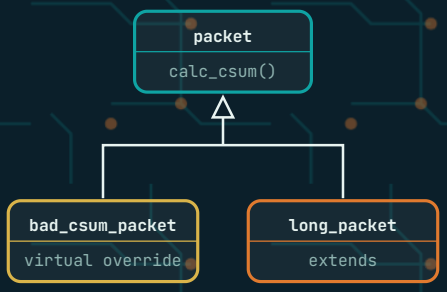
STUDY PAGE · AREA 04 OF 14

SystemVerilog OOP & Classes

04

UVM is a class library. Learn classes, handles, deep copies, inheritance and virtual methods — then debug a class-based testbench whose bugs live in the classes.




```
packet p = new bad_csum_packet
p.calc_csum() → child version
```

18 h of study · **4** modules · **5** diagrams · **2** interactive · **Free** tools · Verilator 5 · **20**-question test · 70% to pass · Lab: packet generator + checksum checker

1 · Classes (objects · handles) → 2 · Copying (shallow vs deep) → 3 · Inheritance (virtual · $cast) → 4 · Static & params (generators) → Lab · Packets (two class bugs) → Area test · 70% (20 questions)

BEFORE YOU START · A

Who this area is for

The bridge to UVM. Every UVM component, sequence item and test you will write is a class.



EE / ECE students

Final-year and master's students preparing for VLSI internships and campus placements in design or verification.

Software engineers

Developers moving into hardware who want the foundations explained from first principles.

QA & test engineers

Test automation people heading into design verification — the same stimulus-and-check mindset applied to silicon.

FPGA & embedded engineers

Engineers who write some HDL already and want the rules and habits that verification teams expect.

RTL designers

Designers who want to move to verification and think like the person checking their code.

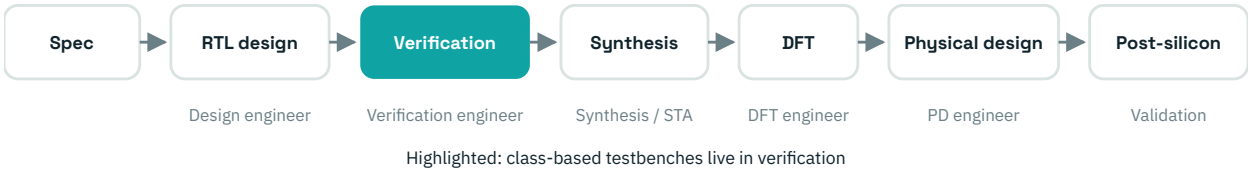
Career returners

Engineers refreshing fundamentals before interviews after time away from hands-on work.

Prerequisites

You need	Level	Where to get it
SystemVerilog types, queues, mailboxes	Comfortable	Area 03
Tasks, functions, automatic	Comfortable	Area 03 · Module 3
Object-oriented programming (any language)	Helpful, not required	This area starts from zero

Where this area sits in the chip flow — and the roles that use it



Highlighted: class-based testbenches live in verification

OOP is how verification scales. Reusable transactions, components and tests are classes; designers rarely write them, verifiers write little else.

Role	How you use this area on the job	Typical first title
Design verification (DV)	Write sequence items, drivers, monitors and scoreboards as classes; debug copy and override bugs	DV Engineer / Verification Engineer I
RTL design	Read class-based testbenches to reproduce failures	ASIC Design Engineer / RTL Engineer
FPGA	Build reusable class-based testbenches for IP blocks	FPGA Engineer
DFT	Model patterns and expected responses as objects	DFT Engineer
Post-silicon validation	Reuse verification classes in post-silicon test content	Validation Engineer

BEFORE YOU START · B

What you will be able to do

Every outcome is practised in the lab and checked by the area test.

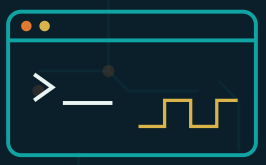


- Write a transaction class with a constructor, methods and convert2string.
- Explain handles vs objects and avoid null-handle crashes.
- Implement deep copy and field-wise compare.
- Extend classes, override methods and call super.
- Use virtual methods for polymorphism and $cast safely.
- Use static members, parameterised classes and typedef class.
- Apply the blueprint pattern to change generated stimulus.
- Fix a shallow-copy bug and a missing-virtual bug in a real testbench.

BEFORE YOU START · C

Tools & software

Classes need a SystemVerilog-capable simulator: Verilator 5 for free, or a commercial simulator on EDA Playground.



Tool	What it does	Used in this area for	Cost
Verilator (5.x)	Fast simulator that compiles Verilog/SystemVerilog to C++; also a strict linter	Every example and the lab (`make`)	Free
GTKWave or **Surfer**	Waveform viewers for `.vcd` / `.fst` dumps	Watching the byte stream into the checker	Free
VS Code + Verilog extension	Editor with syntax highlighting and lint on save	Writing code	Free
EDA Playground	Browser-based simulators, including commercial ones, after a free sign-up	Commercial simulators with full class support	Free
make and **git**	Run builds with one command; track your changes	Lab Makefile; saving your work	Free

Commercial simulators you will meet at work

Simulator	Vendor	Notes
VCS	Synopsys	Widely used for SystemVerilog/UVM regressions
Xcelium	Cadence	Successor of Incisive / NC-Sim
Questa	Siemens EDA	Successor of ModelSim; starter editions ship with FPGA tools

Same language, same rules

Verilator 5 supports classes, inheritance, virtual methods, parameterised classes and mailboxes used here. Randomisation with constraints (Area 05) is where commercial simulators matter most.

BEFORE YOU START · D

Install step by step

Pick your computer. About 15 minutes. If anything fails, the browser option works with nothing installed.



Windows

Recommended: WSL 2 (Ubuntu inside Windows). You get the same commands as Linux and as most companies' compute farms.

1 **Turn on WSL and install Ubuntu**

Open **PowerShell as Administrator**, run the command, then restart.

```
wsl --install
```

After the restart an Ubuntu window opens: choose a user name and password.

2 **Install the tools**

In the Ubuntu window:

```
sudo apt update
sudo apt install -y iverilog verilator gtkwave make git
```

3 **Install VS Code and connect it to WSL**

Install VS Code from code.visualstudio.com, add the **WSL** extension, then type `code .` in Ubuntu to open the current folder.

4 **Add Verilog support**

Install the **Verilog-HDL/SystemVerilog/Bluespec SystemVerilog** extension (publisher mshr-h).

5 **Check it works**

```
iverilog -V | head -1
verilator --version
```

> i **No WSL allowed on your laptop?**
> Install the native Icarus Verilog for Windows build (installer from bleyer.org/icarus, which also bundles GTKWave) and tick "Add executable folder to the user PATH". Use EDA Playground for Verilator.

macOS

1 **Install Apple's command-line tools**

```
xcode-select --install
```

2 **Install Homebrew**

Follow the one-line command on brew.sh, then reopen Terminal.

3 **Install the simulators**

```
brew install icarus-verilog verilator make git
```

4 **Install a waveform viewer**

Download **Surfer** from surfer-project.org (native on Apple silicon), or try `brew install --cask gtkwave`.

5 **Editor**

Install VS Code and the **Verilog-HDL/SystemVerilog/Bluespec SystemVerilog** extension.

6 **Check it works**

```
iverilog -V | head -1
verilator --version
```

Linux

1 **Ubuntu / Debian**

```
sudo apt update
sudo apt install -y iverilog verilator gtkwave make git
```

2 **Fedora**

```
sudo dnf install -y iverilog verilator gtkwave make git
```

3 **Arch**

```
sudo pacman -S iverilog verilator gtkwave make git
```

4 **Check the versions**

```
iverilog -V | head -1      # 11 or newer
verilator --version        # 5.x recommended
```

Browser only

1 **Open EDA Playground**

Go to edaplayground.com and create a free account (simulation needs a login).

2 **Paste the code**

Design on the right, testbench on the left.

3 **Choose a simulator**

Under *Tools & Simulators* pick a SystemVerilog-capable simulator (one of the commercial ones listed); tick *Open EPWave after run* for waveforms.

4

Run

Press **Run**; the log shows your `$display` output.

!

This area needs Verilator 5

Queues, classes, mailboxes and `fork` timing need Verilator 5.0 or newer with `--timing`. Ubuntu 24.04 and current Homebrew ship 5.x; check with `verilator --version`. On an older system, use EDA Playground or build Verilator from source (verilator.org → Installation). Icarus Verilog does not support associative arrays or most class features.

BEFORE YOU START · E

Your first class in 5 minutes

See handle assignment, a shallow copy and a virtual call in one run.



1

Make a folder and a file called `first.sv`

```
class packet;
  bit [7:0] addr;
  function new(bit [7:0] a = 0); addr = a; endfunction
  virtual function string kind(); return "packet"; endfunction
endclass

class bad_packet extends packet;
  function new(bit [7:0] a = 0); super.new(a); endfunction
  virtual function string kind(); return "bad_packet"; endfunction
endclass

module first;
  packet p1, p2, p3;
  bad_packet b;
  initial begin
    p1 = new(8'h10);
    p2 = p1;              // same object
    p2.addr = 8'h20;
    $display("p1.addr = %h (changed through p2)", p1.addr);
    p3 = new p1;          // shallow copy: a new object
    p3.addr = 8'h30;
    $display("p1.addr = %h, p3.addr = %h", p1.addr, p3.addr);
    b = new(8'h40);
    p1 = b;               // parent handle, child object
    $display("p1.kind() = %s", p1.kind());
    $finish;
  end
endmodule
```

2

Compile it

```
verilator --binary --timing first.sv
```

3

Run it

```
./obj_dir/Vfirst
```

Expected output:

```
p1.addr = 20 (changed through p2)
p1.addr = 20, p3.addr = 30
p1.kind() = bad_packet
- first.sv:26: Verilog $finish
```

4

What to notice

Line 1: `p2 = p1` made two names for one object. Line 2: `new p1` made a separate object. Line 3: a `packet` handle ran the child's `kind()` because it is `virtual`.

BEFORE YOU START · F

Study plan

18 hours at about 8 hours a week.



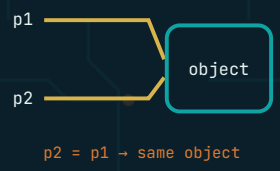
Session	Do this	Time
Day 1	Install Verilator 5 · first class · Module 1 (classes, handles); handles playground	4 h
Day 2	Module 2 (shallow vs deep copy, clone, compare)	3.5 h
Day 3	Module 3 (inheritance, virtual, $cast, access control); virtual toggle	4 h
Day 4	Module 4 (static, parameterised classes, generator pattern)	3 h
Day 5	Lab: find and fix both class bugs · area test	3.5 h

MODULE 1 OF 4

Classes, objects & handles

The three words that make or break class-based testbenches.

- Class anatomy and constructors
- Handles vs objects, null
- this and packages
- Garbage collection



⏱ 4 hours | Area 04 · SystemVerilog OOP

Start from zero

New to this? Work through these steps first, in order. Already comfortable? Skim them and jump to the reference below.

STEP 1

A class is a blueprint

In plain words

A **class** describes a kind of thing: what data it holds and what it can do. An **object** is one thing built from that description.

Think of it like

A house blueprint versus the houses built from it: one plan, many houses, each with its own furniture.

Worked example

```
class Packet;
  bit [7:0] id;
  bit [7:0] len;
  function void show(); $display("id %0d len %0d", id, len
endclass
```

The word for it

Class, **object**; the data are **properties**, the routines are **methods**.

Why it matters

Every transaction, driver, monitor and scoreboard in UVM is a class.

2 STEP 2 — Building an object with new()

≈ In plain words

Declaring a class variable doesn't build anything. Calling `new()` builds the object; a constructor sets its starting values.

☀ Think of it like

Owning the blueprint doesn't give you a house — you have to build one.

✎ Worked example

```
class Packet;
  bit [7:0] id, len;
  function new(int i = 0); id = i; len = 4; endfunction
endclass
Packet p;          // only a handle, no object yet
p = new(7);      // object built: id 7, len 4
```

Aa The word for it

The **constructor** `new()`.

! Why it matters

Forgetting `new()` is the number-one cause of “null object access” crashes.

3 STEP 3 — Handles point to objects

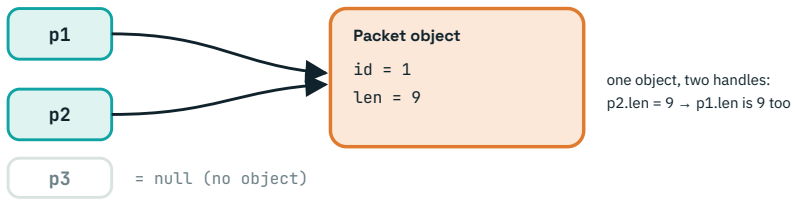
≈ In plain words

A class variable is a **handle**: it points to an object. Copying a handle copies the pointer, not the object.

☀ Think of it like

A TV remote: two remotes can control the same TV. A remote with no TV (null) does nothing — and pressing it crashes.

Worked example



p2 = p1 makes a second remote for the same TV.

1. `p1 = new(1);`
2. `p2 = p1;` — no new object.
3. `p2.len = 9;`
4. `p1.len` is now **9** as well.

The word for it — **Handle**, **null**.

Why it matters — Shared handles are the root of the most confusing scoreboard bugs — Module 2 shows the fix.

PAUSE & TRY

```
Packet a = new(1);
Packet b = a; b =
new(2); b.len = 5;
```

What `a.id` and `a.len` ? is Answer

`b = new(2)` points b at a **new** object, so a is untouched: **a.id = 1, a.len = 4**.

4 STEP 4 this, packages and cleaning up

In plain words — `this` means “my own” property when a name is ambiguous. Packages collect related classes so files can share them. Objects with no handle pointing at them are freed automatically.

Think of it like — Saying “my bag” in a room full of bags; a shared toolbox for the team; rubbish collected when nobody wants it.

Worked example

```
function new(int len); this.len = len; endfunction
package pkt_pkg; class Packet; … endclass endpackage
import pkt_pkg::*;
```

The word for it — **this**, **package**, **garbage collection**.

Why it matters — Lab 04 puts its classes in `pkt_pkg.sv`, the same way real testbenches are organised.

Going further: the reference

The same ideas at the next level: precise definitions, tables, rules and interview questions.

Key definitions

Class
A user-defined type that bundles data (properties) and behaviour (methods).

Object
One instance of a class, created with `new()`.

Handle
A variable that refers to an object (like a pointer you cannot do arithmetic on). `null` until assigned.

Property / method
A variable / function or task that belongs to the class.

Constructor
The special function `new()` that initialises an object.

this
Handle to the current object inside its own methods.

Garbage collection
Objects with no handles left are freed automatically.

package
A named scope that holds classes and types so every file shares them: `import pkg::*;`

class packet

```
// properties (data)
bit [7:0] addr;
byte payload[$];
bit [7:0] csum;

// methods (behaviour)
function new(...)
function bit [7:0] calc_csum()
```

A class is a blueprint.
Nothing exists until new() runs.

An object is one built instance.
It lives on the heap; many can exist.

A handle points at an object.
packet p; → null until p = new();

Blueprint, instance, reference. Keep the three words separate and most OOP bugs become obvious.

```
class packet;
  bit [7:0] addr;
  byte      payload[$];
  bit [7:0] csum;

  function new(bit [7:0] addr = 0);
    this.addr = addr;               // this. separates the property from the arg
  endfunction

  function bit [7:0] calc_csum();
    bit [7:0] s = addr;
    foreach (payload[i]) s ^= payload[i];
    return s;
  endfunction
endclass
```

Try it: handles and objects

```
p1 = null
```

```
p2 = null
```

```
packet p1, p2;       // two handles, both null
```

Code	What happens
`packet p;`	A handle, value `null` — no object yet
`p = new();`	An object is built; p refers to it
`q = p;`	Copies the handle: two names, one object
`p = null;`	p lets go; the object lives while q refers to it
`p.calc_csum()` with p null	Fatal null-handle error — the most common class crash

Common mistakes

Calling a method on a handle that was never constructed · thinking `q = p` copies data · comparing handles with `==` when you meant to compare contents · declaring classes inside each test file instead of one package.

INTERVIEW CORNER

Q. Handle vs object? **A.** The object is the data built by new(); the handle is a reference to it. Several handles can refer to one object.

Q. What happens to an object with no handles? **A.** It is garbage-collected automatically.

Key takeaways

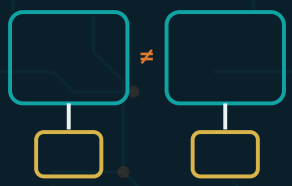
1. A class is a blueprint; `new()` builds an object; a handle refers to it.
2. Assigning handles copies the reference, not the data.
3. Put classes in a package and import it everywhere.

MODULE 2 OF 4

Shallow vs deep copy

Copying objects correctly — the root cause of many scoreboard bugs.

- What `new p` copies
- clone, copy, compare
- Which fields are shared
- Why scoreboards need deep copies



3.5 hours · Area 04 · SystemVerilog OOP

Start from zero

New to this? Work through these steps first, in order. Already comfortable? Skim them and jump to the reference below.

STEP 1

Copying the handle versus copying the object

In plain words

`p2 = p1` copies the handle (same object). `p2 = new p1` builds a new object with the same field values.

Think of it like

Sharing a link to a document versus making your own copy of the file.

Worked example

Code	Objects	Change p2.len — does p1 change?
`p2 = p1;`	1	yes
`p2 = new p1;`	2	no

The word for it

Handle assignment versus **object copy**.

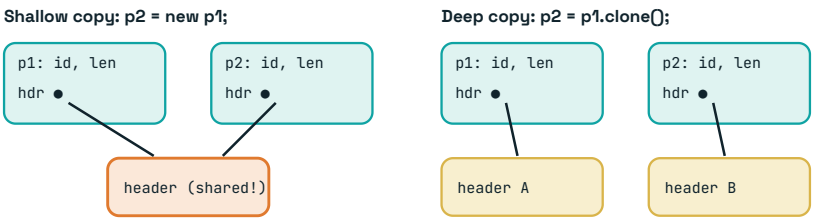
Why it matters — Whenever a component keeps a transaction for later, it needs its own copy.

2 STEP 2 — Shallow copy: nested objects are still shared

In plain words — `new p1` copies each field. If a field is itself a handle (to a header object, say), only that handle is copied — both packets still point to the same header.

Think of it like — Photocopying a folder that contains a shortcut: both folders now hold a shortcut to the same photo.

Worked example



Left: the header is shared. Right: each packet has its own header.

The word for it — **Shallow copy**.

Why it matters — Lab 04's planted bug is in `packet::copy()`: it shares the header instead of cloning it, so changing one packet's header silently changes another's.

3 STEP 3 — Deep copy: copy everything

In plain words — A deep copy also copies the nested objects. You write it yourself, usually as `copy()` or `clone()`.

Think of it like — Copying the folder **and** the photo the shortcut points to.

✎ **Worked example**

```
function Packet clone();
  clone = new this;          // shallow copy of the fields
  clone.hdr = new this.hdr;   // then copy the nested obje
endfunction
```

Aa **The word for it** — **Deep copy**, **clone**, **compare**.

! **Why it matters** — UVM gives every transaction `copy()`, `clone()` and `compare()`; you'll implement exactly this idea.

PAUSE & TRY **p2 = new p1; then p2.hdr.kind = 3. Does p1.hdr.kind change? And after p2 = p1.clone()?** Answer

With `new p1` the header is shared → **yes**, p1 changes. With a deep `clone()` → **no**.

4 STEP 4 — Why scoreboards need deep copies

≈ **In plain words** — A monitor often reuses one object, filling it with each new transaction. If the scoreboard keeps only the handle, the stored "expected" values change under it.

☀ **Think of it like** — Writing every visitor's name on the same sticky note and filing the note — at the end every file shows the last visitor.

✎ **Worked example**

1. Monitor fills `t` with packet 1 and sends the handle.
2. Scoreboard stores the handle in its queue.
3. Monitor overwrites `t` with packet 2.
4. Stored "packet 1" now reads as packet 2 → false pass or false fail.
5. Fix: store `t.clone()`.

Aa **The word for it** — **Object aliasing**.

! **Why it matters** This bug makes tests pass when the design is broken — the most dangerous kind.

≡ **Going further: the reference**

The same ideas at the next level: precise definitions, tables, rules and interview questions.

Key definitions

Shallow copy

`p2 = new p1`: a new object with every field copied; handle fields still point at the same nested objects.

Deep copy

A copy where nested objects are copied too, so nothing is shared.

clone()

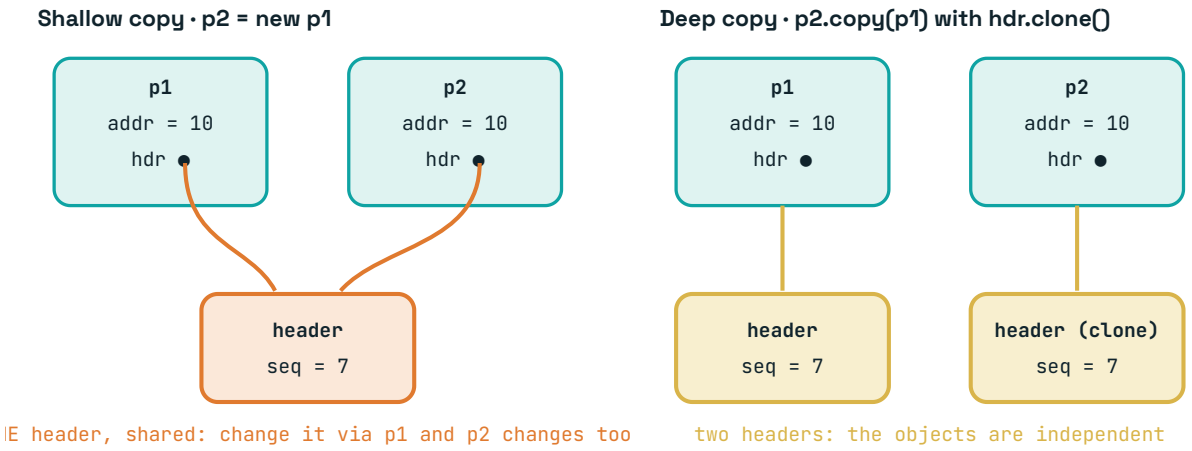
Convention: a method that returns a new object that is a copy of this one.

compare()

Convention: a method that compares field values, not handles.

convert2string()

Convention: returns a printable summary; UVM uses the same name.



IE header, shared: change it via p1 and p2 changes too

two headers: the objects are independent

The difference is what happens to nested objects. Queues and dynamic arrays are values and copy either way; class handles are where shallow and deep copies diverge.

```
class header;
  bit [7:0] kind, seq;
  function header clone();
    clone = new();  clone.kind = kind;  clone.seq = seq;
  endfunction
endclass

class packet;
  header hdr;  byte payload[$];
  function new(); hdr = new(); endfunction
  function void copy(packet rhs);        // deep
    payload = rhs.payload;                 // queue: copied by value
    hdr     = rhs.hdr.clone();             // object: needs its own copy
  endfunction
  function bit compare(packet rhs);
    return hdr.kind == rhs.hdr.kind && hdr.seq == rhs.hdr.seq && payload == rhs.
  endfunction
endclass
```

Field type	Shallow copy (`new p`)	Deep copy
bit, int, enum, string	value copied	value copied
queue, dynamic array	value copied	value copied
class handle	**handle copied — shared object**	object cloned
mailbox, event, semaphore handle	shared	usually shared on purpose

★ **Why scoreboards need deep copies**

The scoreboard keeps the expected transaction for a while. If it shares a nested object with something the generator keeps changing, the expected value changes underneath it — exactly the bug in this area's lab.

× **Common mistakes**

Copying a handle when you meant to copy the object · cloning only the top level of a nested structure · comparing with `==` on handles · forgetting to copy a field added later (keep copy and compare next to the field list).

INTERVIEW CORNER

Q. What does `new p1` copy? **A.** Every property; handle properties are copied as handles, so nested objects are shared.

Q. How does UVM handle copying? **A.** `copy()` calls the `do_copy()` hook you implement — a deep copy written field by field.

Key takeaways

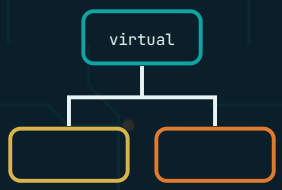
1. Shallow copies share nested objects; deep copies don't.
2. Write clone, copy and compare for every transaction class.
3. The scoreboard's copy must be deep.

MODULE 3 OF 4

Inheritance, virtual & $cast

Extend, override and let the object decide which method runs.

- extends and super
- virtual methods and polymorphism
- Up- and down-casting
- local, protected, const



⏱ 4 hours | Area 04 · SystemVerilog OOP

Start from zero

New to this? Work through these steps first, in order. Already comfortable? Skim them and jump to the reference below.

1 STEP 1 — extends: build on an existing class

≈ In plain words

A child class **extends** a parent: it gets all the parent's properties and methods, and can add or change some.

☀ Think of it like

A smartphone is a phone with extra features: it can still make calls.

✎ Worked example

```
class BadPacket extends Packet;
  bit corrupt = 1;    // added
endclass
```

Aa The word for it

Inheritance: **parent (base)** and **child (derived)** classes.

! Why it matters

Error-injection tests are child classes of normal transactions — no copy-pasting.

2 STEP 2
super: call the parent's version

In plain words

Inside a child, `super.` calls the parent's method, so you extend it instead of rewriting it.

Think of it like

Following the family recipe, then adding your own spice.

Worked example

```
function void show();
  super.show();                  // id and len
  $display("corrupt %0d", corrupt);  // plus the new field
endfunction
```

The word for it

super; a child's constructor must call `super.new()` when the parent's needs arguments.

Why it matters

Keeps base behaviour in one place, so a fix there reaches every child.

3 STEP 3
virtual methods: the object decides

In plain words

A parent handle can point to a child object. If the method is `virtual`, calling it runs the **child's** version. Without `virtual`, the handle's type decides.

Think of it like

A universal remote's “play” button: it does the right thing for whichever device it's pointed at.

Worked example

1. `Packet p = BadPacket_obj;` — parent handle, child object.
2. `p.show();`
3. `show` is `virtual` → BadPacket's show runs ✓
4. `show` not virtual → Packet's show runs, and the corrupt field is never printed ×

Aa **The word for it** — **Virtual method** and **polymorphism**.

! **Why it matters** — Lab 04 has a missing `virtual` on `calc_csum()`: a `bad_csum_packet` held in a `packet` handle still computes the good checksum, so the error test injects no error at all.

PAUSE & TRY **Packet's `calc()` is not virtual; `Packet p = new_bad;`, `calc()` runs for BadPacket which overrides it. With**

Without `virtual`, the handle type decides → **Packet's** calc().

4 STEP 4

$cast: checked down-casting

≈ **In plain words** — Going from a parent handle back to a child handle needs `$cast`, which checks that the object really is that child.

☀ **Think of it like** — Checking a parcel's label before signing for it.

✎ **Worked example**

```
BadPacket b;
if (!$cast(b, p)) $error("not a BadPacket");   // returns
```

Aa **The word for it** — **Upcast** (child → parent, always fine) and **downcast** (`$cast`).

! **Why it matters** — UVM hands you parent-type handles everywhere; `$cast` is how you get the real type back safely.

5 STEP 5

local and protected

In plain words — `local` members are visible only inside the class; `protected` also inside its children.

Think of it like — A diary (local), a family album (protected), a newspaper (public).

Worked example

Keyword	Visible in the class	In children	Outside
(none)	yes	yes	yes
`protected`	yes	yes	no
`local`	yes	no	no

The word for it — **Access control**.

Why it matters — Hiding internals stops tests from poking fields they shouldn't — fewer accidental dependencies.

Going further: the reference

The same ideas at the next level: precise definitions, tables, rules and interview questions.

Key definitions

Inheritance
`class child extends parent`: the child gets every property and method of the parent.

Override
A child method with the same name and signature as a parent method.

super
Refers to the parent's members: `super.new()`, `super.calc_csum()`.

virtual method
A method whose call is resolved by the object's real type, not the handle's type.

Polymorphism

One handle type, many behaviours: a `packet` handle can run a child's override.

Upcast / downcast

Child handle → parent handle (always legal) / parent → child (checked with `$cast`).

Abstract class

`virtual class`: cannot be constructed; children implement its `pure virtual` methods.

```
class bad_csum_packet extends packet;
  function new(); super.new(); endfunction
  virtual function bit [7:0] calc_csum();
    return super.calc_csum() ^ 8'h01;   // wrong by one bit on purpose
  endfunction
endclass
```



virtual decides who answers. Without `virtual` in the base class, the handle's declared type wins and the override is silently ignored.

Try it: virtual or not

```
class packet;
  virtual function bit [7:0] calc_csum(); return sum;        endfunction
endclass
class bad_csum_packet extends packet;
  virtual function bit [7:0] calc_csum(); return sum ^ 8'h01; endfunction
endclass

packet p = bad_csum_packet::new();   // handle type packet, object type ba
p.calc_csum();
```

```
→ runs bad_csum_packet::calc_csum — the object's real type decides.
   Error injection works: the DUT sees a wrong checksum.
```

Casting

```
packet          p;
bad_csum_packet b = new();
p = b;                                  // upcast: always legal
if (!$cast(b, p)) $error("p is not a bad_csum_packet");   // downcast: checked
```

Access control

Keyword	Visible to	Use for
(none)	Everyone	Most testbench fields
`protected`	The class and its children	Internals children may need
`local`	This class only	True internals
`const`	Everyone, set once	IDs, configuration fixed at construction

Common mistakes

Forgetting `virtual` on the base method · not calling `super.new()` when the parent constructor takes arguments · downcasting without checking `$cast`'s result · changing an override's argument list (it no longer overrides).

INTERVIEW CORNER

Q. Why must base-class methods be virtual in UVM? **A.** So tests can swap in child classes (factory overrides) and the framework calls their versions through base handles.

Q. When does `$cast` fail? **A.** When the object isn't actually the target type or one of its descendants.

Key takeaways

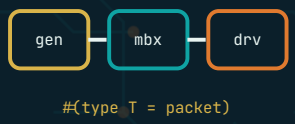
1. Children inherit everything and may override.
2. `virtual` makes the object's real type choose the method.
3. Upcast freely; downcast only through a checked `$cast`.

MODULE 4 OF 4

Static members, parameters & generators

The OOP patterns UVM is built on.

- static properties and methods
- Type-parameterised classes
- typedef class and extern
- The blueprint generator



#(type T = packet)

⏱ 3 hours | Area 04 · SystemVerilog OOP

Start from zero

New to this? Work through these steps first, in order. Already comfortable? Skim them and jump to the reference below.

1

STEP 1

static: one value shared by all objects

In plain words

A `static` property belongs to the class, not to each object, so every object sees the same one.

Think of it like

A ticket dispenser at a deli counter: everyone takes the next number from the same machine.

Worked example

```
class Packet;
  static int count = 0;
  int id;
  function new(); id = count++; endfunction   // 0, 1, 2 …
endclass
```

The word for it

Static property; a **static method** can use only static members.

! **Why it matters** Unique IDs on every transaction make logs and scoreboards traceable.

PAUSE & TRY **Three Packets are created with the class above. What are their ids, and what is `Packet::count` ?** Answer

ids **0, 1, 2**; count = **3**.

2 STEP 2 Classes with type parameters

≈ **In plain words** A class can take a **type** as a parameter, so one piece of code works for ints, packets, or anything else.

✳ **Think of it like** One storage box design that comes in “for shoes” and “for books” versions.

✎ **Worked example**

```
class Stack #(type T = int);
  T items[$];
  function void push(T x); items.push_back(x); endfunction
  function T pop(); return items.pop_back(); endfunction
endclass
Stack #(Packet) ps = new();
```

Aa **The word for it** **Parameterised class**.

! **Why it matters** UVM's FIFOs, ports and sequencers are all parameterised by your transaction type.

3 STEP 3 The blueprint generator

≈ **In plain words**

A generator keeps one **blueprint** object. Each time it needs a new transaction it clones the blueprint, randomises the copy and sends the copy.

☀ **Think of it like**

A cookie cutter: the same cutter, a fresh cookie each time.

✎ **Worked example**

1. `blueprint = new();` — or a child class, to change behaviour.
2. Loop: `t = blueprint.clone(); t.randomize(); mb.put(t);`
3. Swap in `blueprint = BadPacket` and every generated packet is a BadPacket — no generator code changes.

Aa **The word for it**

The **blueprint pattern** (the idea behind UVM's factory).

! **Why it matters**

It shows why both deep copies and virtual methods matter: the lab's generator relies on both.

4 STEP 4 typedef class and extern

≈ **In plain words**

`typedef class` lets two classes refer to each other before both are defined. `extern` lets you declare a method in the class and write its body later.

☀ **Think of it like**

A table of contents that lists a chapter before you reach it.

✎ **Worked example**

```
typedef class Driver;            // "Driver exists, detail
class Packet; extern function void show(); endclass
function void Packet::show(); … endfunction
```

Aa **The word for it**

Forward declaration, **extern method**.

! **Why it matters**

Keeps class declarations short and readable in big testbenches.

Going further: the reference

The same ideas at the next level: precise definitions, tables, rules and interview questions.

Key definitions

static property

One copy shared by every object of the class: `static int count;`

static method

Called on the class without an object: `packet::get_count()`.

Parameterised class

A class with type or value parameters: `class sb #(type T = packet);`

typedef class

Forward declaration so two classes can refer to each other.

extern method

Method declared in the class, body written outside with `packet::calc_csum`.

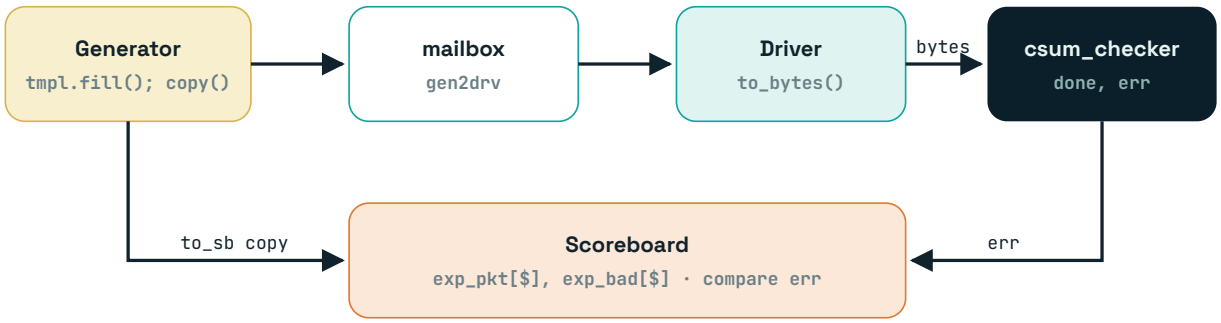
Blueprint (template) pattern

A generator copies a template object; swapping the template changes what is generated.

```
class packet;
  static int unsigned count;       // shared by all packets
  int unsigned id;
  function new(); id = count++; endfunction
endclass

class scoreboard #(type T = packet);   // parameterised by type
  T expected[$];
  function void add(T t); expected.push_back(t); endfunction
endclass
scoreboard #(packet) sb = new();
```

The lab's testbench, as a picture



Generator → mailbox → driver → DUT, with a deep copy to the scoreboard. The generator reuses one template object per kind; every hand-off must be a real copy.

Pre-UVM (this area)	UVM (Areas 08–12)
Transaction class	`uvm_sequence_item`
copy / compare / convert2string	`do_copy` / `do_compare` / `convert2string`
Generator + blueprint	Sequence + factory override
Mailbox	TLM FIFO / analysis port
Scoreboard queue	`uvm_scoreboard` with analysis imports

Common mistakes

Using a static counter for something that should be per-object · instantiating a parameterised class with the wrong type and casting around it · forgetting `typedef class` for mutually-referencing classes.

INTERVIEW CORNER

Q. What is the factory pattern in UVM, in OOP terms? **A.** A registry that constructs objects by type name, so a test can substitute a child class without editing the code that creates it.

Q. Why parameterise a class by type? **A.** One implementation (sequencer, FIFO, scoreboard) works for every transaction type.

Key takeaways

1. Static members are per class, not per object.
2. Type parameters make components reusable.
3. The blueprint pattern is the factory idea in miniature.

PRACTICE

Check yourself

Answer in your head, then open each card. The area test has 20 more.



- [x] **After `p2 = p1; p2.addr = 5;` what is `p1.addr` ?**

  5 — both handles refer to one object.

- [x] **What is a handle's value before `new()` ?**

  `null`.

- [x] **Which fields does `new p1` share with p1?**

  Class-handle fields; values, queues and arrays are copied.

A base method isn't virtual and a child overrides it. Which runs through a base handle?

The base version.

When does `$cast(child, base)` fail?

When the object isn't of the child type or a descendant.

What does `static int count` mean in a class?

One counter shared by every object of the class.

How do you change every generated packet without editing the generator?

Point its blueprint handle at a child-class object.

In Lab 04, what fixes bug 1?

Make `packet::copy` clone the header instead of copying the handle.

REFERENCE

Interview questions

Asked in DV and RTL interviews. Try an answer out loud before opening.



Explain shallow vs deep copy with an example.

A packet with a header object: `new p` shares the header; a deep copy clones it so changing one doesn't change the other.

What is polymorphism and why does UVM rely on it?

A base handle running a child's override via virtual methods; UVM's factory substitutes child classes and calls them through base handles.

Difference between `local` and `protected` ?

local: this class only; protected: this class and its children.

What is an abstract class?

A `virtual class` with `pure virtual` methods; it can't be constructed, only extended.

Why put classes in a package?

One shared definition: the same type everywhere, no duplicate class declarations.

Can you override a constructor?

A child defines its own `new()` and should call `super.new()` first.

What does a parameterised class give you?

Type-safe reuse: one implementation for many transaction types.

How would you debug a scoreboard reporting mismatches on every packet with the same sequence number?

Suspect a shared object: the expected copies all point at one header that the generator keeps updating.

REFERENCE

Glossary

Every term used on this page, A to Z.

Aa

Abstract class
virtual class that can't be constructed.

Class
Blueprint of data and methods.

clone()
Returns a new copy of the object.

compare()
Field-wise equality check.

Constructor
new(): initialises an object.

Deep copy
Copy including nested objects.

Downcast
Parent handle to child handle via $cast.

extends
Declares a child class.

Factory
Creates objects by type so tests can substitute classes.

Handle
Reference to an object.

local / protected
Visibility: this class / this class and children.

null
A handle that refers to nothing.

Object
An instance of a class.

Override
Child method replacing a parent method.

Package
Shared scope for classes and types.

Parameterised class
Class with type or value parameters.

Polymorphism
Base handle running a child's method.

pure virtual
Method a child must implement.

Shallow copy
Copy that shares nested objects.

static
Shared by all objects of a class.

super
The parent class's members.

this
The current object.

Upcast
Child handle to parent handle.

virtual method
Resolved by the object's real type.

REFERENCE

FAQ & troubleshooting

The problems learners hit most in this area, and the fix for each.



You see	Why	Fix
Fatal: null pointer / null handle	A method called before `new()`	Construct every handle; check with `if (p == null)`
Override never runs	Base method not virtual, or the signature differs	Add `virtual` in the base; match argument types exactly
Every scoreboard entry shows the same values	Shallow copy shares a nested object	Deep copy with `clone()`
`$cast` error at run time	Object is not the target type	Check the factory/blueprint; use `if (!$cast(...))`
Duplicate class declaration errors	Class defined in several files	Define it once in a package; import it

REFERENCE

Further reading

Go deeper when you are ready.



Resource	Why read it
Chris Spear & Greg Tumbush, *SystemVerilog for Verification* — OOP chapters	Classes, copying and polymorphism for testbenches
IEEE 1800 — Classes chapter	Exact rules for handles, virtual methods and casting
Accellera UVM 1.2 / IEEE 1800.2 User's Guide	Where these OOP patterns lead next
Verilator manual — class support	What the free simulator supports

HANDS-ON LAB · ABOUT 3.5 HOURS

Lab 04 · Class-based packet generator and checksum checker

The DUT, `csum_checker.v`, is correct. Both bugs are in `pkt_pkg.sv`. `tb_csum.sv` has a generator, a driver and a scoreboard.

1. Run `make`: the scoreboard reports mismatches even on good packets — and every one shows the same sequence number.
2. Bug 1: find which field of `packet::copy` still points at the template's object; fix it with `header::clone`.
3. Re-run: good packets pass, but bad packets are no longer flagged.
4. Bug 2: the generator calls `calc_csum` through a `packet` handle — what must the base method be for the override to run? Fix and re-run.

Done when you see `PASS: 200 packets, 40 bad checksums flagged`.