```
1  //=============================================================
2  // 1st code == demonstrating factory override using inheritance in UVM// Import UVM package and
3  include UVM macros
4  /*
5  import uvm_pkg::*; // Import all UVM base classes
6  `include "uvm_macros.svh" // Include UVM macros (like factory registration)
7  // -------------------------------------------------------------
8  // Base sequence item: write_xtn
9  // ------------------------------------------------------------- class write_xtn extends uvm_sequence_item;
10 rand int a; // Declare a random integer variable
11 `uvm_object_utils(write_xtn) // Register with factory
12 // Constructor
13 function new(string name = "write_xtn");
14 super.new(name); // Call base class constructor
15 endfunction
16 // Constraint: a must be between 6 and 14
17 constraint valid_a { a > 5; a < 15; }
18 endclass
19 // -------------------------------------------------------------
20 // Derived sequence item: small_xtn (child of write_xtn)
21 // ------------------------------------------------------------- class small_xtn extends write_xtn;
22 `uvm_object_utils(small_xtn) // Register derived class with factory
23 // Constructor
24 function new(string name = "small_xtn");
25 super.new(name); // Call parent constructor
26 endfunction
27 // Override constraint: a must be exactly 9
28 constraint valid_a { a == 9; }
29 endclass
30 // -------------------------------------------------------------
31 // Declare a handle of base type (write_xtn)
```

```
32  // ------------------------------------------------------------ write_xtn xtn_h;
33  // ------------------------------------------------------------
34  // Top module
35  // ------------------------------------------------------------ module top;
36  // Task to create and randomize object
37  task call();
38  xtn_h = write_xtn::type_id::create("xtn_h"); // Create object using factory
39  xtn_h.randomize(); // Randomize it
40  $display("The value of a is %0d", xtn_h.a); // Display the value of 'a' endtask
41  initial begin
42  // ---------------- First call without override ---------------- $display("--- Without Override ---");
43  call(); // Will create object of write_xtn
44  // ---------------- Set type override ----------------
45  factory.set_type_override_by_type(write_xtn::get_type(), small_xtn::get_type());
46  // Now any factory creation of write_xtn will return small_xtn
47  // ---------------- Second call with override ---------------- $display("--- With Override ---");
48  call(); // Will create object of small_xtn (overridden)
49  // ---------------- Print factory override info ----------------
50  factory.print();
51  end
52  endmodule
53  */
54  //=========================================================================
55  //=========================================================================
56  // demonstrating object comparison in UVM
57  /*
58  `include "uvm_macros.svh" // Include UVM macros
59  import uvm_pkg::*; // Import all UVM classes
60  // Define a class named 'transaction' which extends uvm_object
61  class transaction extends uvm_object;
62  rand bit[15:0] addr; // Random address field (16-bit)
```

```
63  rand bit[15:0] data; // Random data field (16-bit)
64  // Register fields for UVM automation (factory, copy, compare, print)
65  `uvm_object_utils_begin(transaction)
66  `uvm_field_int(addr, UVM_PRINT) // Enable print for addr
67  `uvm_field_int(data, UVM_PRINT) // Enable print for data
68  `uvm_object_utils_end
69  // Constructor for transaction object
70  function new(string name = "transaction");
71  super.new(name); // Call base class constructor
72  endfunction
73  endclass // End of transaction class
74  // Define base_test class which extends uvm_test
75  class base_test extends uvm_test;
76  transaction tr1, tr2; // Declare two transaction objects to compare
77  uvm_comparer comp; // UVM object for comparing objects
78  `uvm_component_utils(base_test) // Register class with factory
79  // Constructor for base_test
80  function new(string name = "base_test", uvm_component parent = null);
81  super.new(name, parent); // Call base class constructor
82  endfunction
83  // Build phase: create objects using factory
84  function void build_phase(uvm_phase phase);
85  super.build_phase(phase); // Call base build phase
86  // Create tr1 and tr2 using UVM factory
87  tr1 = transaction::type_id::create("tr1", this);
88  tr2 = transaction::type_id::create("tr2", this);
89  // Create comparer object manually
90  comp = new();
91  endfunction
92  // Run phase: do randomization and comparison
93  task run_phase(uvm_phase phase);
```

```
94  super.run_phase(phase); // Call base run phase
95  // Randomize both transaction objects
96  assert(tr1.randomize()); // Randomize tr1 and check success
97  assert(tr2.randomize()); // Randomize tr2 and check success
98  // Configure comparer settings
99  comp.verbosity = UVM_LOW; // Set log level to low (minimal)
100 comp.sev = UVM_ERROR; // Severity for mismatches = error
101 comp.show_max = 100; // Show up to 100 mismatches
102 // Print info before first comparison
103 `uvm_info(get_full_name(), "Comparing objects", UVM_LOW)
104 // Compare the two transaction objects (likely different)
105 comp.compare_object("tr_compare", tr1, tr2);
106 // Now copy tr1 into tr2 (make them identical)
107 tr2.copy(tr1);
108 // Compare again (they should now match)
109 comp.compare_object("tr_compare", tr1, tr2);
110 // Print final comparison result
111 `uvm_info(get_full_name(), $sformatf("Comparing objects: result = %0d", comp.result), UVM_LOW)
112 // Compare mismatched integers (expect error)
113 comp.compare_field_int("int_compare", 5'h2, 5'h4, 5); // Mismatch: 0x2 vs 0x4
114 // Compare mismatched strings (expect error)
115 comp.compare_string("string_compare", "name", "names"); // Mismatch: extra 's' // Show result
116 after mismatches
117 `uvm_info(get_full_name(), $sformatf("Comparing objects: result = %0d", comp.result), UVM_LOW)
118 // Now compare matching values
119 comp.compare_field_int("int_compare", 5'h4, 5'h4, 5); // Match
120 comp.compare_string("string_compare", "name", "name"); // Match
121 endtask
122 endclass // End of base_test
123 // Top-level module to run UVM test
124 module tb_top;
```

```
initial begin
run_test("base_test"); // Run the test named "base_test" end
endmodule
*/
//=========================================================================
// This UVM code is for building a basic UVM testbench structure that
//demonstrates the creation and connection of components: driver, agent, environment, andtest. //It
showcases how UVM phases (build, connect, run)
//work and how components interact in a layered UVM testbench. /*
`include "uvm_macros.svh" // Include UVM macros (e.g., `uvm_info, `uvm_component_utils)
import uvm_pkg::*; // Import all UVM types and classes fromthe UVMpackage
//======================== DRIVER ========================//
class driver extends uvm_driver; // Define a class `driver` that extends uvm_driver (baseclass for
drivers)
`uvm_component_utils(driver) // Register the class with the UVMfactory (enablescreate by name)
function new(string name = "driver", uvm_component parent); // Constructor with default
name and parent
super.new(name, parent); // Call the parent class constructor
endfunction
virtual function void build_phase(uvm_phase phase); // build_phase: for creating sub- components,
setting config
super.build_phase(phase); // Always call base class's build_phase
`uvm_info("driver", // UVM info message with ID "driver"
"am in the build of driver", // Message string
UVM_MEDIUM); // Verbosity level: UVM_MEDIUM
endfunction
virtual function void connect_phase(uvm_phase phase); // connect_phase: used toconnect
ports/exports
super.connect_phase(phase); // Call base class's connect_phase
`uvm_info("driver", // UVM info message
"am in the connect of driver", UVM_MEDIUM); // Medium verbosity
endfunction
```

```
157 task run_phase(uvm_phase phase); // run_phase: simulation run-time behavior
158 phase.raise_objection(this); // Raise objection to keep simulation running
159 `uvm_info("driver", // Log message during run phase
160 "am in the run phase of driver", UVM_MEDIUM);
161 phase.drop_objection(this); // Drop objection to allow phase to end
162 endtask
163 endclass
164 //======================== AGENT ========================//
165 class agent extends uvm_agent; // Agent class: groups sequencer, driver, monitor
166 `uvm_component_utils(agent) // Register agent with factory
167 driver drv_h; // Handle for driver component
168 function new(string name = "agent", uvm_component parent); // Constructor
169 super.new(name, parent); // Call parent class constructor
170 endfunction
171 virtual function void build_phase(uvm_phase phase); // build_phase of agent
172 super.build_phase(phase); // Call base build_phase
173 `uvm_info("agent", "am in the build of agent", UVM_MEDIUM); // Info message
174 drv_h = driver::type_id::create("drv_h", this); // Create driver using factory
175 drv_h.set_report_verbosity_level(UVM_MEDIUM); // Set verbosity of the driver tomedium
176 endfunction
177 virtual function void connect_phase(uvm_phase phase); // connect_phase of agent
178 super.connect_phase(phase); // Base call `uvm_info("agent", "am in the connect of agent",
179 UVM_MEDIUM); // Log connectionendfunction
180 task run_phase(uvm_phase phase); // Agent run phase
181 phase.raise_objection(this); // Raise objection to delay phase end
182 `uvm_info("agent", "am in the run phase of agent", UVM_MEDIUM); // Print
183 messagephase.drop_objection(this); // Drop objection
184 endtask
185 endclass
186 //======================== ENVIRONMENT ========================//
187 class env extends uvm_env; // env class: top-level container for agents, scoreboards, etc.
188 `uvm_component_utils(env) // Factory registration
```

```
189 agent agnt_h; // Agent handle
190 function new(string name = "env", uvm_component parent); // Constructor
191 super.new(name, parent); // Call base class constructor
192 endfunction
193 virtual function void build_phase(uvm_phase phase); // build_phase of env
194 super.build_phase(phase); // Call base method
195 `uvm_info("env", "am in the build of env", UVM_MEDIUM); // Info message
196 agnt_h = agent::type_id::create("agnt_h", this); // Create agent using factory
197 agnt_h.set_report_verbosity_level(UVM_MEDIUM); // Set verbosity for agent
198 endfunction
199 virtual function void connect_phase(uvm_phase phase); // connect_phase of env
200 super.connect_phase(phase); // Base connect
201 `uvm_info("env", "am in the connect of env", UVM_MEDIUM); // Log message
202 endfunction
203 task run_phase(uvm_phase phase); // env run_phase
204 phase.raise_objection(this); // Raise objection to keep run phase active
205 `uvm_info("env", "am in the run phase of env", UVM_MEDIUM); // Info log
206 phase.drop_objection(this); // Drop objection to allow phase to end
207 endtask
208 endclass
209 //======================== TEST ========================//
210 class basetest extends uvm_test; // Test class extending uvm_test (entry point for
211 testbench)
212 `uvm_component_utils(basetest) // Register test with factory
213 env env_h; // Handle to environment
214 function new(string name = "basetest", uvm_component parent); // Constructor
215 super.new(name, parent); // Call base constructor
216 endfunction
217 virtual function void build_phase(uvm_phase phase); // build_phase of test
218 super.build_phase(phase); // Base class build
219 `uvm_info("test", "am in the build of test", UVM_MEDIUM); // Log
```

```
220 env_h = env::type_id::create("env_h", this); // Create env
221 endfunction
222 virtual function void connect_phase(uvm_phase phase); // connect_phase of test
223 super.connect_phase(phase); // Base connect
224 `uvm_info("test", "am in the connect of test", UVM_MEDIUM); // Log
225 endfunction
226 virtual function void end_of_elaboration_phase(uvm_phase phase); // Called after
227 build/connect
228 `uvm_info("test", "am in the end_of_elaboration of test", UVM_MEDIUM); // Info
229 uvm_top.print_topology(); // Print full UVM hierarchy (for debug)
230 endfunction
231 task run_phase(uvm_phase phase); // run_phase of test
232 phase.raise_objection(this); // Keep run phase active
233 #10; // Wait 10 time units
234 `uvm_info("test", "am in the run phase of test", UVM_MEDIUM); // Log
235 phase.drop_objection(this); // Allow run phase to end
236 endtask
237 endclass
238 //========================= TOP MODULE =========================//
239 module top;
240 initial begin
241 uvm_top.set_report_verbosity_level(UVM_MEDIUM); // Set default verbosity for theentiretestbench
242 run_test("basetest"); // Run the test by name ("basetest") using UVMfactory
243 end
244 endmodule
245 */
246 //=====================================================================
247 //This UVM code is for **demonstrating custom `do_print()` and `do_copy()` methods inasequence
248 item**, //showing how to manually implement printing and copying of transaction data (`address`
249 and `data`)
250 //instead of using built-in UVM macros like `uvm_field_int`.
251 /*
```

```
252 // Include the UVM macros file, which provides useful macros for UVMfunctionality
253 `include "uvm_macros.svh" // Import the UVM package to use UVM classes and methods
254 import uvm_pkg::*;
255 // Define a transaction class that extends uvm_sequence_item
256 class write_xtn extends uvm_sequence_item;
257 // Declare two 3-bit randomizable variables for address and data
258 rand bit [2:0] address;
259 rand bit [2:0] data;
260 // Register the class with the UVM Factory to enable dynamic creation
261 `uvm_object_utils(write_xtn)
262 `uvm_object_utils_begin(write_xtn)
263 // Enable UVM automation for 'address' and 'data' (printing, copying, comparing, etc.)
264 `uvm_field_int(address, UVM_ALL_ON)
265 `uvm_field_int(data, UVM_ALL_ON)
266 `uvm_object_utils_end
267 // Constructor for the write_xtn class
268 function new(string name = "write_xtn");
269 // Call the parent class (uvm_sequence_item) constructor
270 super.new(name);
271 endfunction
272 virtual function void do_print(uvm_printer printer);
273 $display("Called By Print Method");
274 printer.print_field("address", address, 3,UVM_DEC);
275 printer.print_field("data", data, 3, UVM_DEC);
276 endfunction
277 virtual function void do_copy(uvm_object rhs);
278 write_xtn temph;
279 $cast(temph,rhs);
280 this.address = temph.address;
281 this.data = temph.data;
282 endfunction
```

```
283 endclass
284 // Define a testbench module
285 module top;
286 // Declare object handles for the transaction class
287 write_xtn xtn_h1, xtn_h2, xtn_h3;
288 // Initial block to execute the test
289 initial begin
290 // Create an instance of write_xtn using the UVM factory
291 xtn_h1 = write_xtn::type_id::create("xtn_h1");
292 // Randomize the values of address and data
293 xtn_h1.randomize();
294 // Print the values of address and data in a formatted UVM output
295 xtn_h1.print();
296 xtn_h1.print(uvm_default_tree_printer);
297 xtn_h1.print(uvm_default_line_printer);
298 xtn_h1 = write_xtn :: type_id ::create("xtn_h2");
299 xtn_h2.copy(xtn_h1);
300 xtn_h1.print();
301 end
302 endmodule
303 */
304 //=========================================================================
305 ///This UVM code is for **demonstrating custom implementation of `do_print()`, //`do_copy()`,
306 object comparison using `compare()`, and cloning using `clone()` in a
307 //`uvm_sequence_item`-based transaction**. It shows manual handling of print, //copy, compare,
308 and clone behaviors typically automated by UVM macros. /*
309 // Include the UVM macros file, which provides useful macros for UVMfunctionality`include
310 "uvm_macros.svh" // Import the UVM package to use UVM classes and methods
311 import uvm_pkg::*;
312 // Define a transaction class that extends uvm_sequence_item
313 class write_xtn extends uvm_sequence_item;
314 // Declare two 3-bit randomizable variables for address and data
```

```
315     rand bit [2:0] address;
316     rand bit [2:0] data;
317     // Register the class with the UVM Factory to enable dynamic creation
318     `uvm_object_utils(write_xtn)
319     // Constructor for the write_xtn class
320     function new(string name = "write_xtn");
321     super.new(name);
322     endfunction
323     // Print method override
324     virtual function void do_print(uvm_printer printer);
325     $display("Called By Print Method");
326     printer.print_field("address", address, 3, UVM_DEC);
327     printer.print_field("data", data, 3, UVM_DEC);
328     endfunction
329     // Copy method override
330     virtual function void do_copy(uvm_object rhs);
331     write_xtn temph;
332     if (!$cast(temph, rhs)) begin
333     `uvm_error("COPY", "Casting failed in do_copy")
334     return;
335     end
336     this.address = temph.address;
337     this.data = temph.data;
338     endfunction
339     endclass
340     // Define a testbench module
341     module top;
342     // Declare object handles for the transaction class
343     write_xtn xtn_h1, xtn_h2, xtn_h3;
344     // Initial block to execute the test
345     initial begin
```

```
346  // Create an instance of write_xtn using the UVM factory
347  xtn_h1 = write_xtn::type_id::create("xtn_h1");
348  xtn_h1.randomize();
349  // Print the values of address and data in a formatted UVM output
350  xtn_h1.print();
351  xtn_h1.print(uvm_default_tree_printer);
352  xtn_h1.print(uvm_default_line_printer);
353  // Create another instance and copy data
354  xtn_h2 = write_xtn::type_id::create("xtn_h2");
355  xtn_h2.copy(xtn_h1);
356  xtn_h2.print();
357  // Compare two objects
358  if (xtn_h2.compare(xtn_h1))
359  $display("H1 AND H2 BOTH ARE SAME");
360  // Clone an object
361  xtn_h3 = write_xtn::type_id::create("xtn_h3");
362  xtn_h3 = write_xtn::type_id::create("xtn_clone");
363  xtn_h3 = write_xtn::type_id::create("xtn_clone");
364  $cast(xtn_h3, xtn_h1.clone());
365  xtn_h3.print();
366  end
367  endmodule
368  */
369  //=========================================================================
370  //6
371  //This UVM code is for **demonstrating UVM automation macros
372  //(`uvm_field_int`) for print, copy, compare, and clone operations** in a
373  //`uvm_sequence_item`-based transaction. It simplifies these operations using macro-
374  based//automation instead of overriding `do_print()`, `do_copy()`, etc.
375  /*
376  // Include the UVM macros file, which provides useful macros for UVM functionality
```

```
377  `include "uvm_macros.svh" // Import the UVM package to use UVM classes and methods
378  import uvm_pkg::*;
379  // Define a transaction class that extends uvm_sequence_item
380  class write_xtn extends uvm_sequence_item;
381  // Declare two 3-bit randomizable variables for address and data
382  rand bit [2:0] address;
383  rand bit [2:0] data;
384  // Register the class with the UVM Factory to enable dynamic creation
385  `uvm_object_utils_begin(write_xtn)
386  // Enable UVM automation for 'address' and 'data' (printing, copying, comparing, etc.)
387  `uvm_field_int(address, UVM_ALL_ON)
388  `uvm_field_int(data, UVM_ALL_ON)
389  `uvm_object_utils_end
390  // Constructor for the write_xtn class
```