OOPs concepts

- **Inheritance**

The procedure in which one class inherits the attributes and methods of another class (Parent class).

OOPs concepts - Inheritance

Class Instantiation	Class Inheritance

class living_creatures;
  int eyes=2;
  int nose=1;
endclass
 	
class birds;
  int eyes, nose;
  living_creatures l1=new;
  this.eyes = l1.eyes;
  this.nose = l1.nose;
  int wings;
endclass
	class living_creatures;
	  int eyes=2;
	  int nose=1;
	endclass
	
	class birds **extends** living_creatures;
	
	  int wings;
	
	endclass

this, super & local (data hiding)

this

to access current class properties

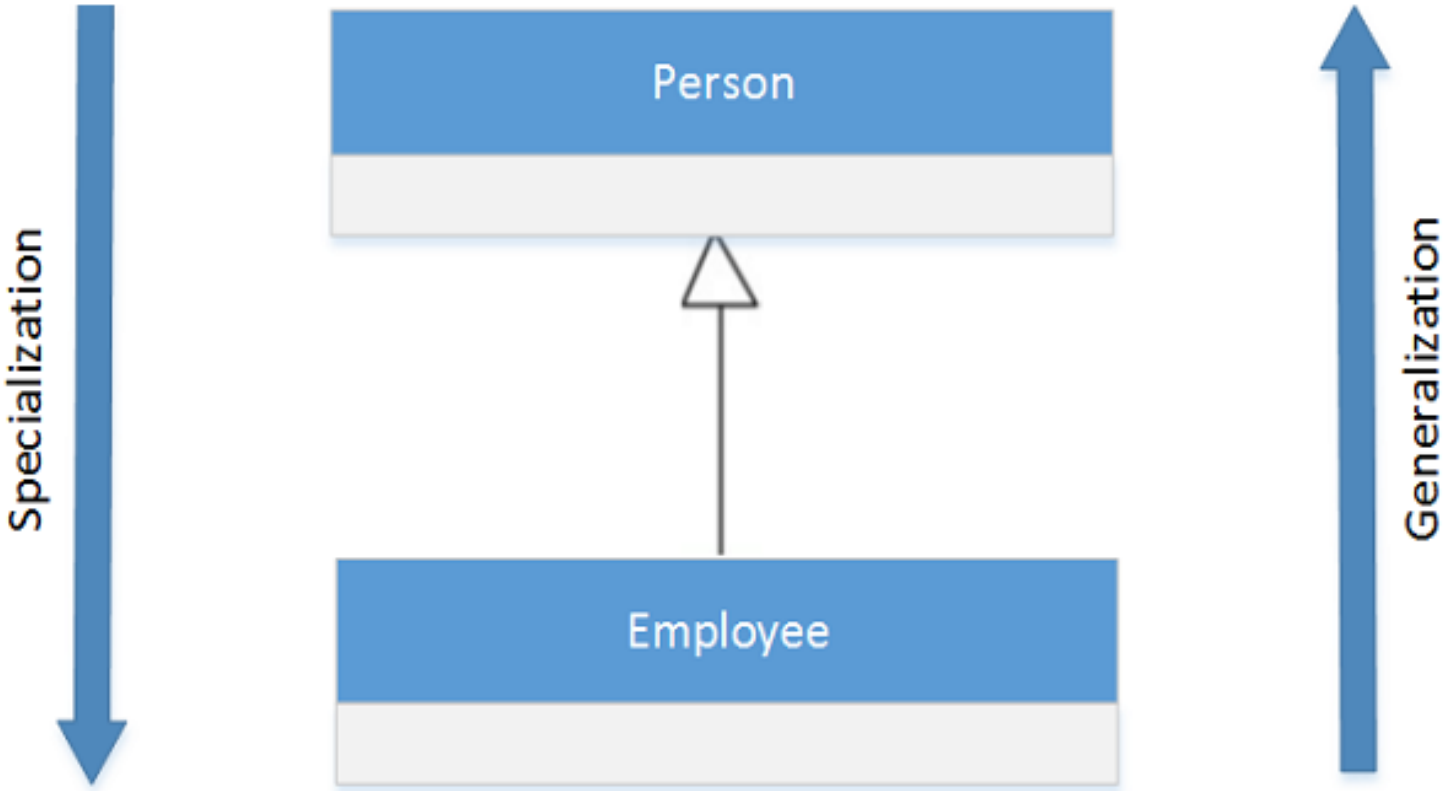
```
class a;
   int i;
   int delay=5;
    local int j;

   function new(int i);
      this.i = i;
   endfunction

endclass
```

super

to access parent class properties

```
class b extends a;
   int result;
   int delay=10;

   function print;
      result = this.delay + super.delay;
   endfunction

endclasss
```

Generalization



The Employee class extends the Person class (Inheritance)

```java
// super or parent class
class Person
{
    static int count = 1;
    String name;
    int age;
    String gender;
}
```

```java
// subclass or child of Person
class Student extends Person
{
    // name, age and gender
    // get inherited from Person
    int rollNo;
    String course;
}
```

```java
// subclass of Person
class Teacher extends Person
{
    // name, age and gender
    // get inherited from Person
    static int count = 20;
    String subject;
    int experience;
}
```

```java
class Test {
    public static void main(String[] s)
    {
        Teacher t = new Teacher();
        t.name = "Shaan";

        // name gets inherited in teacher
        System.out.println(t.name);

         // will give priority to its own count
        System.out.println(t.count);
    }
}
```

Chain Constructor

super.new()

```
class a;
endclass : a


class b extends a;
function new();
   super.new();
endfunction
endclass : b
```

1 level of inheritance

super.super.new X

Chain Constructor

```
class a;
  int a1 = 1;
  function new();
    a1 = 2;
  endfunction
endclass

class b extends a;
  int b1 = 3;
  function new();
    super.new();
    b1 = a1;
  endfunction
endclass

module c();
  b bb;
  initial begin
    bb = new();
    $display(bb.b1);
  end
endmodule
~
```

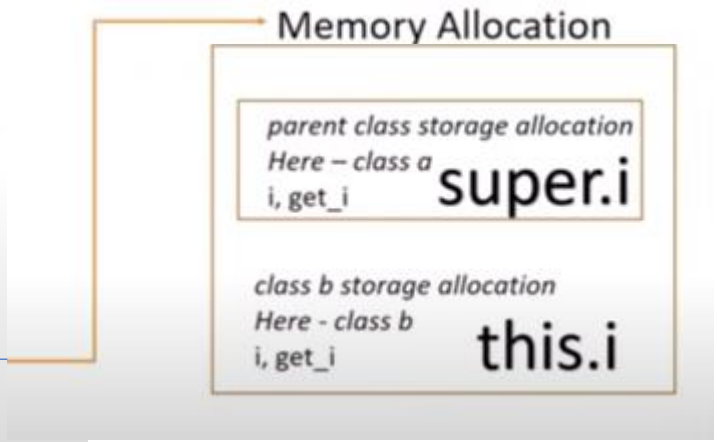
```
class b extends a;
  int b1 = 3;
  function new();
//    super.new();
    b1 = a1;
  endfunction
endclass

module c();
  b bb;
  initial begin
    bb = new();
    $display(bb.b1);
  end
endmodule
~
```

```
class b extends a;
  int b1 = 3;
  function new();
//    super.new(5);
    b1 = a1;
  endfunction
```

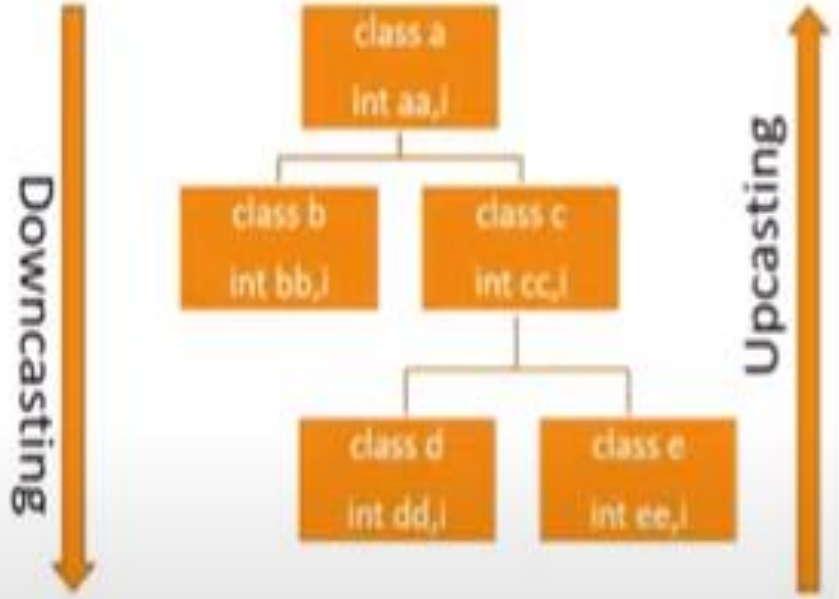
```
class a;
  int a1 = 1;
  function new(int i);
    a1 = 2;
  endfunction
endclass

class b extends a;
  int b1 = 3;
  function new();
    super.new(5);
    b1 = a1;
  endfunction
endclass

module c();
  b bb;
  initial begin
    bb = new();
    $display(bb.b1);
  end
endmodule
~
```

Inheritance Memory Allocation

```
class a;
   int i;
   function get_i; endfunction;
endclass : a

class b extends a;
   int i;
   function get_i; endfunction;
endclass : b

b = new();
```



Overriden members – Upcasting & Downcasting



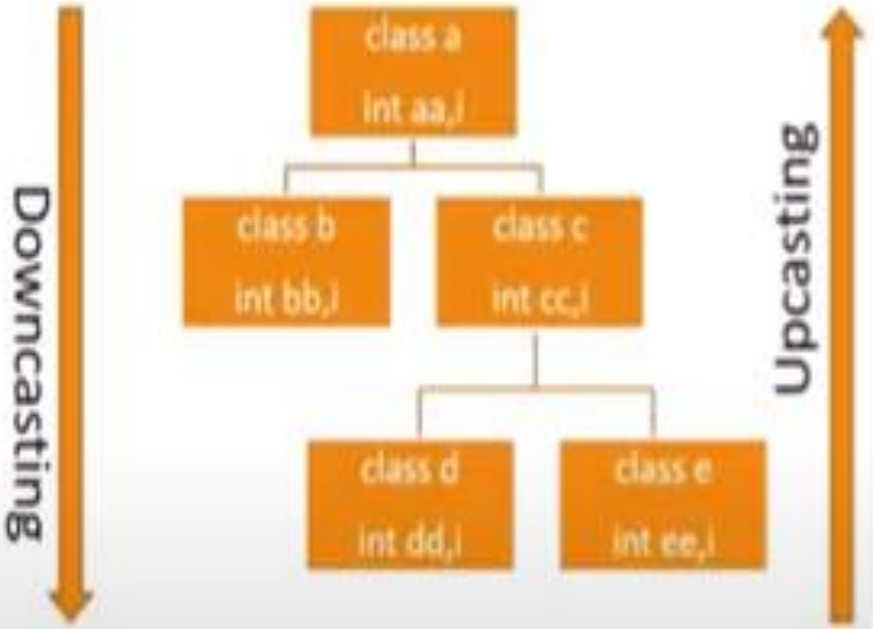
Upcasting – casting e into a – (a = e) **LEGAL**

```
e e1; a a1;
e1 = new;
a1 = e1;
```

Downcasting – a into e – (e = a) **ILLEGAL**

```
a a1; e e1;
a1 = new;
$cast(e1, a1);
```

Overriden members – Upcasting & Downcasting



Upcasting – casting e into a – (a = e) **LEGAL**

```
e e1; a a1;
e1 = new;
a1 = e1;
```

what happens?
when e1.i & a1.i

Downcasting – a into e – (e = a) **ILLEGAL**

```
a a1; e e1;
a1 = new;
$cast(e1, a1);
```

Upcasting

```
class birds;
  int size_inch = 20;
  int lifetime = 10;
  string colour = "brown";

  function void get_properties();
    $display(size_inch, lifetime, colour);
  endfunction
endclass

class parrot extends birds;
  string colour = "green";
  function void get_properties();
    $display(size_inch, lifetime, colour);
  endfunction
endclass

module a;
  birds b;
  parrot p;
  initial begin
    p = new();
    b = p;
    p.get_properties;
    b.get_properties;
  end
endmodule
```

```
20          10greën
20          10brown
  V C S   S i m u l a t i o n   R e p o r t
```

```
class B;
virtual task print();
$display(" CLASS B ");
endtask
endclass
```

```
class E_1 extends B;
virtual task print();
$display(" CLASS E_1 ");
endtask
endclass
```

```
class E_2 extends B;
virtual task print();
$display(" CLASS E_2 ");
endtask
endclass
```

```
program main;
initial
begin
B b;
E_1 e1;
E_2 e2;

e1 = new();
$cast(b,e1);
b.print();

end
endprogram
```

OOPs – Downcasting

The following cast failes because a superclass object not be read as an childclass.

```
m_base = new();
$cast(m_extend, m_base); // destination type != source object type
```

To cast correctly the object of the source handle must be compatible with the destination typetypes must be comparable:

```
m_extend = new();
m_base = m_extend;
$cast(m_extend, m_base); // destination type == source object type
```

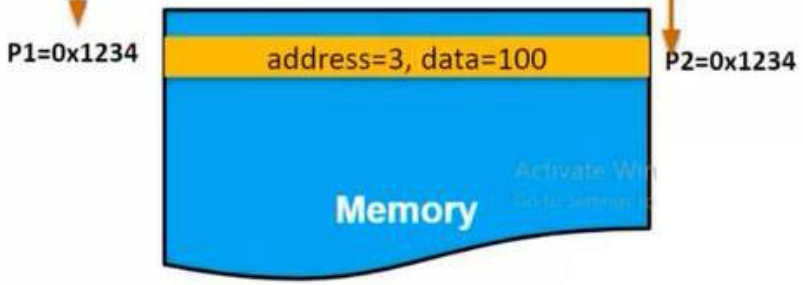
- $cast can be called as a function that will return a boolean indicating whether the cast was successful or not.
- Eg: if ($cast(extended class_handle, base_class handle))

Assignment, Renaming

- Declaring a class variable only creates the name by which object is known.
- Class can be assigned or renamed to one another class handle
- Assigned or renamed class handle shall point to same memory

```
Packet P1;  // P1 hold the handle of object of the class Packet,
P1 = new;
Packet P2;
P2=P1;
// only one object with 2 names P1, P2, created only once
```

```
// Class Declaration
class Packet;
   bit [3:0] address;
   logic [15:0] data;
   function new(); //initialization
        address = 3;
        data = 100;
    endfunction
endclass

// module
module assign_rename;
  Packet P1, P2;  //declare P1, P2 handles for Packet
  initial
    P1 = new();  // Initialize/construct packet
    P2 = P1;
endmodule
```

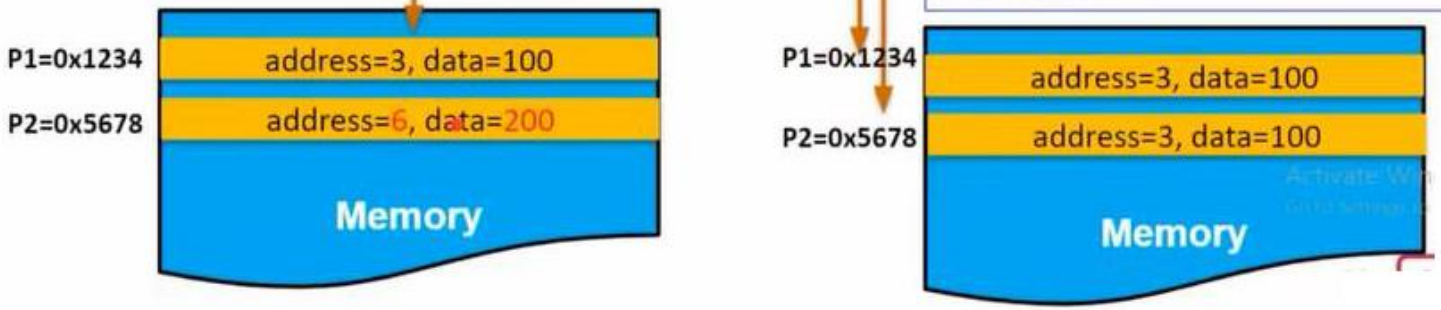


Copying – Shallow Copy

- A shallow copy copies all the variables, handles of the object instance, but not the nested objects.

```
Packet P1;
Packet P2;
P1=new;
P2 = new P1;
```

```
// Class Declaration
class Packet;
   bit [3:0] address;
   logic [15:0] data;
   function new(); //initialization
        address = 3;
        data = 100;
    endfunction
endclass

// module
module shallow_copy;
 Packet P1, P2;  //declare P1, P2 handles for Packet
  initial
    P1 = new();
    P2 = new P1;
    P2.address=6; P2.data=200
 endmodule
```



```
//Shallow Copy New
class header;
  int id;
  function new (int id);
    this.id = id;
  endfunction
endclass

class packet;
  int addr;
  int data;
  header hdr;

  function new(int addr, int data, int id);
    hdr = new(id);
    this.addr = addr;
    this.data = data;
  endfunction

  function void print (string name);
    $display("[%s] addr=0x%0h, data=0x%0h,id=%0d", name, addr, data, hdr.id);
  endfunction

endclass
```

```
module shallow_copy;
  packet p1, p2;

  initial begin
    p1 = new(32'hAB, 32'h1234, 26);
    p1.print("Packet P1");
    p2 = new p1;
    p2.print("Packet P2");
    p2.addr   = 'hFE;
    p2.hdr.id = 36;
    p2.print("Packet P2");
    p1.print("Packet P1");
  end

endmodule
```

```
xcelium> run
[Packet P1] addr=0xab, data=0x1234,id=26
[Packet P2] addr=0xab, data=0x1234,id=26
[Packet P2] addr=0xfe, data=0x1234,id=36
[Packet P1] addr=0xab, data=0x1234,id=36
```

Copying – deep copy

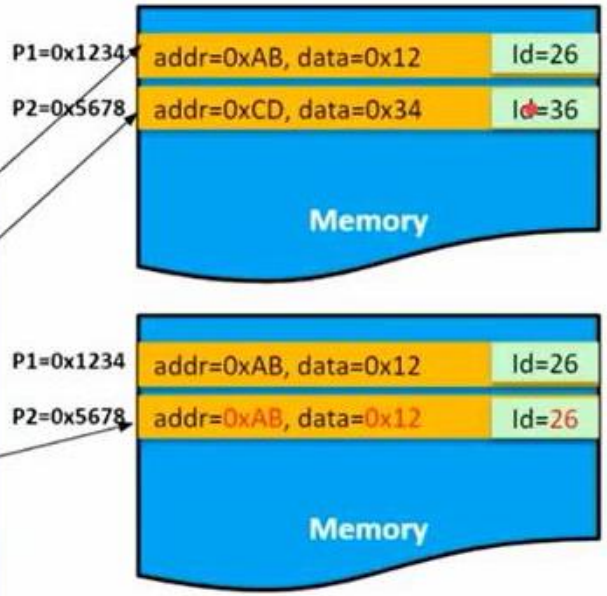
- A deep copy copies all fields, and makes copies of dynamically allocated memory pointed to by the fields.
- To make a deep copy, you must write a copy constructor and overload the assignment operator, otherwise the copy will point to the original, with disastrous consequences

```
Packet P1 = new;
Packet P2 = new;
P2 = copy(P1);
```

```
// header class
class Header;
 int id;

 function new (int id);
   this.id = id;
 endfunction

Endclass
```

```
// class packet
class Packet;
   int    addr;
   int    data;
   Header       hdr;
   function new (int addr, int data, int id);
          hdr = new (id);
          this.addr = addr;
          this.data = data;
   endfunction

endclass
```

```
// module
module deep_copy ;
  Packet P1, P2;

  initial begin
    // Create a new pkt object called P1
    P1 = new (32'hAB, 32'h12, 26);

    // deep copy P1 into P2
    P2 = new (32'hCD, 32'h34, 36);
    P2.copy(P1);
  end


endmodule
```



Deep Copy Example

```
// header class
class Header;
 int id;
 function new (int id);
   this.id = id;
 endfunction
 function showId();
   $display("id=0x%0d", id);
 endfunction
endclass
// class packet
class Packet;
  int    addr;
  int    data;
  Header       hdr;
  function new (int addr, int data, int id);
        hdr = new (id);
        this.addr = addr;
        this.data = data;
  endfunction
  function copy (Packet p);
    this.addr = p.addr;
    this.data = p.data;
    this.hdr.id = p.hdr.id;
  endfunction

  function print (string name);
   $display ("[%s] addr=0x%0h data=0x%0h id=%0d", name, addr, data, hdr.id);
  endfunction
endclass;
```

```
// module
module deep_copy;
       Packet P1, P2;

        initial begin
        // Create a new pkt object called P1
        P1 = new (32'hAB, 32'h1234, 26);
        P1.print("Packet P1");

        // deep copy P1 into P2
        P2 = new (32'hCD, 32'h5678, 36);
        P2.copy(P1);
        P2.print("Packet P2");

        // Now let's change the addr, data and id in P1
        P1.addr = 32'h12EF;
        P1.data = 32'hA5A5;
        P1.hdr.id = 46;
        P1.print("Packet P1");

        // addr is not changed
        P2.print ("Packet P2");
      end
endmodule
```

```
[Packet P1] addr=0xab data=0x1234 id=26
[Packet P2] addr=0xab data=0x1234 id=26
[Packet P1] addr=0x12ef data=0xa5a5 id=46
[Packet P2] addr=0xab data=0x1234 id=26
```

OOPs concepts

- **Inheritance**

The procedure in which one class inherits the attributes and methods of another class (Parent class).

- **Polymorphism**

Ability of the object in belonging to different types with specific behavior of each type.

OOPs concepts - Polymorphism

- Method overloading would be an example of static polymorphism
- Method overriding would be an example of dynamic polymorphism

```
class Calculation {
  void sum(int a,int b){System.out.println(a+b);}
  void sum(int a,int b,int c){System.out.println(a+b+c);}

  public static void main(String args[]) {
    Calculation obj=new Calculation();
    obj.sum(10,10,10);  // 30
    obj.sum(20,20);     //40
  }
}
```

Static binding/Compile-Time binding/Early binding/Method overloading.(in same class)

OOPs concepts - Polymorphism

- Method overloading would be an example of static polymorphism
- Method overriding would be an example of dynamic polymorphism

Dynamic binding/Run-Time binding/Late binding/Method overriding.(in different classes)

```
class Animal {
   public void move(){
      System.out.println("Animals can move");
   }
}
class Dog extends Animal {

   public void move() {
      System.out.println("Dogs can walk and run");
   }
}

public class TestDog {

   public static void main(String args[]) {
      Animal a = new Animal(); // Animal reference and object
      Animal b = new Dog(); // Animal reference but Dog object

      a.move();//output: Animals can move

      b.move();//output:Dogs can walk and run
   }
}
```

OOPs concepts - Polymorphism

Same code – works differently – for different types.

Static – parameterized types … *Dynamic – virtual method*

virtual function

code reuse ---------- HOW ?

OOPs concepts - Polymorphism

```
class birds;
  int size_inch = 20;
  int lifetime = 10;
  string colour = "brown";

  virtual function void get_properties();
    $display(size_inch, lifetime, colour);
  endfunction
endclass

class parrot extends birds;
  string colour = "green";
  virtual function void get_properties();
    $display(size_inch, lifetime, colour);
  endfunction
endclass

class m_parrot extends parrot;
  string colour = "multi";
  virtual function void get_properties();
    $display(size_inch, lifetime, colour);
  endfunction
endclass

module a;
  birds b;
  parrot p;
  m_parrot m;
  initial begin
    m = new();
    p = m;
    p.get_properties;
  end
endmodule
```

```
20          10multi
  V C S   S i m u l a t i o n   R e p o r t
```

OOPs concepts – Abstract class and pure virtual method

virtual class;

endclass

-- can't take its instance ; only can extend

in virtual class – can't define method ; so can declare pure virtual method without body

virtual class;

```
  pure virtual function void get_properties;    // no – implementation
endclass
```

```
virtual class birds;
  int size_inch = 20;
  int lifetime = 10;
  string colour = "brown";

  pure virtual function void get_properties();
endclass
```

OOPs concepts - Polymorphism

- Polymorphism is a process of accessing a child methods from the parent handle with
  - Both the parent and the child method should have same prototype (i.e. both function/task having same names)
  - Method of a parent class should be virtual.

OOPs concepts

- **Inheritance**

The procedure in which one class inherits the attributes and methods of another class (Parent class).

- **Polymorphism**

Ability of the object in belonging to different types with specific behavior of each type.

- **Abstraction**

Used to hide certain details and only show essential features of the object (deals with outside view of the object).

OOPs – Pure virtual function

There are two major differences between a virtual and a pure virtual function, these are below:

- There CAN'T be a definition of the pure virtual function in the base class.
- There MUST be a definition of the pure virtual function in the derived class.

OOPs concepts – Parameterization

- Writing code that can be reused across wide range of applications.

```
1  // Declare parameterized class
2  class <name_of_class> #(<parameters>);
3  class Trans #(addr = 32);
4
5  // Override class parameter
6  <name_of_class>  #(<parameters>) <name_of_inst>;
7  Trans #(.addr(16)) obj;
```

```
class Base#(int size = 3);
bit [size:0] a;

task disp();
$display(" Size of the vector a is %d ",$size(a));
endtask
endclass

program main();

initial
begin
Base B1;
Base#(4) B2;
Base#(5) B3;
B1 = new();
B2 = new();
B3 = new();
B1.disp();
B2.disp();
B3.disp();
end
endprogram
```

```
Size of the vector a is 4
Size of the vector a is 5
Size of the vector a is 6
```

```
class something #(int size = 8);
    bit [size-1:0] out;
endclass
```

```
module tb;

  // Override default value of 8 with the given values in #()
  something  #(16) sth1;              // pass 16 as "size" to this class object
  something  #(.size (8)) sth2;       // pass 8 as "size" to this class object
  typedef something #(4) td_nibble;   // create an alias for a class with "size" = 4 as "nibble"
  td_nibble nibble;

  initial begin
    // 1. Instantiate class objects
    sth1 = new;
    sth2 = new;
    nibble = new;

    // 2. Print size of "out" variable. $bits() system task will return
    // the number of bits in a given variable
    $display ("sth1.out   = %0d bits", $bits(sth1.out));
    $display ("sth2.out   = %0d bits", $bits(sth2.out));
    $display ("nibble.out = %0d bits", $bits(nibble.out));
  end
endmodule
```

OOPs concepts – Parameterization

```
class List #(type T = int);
//attributes:
T data_node;
......
......
// methods:
task append(T element);
function T getFirst();
function T getNext();
......
......
endclass
```

```
List#(packet) pl; // Object pl is a list of packet
List#(string) sl; // Object sl is a list of strings
```

OOPs concepts – Parameterization

- If a class A is parameterized with a data type B, A is called template class.
  - Once an object of A is created, B is replaced by an actual data type.
- This allows the definition of an actual class based on the template specified for A and the actual data type.

Extending Parameterized Class

```
class C #(type T = bit);
...
endclass // base class

class D1 #(type P = real) extends C; // T is bit (the default)

class D2 #(type P = real) extends C #(integer); // T is integer

class D3 #(type P = real) extends C #(P); // T is P
```

OOPs concepts – Encapsulation

- **Parameterization**

  - Writing code that can be reused across wide range of applications.

- **Encapsulation**

  - Separates an object's state from its behavior.
  - Helps in hiding an object's data describing its state from further modification by external component.

OOPs concepts - Encapsulation

```
class base;
local int i;
endclass

program main;
initial
begin
base b = new();
b.i = 123;
end
endprogram
```

```
Local member 'i' of class 'base' is not accessible from scope 'main'
```

OOPs concepts - Encapsulation

```
class base;
local int i;

task set(int j);
i = j;
$display(i);
endtask
endclass

program main;
initial
begin
base b = new();
b.set(123);
end
endprogram
```

OOPs concepts - Encapsulation

```
class base;
local int i;
endclass

class ext extends base;
function new();
i = 10;
endfunction
endclass
```

Local member 'i' of class 'base' is not accessible from scope 'ext'

OOPs concepts - Encapsulation

```
class base;
protected int i;
endclass

class ext extends base;
function new();
i = 10;
endfunction
endclass
```

Local member 'i' of class 'base' is not accessible from scope 'ext'

```
class base;
protected int i;
endclass

program main;
initial
begin
base b = new();
b.i = 123;
end
endprogram
```

Protected member 'i' of class 'base' is not accessible from scope 'main'

- A protected class property or method has all of the characteristics of a local member, except that it can be inherited; it is visible to subclasses

OOPs concepts - Encapsulation

- If a class item is declared as *local*, then it can be used within the method of that class only and won't be visible to its subclass
- If a class item is declared as *protected*, the item will be visible to the class inheriting the super-class.

OOPs concepts - Packages

- Provide mechanism to store and share data, methods, property, parameters that can be re-used in multiple other modules, interfaces, or programs.
- SystemVerilog provides package support to help share following:
  - Parameters
  - Data
  - Type
  - Task
  - Function
  - Sequence
  - Property
- Few rules that should be followed with packages:
  - Packages cannot contain any assign statement.
  - Variable declaration assignments within the package shall occur before any initial, always (_ff, _comb, _latch).
  - Items within packages cannot have hierarchical references.
  - Assign statement on any net type is not allowed.

A package is to define a utility for common use.

OOPs concepts - Packages

- Access data, functions or types in packages using:
  - class scope resolution operator ::
  - import statement (the import statement provides direct visibility of identifiers within packages.

```
1  package my_pkg;
2      typedef enum bit [1:0] { RED, YELLOW, GREEN, RSVD } e_signal;
3      typedef struct { bit [3:0] signal_id;
4                       bit       active;
5                       bit [1:0] timeout;
6                     } e_sig_param;
7
8      function common ();
9          $display ("Called from somewhere");
10     endfunction
11
12     task run ( ... );
13         ...
14     endtask
15 endpackage
```

OOPs concepts - Packages

```
1  // Import the package defined above to use e_signal
2  import my_pkg::*;
3
4  class myClass;
5      e_signal    my_sig;
6  endclass
7
8  module tb;
9      myClass cls;
10
11     initial begin
12         cls = new ();
13         cls.my_sig = GREEN;
14         $display ("my_sig = %s", cls.my_sig.name());
15         common ();
16     end
17 endmodule
```

```
ncsim> run
my_sig = GREEN
Called from somewhere
ncsim: *W,RNQUIE: Simulation is complete.
```

OOPs concepts – Namespace collision

```
1  package my_pkg;
2      typedef enum bit { READ, WRITE } e_rd_wr;
3  endpackage
4
5  import my_pkg::*;
6
7  typedef enum bit { WRITE, READ } e_wr_rd;
8
9  module tb;
10     initial begin
11         e_wr_rd     opc1 = READ;
12         e_rd_wr     opc2 = READ;
13       $display ("READ1 = %0d READ2 = %0d ", opc1, opc2);
14     end
15 endmodule
```

```
ncsim> run
READ1 = 1 READ2 = 1
ncsim: *W,RNQUIE: Simulation is complete.
```

OOPs concepts – Namespace collision

```
1  module tb;
2      initial begin
3          e_wr_rd     opc1 = READ;
4          e_rd_wr     opc2 = my_pkg::READ;
5        $display ("READ1 = %0d READ2 = %0d ", opc1, opc2);
6      end
7  endmodule
```

```
ncsim> run
READ1 = 1 READ2 = 0
ncsim: *W,RNQUIE: Simulation is complete.
```

OOPs concepts – package vs `include

```
class A;
int i;
endclass : A
```

```
class B;
int i;
endclass : B
```

```
package P;
class A;
int i;
endclass : A
A a1;
endpackage : P
```

```
package Q;
class A;
int i;
endclass : A
A a1;
endpackage : Q
```

File A.sv

```
class A;
int i;
endclass : A
```

File P.sv

```
package P;
`include "A.sv"
A a1;
endpackage : P
```

File Q.sv

```
package Q;
`include "A.sv"
A a1;
endpackage : Q
```

File A.sv

```
class A;
int i;
endclass : A
```

File P.sv

```
package P;
`include "A.sv"
endpackage : P
```

File R.sv

```
package R;
import P::A;
A a1;
endpackage : R
```

File S.sv

```
package S;
import P::A;
A a1;
endpackage : S
```

OOPs concepts – Class library

- A class library is a pre-coded OOPs template collection.

- Class libraries enhance code reuse by providing implementations of repetitive jobs.

OOPs concepts

Classes

Objects

Static Variables

Static Methods

Inheritance

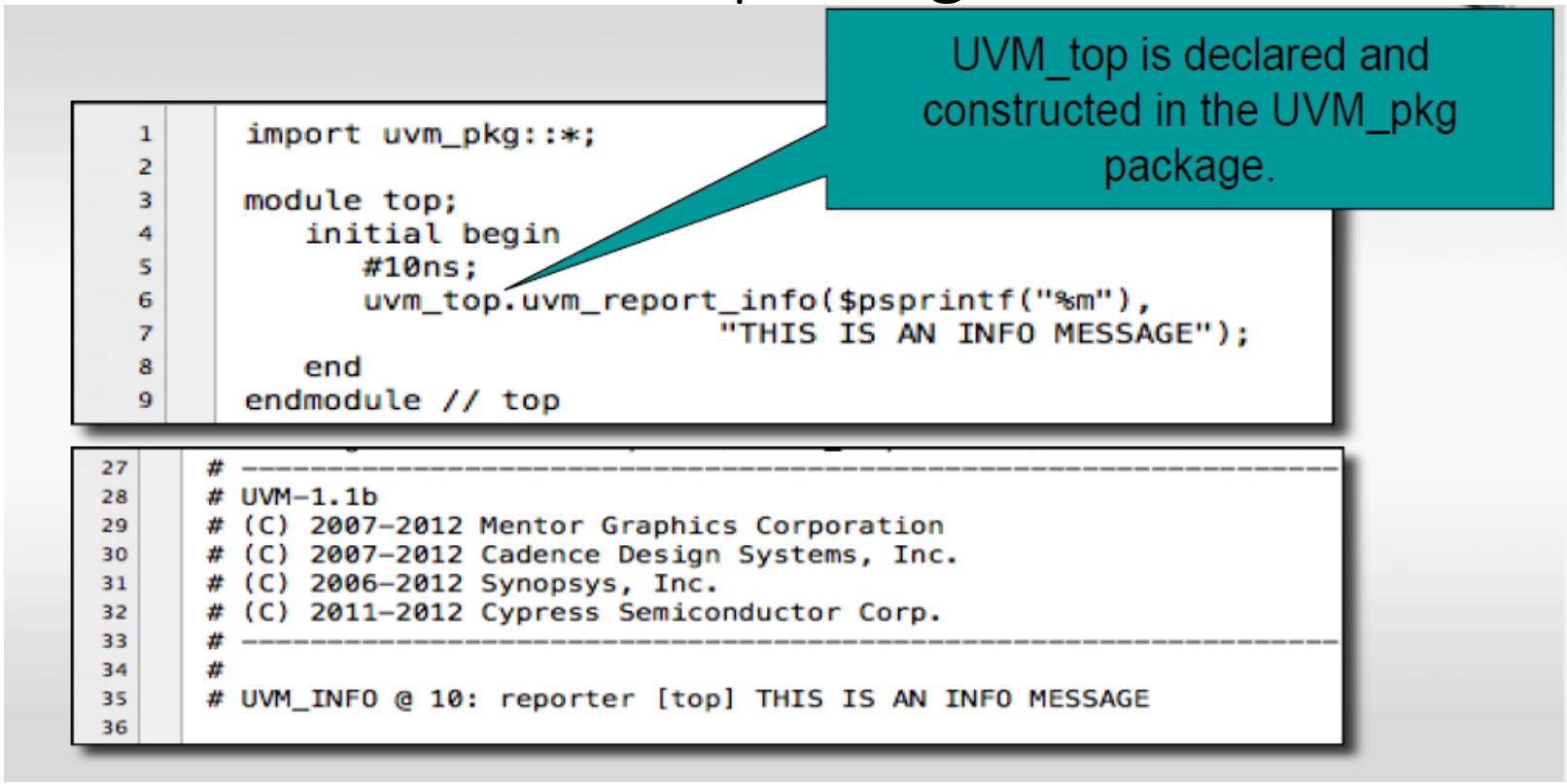
Polymorphism

Parameterization

Encapsulation

Packages

UVM is delivered as a package



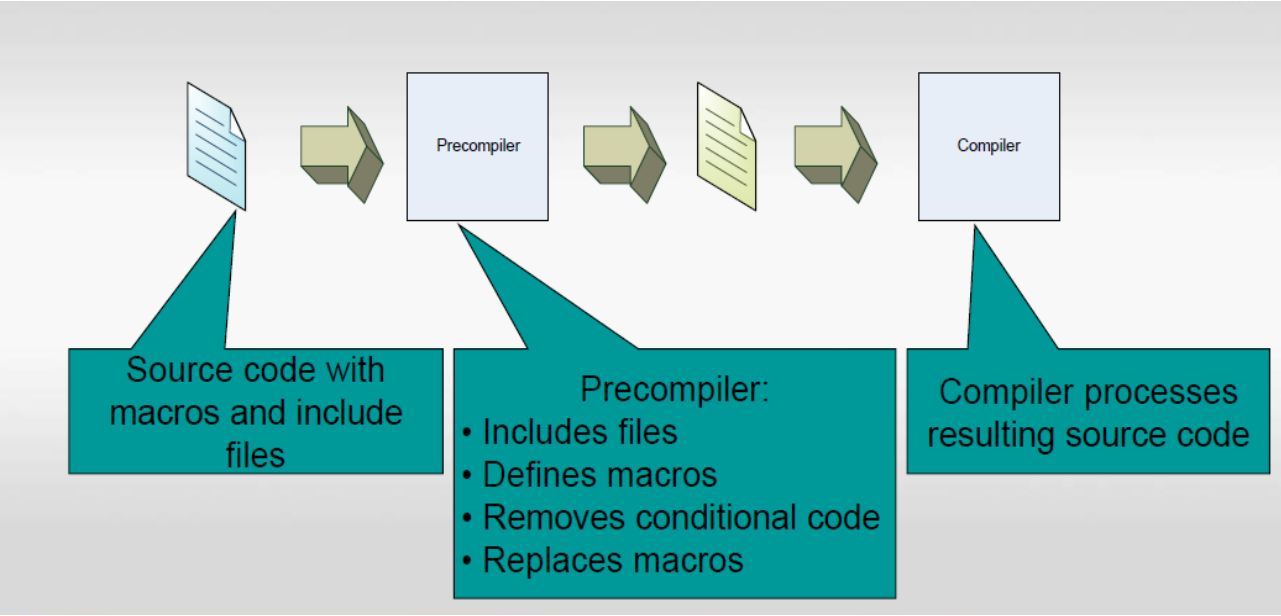
- uvm_top is declared and instantiated inside uvm_pkg
  - Here, uvm_top calls the reporting function

Packages

- **Provide a common name space**
- **Can be imported**
- **Can contain common objects**

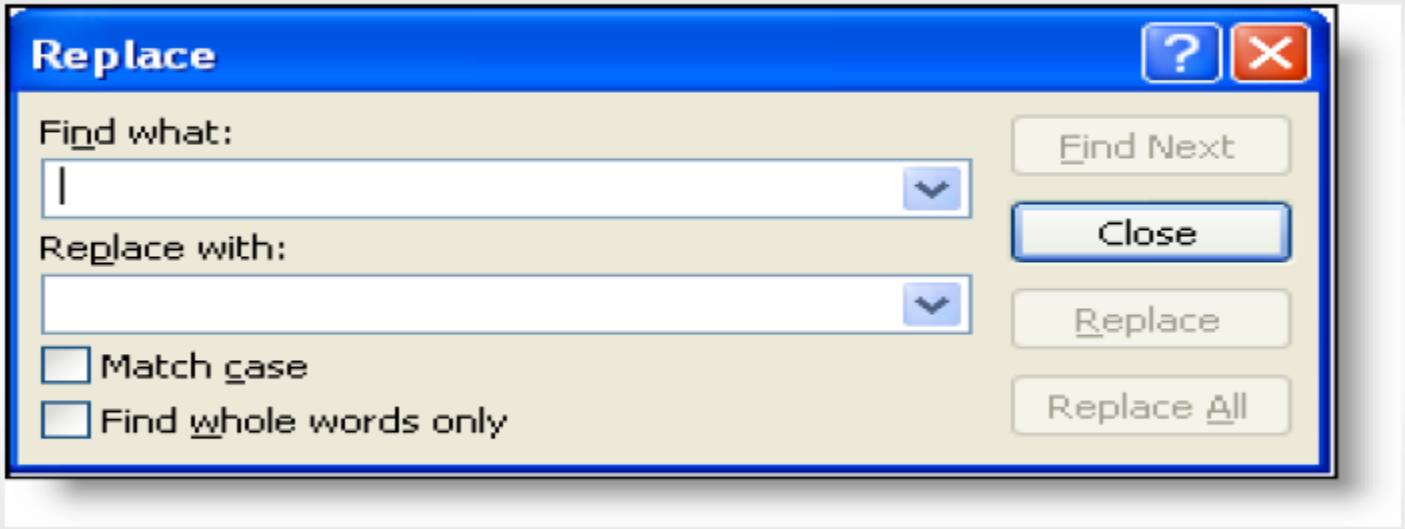
MACROS

SystemVerilog Pre-Compiler



- SV, like C, has a precompiler, some source code that has macros and source code etc in it.
- The precompiler includes files, processes macros, creates a bit of code that's ready to be compiled.
- That code goes into the compiler. Just like C, there's macros and include files that get managed by the precompiler.

Macros deliver ‘find and replace’ functions



Macro Example

```
1   `define info(msg)      uvm_top.uvm_report_info($psprintf("%m"), msg);
2   `define warning(msg)   uvm_top.uvm_report_warning($psprintf("%m"), msg);
3   `define error(msg) \
4                          uvm_top.uvm_report_error($psprintf("%m"), msg);
5   `define fatal(msg)     uvm_top.uvm_report_fatal($psprintf("%m"), msg);
6
7   import uvm_pkg::*;
8
9   module top;
10    initial begin
11        `info ("My INFO message");
12        `warning("My WARNING message");
13        `error("My ERROR message");
14        `fatal("It's FATAL!");
15    end
16  endmodule // top
17
```

Macros are not function calls. They are textual replacements

```
`info ("My INFO message");
```

becomes

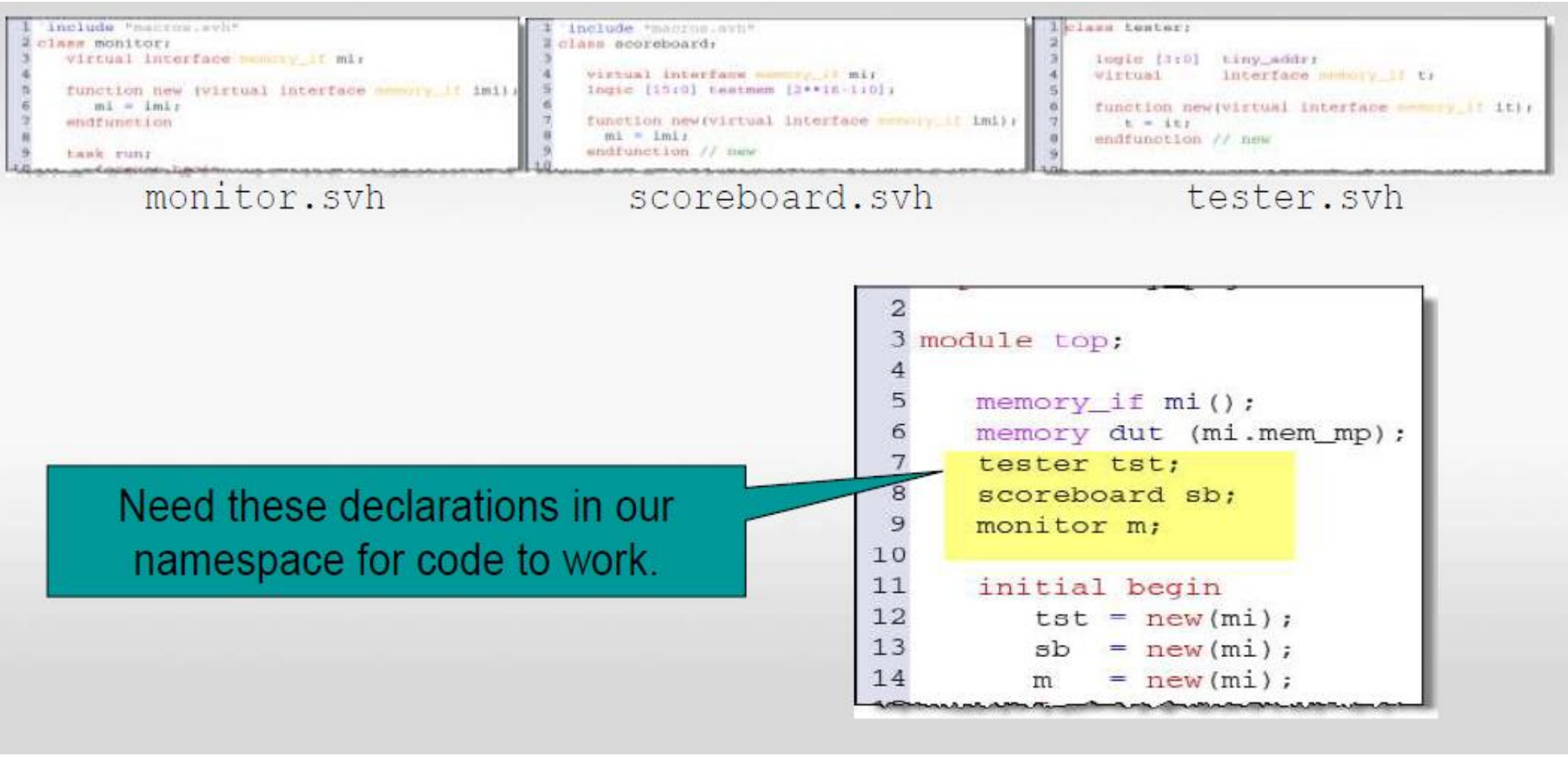
```
uvm_top.uvm_report_info($psprintf("%m"), "My INFO message");;
```

Running Report Macros

```
27    # ----------------------------------------------------------------
28    # UVM-1.1b
29    # (C) 2007-2012 Mentor Graphics Corporation
30    # (C) 2007-2012 Cadence Design Systems, Inc.
31    # (C) 2006-2012 Synopsys, Inc.
32    # (C) 2011-2012 Cypress Semiconductor Corp.
33    # ----------------------------------------------------------------
34    #
35    # UVM_INFO @ 0: reporter [top] My INFO message
36    # UVM_WARNING @ 0: reporter [top] My WARNING message
37    # UVM_ERROR @ 0: reporter [top] My ERROR message
38    # UVM_FATAL @ 0: reporter [top] It's FATAL!
39    #
40    # --- UVM Report Summary ---
41    #
42    # ** Report counts by severity
43    # UVM_INFO :    3
44    # UVM_WARNING :    1
45    # UVM_ERROR :    1
46    # UVM_FATAL :    1
47    # ** Report counts by id
48    # [Questa UVM]     2
49    # [top]     4
50    # ** Note: $finish    : /tools/mentor/questa/10.1c_1/questasim/linux
51    #    Time: 0 ns  Iteration: 0  Instance: /top
```

Sharing Declarations

How do I get my declarations into many files?



One Solution: Include Files

```
1   import uvm_pkg::*;
2
3   `include "tester.svh"
4   `include "monitor.svh"
5   `include "scoreboard.svh"
6
7   module top;
8
9       memory_if mi();
10      memory dut (mi.mem_mp);
11      tester tst;
12      scoreboard sb;
13      monitor m;
14
15      initial begin
16          tst = new(mi);
```

Pro: Easy to Understand

Con: Long List of Files

Con: Need to change filenames in many places

Another solution: Packages

```
1   package memory_pkg;
2      import uvm_pkg::*;
3
4   class tester;
5
6      logic [3:0]  tiny_addr;
```

```
3
4   class tester;
5
6      logic [3:0]  tiny_addr;
7      virtual      interface memory_if t;
8
9      function new(virtual interface memory_if it);
10        t = it;
11     endfunction // new
```

```
34
35  class monitor;
36     virtual interface memory_if mi;
37
38     function new (virtual interface memory_if imi);
39        mi = imi;
40     endfunction
41
42     task run;
```

```
51
52  class scoreboard;
53
54     virtual interface memory_if mi;
55     logic [15:0] testmem [2**16-1:0];
56
57     function new(virtual interface memory_if imi);
```

```
69              end
70
71              if (mi.wr)
72                testmem[mi.addr] = mi.data;
73           end
74        endtask // run
75     endclass // scoreboard
76
77
```

Pros:

- Single file
- One compile

Con:

- One very big file

Suggested Solution: Combine Includes and Packages

Import UVM package into your package

```
1  package memory_pkg;
2      import uvm_pkg::*;
3  `include "tester.svh"
4  `include "monitor.svh"
5  `include "scoreboard.svh"
6  endpackage // memory_pkg
7
```

Notice that imports do not chain. You must import all namespaces explicitly

We must import UVM_pkg into source file

```
1  import uvm_pkg::*;
2  import memory_pkg::*;
3
4  module top;
5
6      memory_if mi();
7      memory dut (mi.mem_mp);
8      tester tst;
9      scoreboard sb;
```

Pros:

- One file per class
- Only compiled once
- All changes in one place

Con:

- Package must be compiled before it is imported.

Summary

- **Create .svh files to hold each class**
- **Create a package to deliver all class declarations**
- **Include the .svh files in the package file**