
Declarative part
subprogram
end;
Declarations
Sequential statements

Declarations and Visibility



```
architecture example of nested_scopes is


    constant N: integer := 1;

begin


    process
        constant N: integer := 2;
    begin



    end process;

end architecture;
```

declaration #1

scope of declaration #1
- declaration #1 is *directly visible*

declaration #2

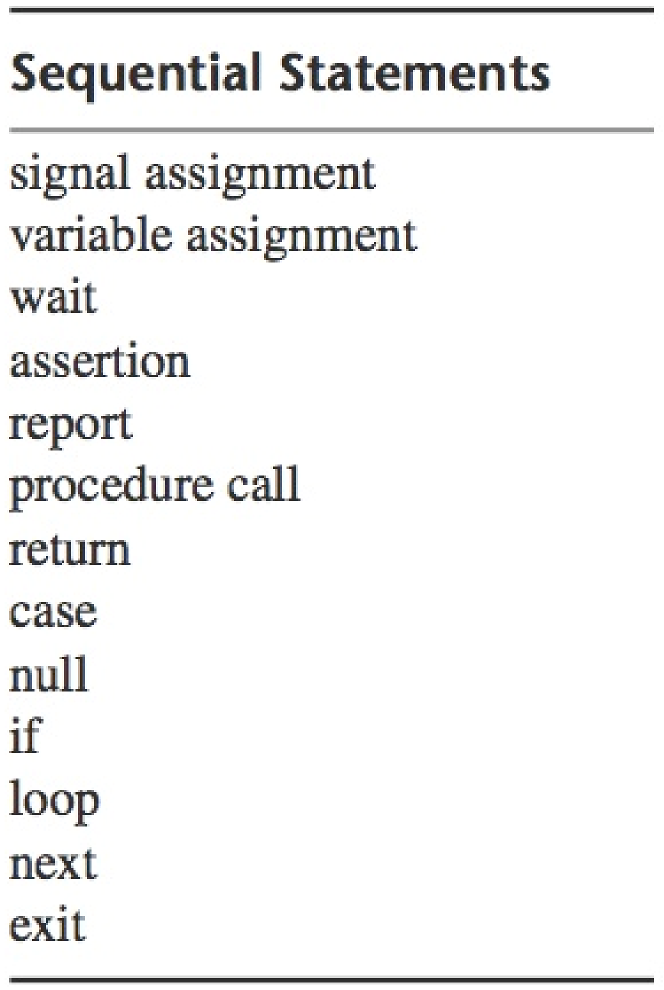
scope of declaration #2
- declaration #2 is *directly visible*
- declaration #1 is *visible by selection*

Statements

- Sequential
  - are executed one after another in the order they appear in the source code
  - can only exist in sequential code regions, namely, inside *processes* or *subprograms*
- Concurrent
  - they execute side by side, conceptually in parallel
  - they execute continuously
  - no predefined order of execution

Sequential Statements

- Can be used *only* within a process
- Executed one after another



Sequential Statements
signal assignment
variable assignment
wait
assertion
report
procedure call
return
case
null
if
loop
next
exit

Processes

Processes

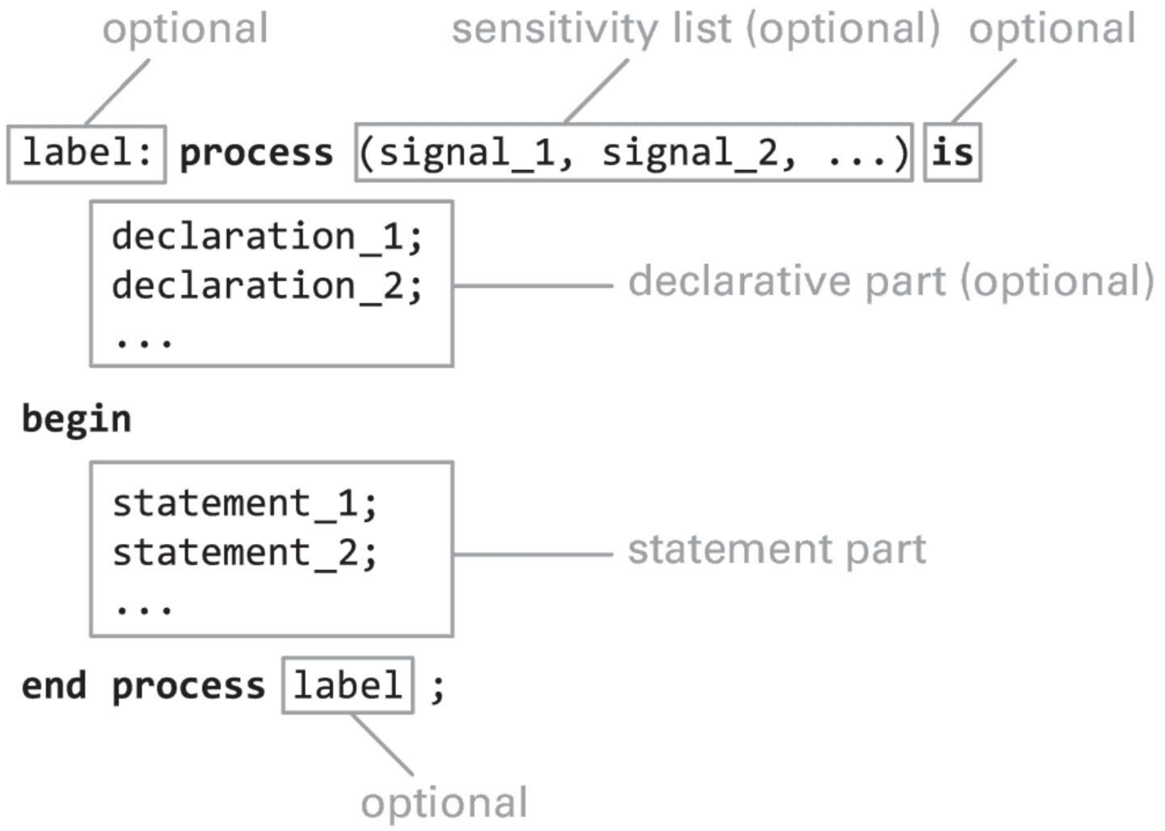
- Processes are executed

  in parallel

- Statements within a process are executed

  in sequece

- After all process's statements are executed, the process starts again from the beginning
  - Signals in the sensitity list awakes the process for executions



```
label: process (signal_1, signal_2, ...) is
    declaration_1;
    declaration_2;
    ...
begin
    statement_1;
    statement_2;
    ...
end process label ;
```

Facts

- Once activated, a process keeps running until it executes a wait statement
  - a process with a sensitivity list has an implicit wait statement immediately before the end process keywords
- A processes is always in one of two states: executing or suspended
- A process can declare variables, which are local to the process
- Variables declared in a process are initialized only once when the simulation begins
- A process should contain a sensitivity list or wait statements but not both

UNIVERSITY OF TURKU

Fact

- Keep in mind that processes, like real hardware components, exist and execute independently of each other

AND

- The order of execution among processes is random and should be irrelevant

Sensitivity List

- A list of *signals* to which a process is sensitive
- Signals in the sensitity list awakes the process for executions
- **NOTE**, only those signals that are read, that is, are on the right side of a statement are allowed to be in the sensitivity list
  - Improper use of sensitivity list causes mismatch between the design and its implementation
- The keyword *all* can be used to denote all the signals that are read within a process
  - VHDL-2008 only

Sensitivity List

- The keyword *all* can be used to denote all the signals that are read within a process
  - VHDL-2008 only

```
next_state_logic: process (all)
begin
    case current_state is
        when accepting_coins =>
            ...
    end case;
end process;
```

Sensitivity List

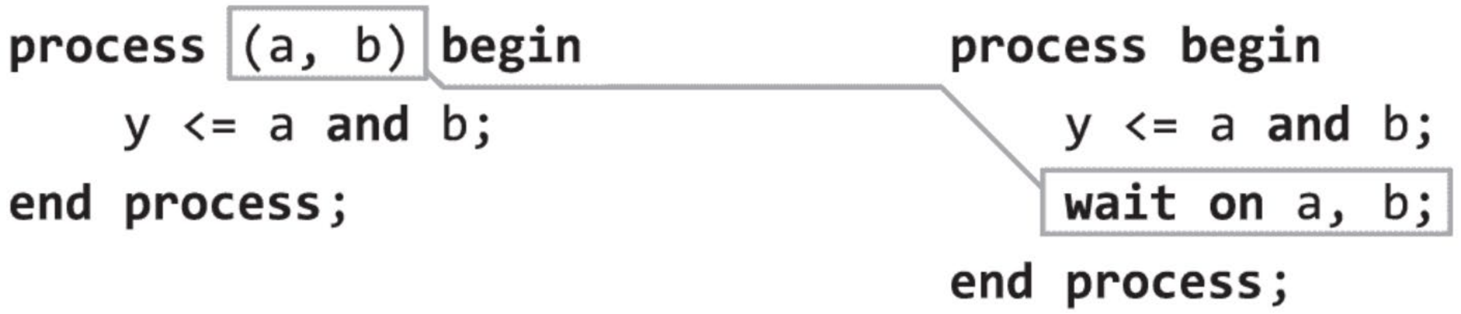
- Two ways to write the sensitivity list for a combinational process

```
next_state_logic: process (all)
begin
    case current_state is
        when accepting_coins =>
            ...
    end case;
end process;
```

```
next_state_logic: process (current_state, nickel_in, dime_in, quarter_in, coin_return)
begin
    case current_state is
        when accepting_coins =>
            ...
    end case;
end process;
```

Sensitivity List

- Equivalence between a process sensitivity list and a wait on statement



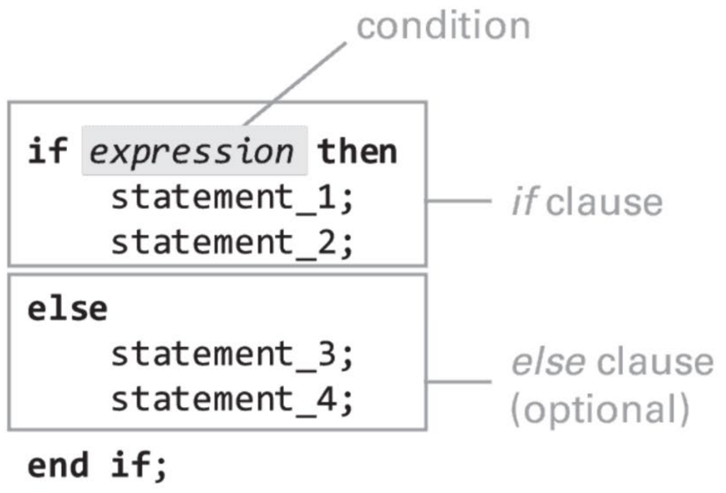
```
process (a, b) begin
    y <= a and b;
end process;
```

```
process begin
    y <= a and b;
    wait on a, b;
end process;
```

Control Structures

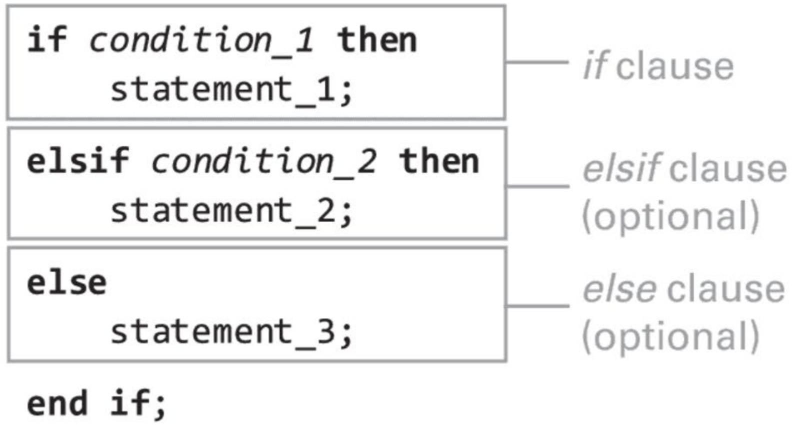
If Statement

- Chooses one of the given statements
- Chooses the first matching condition for execution
- If none of the conditions are met, the construct is terminated and not statements are executed



```
if expression then
    statement_1;
    statement_2;
else
    statement_3;
    statement_4;
end if;
```

(a) A simple *if-else* statement.



```
if condition_1 then
    statement_1;
elsif condition_2 then
    statement_2;
else
    statement_3;
end if;
```

(b) An *if-elsif-else* statement.

If Statement

- Choose the order of the if and else clauses consciously
- Replace complicated tests with a boolean variable

```
if (sync_counter = 63 and (data_in_sync(7 downto 0) = "01000111"
            or  data_in_sync(7 downto 0) = "00100111")
            and (current_state =  acquiring_lock or current_state = recovering_lock) )
then  locked <= '1';  ... end if;
```

If Statement

- Replace complicated tests with a boolean variable

```
if (sync_counter = 63 and (data_in_sync(7 downto 0) = "01000111"
              or  data_in_sync(7 downto 0) = "00100111")
              and (current_state =  acquiring_lock or current_state = recovering_lock) )
then  locked <= '1';  ... end if;
```

```
alias current_byte is data_in_sync( 7 downto 0);
...
waiting_lock := (current_state = acquiring_lock) or (current_state = recovering_lock);
last_sync_bit := (sync_counter = 63);
preamble_detected := (current_byte = "01000111") or (current_byte = "00100111");

if waiting_lock and last_sync_bit and preamble_detected
then  locked <= '1';  .. . end if;
```

ERSITY
JRKU

If Statement

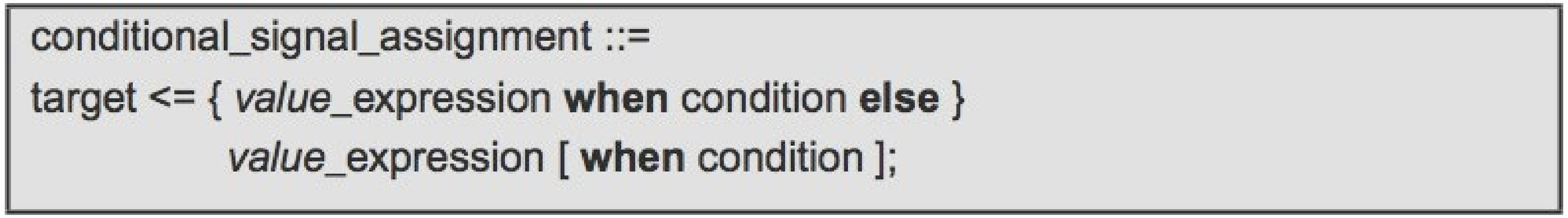
- Replace complicated tests with a boolean variable

```
if chip_sel = '1' and enable = '1' and write = '1' then ...

-- In VHDL2008 use instead

if chip_sel and enable and write then ...
```

UNIVERSITY OF TURKU

Conditional Signal Assignments

- Shorthand notation for the IF statement



```
conditional_signal_assignment ::=
target <= { value_expression when condition else }
            value_expression [ when condition ];
```

- Conditions are evaluated in the order of apperance
- Last values must NOT have an associated condition
- Only available in VHDL 2008
  - VHDL 2008 provides a conditional variable assignement as well (Older versions must use the if statement)

Conditional Signal Assignments

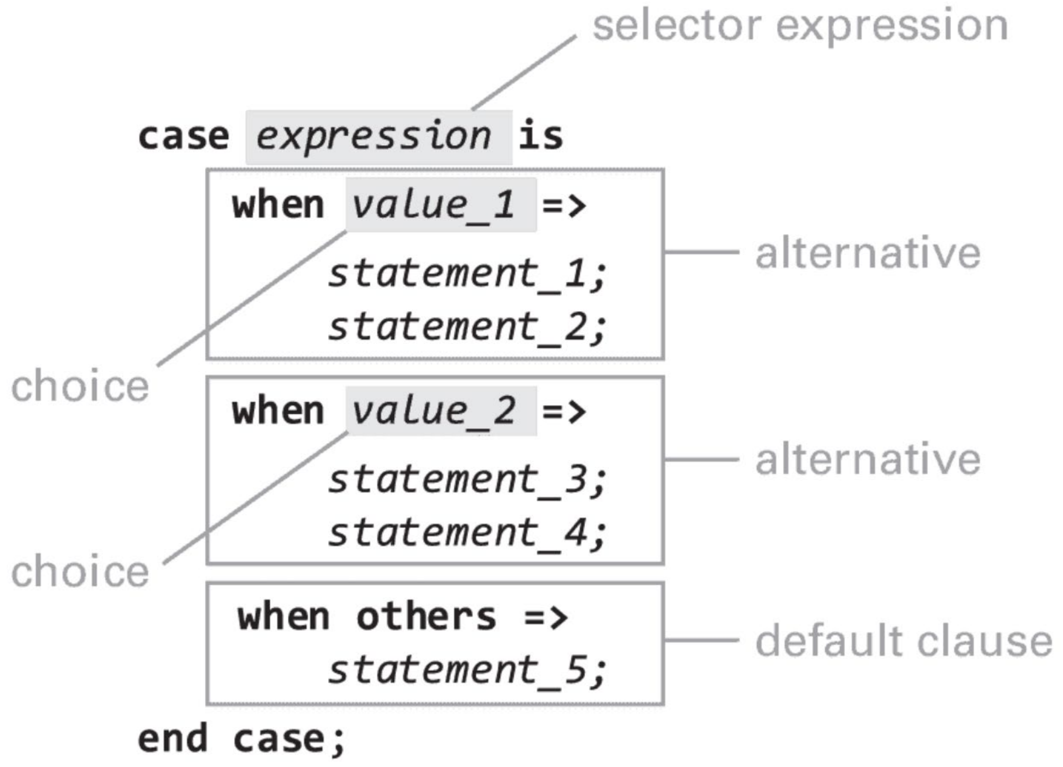
- Shorthand notation for the IF statement

```
if error_detected then
    next_state <= locking_error;
else
    next_state <= locked;
end if;
```

```
next_state <= locking_error when error_detected else locked;
```

Case Statement

- Chooses one of the given statements
- *One and only one* choice can and must match
- Choices must be *mutually exclusive,* that is, must cover ALL possible choices
- *Others* can be used to denote unspecified choices



```
case expression is
    when value_1 =>
        statement_1;
        statement_2;
    when value_2 =>
        statement_3;
        statement_4;
    when others =>
        statement_5;
end case;
```

(a) Elements of a *case* statement.

Case Statement

```
entity simple is
port (x1: in  std_logic_vector(1 downto 0);
      x2: out std_logic_vector(7 downto 0)
);
end;

architecture behaviour of simple is
begin
  casex :Process ( x1)
  begin
    case x1 is
            when ”00” => x2 <= X”01”;
            when ”01” => x2 <= X”10”;
            when ”10” => x2 <= X”11”;
            when ”11” => x2 <= X”11”;
    end case;
  end process;
end architecture;
```

Case Statement (equivalent?)

```
entity simple is
port (x1: in std_logic_vector(1 downto 0);
   x2: out std_logic_vector(7 downto 0)
);
end;

architecture behaviour of simple is
begin
 casex :Process ( x1)
 begin
  case x1 is
   when ”00” => x2 <= X”01”;
   when ”10” => x2 <= X”10”;
   when others => x2 <= X”11”;
  end case;

 end process;
end architecture;
```

UNIVERSITY OF TURKU

Don’t Care Inputs and Outpus

- Don’t care indicates that we are not interested is the value ’1’ or ’0’
- Don’t care value is represented with the character ’-’
- NOTE, don’t care value is treated differently by simulators and synthesisers
  - Synthesisers take advantage of the don’t care value to mimise the logic it synthesises

```
x1   <= ”00” when s = ”0---” else
       ”10” when s = ”1---” else
       ”11”;
```

```
x1  <=  ”0” when s = ”1000” else
       ”1” when s = ”1100” else
       ”-”  when others;
```

Matching case Statement

- case which allows don't care behaviour

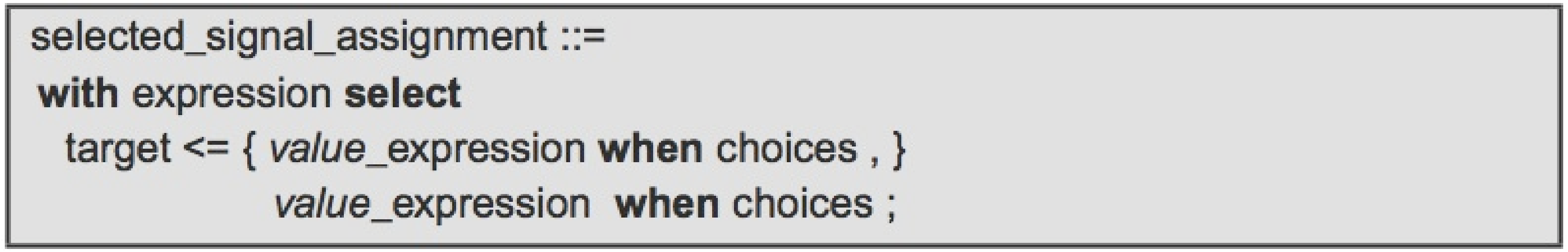
```
case? data_word is
            when "1---"    =>  leading_zeros_count := 0;
            when "01--"    =>  leading_zeros_count := 1;
            when "001-"    =>  leading_zeros_count := 2;
            when "0001"    =>  leading_zeros_count := 3;
            when "0000"    =>  leading_zeros_count := 4;
            when others =>  null;
end case?;
```

- The don’t care terms ('-') match any value at their corresponding positions. Thus, the first choice in the earlier example ("1---") would match the values "1000", "1111", "1010", and so on

UNIVERSITY OF TURKU

Selected Signal Assignment

- Shorthand notation for the CASE statement



```
selected_signal_assignment ::=
 with expression select
   target <= { value_expression when choices , }
                     value_expression  when choices ;
```

- The value of *expression* is first evaluated after which the value is simultaneously compared againts all the *choices*
- Only available in VHDL 2008
  - VHDL 2008 provides a selected variable assignement as well (Older versions must use the if statement)

Null Statement

- Null statement explicitly indicates that there is nothing to be done
- The null statement can be used anywhere where a sequential statement is required
  - Used mainly in case statements
  - For example, in processor models, one may use *null* when *nop* (*no operation)* is performed

```
case x1 is
  when ”00” => x2 <= X”01”;
  when ”10” => x2 <= X”10”;
  when others => null;
end case;
```

UNIVERSITY
OF TURKU