


# SYSTEM MODELLING AND SYNTHESIS WITH HDL

DTEK0078

2023 Lecture 4



# 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



VHDL Object			
Type	Class	Name	Value/Internal representation
<ul style="list-style-type: none"> <li>Defines the set of allowed values.</li> <li>Defines the allowed operations.</li> <li>Can be predefined or custom.</li> <li>Ex: boolean, bit, std_logic, integer, real...</li> </ul>	<ul style="list-style-type: none"> <li>Defines how the object can be used in a design.</li> <li>There are only four object classes: <ul style="list-style-type: none"> <li>constant;</li> <li>variable;</li> <li>signal;</li> <li>file.</li> </ul> </li> </ul>	<ul style="list-style-type: none"> <li>Every object explicitly declared in the code must have a name.</li> <li>Dynamically created objects do not have a name and are accessed using pointers.</li> </ul>	<ul style="list-style-type: none"> <li>For simple classes (constant, variable), a value is enough to describe the object state.</li> <li>Complex classes (signal, file) need to keep extra information</li> </ul>
<p>Examples:</p> <div> integer real boolean bit std_ulogic </div>	<p>Must be one of:</p> <div> constant variable signal file </div>	<p>Examples:</p> <div> write_enable sum clock address </div>	<p>Examples:</p> <div> 0 1 2 'a' 'b' "abc" 0, 1, 2, 3 true false </div>

# Objects in VHDL

# Four Classes of Objects

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

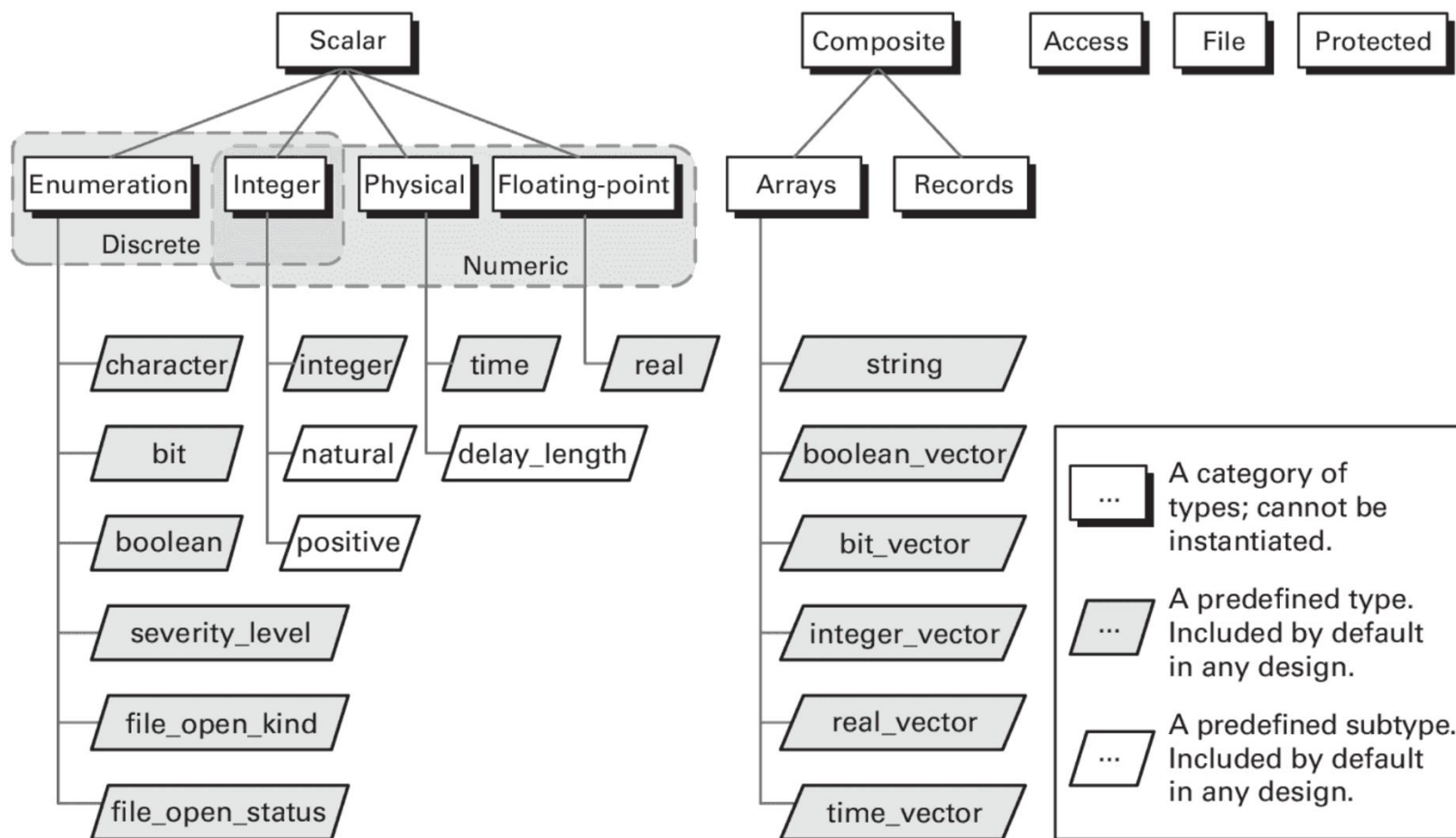
One of the four object classes: signal, variable, constant, or file. Determines how an object can be used, but does not define its data structure or operations.

## *Object, OOP language*

An instance of a class.

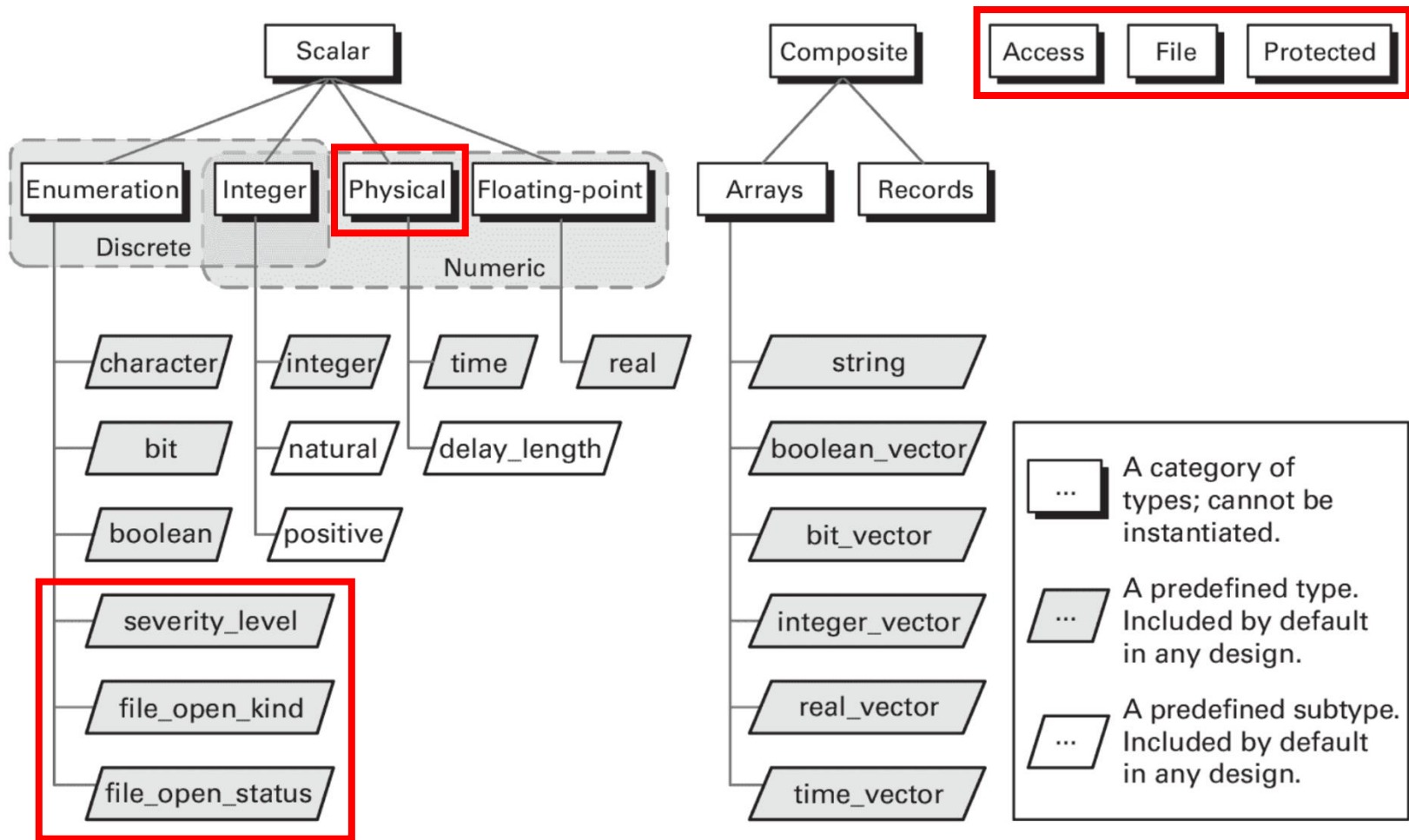
## *Object, VHDL*

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

# 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



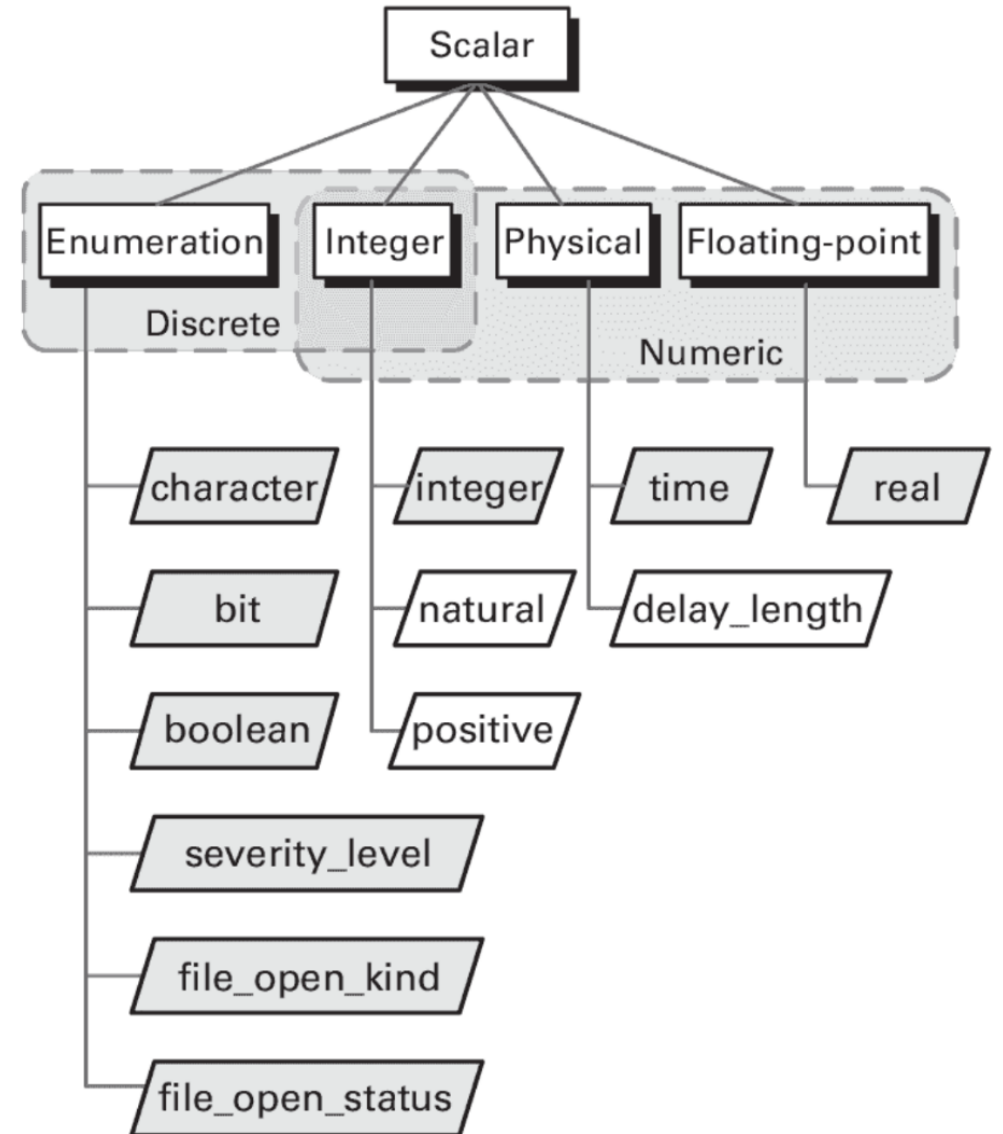
# Scalar Types

---



# 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;
```

# 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
```



# Data Types

Do not use *bit* or *bit\_vector*. Use *std\_logic* and *std\_logic\_vector* instead

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

# Types in C++

- In C++, there are different **types** of variables (defined with different keywords), for example:

`int` - stores integers (whole numbers), without decimals, such as 123 or -123

`double` - stores floating point numbers, with decimals, such as 19.99 or -19.99

`char` - stores single characters, such as 'a' or 'B'. Char values are surrounded by single quotes

`string` - stores text, such as "Hello World". String values are surrounded by double quotes

`bool` - stores values with two states: true or false

# Variables in C++

Canonical Type Specifier	Minimum Width	Minimum Range
short unsigned short	16 bit	$-2^{15}$ to $2^{15} - 1$ 0 to $2^{16} - 1$
int unsigned	16 bit	$-2^{15}$ to $2^{15} - 1$ 0 to $2^{16} - 1$
long unsigned long	32 bit	$-2^{31}$ to $2^{31} - 1$ 0 to $2^{32} - 1$
long long unsigned long long	64 bit	$-2^{63}$ to $2^{63} - 1$ 0 to $2^{64} - 1$



# 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);
```

# Enumerated Types

- Enumeration types are great for improving readability, especially when they are used to replace mysterious 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

---



UNIVERSITY  
OF TURKU



# 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 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;
```

- 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(255 downto 0)(127 downto 0);
```

# Array

```
variable memory_word: memory_word_type(127 downto 0);  
variable memory_contents: memory_type(255 downto 0)(127 downto 0);
```

OR

```
type memory_word_type is array (127 downto 0) of std_logic;  
type memory_type is array (255 downto 0) 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

# Endiannes in Arrays

- `downto` corresponds to little endian
- `to` corresponds to big endian

```
signal big_endian : std_logic_vector(0 to 7) := ( 0 => '1', others => '0');  
-- big_endian is initially "1000_0000"
```

```
signal little_endian : std_logic_vector(7 downto 0) := ( 0 => '1', others => '0');  
-- little_endian is initially "0000_0001"
```

- Do not mix *downto* and *to* because this leads to confusion and bugs



# Assigning Values to Signals

```
signal my_array : std_logic_vector(7 downto 0);

-- assign to element
my_array(0)      <= '0';

-- assign slice
my_array(3 downto 0) <= "0001";

-- positional association
my_array         <= ('0', '0', '0', '0', '0', '0', '0',
'1');

-- named association variants
my_array         <= (0 => '1', others => '0');
my_array         <= (0 => '1', 1 => '1', others => '0');
my_array         <= (0 | 1 => '1', others => '0');
my_array         <= (1 downto 0 => '1', others => '0');
```

# Assigning Values: Literals

Base Category	Subcategory	Examples
Numeric literal	Integer, decimal literal	51, 33, 33e1
Numeric literal	Integer, based literal	16#33#, 2#0011_0011#
Numeric literal	Real, decimal literal	33.0, 33.0e1
Numeric literal	Real, based literal	16#33.0#, 2#0011_0011.0011#
Numeric literal	Physical literal	15 ns, 15.0 ns, 15.0e3 ns
Enumeration literal	Identifier	true, ESC, warning, failure
Enumeration literal	Character literal	'a', ' ', '0', '1', '½', '@', ''
String literal	String literal	"1100", "abc", "a", "0", "1", "123"
Bit-string literal	Bit-string literal	b"1100", X"7FFF", x"7fff", 16d"123"
Null literal	Null literal	null

# Assigning Values: Literals

<code>8d"123"</code>	-- Expands to <code>"01111011"</code>
<code>8b"111_1011"</code>	-- Expands to <code>"01111011"</code>
<code>8ub"111_1011"</code>	-- Expands to <code>"01111011"</code> (unsigned, no sign extension)
<code>8sb"111_1011"</code>	-- Expands to <code>"11111011"</code> (signed, perform sign extension)
<code>x"7b"</code>	-- Expands to <code>"01111011"</code>
<code>ux"7b"</code>	-- Expands to <code>"01111011"</code> (unsigned, no sign extension)
<code>8sx"b"</code>	-- Expands to <code>"11111011"</code> (signed, perform sign extension)

# Assigning Values to Signals

```
signal x1: std_logic;  
signal x2: std_logic_vector(3 downto 0);  
signal x3: std_logic_vector(15 downto 0);  
signal x4: std_logic_vector(9 downto 0);
```

```
x1 <= '1';  
x2 <= "0101"; -- Binary base assumed by default (B"0000")  
x3 <= X"FF67"; -- Hexadecimal base  
x4 <= O"713"; -- Octal base
```

# Assigning Values to Signals (Vectors)

```
signal x1: std_logic;  
signal x2: std_logic_vector(3 downto 0);  
signal x3: std_logic_vector(15 downto 0);  
signal x4: std_logic_vector(9 downto 0);
```

```
x2(0) <= '1';           -- assigns value to the rightmost element of x2  
x2    <= x3(3 downto 0); -- assigns the 4 rightmost values to x2  
x2    <= (3 => '1', 0 => '1', 2 => '1', 1 => '0');  
x2    <= (3|0 |2 => '1', 1 => '0');  
x2    <= (others => '1', 1 => '0');  
x2    <= (others => '0');
```



# Assigning Values to Signals - VHDL-2008

- An array aggregate forms an array from a collection of elements  
(`'0'`, `"1001"`, `'1'`)
- Can be used for the target of an assignment statement as well

```
signal a, b, sum : unsigned(7 downto 0);  
signal carry : std_ulogic;  
...  
(carry, sum) <= ('0' & a) + ('0' & b);  
  
(13 downto 8 => opcode,  
 6 downto 0 => register_address,  
 7 => destination) := instruction;
```

# Assigning Values to Signals – VHDL 2008

```
signal x1: std_logic_vector(5 downto 0);
```

```
x1 <= 6x"0f";
```

```
x1 <= 6SX"F"; (sign extension)
```

```
x1 <= 6Ux"f"; (zero extension)
```

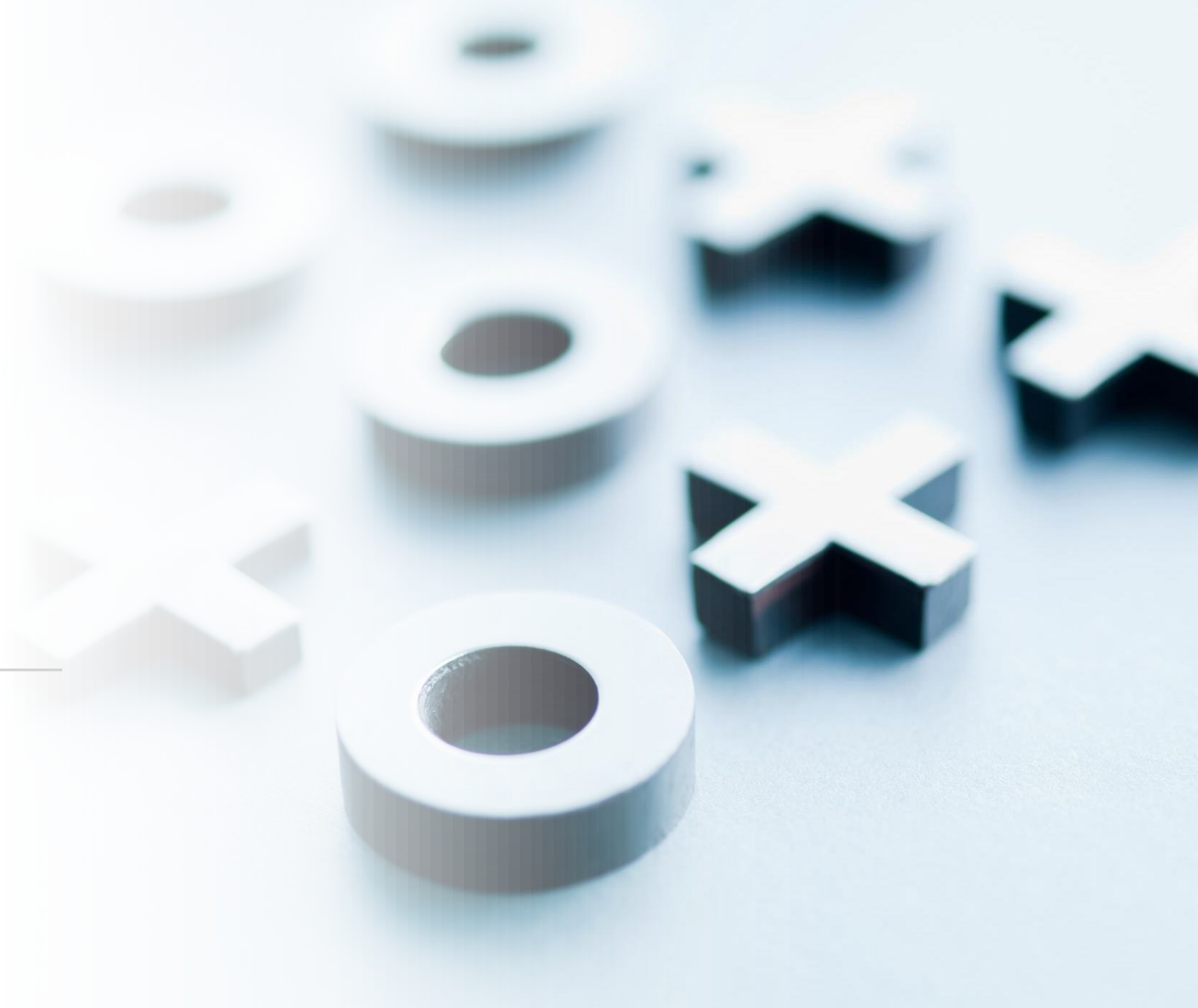
```
x1 <= 6sb"11"; (binary format)
```

```
x1 <= 6u0"7"; (octal format)
```



# Statements


---



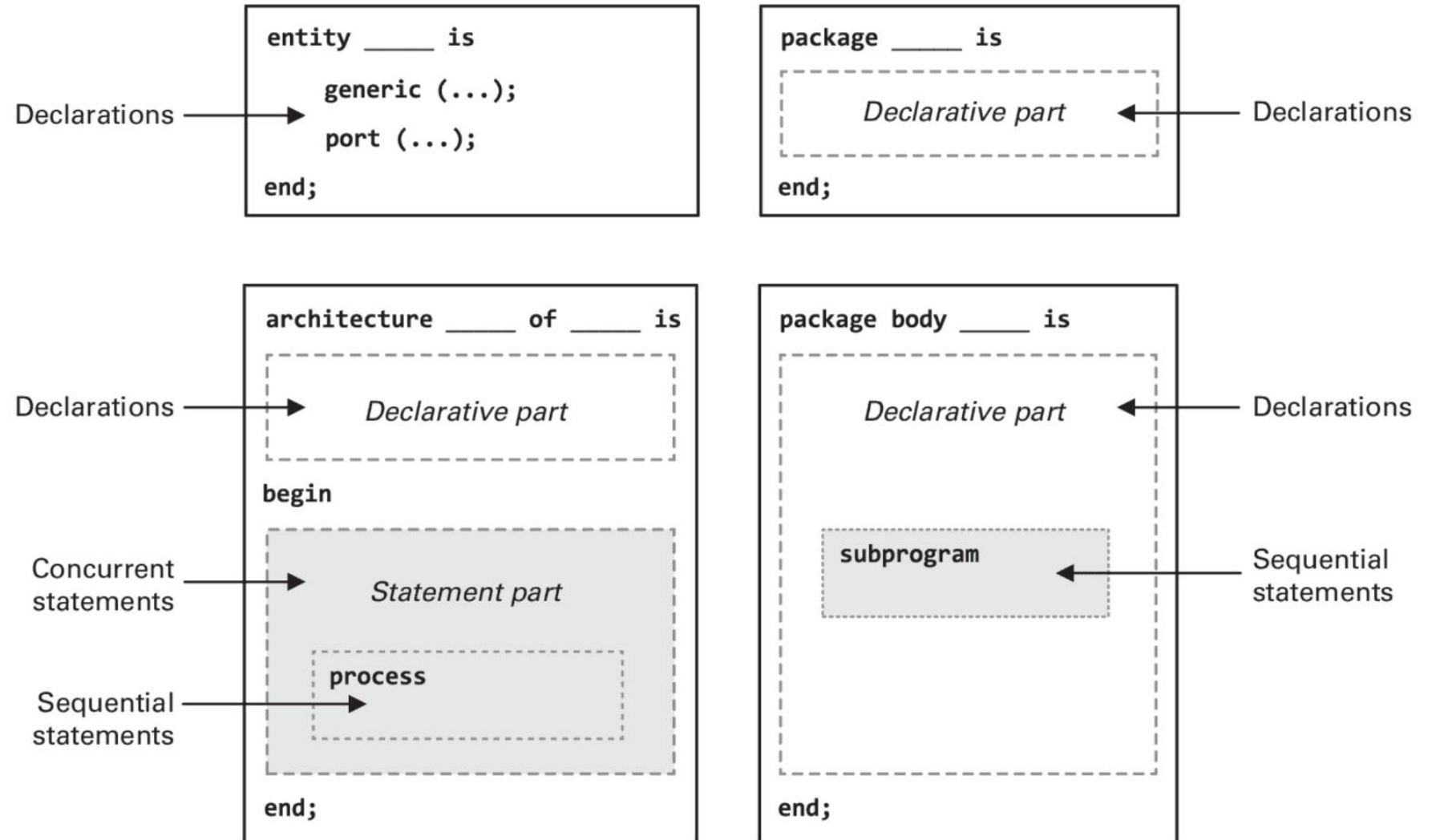


# Statements

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.

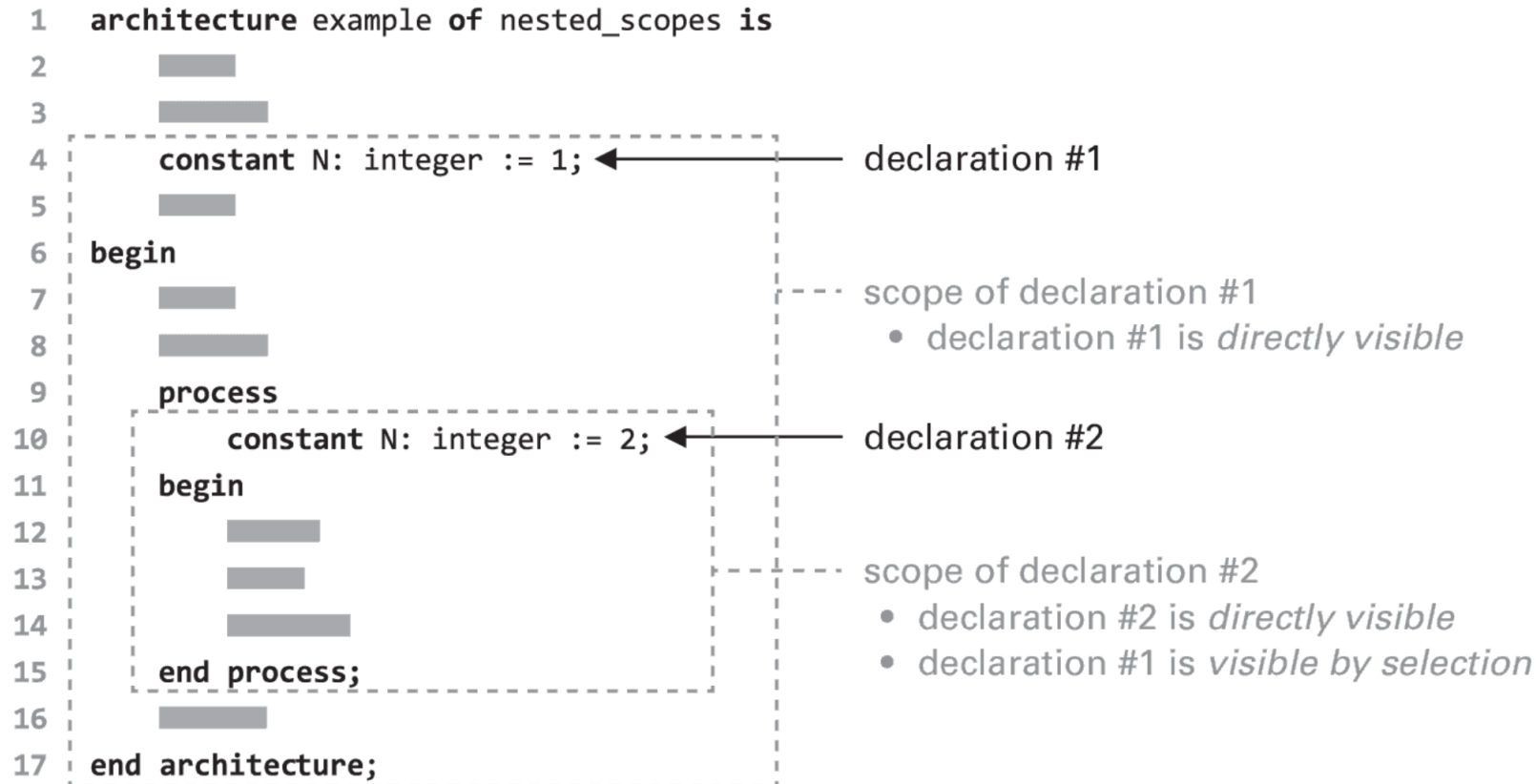


# Statements





# Declarations and Visibility



# 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

---



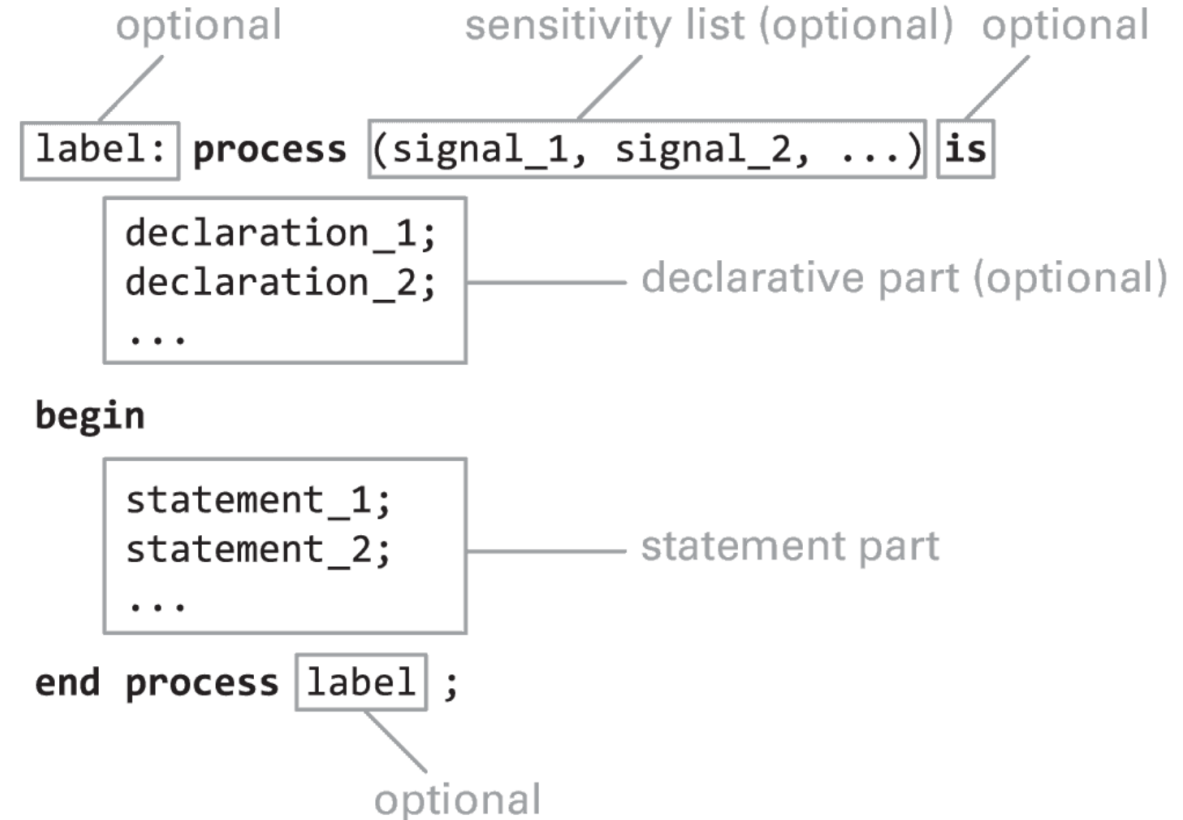
# Processes

---



# 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 sensity list awakes the process for executions



# 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





# 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 sensitivity 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;
```





**UNIVERSITY  
OF TURKU**