Table of Contents

1. **Class**
2. **Handle**
3. **Object**
4. **Property**
5. **Method**
6. **Prototype**
7. **Static, Local, and Global Variables**
8. **Shallow Copy and Deep Copy**
9. **Static Methods**
10. **Inheritance**
11. **Super Keyword**
12. **This Keyword**
13. **Polymorphism**
14. **Casting**
15. **Abstract Class**
16. **Extern Keyword**

Object-Oriented Programming in SystemVerilog

In SystemVerilog, **Object-Oriented Programming (OOP)** concepts are supported, allowing you to create reusable and modular code. Below is an explanation of the key OOP terminology in SystemVerilog.

1. Class

A **class** is a blueprint or template for creating objects. It defines the properties (data members) and methods (functions/tasks) that the objects of the class will have.

2. Handle

A **handle** is a reference to an object of a class. It is similar to a pointer in other programming languages. You use a handle to access the properties and methods of an object.

3. Object

An **object** is an instance of a class. It is created dynamically using the new() method. Each object has its own set of properties and can call the methods defined in the class.

4. Property

A **property** is a data member of a class. It holds the state or attributes of an object. Properties can be of any data type, such as int, string, or even another class.

VIKRAM RENESAS

5. Method

A **method** is a function or task defined within a class. It defines the behavior or actions that an object can perform.

6. Prototype

A **prototype** is a declaration of a method (function or task) within a class. It specifies the method's name, return type (if any), and arguments (if any). The actual implementation of the method is defined later.

Static, Local, and Global Variables

In SystemVerilog, **static**, **local**, and **global** variables have specific scoping and lifetime rules.

1. Static Variable

- A static variable retains its value between function/task calls.
- It is initialized only once and persists throughout the simulation.
- Declared using the static keyword.

2. Local Variable

- A local variable is declared inside a function or task.
- It is created when the function/task is called and destroyed when the function/task exits.
- Declared without any special keyword.

3. Global Variable

- A global variable is declared outside any function, task, or class.
- It is accessible throughout the entire module or program.
- Declared at the module or program level.

Shallow Copy and Deep Copy

In SystemVerilog, **shallow copy** and **deep copy** are two ways to copy objects. The key difference lies in how they handle **nested objects** or **dynamic data** within the object being copied.

Shallow Copy

- A **shallow copy** creates a new object and copies all the fields of the original object.
- If the object contains references to other objects, the references are copied, but the referenced objects themselves are **not duplicated**.
- Both the original and the copied object will **share the same nested objects**.

VIKRAM RENESAS

Deep Copy

- A **deep copy** creates a new object and recursively copies all the fields of the original object, including any nested objects.
- If the object contains references to other objects, new instances of those objects are created, and their contents are copied.
- The original and the copied object will have **independent copies of nested objects**.

Static Methods

In SystemVerilog, a **static method** is a method that belongs to a class itself, rather than to an instance of the class. This means that the method can be called without creating an object of the class.

Characteristics of Static Methods:

1. **No Access to Instance Variables**: Static methods do not have access to instance-specific variables (non-static fields) or this. They can only access static fields and other static methods.
2. **Called Using Class Name**: Static methods are typically called using the class name, not an instance of the class.
3. **Used for Utility Functions**: Static methods are ideal for utility functions or operations that don't need object state, such as mathematical calculations or managing static data shared among all instances of the class.

Inheritance

In SystemVerilog, **inheritance** is a key feature of Object-Oriented Programming (OOP) that allows you to create a new class (called the **derived class** or **subclass**) based on an existing class (called the **base class** or **superclass**). The derived class inherits all the properties and methods of the base class and can also add new properties and methods or override existing ones.

Key Concepts of Inheritance:

1. **Base Class (Superclass)**: The original class from which properties and methods are inherited.
2. **Derived Class (Subclass)**: The new class that inherits from the base class.
3. **extends Keyword**: Used to specify that a class inherits from another class.
4. **super Keyword**: Used to refer to the base class from within the derived class.

Polymorphism

Polymorphism is one of the core concepts of Object-Oriented Programming (OOP), and it allows for the ability to use a single interface to represent different types of objects. In SystemVerilog, polymorphism is primarily used with **classes**, and it enables you to write more flexible and reusable code by allowing objects of different classes to be treated through a common interface.

VIKRAM RENESAS

Key Concepts of Polymorphism:

1. **Base Class and Derived Classes**: A **base class** defines a common interface (methods and properties). **Derived classes** inherit from the base class and can override its methods to provide specific implementations.
2. **Virtual Methods**: Methods in the base class must be declared as virtual to allow overriding in derived classes.
3. **Dynamic Method Dispatch**: The actual method to be called is determined at runtime based on the type of the object, not the type of the handle.

Casting

Dynamic Casting

- Dynamic casting is used to safely cast a super-class pointer (reference) into a subclass pointer (reference) in a class hierarchy.
- Dynamic casting will be checked during runtime, and an attempt to cast an object to an incompatible object will result in a runtime error.
- Dynamic casting is done using the $cast(destination, source) method.

Abstract Class

SystemVerilog prohibits a class declared as virtual to be directly instantiated and is called an **abstract class**.

Pure Virtual Method

- A virtual method inside an abstract class can be declared with the keyword pure and is called a **pure virtual method**.
- Such methods only require a prototype to be specified within the abstract class, and the implementation is left to be defined within the sub-classes.

Extern Keyword

The **extern** keyword is used to declare methods that are defined outside the class body. This allows for separation of method declaration and implementation.

This document provides a comprehensive overview of Object-Oriented Programming concepts in SystemVerilog, including classes, handles, objects, methods, inheritance, polymorphism, and more.

VIKRAM RENESAS

Object-Oriented Programming (OOP)

Concepts are supported, allowing you to create reusable and modular code. Below is an explanation of the key OOP terminology in SystemVerilog

1. Class

A **class** is a blueprint or template for creating objects. It defines the properties (data members) and methods (functions/tasks) that the objects of the class will have.

```
1 class Animal;
2   // Properties (data members)
3   string name;
4   int age;
5
6   // Method (function)
7   function void display();
8     $display("Name: %s, Age: %0d", name, age);
9   endfunction
10 endclass
```

2. Handle

A **handle** is a reference to an object of a class. It is similar to a pointer in other programming languages. You use a handle to access the properties and methods of an object.

```
1 Animal my_animal; // 'my_animal' is a handle to an object of type 'Animal'
2 my_animal = new(); // Create a new object and assign it to the handle
```

3. Object

An **object** is an instance of a class. It is created dynamically using the new() method. Each object has its own set of properties and can call the methods defined in the class.

```
1 Animal my_animal; // Declare a handle
2 my_animal = new(); // Create an object and assign it to the handle
3 my_animal.name = "Dog"; // Set the 'name' property
4 my_animal.age = 5;      // Set the 'age' property
5 my_animal.display();    // Call the 'display' method
```

4. Property

A **property** is a data member of a class. It holds the state or attributes of an object. Properties can be of any data type, such as int, string, or even another class.

VIKRAM RENESAS

```
1 class Animal;
2   string name; // Property
3   int age;     // Property
4 endclass
```

5. Method

A **method** is a function or task defined within a class. It defines the behavior or actions that an object can perform.

```
1 class Animal;
2   string name;
3   int age;
4
5   // Method to display the properties
6   function void display();
7     $display("Name: %s, Age: %0d", name, age);
8   endfunction
9 endclass
```

6. Prototype

A **prototype** is a declaration of a method (function or task) within a class. It specifies the method's name, return type (if any), and arguments (if any). The actual implementation of the method is defined later.

```
1  class Animal;
2    string name;
3    int age;
4
5    // Prototype of the method
6    extern function void display();
7  endclass
8
9  // Implementation of the method
10 function void Animal::display();
11   $display("Name: %s, Age: %0d", name, age);
12 endfunction
13
```

VIKRAM RENESAS

Complete code:

```
 3 class Animal;
 4   // Properties
 5   string name;
 6   int age;
 7
 8   // Method prototype
 9   extern function void display();
10 endclass
11
12 // Method implementation
13 function void Animal::display();
14   $display("Name: %s, Age: %0d", name, age);
15 endfunction
16
17 module oop_example;
18   initial begin
19     // Create a handle to an object of type 'Animal'
20     Animal my_animal;
21
22     // Create a new object and assign it to the handle
23     my_animal = new();
24
25     // Set the properties of the object
26     my_animal.name = "Dog";
27     my_animal.age = 5;
28
29     // Call the method to display the properties
30     my_animal.display();
31   end
32 endmodule
```

```
../simv up to date
CPU time: .460 seconds to compile + .382 seconds to elab + .390 seconds to link
Chronologic VCS simulator copyright 1991-2023
Contains Synopsys proprietary information.
Compiler version U-2023.03-SP2_Full64; Runtime version U-2023.03-SP2_Full64;  Feb  5 10:23 2025

------------------------------->
Name: Dog, Age: 5

------------------------------->
           V C S   S i m u l a t i o n   R e p o r t
```

Feature	new()	new[]
Purpose	Creates a single object of a class.	Creates a dynamic array of objects of a class.
Usage	Used when you want to instantiate a single object.	Used when you want to create a dynamic array of objects.
Syntax	ClassName obj = new();	ClassName arr[size] = new[size];
Creates	One instance of a class.	An array of instances of a class with a specified size.
Example	MyClass obj = new();	MyClass arr[10] = new[10];
Memory Allocation	Allocates memory for a single object.	Allocates memory for an array of objects.
Dynamic Size	Not applicable (single object).	Allows dynamic size for the array (can specify the number of elements).
Arguments	Can pass arguments to the constructor of the class.	Can pass arguments to the constructor of each object in the array (or to the array itself).
Argument Example	MyClass obj = new(5); (if the constructor takes an argument)	`MyClass arrct in the array needs an argument)

Feature	time	realtime
Type	time is a built-in type.	realtime is a built-in type.
Precision	64-bit unsigned integer.	64-bit signed floating-point number.
Range	Maximum value of time is 2^64 - 1 (for unsigned integer).	Can represent both positive and negative values.

VIKRAM RENESAS

Use Case	Used to represent **simulation time** in clock cycles or as a counter.	Used for **real-time** calculations or time intervals (e.g., in seconds).
Unit of Measurement	Typically represents **time in simulation cycles**.	Represents **real time in seconds** (floating point).
Resolution	Limited by simulation time resolution.	Can represent fractional seconds (e.g., 1.5, 0.0001).
Example	time t = 100; (Represents 100 time units in simulation cycles).	realtime rt = 0.01; (Represents 10 milliseconds).
Arithmetic	Cannot support fractional values.	Supports arithmetic with fractions (e.g., 5.5 or 0.1).
Conversion	Can be converted to/from real and int but not directly to/from time.	Can be used in expressions with floating-point numbers.

Null –object de-allocation

In SystemVerilog, setting an object reference to `null` is often used in conjunction with deallocation, but it **does not** actually deallocate the memory itself. Instead, setting an object reference to `null` simply makes the object reference **invalid** and removes the reference to the object, which means you can no longer access the object through that reference.

The **actual deallocation** of memory is done using the `delete` operator. Setting an object to `null` just **nullifies** the reference and marks it as no longer pointing to the object. The object itself will be deallocated only if there are **no other references** to it and the `delete` operator is used.

VIKRAM RENESAS

```
 1 class MyClass;
 2     int value;
 3
 4     function new(int val);
 5         value = val;
 6     endfunction
 7 endclass
 8
 9 module test;
10     MyClass obj;
11
12     initial begin
13         // Dynamically allocate an object
14         obj = new(10);
15
16       $display("------------------------------->");
17         // Access and display object value
18         $display("Object value: %0d", obj.value);
19
20         // Nullify the reference (does NOT deallocate memory)
21         obj = null;
22
23        $display("------------------------------->");
24       $display("Object value: %0d", obj.value);
25         // At this point, obj is null and cannot be accessed
26
27     end
28 endmodule
```

```
Compiler version U-2023.03-SP2_Full64; Runtime version U-2023.03-SP2_Full64;  Feb  5 10:38 2025
------------------------------>
Object value: 10
------------------------------>

Error-[NOA] Null object access
testbench.sv, 24
  The object at dereference depth 0 is being used before it was
  constructed/allocated.
  Please make sure that the object is allocated before using it.

  #0 in test at testbench.sv:24
  #1 in test

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

STATIC LOCAL AND GLOBAL

In SystemVerilog, **static**, **local**, and **global** variables have specific scoping and lifetime rules. Below is an example that demonstrates the use of **static**, **local**, and **global** variables in SystemVerilog.

1. **Static Variable**:
    - A static variable retains its value between function/task calls.
    - It is initialized only once and persists throughout the simulation.
    - Declared using the static keyword.
2. **Local Variable**:
    - A local variable is declared inside a function or task.
    - It is created when the function/task is called and destroyed when the function/task exits.
    - Declared without any special keyword.
3. **Global Variable**:
    - A global variable is declared outside any function, task, or class.
    - It is accessible throughout the entire module or program.
    - Declared at the module or program level.

RENESAS

EXAMPLE:

```
1 module variable_example;
2
3   // Global variable
4   int global_var = 0;
5
6   // Task to demonstrate static and local variables
7   task automatic my_task;
8     // Static variable (retains value between calls)
9     static int static_var = 0;
10
11    // Local variable (reinitialized on each call)
12    int local_var = 0;
13
14    // Increment variables
15    static_var++;
16    local_var++;
17    global_var++;
18
19    // Display values
20    $display("Static Var: %0d, Local Var: %0d, Global Var: %0d", static_var, local_var, global_var);
21  endtask
22
23  initial begin
24    // Call the task multiple times
25    my_task(); // Call 1
26    my_task(); // Call 2
27    my_task(); // Call 3
28  end
29
30 endmodule
```

```
Contains Synopsys proprietary information.
Compiler version U-2023.03-SP2_Full64; Runtime version U-2023.03-SP2_Full64;   Feb  5 12:25 2025
Static Var: 1, Local Var: 1, Global Var: 1
Static Var: 2, Local Var: 1, Global Var: 2
Static Var: 3, Local Var: 1, Global Var: 3
           V C S   S i m u l a t i o n   R e p o r t
```

Explanation of the Code

1. **Global Variable (global_var)**:
   - Declared outside the task, so it is accessible throughout the module.
   - Retains its value throughout the simulation.
2. **Static Variable (static_var)**:
   - Declared inside the task with the static keyword.
   - Retains its value between task calls.
3. **Local Variable (local_var)**:
   - Declared inside the task without any special keyword.
   - Reinitialized to 0 every time the task is called.

When to Use Each

- Use **static variables** when you need to retain values between function/task calls.
- Use **local variables** for temporary storage within a function/task.
- Use **global variables** for data that needs to be shared across multiple functions/tasks or throughout the module.

EXAMPLE 2:

```
 1 class MyClass;
 2   // Static variable (shared across all instances)
 3   static int static_var = 0;
 4
 5   // Non-static variable (unique to each instance)
 6   int non_static_var = 0;
 7
 8   // Method to demonstrate variable behavior
 9   function void display();
10     // Increment variables
11     static_var++;
12     non_static_var++;
13
14     // Display values
15     $display("Static Var: %0d, Non-Static Var: %0d", static_var, non_static_var);
16   endfunction
17 endclass
18
19 module class_variable_example;
20   initial begin
21     // Create two instances of the class
22     MyClass obj1 = new();
23     MyClass obj2 = new();
24
25     // Call the display method on both objects
26     $display("Object 1:");
27     obj1.display(); // Call 1
28     obj1.display(); // Call 2
29
30     $display("\nObject 2:");
31     obj2.display(); // Call 1
32     obj2.display(); // Call 2
33   end
34 endmodule
```

```
Compiler version U-2023.03-SP2_Full64; Runtime version U-2023.03-SP2_Full64;  Feb  5 12:30 2025
Object 1:
Static Var: 1, Non-Static Var: 1
Static Var: 2, Non-Static Var: 2

Object 2:
Static Var: 3, Non-Static Var: 1
Static Var: 4, Non-Static Var: 2
           V C S   S i m u l a t i o n   R e p o r t
```

PROTECTED KEYWORD

The `protected` keyword is used to specify that a class member (variable or method) can be accessed within the **class itself** and **by derived (child) classes**. However, it cannot be accessed from outside the class hierarchy, i.e., it is not accessible by instances of the class or non-derived classes.

```
 1 class BaseClass;
 2   protected int protected_data;  // Protected variable
 3
 4   // Protected method
 5   protected function void display_protected_data();
 6     $display("Protected Data: %0d", protected_data);
 7   endfunction
 8 endclass
 9
10 class DerivedClass extends BaseClass;
11   function new();
12     protected_data = 100;  // Accessible in derived class
13     display_protected_data();  // Can call protected method in derived class
14   endfunction
15 endclass
16
17 module test;
18   initial begin
19     DerivedClass obj = new();  // Creating an object of DerivedClass
20     // obj.protected_data = 50;  // This would cause an error (protected member can't be accessed directly)
21   end
22 endmodule
```

```
Contains Synopsys proprietary information.
Compiler version U-2023.03-SP2_Full64; Runtime version U-2023.03-SP2_Full64;  Feb  5 13:47 2025
Protected Data: 100
           V C S   S i m u l a t i o n   R e p o r t
Time: 0 ns
CPU Time:      0.460 seconds;       Data structure size:   0.0Mb
```

Keyword	Scope	Access Control
local	Limited to the block or function where it is declared (e.g., within an initial block or a function).	Only accessible within the block or function it is declared in. Cannot be accessed outside the scope of that block or function.
protected	Accessible within the current class and its derived (child) classes, but not outside of the class hierarchy.	Accessible only by the class that declares it and any subclasses (derived classes), not by outside classes or modules.

VIKRAM RENESAS

global	Accessible anywhere within the scope of the module, class, or program in which it is declared.	Accessible from anywhere within the module, class, or program where it is declared.

SHALLOW COPY AND DEEP COPY

In SystemVerilog, **shallow copy** and **deep copy** are two ways to copy objects. The key difference lies in how they handle **nested objects** or **dynamic data** within the object being copied.

Shallow Copy

- A **shallow copy** creates a new object and copies all the fields of the original object.
- If the object contains references to other objects (e.g., handles to class objects), the references are copied, but the referenced objects themselves are **not duplicated**.
- Both the original and the copied object will **share the same nested objects**.

Deep Copy

- A **deep copy** creates a new object and recursively copies all the fields of the original object, including any nested objects.
- If the object contains references to other objects, new instances of those objects are created, and their contents are copied.
- The original and the copied object will have **independent copies of nested objects**.

VIKRAM RENESAS

Aspect	Shallow Copy	Deep Copy
Definition	Copies the object and its fields, but not the objects referenced by its fields.	Copies the object, its fields, and recursively copies all nested objects.

Nested Objects	References to nested objects are shared between the original and copied object.	New instances of nested objects are created, and their contents are copied.
Memory Usage	Less memory is used because nested objects are not duplicated.	More memory is used because nested objects are duplicated.
Performance	Faster because it does not recursively copy nested objects.	Slower because it recursively copies nested objects.
Use Case	Suitable when nested objects are immutable or shared intentionally.	Suitable when nested objects need to be independent.

VIKRAM RENESAS

STATIC

In SystemVerilog, a **static method** is a method that belongs to a class itself, rather than to an instance of the class. This means that the method can be called without creating an object of the class, which is particularly useful when you need to perform a task that does not depend on any instance-specific data.

Characteristics of Static Methods in SystemVerilog:

1. **No Access to Instance Variables**: Static methods do not have access to instance-specific variables (non-static fields) or `this`. They can only access static fields and other static methods.
2. **Called Using Class Name**: Static methods are typically called using the class name, not an instance of the class.
3. **Used for Utility Functions**: Static methods are ideal for utility functions or operations that don't need object state, such as mathematical calculations or managing static data shared among all instances of the class.

Syntax to Define a Static Method:

To define a static method, you use the `static` keyword before the `function` or `task` keyword.

EXAMPLE FOR STATIC METHOD

```
1 class MyClass;
2     // Static variable
3     static int counter = 2;
4
5     // Non-static variable
6     int counter2 = 1;
7
8     // Static method to increment static counter
9     static function void increment_counter();
10         counter = counter + 1;
11       $display("\n-------------------->");
12         $display("Counter incremented: %0d", counter);
13       $display("--------------------> \n");
14     endfunction
15
16     // Non-static method to display both counters
17     function void display_counter();
18         $display("-------------------->");
19         $display("Current counter value (static): %0d", counter);
20         $display("Current counter2 value (non-static): %0d", counter2);
21         $display("-------------------->");
22     endfunction
23 endclass
24
25 // Usage
26 module test;
27    MyClass obj;
28
29     initial begin
30         // Calling static method without creating an object
31         MyClass::increment_counter();  // Output: Counter incremented: 1
32         MyClass::increment_counter();  // Output: Counter incremented: 2
33
34         // Creating an object and using non-static method
35
36         obj = new;
37         obj.display_counter();  // Output: Displays static and non-static counter values
38
39         // Modifying non-static counter and displaying again
40         obj.counter2 = 5;
41         obj.display_counter();  // Output: Displays updated static and non-static counter values
42     end
43 endmodule
```

RESULT:

```
Compiler version U-2023.03-SP2_Full64; Runtime version U-2023.03-SP2_Full64;  Feb  8 04:16 2025

-------------------->
Counter incremented: 3
-------------------->



-------------------->
Counter incremented: 4
-------------------->


-------------------->
Current counter value (static): 4
Current counter2 value (non-static): 1
-------------------->
-------------------->
Current counter value (static): 4
Current counter2 value (non-static): 5
-------------------->
           V C S   S i m u l a t i o n   R e p o r t
Time: 0 ns
CPU Time:      0.420 seconds;       Data structure size:   0.0Mb
```

Note : you can't call non static member in static method

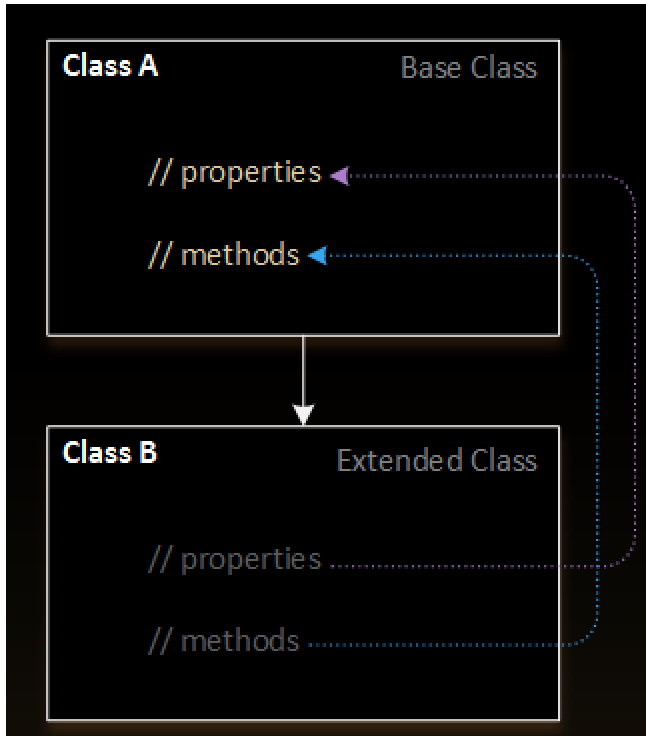
```
1 class MyClass;
2     // Static variable
3     static int counter = 2;
4 
5     // Non-static variable
6     int counter2 = 1;
7 
8     // Static method to increment static counter
9     static function void increment_counter();
10        counter = counter + 1;
11        counter2 = counter2 + 1;
12      $display("\n-------------------->");
13        $display("Counter incremented: %0d", counter);
14      $display("Counter2  incremented: %0d", counter2);
15      $display("--------------------> \n");
16    endfunction
17 
18    // Non-static method to display both counters
19    function void display_counter();
20        $display("-------------------->");
21        $display("Current counter value (static): %0d", counter);
22        $display("Current counter2 value (non-static): %0d", counter2);
23        $display("-------------------->");
24    endfunction
25 endclass
26 
27 // Usage
28 module test;
29    MyClass obj;
30 
31    initial begin
32        // Calling static method without creating an object
33        MyClass::increment_counter();  // Output: Counter incremented: 1
34        MyClass::increment_counter();  // Output: Counter incremented: 2
35 
36        // Creating an object and using non-static method
37 
38        obj = new;
39        obj.display_counter();  // Output: Displays static and non-static counter values
40 
41        // Modifying non-static counter and displaying again
42        obj.counter2 = 5;
43        obj.display_counter();  // Output: Displays updated static and non-static counter values
44    end
45 endmodule
```

ERROR:

```
Error-[SV-AMC] Non-static member access
testbench.sv, 13
$unit, "counter2"
  Illegal access of non-static member 'counter2' from static method
  'MyClass::increment_counter'.
```

INHERETANCE

In SystemVerilog, **inheritance** is a key feature of Object-Oriented Programming (OOP) that allows you to create a new class (called the **derived class** or **subclass**) based on an existing class (called the **base class** or **superclass**). The derived class inherits all the properties and methods of the base class and can also add new properties and methods or override existing ones.



Key Concepts of Inheritance

1. **Base Class (Superclass)**:
   - The original class from which properties and methods are inherited.
2. **Derived Class (Subclass)**:
   - The new class that inherits from the base class.
   - Can add new properties and methods.
   - Can override methods from the base class.
3. extends **Keyword**:
   - Used to specify that a class inherits from another class.
4. super **Keyword**:
   - Used to refer to the base class from within the derived class.
   - Commonly used to call the base class constructor or access base class methods.

VIKRAM RENESAS

EXAMPLE 1:

```
1 class parent_class;
2   bit [31:0] addr;
3 endclass
4
5 class child_class extends parent_class;
6   bit [31:0] data;
7 endclass
8
9 module inheritence;
10   initial begin
11     child_class c = new();
12     c.addr = 10;
13     c.data = 20;
14     $display("\n\n ===========================>>>>>>>>>>>>");
15     $display("Value of addr = %0d data = %0d",c.addr,c.data);
16     $display("\n\n ===========================>>>>>>>>>>>>");
17   end
18 endmodule
```

RESULT:

```
Contains Synopsys proprietary information.
Compiler version U-2023.03-SP2_Full64; Runtime version U-2023.03-SP2_Full64;  Feb  5 12:51 2025


 ===========================>>>>>>>>>>>>
Value of addr = 10 data = 20
===========================>>>>>>>>>>>>

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

parent_class is a class that has one member variable:

- **addr** is a 32-bit wide `bit` (which is similar to a `reg` in Verilog but more suited for use in classes). This variable is intended to hold an address.

child_class is another class, which **extends** the **parent_class**, meaning it inherits all the properties and behaviors of `parent_class`.

- In addition to the `addr` variable inherited from the `parent_class`, **child_class** defines its own member:
  - **data** is another 32-bit wide `bit` variable.

The `initial` block is used for initialization in simulation. It runs once at the start of the simulation.

- **`child_class c = new();`**: This creates an object `c` of type `child_class` using the `new()` constructor. This object will have both the `addr` (from the parent) and `data` (from the child) properties.
- **`c.addr = 10;`**: Sets the `addr` member of the object `c` to `10`.
- **`c.data = 20;`**: Sets the `data` member of the object `c` to `20`.
- **`$display(...)`**: This is a SystemVerilog task that displays the output to the console.

VIKRAM RENESAS

- The first `$display` adds some formatting (a divider line).
- The second `$display` prints the values of `c.addr` and `c.data` (i.e., 10 and 20).
- The third `$display` adds another divider line.

SUPER KEYWORD

In SystemVerilog, the **super keyword** is used in the context of **inheritance** to refer to the **base class (superclass)** from within a **derived class (subclass)**. It is primarily used for two purposes:

1. **Calling the Base Class Constructor**:
   - When a derived class is instantiated, the base class constructor must be called to initialize the inherited properties.
   - The super keyword is used to explicitly call the base class constructor from the derived class constructor.
2. **Accessing Base Class Methods**:
   - If a derived class overrides a method from the base class, the super keyword can be used to call the base class version of the method from within the derived class.

```
1 class parent_class;
2   bit [31:0] addr;
3
4   task printf();
5   $display(" THIS IS PARENT CLASS ");
6   endtask
7 endclass
8
9 class child_class extends parent_class;
10   bit [31:0] data;
11
12   task printf();
13   super.printf();
14   endtask
15
16 endclass
17
18 module inheritence;
19   initial begin
20     child_class c;
21     c = new();
22     c.addr = 10;
23     c.data = 20;
24
25     $display("\n ===========================>>>>>>>>>>>>");
26     c.printf();
27     $display("Value of addr = %0d data = %0d",c.addr,c.data);
28     $display("\============================>>>>>>>>>>>>\n");
29   end
30 endmodule
```

```
Chronologic VCS simulator copyright 1991-2023
Contains Synopsys proprietary information.
```

```
Compiler version U-2023.03-SP2_Full64; Runtime version U-2023.03-SP2_Full64;  Feb  5 13:04 2025

 ===========================>>>>>>>>>>>>
 THIS IS PARENT CLASS
Value of addr = 10 data = 20
===========================>>>>>>>>>>>>

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

EXAMPLE3

```
1 class parent_class;
2   bit [31:0] addr;
3
4   task printf();
5   $display(" THIS IS PARENT CLASS ");
6   endtask
7 endclass
8
9 class child_class extends parent_class;
10   bit [31:0] data;
11
12   task printf();
13     $display(" THIS IS CHILD CLASS ");
14   super.printf();
15   endtask
16
17 endclass
18
19 module inheritence;
20   initial begin
21     child_class c;
22     c = new();
23     c.addr = 10;
24     c.data = 20;
25
26     $display("\n ===========================>>>>>>>>>>>>");
27     c.printf();
28     $display("Value of addr = %0d data = %0d",c.addr,c.data);
29     $display("\===========================>>>>>>>>>>>>\n");
30   end
31 endmodule
```

```
Contains Synopsys proprietary information.
Compiler version U-2023.03-SP2_Full64; Runtime version U-2023.03-SP2_Full64;  Feb  5 13:07 2025

 ===========================>>>>>>>>>>>>
 THIS IS CHILD CLASS
 THIS IS PARENT CLASS
Value of addr = 10 data = 20
===========================>>>>>>>>>>>>

           V C S   S i m u l a t i o n   R e p o r t
Time: 0 ns
CPU Time:      0.380 seconds;       Data structure size:   0.0Mb
```

EXAMPLE 4

```
 1 class parent_class;
 2   int N = 3; // Parent class variable
 3
 4   // Function to get the value of N
 5   function int get_N();
 6     $display("Value PARENT variable N = %0d", N);
 7     return N;
 8   endfunction
 9 endclass
10
11 class child_class extends parent_class;
12   int N = 4; // Child class variable (shadows parent class variable)
13
14   // Override the get_N function
15   function int get_N();
16     $display("Value CHILD variable N = %0d", N);
17     return N;
18   endfunction
19
20   // Function to access parent class variable N
21   function int get_N1();
22     $display("Value PARENT variable N (via super) = %0d", super.N);
23     return super.N; // Return parent class variable N
24   endfunction
25 endclass
26
27 module testbench;
28   initial begin
29     // Create an object of the child class
30     child_class child_obj = new();
31
32     // Call get_N (child class version)
33     $display("Child get_N(): %0d", child_obj.get_N());
34
35     // Call get_N1 (access parent class variable N)
36     $display("Child get_N1(): %0d", child_obj.get_N1());
37   end
38 endmodule
```

RESULT:

```
Chronologic VCS simulator copyright 1991-2023
Contains Synopsys proprietary information.
Compiler version U-2023.03-SP2_Full64; Runtime version U-2023.03-SP2_Full64;  Feb  5 13:19 2025


 ===========================>>>>>>>>>>>>
Value CHILD variable N = 4
Child get_N(): 4


 ===========================>>>>>>>>>>>>
Value PARENT variable N (via super) = 3
Child get_N1(): 3


 ===========================>>>>>>>>>>>>
          V C S   S i m u l a t i o n   R e p o r t
```

“THIS” KEYWORD

The keyword is used to refer to class properties, parameters and methods of the current instance. It can only be used within non-static methods, constraints and covergroups. this is basically a pre-defined object handle that refers to the object that was used to invoke the method in which this is used.

- **This** is used inside a class method to refer to the object that is currently calling the method.
- It can be useful when you need to distinguish between the instance's own variables and parameters that might be passed to a method with the same name.
- It’s typically used in constructor methods, for chaining, or when you need to call another method or access another member within the same class.

EXAMPLE:

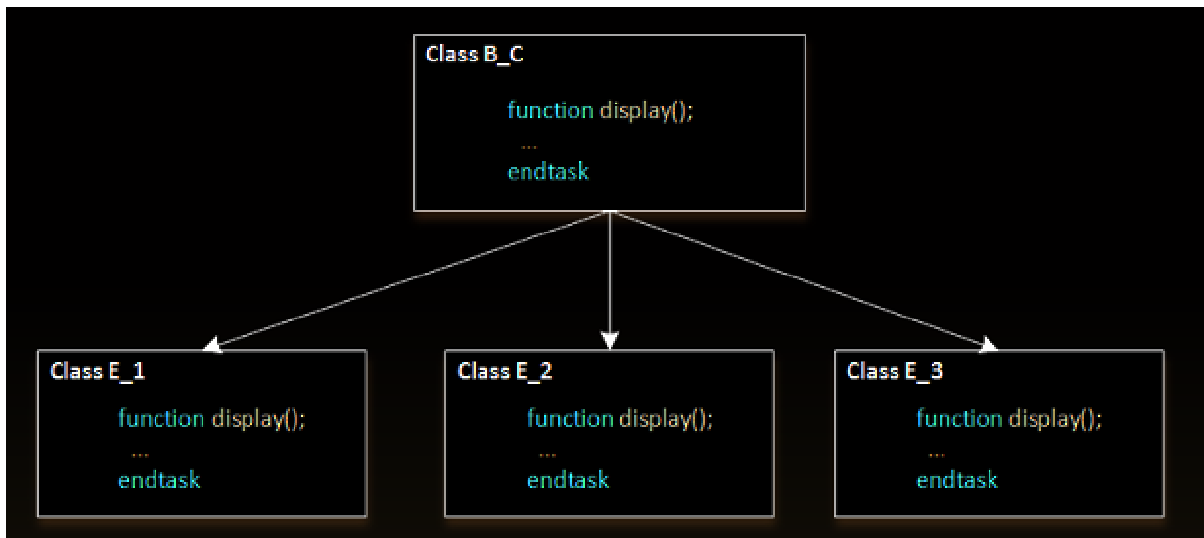
```
 1 class MyClass;
 2   int x;
 3
 4   // Constructor
 5   function new(int x);
 6     this.x = x;  // Use 'this' to refer to the current object's 'x'
 7   endfunction
 8
 9   // Method to display the value
10   function void display();
11     $display("Value of x: %0d", this.x); // Using 'this' to refer to current instance's 'x'
12   endfunction
13 endclass
14
15 module test;
16   initial begin
17     MyClass obj = new(10);  // Create an object of MyClass and initialize 'x' to 10
18      $display("\n ===========================>>>>>>>>>>>>");
19     obj.display();           // Call the display method to print the value of 'x'
20     $display("===========================>>>>>>>>>>>>\n");
21   end
22 endmodule
~
```

```
CPU time: .406 seconds to compile + .426 seconds to elab + .485 seconds to link
Chronologic VCS simulator copyright 1991-2023
Contains Synopsys proprietary information.
Compiler version U-2023.03-SP2_Full64; Runtime version U-2023.03-SP2_Full64;  Feb  5 13:24 2025

 ===========================>>>>>>>>>>>>
Value of x: 10
===========================>>>>>>>>>>>>

           V C S   S i m u l a t i o n   R e p o r t
Time: 0 ns
CPU Time:      0.460 seconds;       Data structure size:   0.0Mb
```

POLYMORPHISM IN SYSTEMVERILOG



Polymorphism in SystemVerilog

Polymorphism is one of the core concepts of Object-Oriented Programming (OOP), and it allows for the ability to use a single interface to represent different types of objects. In SystemVerilog, polymorphism is primarily used with **classes**, and it enables you to write more flexible and reusable code by allowing objects of different classes to be treated through a common interface.

Polymorphism means many forms. Polymorphism in SystemVerilog provides an ability to an object to take on many forms.

Key Concepts of Polymorphism

1. **Base Class and Derived Classes**:
   - A **base class** defines a common interface (methods and properties).
   - **Derived classes** inherit from the base class and can override its methods to provide specific implementations.
2. **Virtual Methods**:
   - Methods in the base class must be declared as virtual to allow overriding in derived classes.
   - When a base class handle refers to a derived class object, the overridden method in the derived class is called.
3. **Dynamic Method Dispatch**:
   - The actual method to be called is determined at runtime based on the type of the object, not the type of the handle

VIKRAM RENESAS

1. **Polymorphism** allows a base class handle to refer to objects of derived classes.
2. **Virtual methods** enable dynamic method dispatch.
3. Polymorphism is a powerful feature for writing flexible and reusable code

SystemVerilog supports **two types of polymorphism**:

1. **Compile-time Polymorphism** (Method Overloading)
2. **Runtime Polymorphism** (Method Overriding)

1. Compile-time Polymorphism (Method Overloading)

This is achieved by having multiple methods in the same class with the same name but different argument types or numbers of arguments. This allows the compiler to choose the correct method at compile-time based on the method signature.

Example 1:

```
1 class Shape;
2     // Virtual method
3     function void draw();
4         $display("Drawing a generic shape");
5     endfunction
6 endclass
7
8 class Circle extends Shape;
9     // Overloading the draw() method
10    function void draw(int radius);
11        $display("Drawing a circle with radius %0d", radius);
12    endfunction
13 endclass
14
15 class Rectangle extends Shape;
16    // Overloading the draw() method
17    function void draw(int length, int width);
18        $display("Drawing a rectangle with length %0d and width %0d", length, width);
19    endfunction
20 endclass
21
22 module test;
23    initial begin
24        Shape s = new();
25        Circle c = new();
26        Rectangle r = new();
27
28        s.draw();                // Calls Shape's draw method
29        c.draw(5);               // Calls Circle's draw method
30        r.draw(10, 20);          // Calls Rectangle's draw method
31    end
32 endmodule
```

RESULT:

```
Chronologic VCS simulator copyright 1991-2023
Contains Synopsys proprietary information.
Compiler version U-2023.03-SP2_Full64; Runtime version U-2023.03-SP2_Full64;  Feb  5 13:54 2025
Drawing a generic shape
Drawing a circle with radius 5
Drawing a rectangle with length 10 and width 20
           V C S   S i m u l a t i o n   R e p o r t
Time: 0 ns
CPU Time:      0.400 seconds;       Data structure size:   0.0Mb
```

In this example, the `draw()` method is overloaded in the derived classes (`Circle` and `Rectangle`) with different parameters. The method that gets called is determined at compile-time based on the arguments passed.

2. Runtime Polymorphism (Method Overriding)

Runtime polymorphism is achieved through **method overriding**. In this case, a derived class provides its specific implementation of a method that was declared in the base class. The method to be called is determined at runtime based on the actual object type (i.e., whether it's a `Circle`, `Rectangle`, or a `Shape`).

```
 1 class Shape;
 2     // Virtual method
 3     virtual function void draw();
 4         $display("Drawing a generic shape");
 5     endfunction
 6 endclass
 7
 8 class Circle extends Shape;
 9     // Overriding the draw() method
10     function void draw();
11         $display("Drawing a circle");
12     endfunction
13 endclass
14
15 class Rectangle extends Shape;
16     // Overriding the draw() method
17     function void draw();
18         $display("Drawing a rectangle");
19     endfunction
20 endclass
21
22 module test;
23     initial begin
24         Shape s;          // Shape reference
25         Circle c = new();
26         Rectangle r = new();
27
28         s = c;            // Reference to Circle
29         s.draw();         // Calls Circle's draw method at runtime
30
31         s = r;            // Reference to Rectangle
32         s.draw();         // Calls Rectangle's draw method at runtime
33     end
34 endmodule
~
```

```
Chronologic VCS simulator copyright 1991-2023
Contains Synopsys proprietary information.
Compiler version U-2023.03-SP2_Full64; Runtime version U-2023.03-SP2_Full64;  Feb  5 13:57 2025
Drawing a circle
Drawing a rectangle
           V C S   S i m u l a t i o n   R e p o r t
Time: 0 ns
CPU Time:      0.440 seconds;       Data structure size:   0.0Mb
```

EXAMPLE 1 WITHOUT USING VIRTUAL KEYWORD

```
 1 class parent;
 2
 3   function void display();
 4     $display("Inside parent class");
 5   endfunction
 6
 7 endclass
 8
 9 class child_1 extends parent;
10   function void display();
11     $display("Inside child class 1");
12   endfunction
13 endclass
14
15 module class_polymorphism;
16   parent p;
17   child_1 c1;
18
19   initial begin
20     p=new();
21     c1=new();
22
23     p=c1;
24     //accessing extended class methods using base class handle
25     $display("\n-------------------->");
26     p.display();
27     c1.display();
28     $display("--------------------> \n");
29
30   end
31
32 endmodule
```

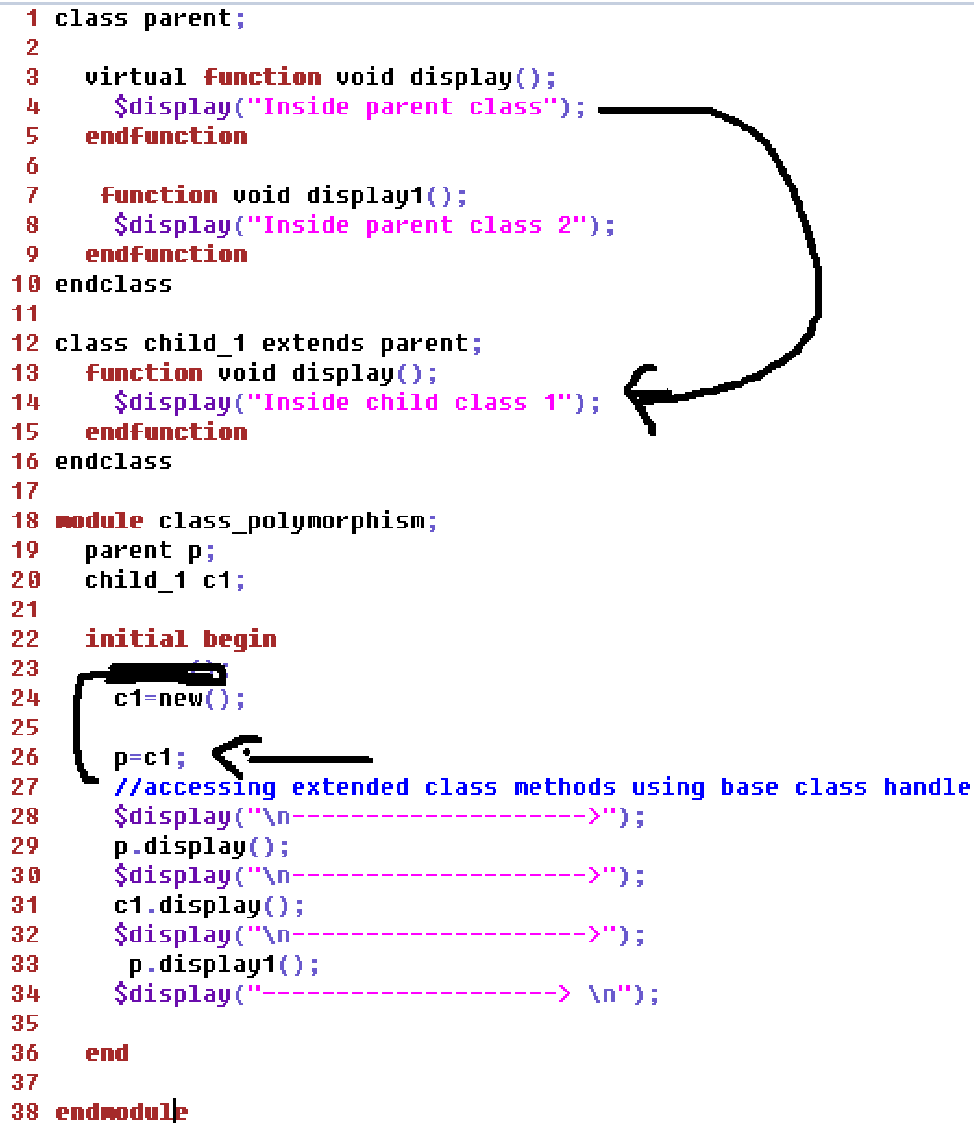
RESULT:

```
Contains Synopsys proprietary information.
Compiler version U-2023.03-SP2_Full64; Runtime version U-2023.03-SP2_Full64;  Feb  8 04:28 2025

-------------------->
Inside parent class
Inside child class 1
-------------------->

           V C S   S i m u l a t i o n   R e p o r t
Time: 0 ns
CPU Time:      0.340 seconds;       Data structure size:   0.0Mb
Sat Feb  8 04:28:58 2025
```

WITH USING VIRTAL KEYWORD



```
class parent;

  virtual function void display();
    $display("Inside parent class");
  endfunction

   function void display1();
    $display("Inside parent class 2");
  endfunction
endclass

class child_1 extends parent;
  function void display();
    $display("Inside child class 1");
  endfunction
endclass

module class_polymorphism;
  parent p;
  child_1 c1;

  initial begin

    c1=new();

    p=c1;
    //accessing extended class methods using base class handle
    $display("\n-------------------->");
    p.display();
    $display("\n-------------------->");
    c1.display();
    $display("\n-------------------->");
     p.display1();
    $display("--------------------> \n");

  end

endmodule
```

RESULT :

```
Compiler version U-2023.03-SP2_Full64; Runtime version U-2023.03-SP2_Full64;  Feb  8 04:35 2025

-------------------->
Inside child class 1

-------------------->
Inside child class 1

-------------------->
Inside parent class 2
-------------------->

           V C S   S i m u l a t i o n   R e p o r t
Time: 0 ns
CPU Time:      0.320 seconds;       Data structure size:   0.0Mb
```

Explanation:

The parent class contains two methods:

- **display()**: This is a **virtual function**, which means it can be overridden by derived classes (child classes). The virtual keyword allows subclasses to provide their own implementation of this method, enabling **run-time polymorphism**.
- **display1()**: This is a regular (non-virtual) function. It cannot be overridden by child classes and will always call the parent class's implementation.

The display() function is designed to print "Inside parent class", and display1() prints "Inside parent class 2".

- The child_1 class inherits from the parent class.
- The display() function in child_1 **overrides** the display() function in the parent class. The display() method in child_1 prints "Inside child class 1".
- Since display1() is not marked as virtual, the child_1 class cannot override it. So, the version of display1() in the parent class will be used when called from child_1 objects.

Key Concepts in the Code:

1. **Polymorphism**:
   - The line p = c1; demonstrates **polymorphism**. Here, the parent class handle p is assigned an instance of child_1. Even though p is of type parent, it points to an object of child_1. This allows us to call the display() method in a polymorphic way.
   - When p.display() is called, the **overridden version** of display() in child_1 is executed, not the one in parent. This is **run-time polymorphism**.
   - The p.display1() call will invoke the display1() method from the **parent class**, because display1() is not virtual and cannot be overridden in child_1.
2. **Method Overriding**:
   - The display() method in child_1 overrides the display() method in parent. This is done by providing a new implementation for display() in the child_1 class.
   - When p.display() is called (even though p is a parent type handle), the method in child_1 is executed because display() is virtual.
3. **Base Class Handle**:
   - The parent class handle p can point to an object of any class that extends parent, like child_1. This allows accessing methods from the derived class through the base class reference.
   - c1.display() is a direct call to the display() method of child_1, and it also calls the overridden display() in the child_1 class.

VIKRAM RENESAS

OVERRIDE

```
1 class parent;
2
3   int addr;
4
5    function void display();
6     $display("Inside parent class 2");
7     $display("Addr = %0d",addr);
8   endfunction
9
10 endclass
11
12 class child_1 extends parent;
13   int data;
14
15   function void display();
16     $display("Inside child class 1");
17     $display("Data = %0d",data);
18     $display("Addr = %0d",addr);
19   endfunction
20
21 endclass
22
23 module class_polymorphism;
24   parent p;
25   child_1 c1;
26
27   initial begin
28
29     c1=new();
30     p=c1;
31
32     //accessing extended class methods using base class handle
33     $display("\n-------------------->");
34     p.display();
35     $display("\n-------------------->");
36     c1.addr=50;
37     c1.data=60;
38     c1.display();
39     $display("\n-------------------->");
40   end
41
42 endmodule
```

```
-------------------->
Inside parent class 2
Addr = 0

-------------------->
Inside child class 1
Data = 60
Addr = 50

-------------------->
           V C S   S i m u l a t i o n   R e p o r t
Time: 0 ns
CPU Time:      0.350 seconds;       Data structure size:   0.0Mb
Sat Feb  8 05:00:34 2025
```

EXAMPLE 2

```
1 class parent;
2
3   // Virtual function in parent class
4   virtual function void display();
5     $display("Inside parent class");
6   endfunction
7
8   // Non-virtual function
9   function void display1();
10    $display("Inside parent class 2");
11  endfunction
12 endclass
13
14 // Child class 1 with virtual keyword
15 class child_1 extends parent;
16   virtual function void display();  // This will allow this method to be overridden by child classes
17     $display("Inside child class 1");
18   endfunction
19 endclass
20
21 // Child class 2 with virtual keyword
22 class child_2 extends child_1; // child_2 extends child_1 now
23   virtual function void display();  // This will override child_1's display method
24     $display("Inside child class 2");
25   endfunction
26 endclass
27
28 // Child class 3 with overridden display method
29 class child_3 extends child_2;  // Inherits from child_2
30   function void display();  // This will override the display method of child_2
31     $display("Inside child class 3");
32   endfunction
33 endclass
34
```

```
35 module class_polymorphism;
36   parent p;    // Base class handle (this can hold a reference to any child class object)
37   child_1 c1; // Object of child_1
38   child_2 c2; // Object of child_2
39   child_3 c3; // Object of child_3
40
41   initial begin
42     p = new();  // Create a new object of the parent class
43     c1 = new(); // Create a new object of child_1
44     c2 = new(); // Create a new object of child_2
45     c3 = new(); // Create a new object of child_3
46
47     // Using polymorphism to point to different child objects
48
49     p = c1;  // Now p points to an object of child_1
50     $display("\n-------------------->");
51     p.display();  // Will call display() of child_1 because p points to child_1 object
52     $display("-------------------->\n");
53
54     p = c2;  // Now p points to an object of child_2
55     $display("\n-------------------->");
56     p.display();  // Will call display() of child_2 because p points to child_2 object
57     $display("-------------------->\n");
58
59     p = c3;  // Now p points to an object of child_3
60     $display("\n-------------------->");
61     p.display();  // Will call display() of child_3 because p points to child_3 object
62     $display("-------------------->\n");
63
64     // Accessing the parent class method
65     p.display1(); // Always calls the parent class display1() method
66     $display("--------------------> \n");
67
68   end
69
70 endmodule
```

```
Contains Synopsys proprietary information.
Compiler version U-2023.03-SP2_Full64; Runtime version U-2023.03-SP2_Full64;  Feb  8 05:10 2025

--------------------->
Inside child class 1
--------------------->


--------------------->
Inside child class 2
--------------------->


--------------------->
Inside child class 3
--------------------->

Inside parent class 2
--------------------->

           V C S   S i m u l a t i o n   R e p o r t
Time: 0 ns
CPU Time:      0.380 seconds;       Data structure size:   0.0Mb
Sat Feb  8 05:10:06 2025
```

VIKRAM RENESAS

IMPORTANT:

Here child can get all property from parents ex father took 2bh property it will go to child

Parents cannot get child property ex child took 2bhk that cannot be in herniated to parents

Another way father has cricket or singing or blue eye pr sugar it will be inherited to child but child skills will not in to parent if child learn cooking or any other it will not got to father

P=C1; -> ALLOWED

C1=P -> NOT ALLOWED

ERROR

```
Error-[SV-ICA] Illegal class assignment
testbench.sv, 27
"c1 = p;"
  Expression 'p' on rhs is not a class or a compatible class and hence cannot
  be assigned to a class handle on lhs.
  Source type: class $unit::parent
  Target type: class $unit::child_1
  Please make sure that the lhs and rhs expressions are compatible.
```

VIKRAM RENESAS

CASTING

DYNAMIC CASTING

- Dynamic casting is used to, safely cast a super-class pointer (reference) into a subclass pointer (reference) in a class hierarchy
- Dynamic casting will be checked during run time, an attempt to cast an object to an incompatible object will result in a run-time error
- Dynamic casting is done using the $cast(destination, source) method
- With $cast compatibility of the assignment will not be checked during compile time, it will be checked during run-time
  - It is never legal to directly assign a super-class (parent class) variable to a variable of one of its subclasses (child class).

    `child_class = parent_class;` //not-allowed

  - However, it is legal to assign a super-class (parent class) handle to a subclass (child class) variable **if** the super-class (parent class) handle refers to an object of the given subclass(child class).

    `parent_class = child_class ;`
    `child_class = parent_class;` //allowed because parent_class is pointing to child_class.

ALLOWED

```
 1 class parent_class;
 2   bit [31:0] addr;
 3   function display();
 4     $display("Addr = %0d",addr);
 5   endfunction
 6 endclass
 7
 8 class child_class extends parent_class;
 9   bit [31:0] data;
10   function display();
11     super.display();
12     $display("Data = %0d",data);
13   endfunction
14 endclass
15
16 module inheritence;
17   initial begin
18     parent_class p=new();
19     child_class  c=new();
20     c.addr = 10;
21     c.data = 20;
22     p = c;        //assigning child class handle to parent class handle
23     c.display();
24   end
25 endmodule
~
```

RESULT

```
Contains Synopsys proprietary information.
Compiler version U-2023.03-SP2_Full64; Runtime version U-2023.03-SP2_Full64;  Feb  8 05:18 2025
```

```
Addr = 10
Data = 20
           V C S   S i m u l a t i o n   R e p o r t
Time: 0 ns
CPU Time:      0.580 seconds;       Data structure size:   0.0Mb
Sat Feb  8 05:18:20 202
```

NOT ALLOWED

```
1 class parent_class;
2   bit [31:0] addr;
3   function display();
4     $display("Addr = %0d",addr);
5   endfunction
6 endclass
7
8 class child_class extends parent_class;
9   bit [31:0] data;
10  function display();
11    super.display();
12    $display("Data = %0d",data);
13  endfunction
14 endclass
15
16 module inheritence;
17   initial begin
18     parent_class p=new();
19     child_class  c=new();
20     c.addr = 10;
21     c.data = 20;
22     c = p;          //assigning parent class handle to child class handle
23     c.display();
24   end
25 endmodule
```

ERROR

```
Error-[SV-ICA] Illegal class assignment
testbench.sv, 22
"c = p;"
  Expression 'p' on rhs is not a class or a compatible class and hence cannot
  be assigned to a class handle on lhs.
  Source type: class $unit::parent_class
  Target type: class $unit::child_class
  Please make sure that the lhs and rhs expressions are compatible.
```

```
1 class parent_class;
2   bit [31:0] addr;
3   function display();
4     $display("Addr = %0d",addr);
5   endfunction
6 endclass
7
8 class child_class extends parent_class;
9   bit [31:0] data;
10  function display();
11    super.display();
12    $display("Data = %0d",data);
13  endfunction
14 endclass
15
16 module inheritence;
17   initial begin
18     parent_class p=new();
19     child_class  c=new();
20     c.addr = 10;
21     c.data = 20;
22     c = p;       //assigning parent class handle to child class handle
23     c.display();
24   end
25 endmodule
```

ALLOWED

NOT ALLOWED

Use of $cast or casting

In the above example, assigning parent class handle (which is pointing to child class handle) to child class handle is valid but compilation error is observed.
During the compile time, as the handle of p is of parent class type which leads to compile error.

With the use of $cast(), type check during compile time can be skipped.

```
 1 class parent_class;
 2   bit [31:0] addr;
 3
 4   function display();
 5     $display("Addr = %0d",addr);
 6   endfunction
 7 endclass
 8
 9 class child_class extends parent_class;
10   bit [31:0] data;
11
12   function display();
13     super.display();
14     $display("Data = %0d",data);
15   endfunction
16 endclass
17
18 module inheritence;
19   initial begin
20     parent_class p;
21     child_class  c=new();
22     child_class  c1;
23     c.addr = 10;
24     c.data = 20;
25     p = c;          //p is pointing to child class handle c.
26     $cast(c1,p);    //with the use of $cast, type chek will occur during runtime
27     c1.display();
28   end
29 endmodule
```

```
Compiler version U-2023.03-SP2_Full64; Runtime version U-2023.03-SP2_Full64;  Feb  8 05:29 2025
Addr = 10
Data = 20
           V C S   S i m u l a t i o n   R e p o r t
Time: 0 ns
CPU Time:      0.380 seconds;       Data structure size:   0.0Mb
```

ABSTRACT CLASS

SystemVerilog prohibits a class declared as `virtual` to be directly instantiated and is called an abstract class

EXAMPLE NORMAL CLASS

```
1 class parent_class;
2   bit [31:0] addr;
3
4   function display();
5     $display("Addr = %0d",addr);
6   endfunction
7 endclass
8
9 module inheritence;
10
11   parent_class p;
12
13   initial begin
14
15     p=new();
16     p.addr = 10;
17     $display("\n-------------------->");
18     p.display();
19     $display("--------------------> \n");
20
21   end
22 endmodule
```

RESULT

```
Compiler version U-2023.03-SP2_Full64; Runtime version U-2023.03-SP2_Full64;  Feb  8 05:50 2025

-------------------->
Addr = 10
-------------------->

           V C S   S i m u l a t i o n   R e p o r t
Time: 0 ns
CPU Time:      0.370 seconds;       Data structure size:   0.0Mb
```

VIKRAM RENESAS

- **ABSTRACT CLASS EXAMPLE**

Let us declare the class base as virtual to make it an abstract class and see what happens.

PURE VIRTUAL METHOD

A virtual method inside an abstract class can be declared with the keyword pure and is called a pure virtual method. Such methods only require a prototype to be specified within the abstract class and the implementation is left to defined within the sub-classes.

```
 1 virtual class parent_class;
 2   bit [31:0] addr;
 3
 4  pure virtual function int get();
 5
 6 endclass
 7
 8 class child extends parent_class;
 9   bit [31:0] addr1=60;
10
11   function int get();
12     $display("Addr = %0d",addr1);
13     return addr1;
14   endfunction
15 endclass
16
17
18
19 module inheritence;
20
21   child  c;
22
23   initial begin
24
25     c=new();
26      $display("\n-------------------->");
27     $display ("addr1 = 0x%0h", c.get());
28     $display("--------------------> \n");
29
30   end
31 endmodule
~
```

```
CPU time: .376 seconds to compile + .360 seconds to elab + .393 seconds to link
Chronologic VCS simulator copyright 1991-2023
Contains Synopsys proprietary information.
Compiler version U-2023.03-SP2_Full64; Runtime version U-2023.03-SP2_Full64;  Feb  8 05:29 2025
Addr = 10
Data = 20
           V C S   S i m u l a t i o n   R e p o r t
Time: 0 ns
CPU Time:      0.380 seconds;       Data structure size:   0.0Mb
Sat Feb  8 05:29:25 2025
```

EXTERN KEYWORD

```
 1 class parent_class;
 2   bit [31:0] addr = 10;
 3
 4   // Declare the display function as extern
 5   extern function void display();
 6 endclass
 7
 8 // Define the extern function outside the class, scoped to parent_class
 9 function void parent_class::display();
10   $display("INSIDE PARENT CLASS: addr = %0d", addr);
11 endfunction
12
13 module inheritance;
14   parent_class c;
15
16   initial begin
17     c = new();
18
19     // Calling the extern function display inside the class
20     $display("\n-------------------->");
21     c.display();
22     $display("--------------------> \n");
23   end
24 endmodule
```

```
ompiler version U-2023.03-SP2_Full64; Runtime version U-2023.03-SP2_Full64;  Feb  8 06:18 2025

-------------------->
INSIDE PARENT CLASS: addr = 10
-------------------->

           V C S   S i m u l a t i o n   R e p o r t
Time: 0 ns
CPU Time:      0.420 seconds;       Data structure size:   0.0Mb
```

VIKRAM

```
 1 virtual class parent_class;
 2   bit [31:0] addr;
 3
 4   function display();
 5     $display("Addr = %0d",addr);
 6   endfunction
 7 endclass
 8
 9 class child extends parent_class;
10   bit [31:0] addr1;
11
12   function display1();
13     $display("Addr = %0d",addr1);
14   endfunction
15 endclass
16
17
18
19 module inheritence;
20
21   parent_class p;
22
23   initial begin
24
25     p=new();
26     p.addr = 10;
27     $display("\n-------------------->");
28     p.display();
29     $display("--------------------> \n");
30
31   end
32 endmodule
~
```

Error

```
Error-[SV-ACCNBI] An abstract class cannot be instantiated
testbench.sv, 25
inheritence, "p = new();"
  Instantiation of the object 'p' can not be done because its type
  'parent_class' is an abstract base class.
  Perhaps there is a derived class that should be used.

1 warning
1 error
```