

# UVM Cheat Sheet

---

## Concept 1 – Transactions (`uvm_sequence_item`)

**Purpose** : Represent a single DUT operation (e.g., counter's reset + enable signals).

**Base Class** : `uvm_sequence_item`

### \* How to Define

```
class counter_item extends uvm_sequence_item;
  rand bit reset;
  rand bit enable;
```

```
  constraint valid { !(reset && enable); }
```

```
  `uvm_object_utils(counter_item)
  function new(string name="counter_item"); super.new(name); endfunction
endclass
```

### Action

- ① Declare `rand` fields for stimulus
- ② Add constraints for valid combinations
- ③ Register with factory →  
`uvm_object_utils()`
- ④ Optionally add `copy`, `print`, `compare` for debugging

→ “The data the DUT sees on its pins.”

### Randomization

```
item.randomize() with { enable == 1; count < 15; };
```

---

## Concept 2 – Sequences & Sequencer Basics

**Key Classes** : `uvm_sequence`, `uvm_sequencer`, `uvm_driver`

### Flow

Sequence → Sequencer → Driver → DUT

### Example

```
class counter_seq extends uvm_sequence#(counter_item);
  `uvm_object_utils(counter_seq)
  task body();
    repeat(5) begin
      counter_item tr = counter_item::type_id::create("tr");
      start_item(tr);
      assert(tr.randomize());
      finish_item(tr);
    end
  endtask
endclass
```

### Handshake

```
drv.seq_item_port.connect(seqr.seq_item_export);
seq_item_port.get_next_item(tr);
seq_item_port.item_done();
```

### Important APIs / Macros

API / Macro	Use
<code>start_item</code> , <code>finish_item</code>	Explicit sequence control
<code>`uvm_do</code> / <code>`uvm_do_with</code>	Legacy shorthand
<code>seq.start(seqr)</code>	Launch sequence
<code>seq_item_port</code> / <code>seq_item_export</code>	Link sequencer ↔ driver

---

## ? Common Doubts (Answered)

**Q 1. Why do we need both `my_if_if` and `vif` variables?**

→ `my_if_if` is the *real* interface instance connected to the DUT; `vif` is a *virtual handle* inside UVM classes that points to it via `uvm_config_db`.

**Q 2. Why `super.build_phase(phase)` ?**

→ It calls the parent class implementation to preserve hierarchy and factory control.

**Q 3. Why `::type_id::create()` inside build phase ?**

→ It uses the factory to create components so overrides can work later.

---

# Advanced Sequences & Communication

---

## Concept 3 – Nested / Virtual Sequences + Virtual Sequencer

**Virtual Sequence** : Coordinates multiple child sequences in multi-agent SoCs.

**Virtual Sequencer** : Holds handles to sub-sequencers (APB, UART, FIFO).

### Example

```
class system_virtual_sequencer extends uvm_sequencer;
  `uvm_component_utils(system_virtual_sequencer)
  apb_sequencer apb_seqr;
  uart_sequencer uart_seqr;
endclass
```

```
class system_virtual_seq extends uvm_sequence;
  `uvm_object_utils(system_virtual_seq)
  task body();
    apb_seq apb_s = apb_seq::type_id::create("apb_s");
    uart_seq uart_s = uart_seq::type_id::create("uart_s");
    fork
      apb_s.start(p_sequencer.apb_seqr);
      uart_s.start(p_sequencer.uart_seqr);
    join
  endtask
endclass
```

### Why use it ?

- To run multiple protocols in sync
- To create system-level tests (APB writes before UART transmit)
- To reuse per-agent sequences without rewriting logic

**Remember:** Not mandatory for beginners; essential for multi-interface or SoC verification later.

---

## Concept 4 – Objection Mechanism

Used to control when simulation ends.

### Typical Structure

```
task run_phase(uvm_phase phase);  
  phase.raise_objection(this);  
  `uvm_info("TEST","Running sequence",UVM_LOW)  
  seq.start(env.agt.seqr);  
  phase.drop_objection(this);  
endtask
```

### Purpose

- Prevents UVM from ending run\_phase prematurely
- Ensures simulation ends only after all activity completes

Function	Meaning
<code>raise_objection (this)</code>	Keeps simulation alive
<code>drop_objection( this)</code>	Allows phase to finish

**Common Mistake :** Missing `drop_objection` → simulation hangs forever.

**Other Mistake :** Missing `raise_objection` → simulation ends instantly at time 0.

 Rule of Thumb → Always use `raise` / `drop` in balanced pairs.

---

## Concept 5 – Configuration Propagation & Factory Overrides

### Configuration Propagation ( `uvm_config_db` )

Used to pass **virtual interfaces** and parameters through the hierarchy.

#### Top Level :

```
uvm_config_db#(virtual counter_if)::set(null,"*", "vif", cif);
```

#### Inside Driver:

```
if(!uvm_config_db#(virtual counter_if)::get(this,"", "vif", vif))  
    `uvm_fatal("DRV", "Virtual interface not set!")
```

✅ Allows higher-level components (test/env) to configure lower-level ones without hard-coding.

#### Debug Tip:

Use `uvm_config_db::dump()` to view what's stored and where.

---

### Factory Overrides Review

Type	Scope	Example	Use
<b>Type Override</b>	Global – replaces all instances	<code>set_type_override_by_type(old_driver::get_type(), new_driver::get_type())</code>	For global behavior change
<b>Instance Override</b>	Specific path only	<code>set_inst_override_by_name("old_drv", "new_drv", "uvm_test_top.env.agt.drv")</code>	For targeted replacement

✅ Used to subclass drivers/monitors without editing their original files.

✅ Remember to use `::type_id::create()` so the factory can replace instances.

### Typical Use :

Replace a base driver with a coverage-enabled or error-inject driver.

---

### Doubt Recap

**Q : Why do we use `::type_id::create()` for sequences even though we don't build them in `build_phase`?**

→ Sequences are objects (`uvm_object`, not components); they don't use build phase but still need factory awareness for overrides.

**Q : Why use factory instead of `new()`?**

→ To enable reusability and swapping subclasses without changing code.

**Q : Why does simulation hang sometimes?**

→ Usually due to missing `drop_objection(this)` or driver waiting forever for `get_next_item()`.

---

## Revision Checklist

Concept	Must Know	Notes
Transaction	<code>uvm_sequence_item</code> basics, <code>randomize()</code>	Foundation
Sequence & Driver Handshake	<code>start_item</code> , <code>finish_item</code> , <code>seq_item_port</code>	Core flow
Virtual Sequence	Controls multiple agents	Advanced but important later
Objections	<code>raise/drop</code> mechanism	Prevent hang or early exit
Config DB	Pass interfaces/params	Reduces manual connections
Factory Overrides	Type vs Instance	For reusability & subclassing

---