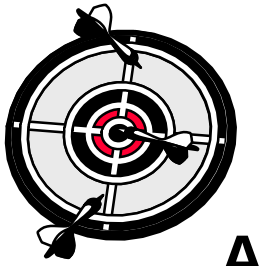
Agenda

DAY 3

8	Object Oriented Programming (OOP) – Inheritance
9	Inter-Thread Communications 
10	Functional Coverage 
11	SystemVerilog UVM Preview
CS	Customer Support

Unit Objectives



After completing this unit, you should be able to:

- **Create OOP extended classes**
- **Access class members in inheritance hierarchy**

Day 2 Review - Creating Concurrent Threads

- Concurrent threads are created in a `fork-join` block:

```
int a, b, c;
fork
  statement0;
  begin
    statement1;
    statement2;
  end
join | join_any | join_none
statement3;
```

- Statements enclosed in `begin-end` in a `fork-join` block are executed sequentially as a single concurrent child thread
- No pre-determined execution order for concurrent threads
- All child threads share the parent variables

Day 2 Review - OOP Class

- **Similar to a module, an OOP *class* encapsulates:**
  - Variables (properties) used to model a system
  - Subroutines (methods) to manipulate the data
  - Properties & methods are called members of class

Class properties and methods are visible inside the class

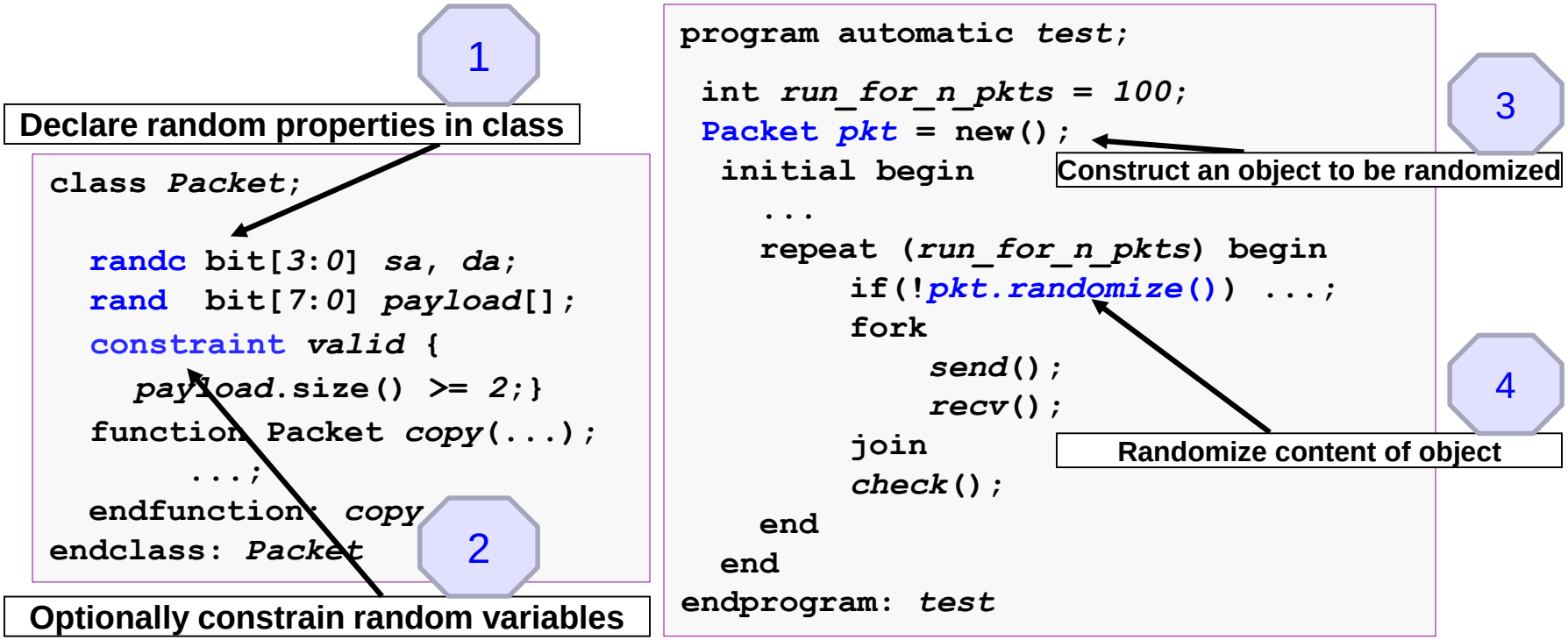
```
class Packet;
  string    name;
  bit[3:0]  sa, da;      //copy of Packet properties
  bit[7:0]  payload[];   //copy of Packet properties

  task send();
    send_addrs();
    send_pad();
    send_payload();
  endtask: send

  task send_addrs();     ...  endtask
  task send_pad();       ...  endtask
  task send_payload();   ...  endtask
endclass: Packet
```

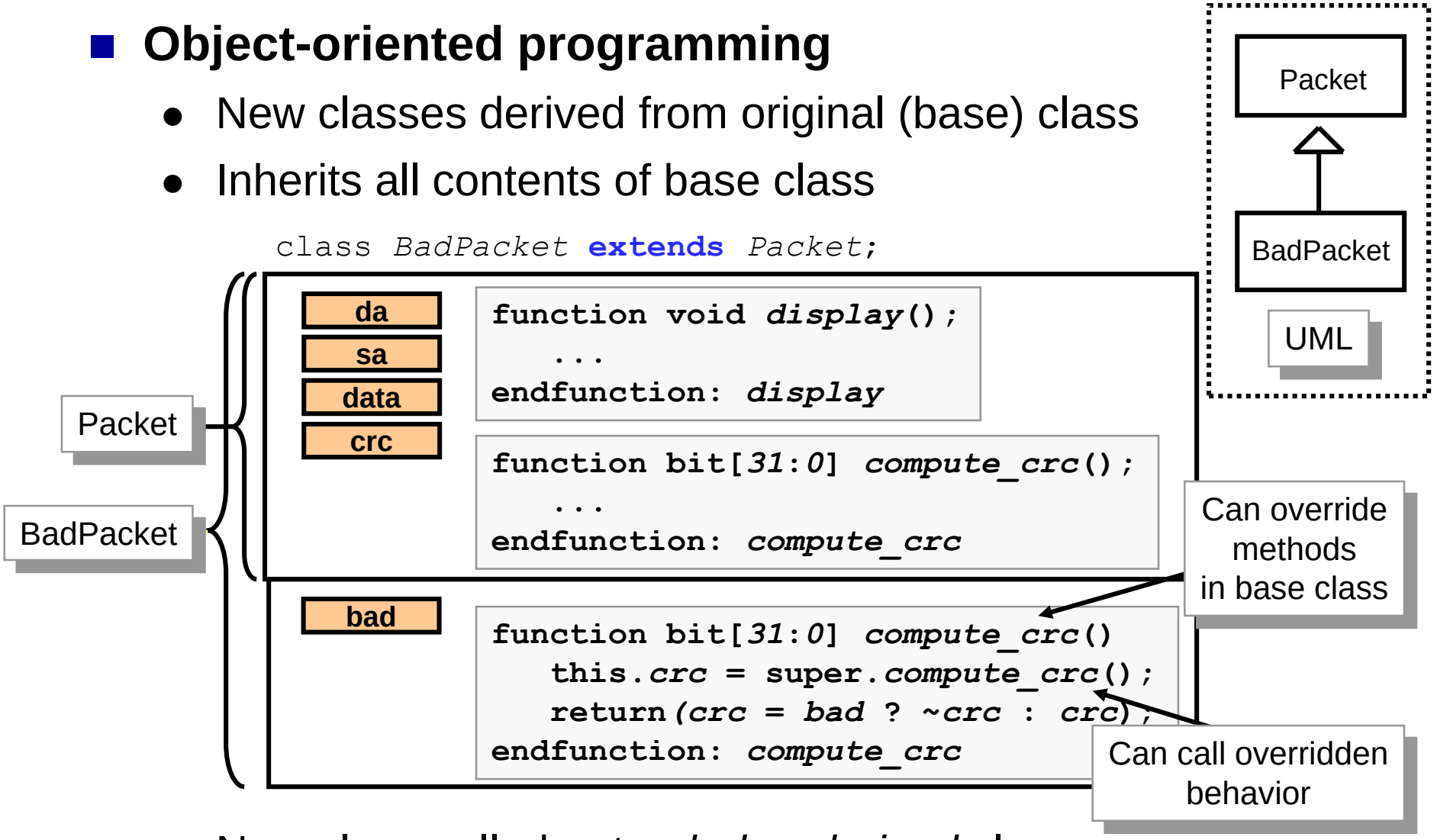
Day 2 Review - OOP Based Randomization

- **Randomization is achieved via classes**
  - **`randomize()` function built into class**
  - Randomizes each **`rand`** and **`randc`** property value to full range of its data type unless constrained with a **`constraint`** block.



Object Oriented Programming: Inheritance

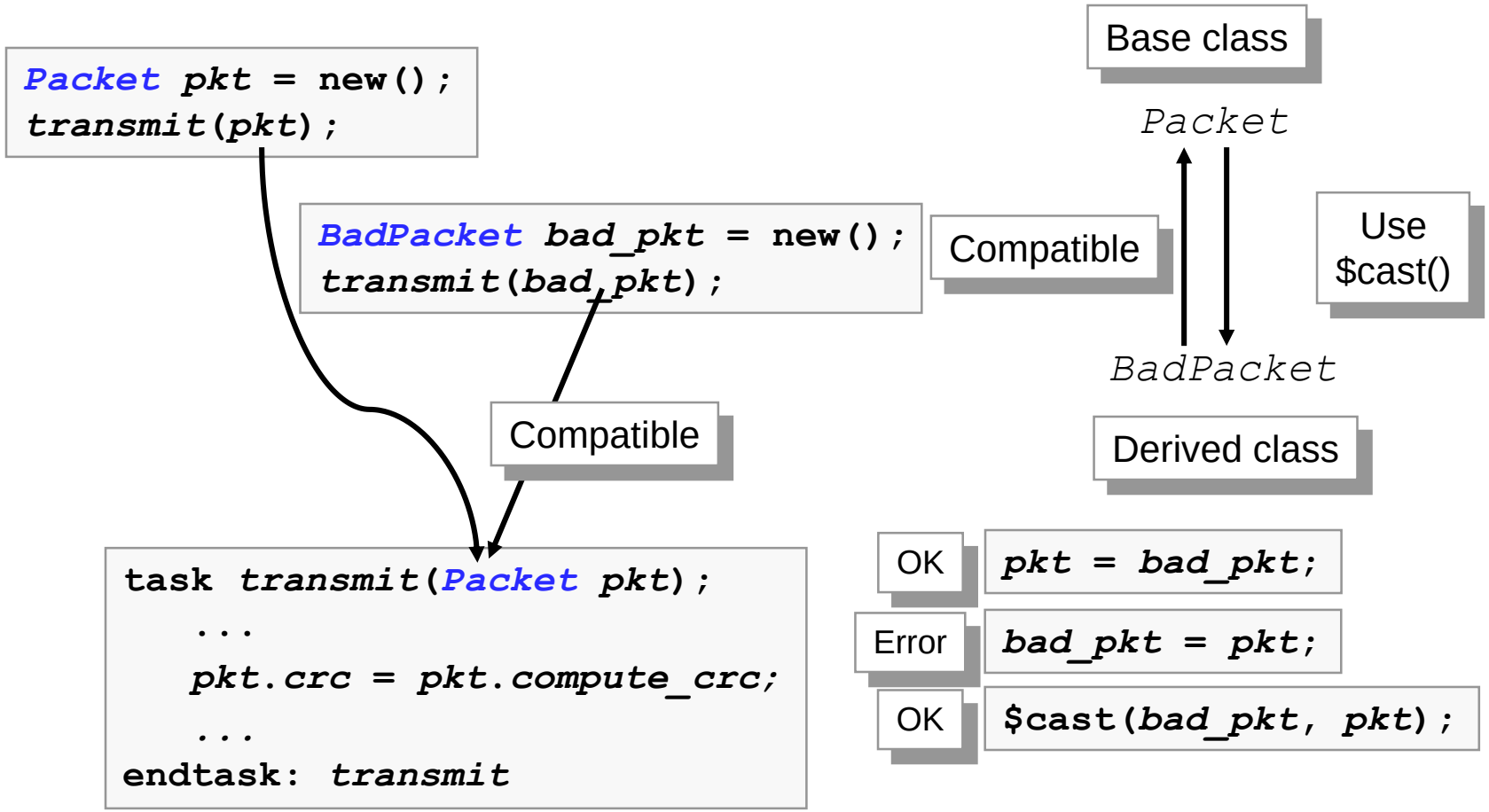
- **Object-oriented programming**
  - New classes derived from original (base) class
  - Inherits all contents of base class



  - New class called *extended* or *derived* class

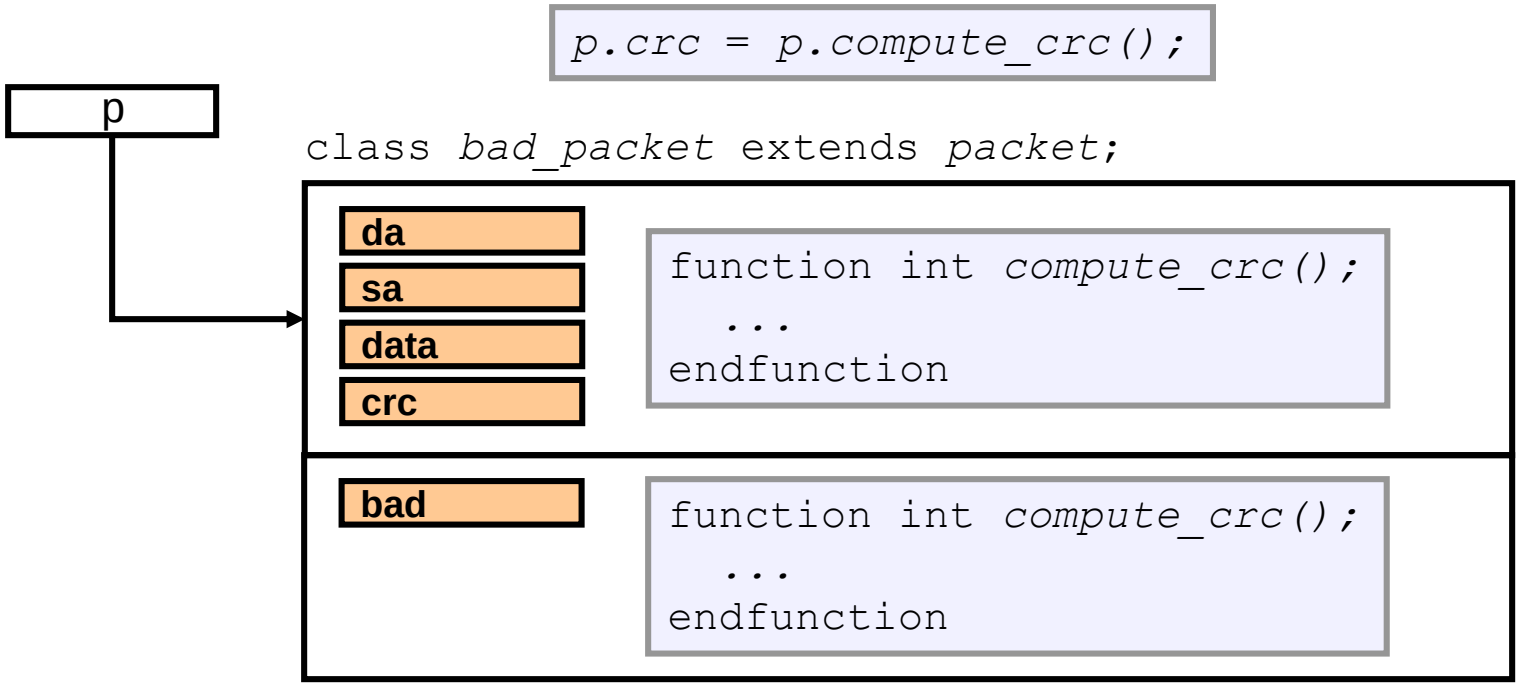
Object Oriented Programming: Inheritance

- **Derived classes compatible with base class**
  - Can reuse code



OOP: Polymorphism

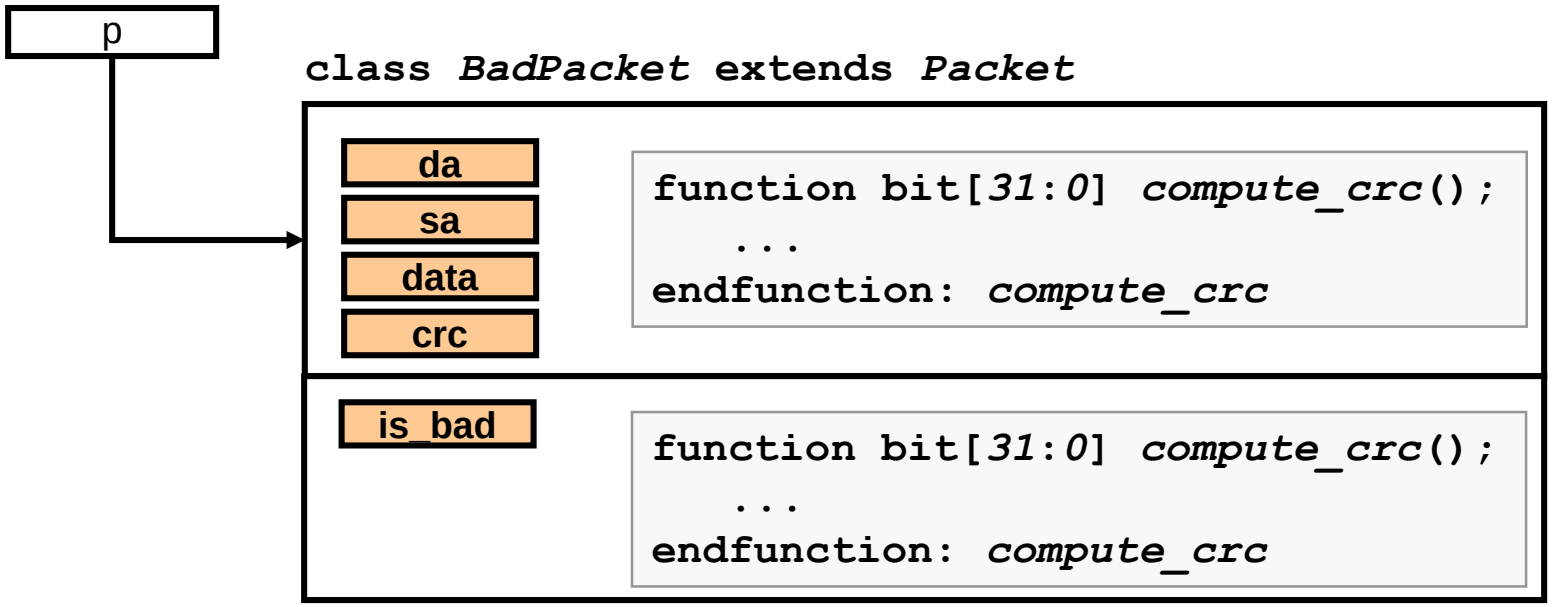
- **Which method gets called?**



- **Depends on**
  - Type of handle *p* (e.g. "***packet***" or "***bad_packet***" ?)
  - Whether ***compute_crc()*** is **virtual** or not

OOP: Polymorphism

- If *compute_crc()* **is not** virtual

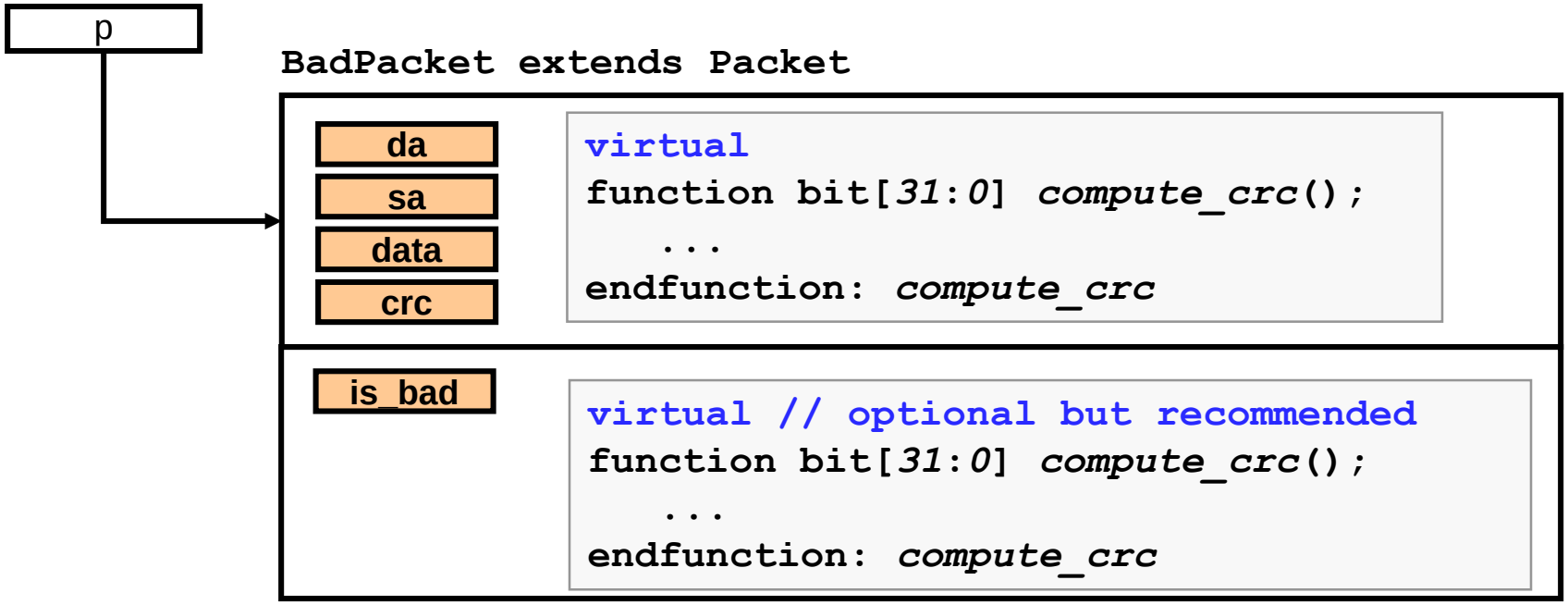



```
Packet p = new();
BadPacket bp = new();
p.crc = p.compute_crc();
bp.crc = bp.compute_crc();
transmit(p);
transmit(bp);
```

```
task transmit(Packet pkt);
  ...
  pkt.crc = pkt.compute_crc();
  ...
endtask: transmit
```

OOP: Polymorphism

- If *compute_crc()* is virtual




```
Packet p = new();
BadPacket bp = new();
p.crc = p.compute_crc();
bp.crc = bp.compute_crc();
transmit(p);
transmit(bp);
```

```
task transmit(Packet pkt);
  ...
  pkt.crc = pkt.compute_crc();
  ...
endtask: transmit
```

Modifying Constraints for Testcases

- **Define test specific constraints in derived classes**
  - Can also override existing constraints

```
class data;
  rand bit[31:0] x, y;
  constraint valid {
    x > 0;
    y >= 0;
  }
endclass: data
```

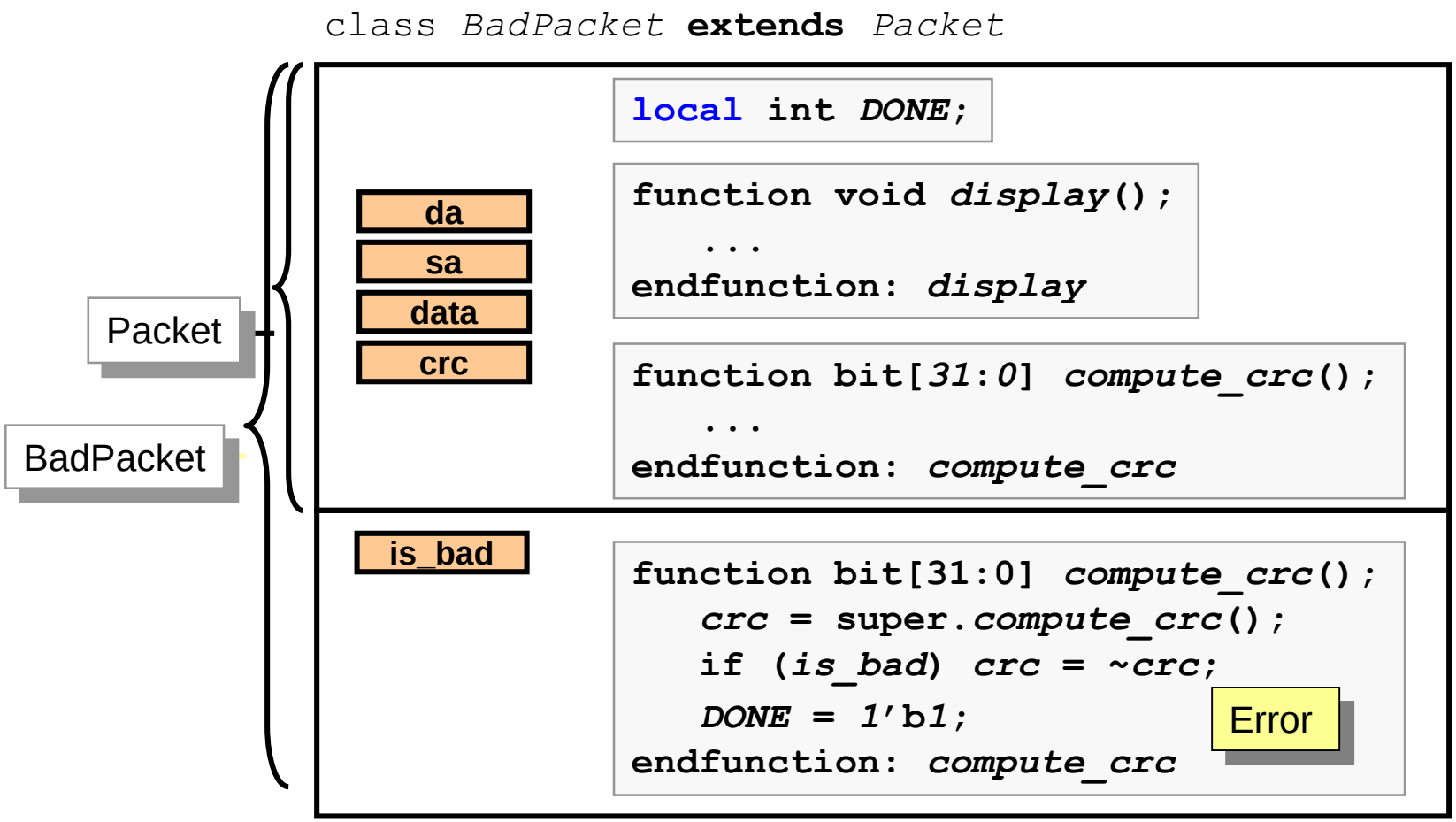
```
class Generator;
 data blueprint;
 …
 while(…)
  …
  blueprint.randomize();
  …
endclass: Generator
```

```
program automatic test_corner_case;
  class test_data extends data;
    constraint corner_case {
      x == 5; y == 10;
    }
  endclass: test_data

  initial begin
    test_data tdata = new();
    Generator gen = new();
    gen.blueprint = tdata; //polymorphism
    …
  end
endprogram: test_corner_case
```

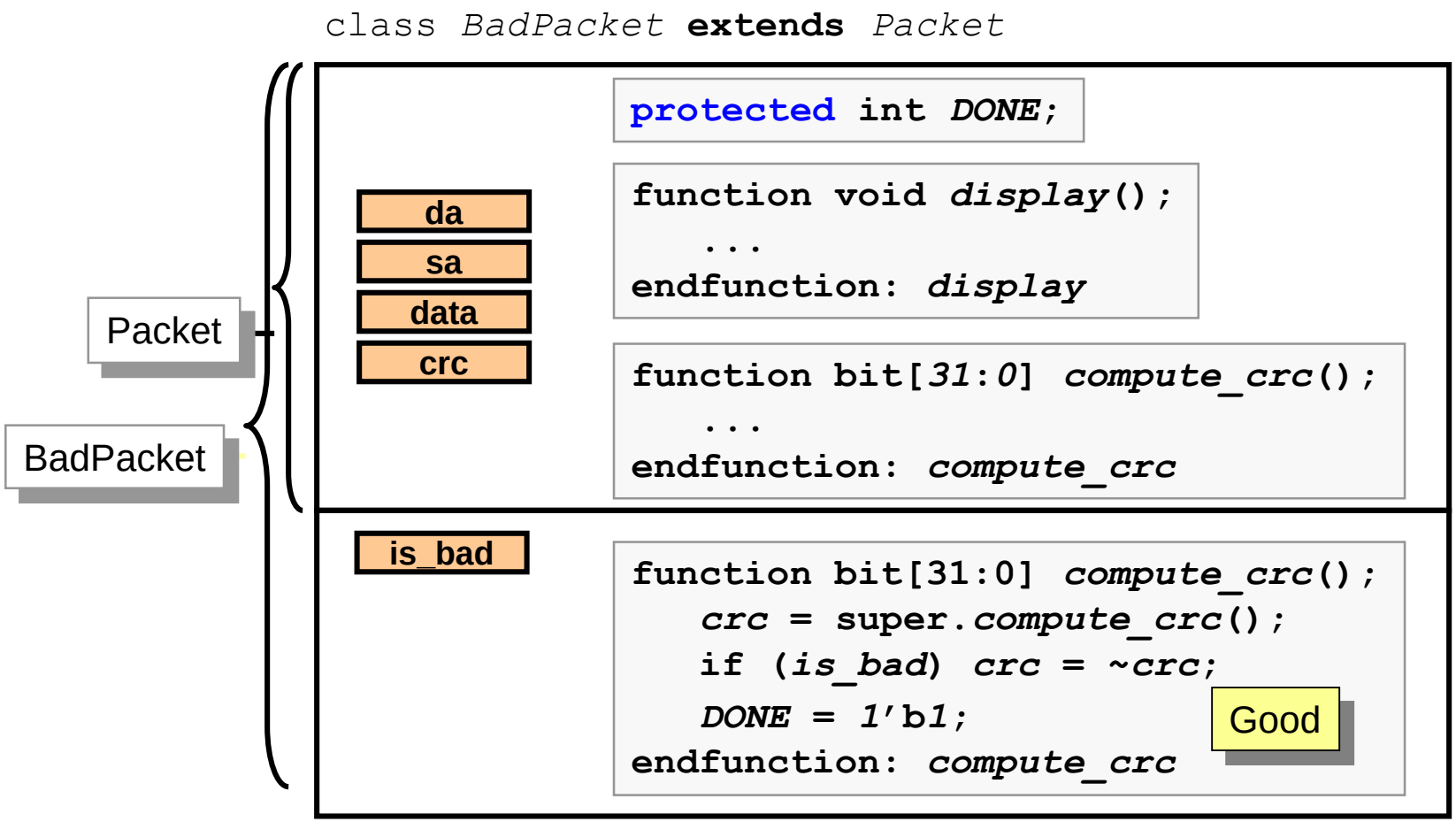
Data Protection: Local

- **Local members of a base class are not accessible in the derived class**



Data Protection: Protected

- **Protected members of a base class are accessible in the derived class, but not to external code**



Constructing Derived Class Objects

- **When constructing an object of a derived class**
  - If the derived class does not have a constructor defined VCS inserts one:

```
function new();
    super.new();
endfunction
```

  - If the derived class defines a constructor, SystemVerilog expects its first procedural statement to be

    `super.new([`*`args`*`])`

    - A syntax error results if called anywhere except as the first statement
    - User provides the correct argument set
    - If missing, the compiler inserts a call without arguments, as the first procedural statement of the function

    `super.new()`

Test For Understanding 1

- Will the following code compile?
- Which one of the task `new()` is executed?

```
class A;
  protected int a;
  function int get_a();
    get_a = a;
  endfunction: get_a
  function new(int b);
    a = b;
  endfunction
endclass: A
```

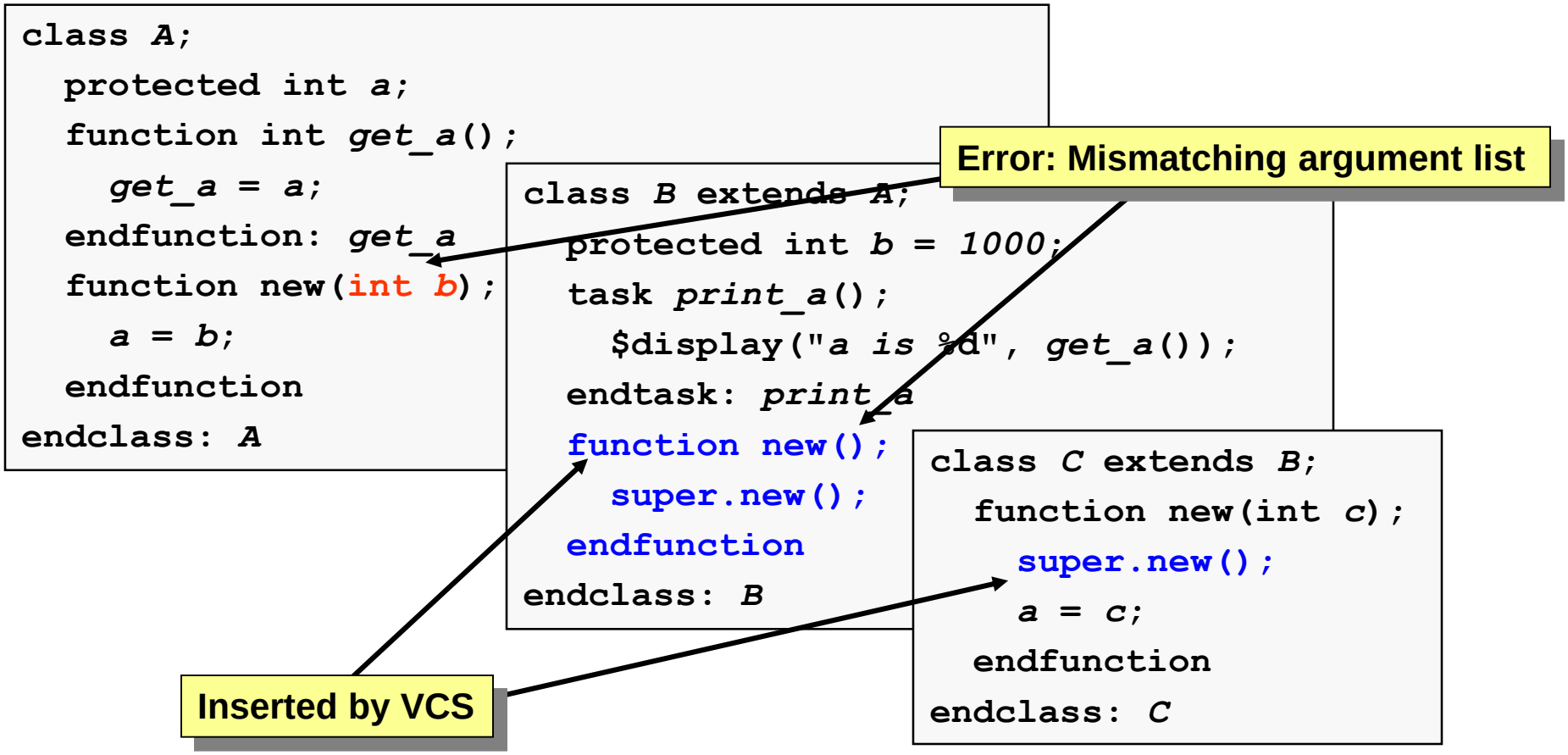
```
class B extends A;
  protected int b = 1000;
  task print_a();
      $display("a is %d", get_a());
  endtask: print_a
endclass: B
```

```
program automatic test;
  C test_c = new(10);
  test_c.print_a();
endprogram: test
```

```
class C extends B;
  function new(int c);
    a = c;
  endfunction
endclass: C
```

Test For Understanding 1: Answers

- VCS will attempt to execute every task `new()` starting with `new()` of `class C,` resulting in syntax error:



Test For Understanding 1: Guideline

- Always call `super.new()` as the first procedural statement in constructor, with correct argument set

```
class A;
  protected int a;
  function int get_a();
    get_a = a;
  endfunction: get_a
  function new(int b);
    a = b;
  endfunction
endclass: A
```

```
class B extends A;
  protected int b = 1000;
  task print_a();
    $display("a is %d", get_a());
  endtask: print_a
  function new(int b);
    super.new(b);
  endfunction
endclass: B
```

```
class C extends B;
  function new(int c);
    super.new(c);
    a = c;
  endfunction
endclass: C
```

Inserted by user

Test For Understanding 2

- Given the following class inheritance hierarchy, is the program code legal?

```
class A;
  protected int a;
  function int get_a();
    get_a = a;
  endfunction: get_a
  function new(int b);
    a = b;
  endfunction
endclass: A
```

```
class B extends A;
  protected int b = 1000;
  task print_a();
    $display("a is %d", get_a());
  endtask: print_a
  function new(int b);
    super.new(b);
  endfunction
endclass: B
```

```
class C extends A;
  function new(int c);
    super.new(c);
    a = c;
  endfunction
endclass: C
```

```
program automatic test;
  C obj_c = new(10);
  B obj_b = obj_c;
endprogram: test
```

Test For Understanding 2: Answer

- Both classes, B and C extend from base class A, but they are unrelated. A handle of one object can not point to its sibling object.

```
class A;
  protected int a;
  function int get_a();
    get_a = a;
  endfunction: get_a
  function new(int b);
    a = b;
  endfunction
endclass: A
```

```
class B extends A;
  protected int b = 1000;
  task print_a();
    $display("a is %d", get_a());
  endtask: print_a
  function new(int b);
    super.new(b);
  endfunction
endclass: B
```

```
class C extends A;
  function new(int c);
    super.new(c);
    a = c;
  endfunction
endclass: C
```

B and C are both derived from A

```
program automatic test;
  C obj_c = new(10);
  B obj_b = obj_c; // not legal
endprogram: test
```

Quiz Time

Inheritance: Quiz 1

```
program automatic test1;

class abc;
    rand int a;
endclass

class xyz extends abc;
    rand int b;
endclass

initial begin

  abc o1 = new(); xyz o2 = new();
  o1 = o2;
  $display("test: o1 = %d", o1.b );

end

endprogram: test1
```

1. Will this code compile without errors?
   - If not, why not?
2. Will it throw any runtime errors?
3. What will the program display?

Inheritance: Quiz 2

```
program automatic test1;
class abc;
    rand int a = 10;
    function void prnt_a();
      $display("abc: a= ", a);
    endfunction
endclass
class xyz extends abc;
     function void prnt_a();
      $display("xyz: a= ", a);
    endfunction
endclass
initial begin
  abc o1 = new(); xyz o2 = new();
  o1 = o2;
  o1.prnt_a();
end
endprogram: test1
```

1. Will this code compile without errors?
   - If not, why not?
2. Will it throw any runtime errors?
3. What will the program display?

Inheritance: Quiz 3

```
program automatic test1;
class abc;
    rand int a = 10;
    virtual function void prnt_a();
      $display("abc: a= ", a);
    endfunction
endclass
class xyz extends abc;
    virtual function void prnt_a();
      $display("xyz: a= ", a);
    endfunction
endclass
initial begin
  abc o1 = new(); xyz o2 = new();
  o1 = o2;
  o1.prnt_a();
end
endprogram: test1
```

1. Will this code compile without errors?
2. Will it throw any runtime errors?
3. What will the program display?
4. Why did display change from Quiz 2?

Inheritance: Quiz 4

```
program automatic test1;
class abc;
    rand int a = 10;
    virtual function void prnt_a();
      $display("abc: a= ", a);
    endfunction
endclass
class xyz extends abc;
    virtual function void prnt_a();
      $display("xyz: a= ", a);
    endfunction
endclass
initial begin
  abc o1 = new(); xyz o2 = new();
  o2 = o1;
  o2.prnt_a();
end
endprogram: test1
```

1. Will this code compile without errors?
   - If not, why not?
2. Will it throw any runtime errors?
3. What will the program display?

Inheritance: Quiz 5

```
program automatic test1;
class abc;
    rand int a = 10;
    virtual function void prnt_a();
      $display("abc: a= ", a);
    endfunction
endclass
class xyz extends abc;
    virtual function void prnt_a();
      $display("xyz: a= ", a);
    endfunction
endclass
initial begin
  abc o1 = new(); xyz o2 = new();
  $cast(o2 , o1);
  o2.prnt_a();
end
endprogram: test1
```

1. Will this code compile without errors?
   - If not, why not?
2. Will it throw any runtime errors?
   - If yes, why?
3. What will the program display?

Unit Objectives Review

Having completed this unit, you should be able to:

- **Create OOP extended classes**
- **Access class members in inheritance hierarchy**

Appendix

Interface Class

Interface Class

Interface Class

- **Interface class**
  - Defined using keyword **`interface`**
    - Contains only pure virtual methods, type declarations and parameter declarations
    - Can not contain properties, cover groups or constraint blocks
  - Is implemented by non interface class using **`implements`** keyword
  - Can inherit from multiple interface classes (multiple inheritance) through **`extends`** keyword

```
interface class intf_cls;
  pure virtual task print();
endclass
```

```
class myclass implements intf_cls;
  virtual task print();
    $display("myclass");
  endfunction
endclass
```

myclass must implement the methods prototyped in the interface classes it implements

Interface Class - Rules

- **`implements` v/s `extends`**
  - **`extends`** - used to add to or modify the behavior of a base class
  - **`implements`** - a requirement to provide implementations for the pure virtual methods defined in an interface class
- **Interface class**
  - may extend zero or multiple interface classes
  - can not implement interface class
  - can not extend a non-interface class
- **Non interface class**
  - can not extend interface class
  - can implement zero or more interface classes
  - can extend only one other class
  - can simultaneously implement interface class(es) and extend a class

Interface Class - Multiple Inheritance

- By implementing multiple interface classes one can achieve multiple inheritance

```
interface class PutImp#(type PUT_T = logic);
    pure virtual function void put(PUT_T a);
endclass
interface class GetImp#(type GET_T = logic);
    pure virtual function GET_T get();
endclass

class Fifo#(type T = logic, int DEPTH = 1) implements
                                    PutImp#(T), GetImp#(T);
    T myFifo [$:DEPTH-1];
    virtual function void put(T a);
        myFifo.push_back(a);
    endfunction
    virtual function T get();
        get = myFifo.pop_front();
    endfunction
endclass
```

Fifo "inherits" from both PutImp and GetImp

Interface Class – Partial Implementation

- **Partial implementation of interface class**
  - Use virtual class to create partially defined class
    - Virtual class can implement some interface methods
    - Virtual class must define prototype for methods not implemented by it

```
interface class IntfClass;
    pure virtual function bit funcA();
    pure virtual function bit funcB();
endclass
```

```
virtual class ClsA implements IntfClass;
    virtual function bit funcA();
        return (1);
    endfunction
    pure virtual function bit funcB();
endclass
```

ClsA only implements funcA
ClsA must be virtual
funcB prototype required

```
class ClsB extends ClsA;
  virtual function bit funcB();
    return (0);
  endfunction
endclass
```

ClsB is complete implementation
funcA inherited from ClsA
funcB implemented

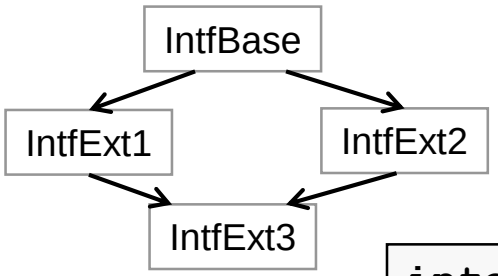
Interface Class – Multiple Extends

- **Multiple extends and Diamond Relationship**
  - Only interface classes can extend from multiple interface classes
  - A *diamond relationship* occurs if an interface class is implemented by the same class or inherited by the same interface class in multiple ways
  - Only one copy of the symbols from any single interface class will be merged, to avoid a name conflict

```
interface class IntfBase;
parameter SIZE = 64;
endclass
```

```
interface class IntfExt1 extends IntfBase;
    pure virtual function bit funcExt1();
endclass

interface class IntfExt2 extends IntfBase;
    pure virtual function bit funcExt2();
endclass
```




```
interface class IntfExt3 extends IntfExt1,IntfExt2;
endclass
```