ENCS5337: Chip Design Verification

Spring 2023/2024

SystemVerilog II

Dr. Ayman Hroub

Outline

- **Classes**
- Static Properties and Methods
- Aggregate Classes
- Inheritance
- Polymorphism
- Randomization

Classes (1)

- Class is a user-defined data type
- Classes must be declared in a module, package, interface, or a program.
- Class declaration contains data items (attributes/properties), and subroutines (tasks and functions) operate on these data items
- Class objects can be dynamically created and deleted during simulation
- Used in Object-Oriented (OO) programming for testbenches and simulation models

```
module myModule
    class myClass
       bit[7:0] address;
       bit[7:0] data;
   endclass
…
```

Classes (2)

- A variable of a class type is called a handle whose unintialized value is **`null`**.
- A class instance must be created for handle using a constructor.
- Procedural call to function **`new`**
- It allocates memory area to hold the instance
- SystemVerilog has no destructor, i.e., automatic garbage collection

```
module myModule
     class myClass
          bit[7:0] address;
          bit[7:0] data;
     endclass

     myClass obj1 = new;
…
```

Class Attributes and Methods

```
class myClass
    int address;
    int data;

    task setData (input int newData)
        data = newData
    endtask

    function int getData()
        return data;
    endfunction

...
endclass

myClass obj1 = new;
initial begin
obj1.data = 10; // or obj1.setData(10)
end
```

External Method Declaration

- It is used for better readability
- Define the method prototype in the class prefixed by the keyword **`extern`**
- The prototype is the first line that identifies the method type, name, and arguments

```
extern function int getData();
```

- Then, the method is implemented outside the class declaration, but in the same scope

```
function int myClass::getData();
    return data;
endfunction
```

External Method Example

```
class myclass;
  int number;

  task set (input int i);
      number = i;
  endtask

  extern function int get();

endclass

function int myclass::get();
  return number;
endfunction
```

```
myclass c2 = new;

initial begin
  c2.set(3);
  $display("c2: %d", c2.get());
end
```

Implicit (Default) Class Constructor

- Method **`new`** is special class method called constructor.
- Defined by default for all classes.
- When the object is created using the **`new`** constructor, the object's fields are initialized to their default initial values based on their data type

Explicit Class Constructor

- You can explicitly define the constructor to initialize the object's fields
- The function **new** will not have a return type, even it is not allowed to have a **void** as a return value

```
class myClass
    int address;
    function new()
        address = 100;;
    endfunction
...
endclass

myClass obj1 = new; //obj1.data is 100
```

Explicit Constructor with Arguments

```
class myClass
    int address;
    function new(input int a)
        address = a;;
    endfunction
...
endclass

myClass obj1 = new (200); //obj1.data is initialized to 200
```

Complete Class Example

```
class frame;
  logic [4:0] addr;
  logic [7:0] payload;
  logic parity = 0;

  function new (input int add, dat);
    addr = add;
    payload = dat;
    genpar();
  endfunction

  function void genpar();
    parity = ^{addr, payload};
  endfunction

  function logic [13:0] getframe();
    return({addr, payload, parity});
  endfunction

endclass
```

```
logic [13:0] framedata;
frame one = new(3, 16);

initial begin
  ...
  @(negedge clk);
  framedata = one.getframe();
  ...
end
```

`this` Keyword

- The **`this`** keyword is used to unambiguously refer to class properties, value parameters, local value parameters, or methods of the current instance.
- **`this`** keyword shall be used within non-static class methods, constraints, etc.

```
class Demo;
  integer x;
  function new (integer x);
    this.x = x;
  endfunction
endclass
```

Outline

- Classes
- **Static Properties and Methods**
- Aggregate Classes
- Inheritance
- Polymorphism
- Randomization

Static Attributes

- The class attributes are dynamic by default, i.e., each class instance has its own copy of the attributes
- Regarding static attributes, one copy of the attribute is shared among all class objects
- They are allocated in memory at elaboration
- They can be accessed using **`null`** handles

Static Attributes: Example

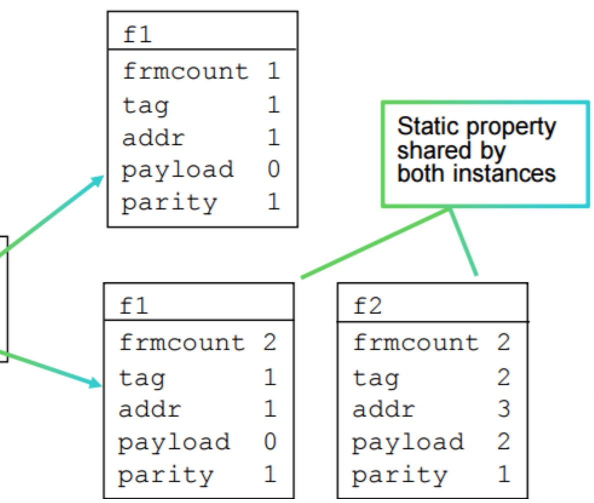
```
class frame;
  static int frmcount;
  int tag;
  logic [4:0] addr;
  logic [7:0] payload;
  logic parity;

  function new(input int add, dat);
    addr = add;
    payload = dat;
    genpar();
    frmcount++;
    tag = frmcount;
  endfunction

...
endclass
```

```
frame f1 = new(1, 0);

frame f2 = new(3, 2);
```



Static Methods

- Can only access static attributes or other static methods
- Besides any class handle, they can be called from the class name using the resolution operator **::**

Static Methods Example

```
class frame;
  static int frmcount;
  int tag;
  logic [4:0] addr;
  logic [7:0] payload;
  logic parity;

  function new(input int add, dat);
    addr = add;
    payload = dat;
    genpar();
    frmcount++;
    tag = frmcount;
  endfunction

  static function int getcount();
    return (frmcount);
  endfunction
...
endclass
```

```
frame f1, f2;
int frames;
initial begin
  frames = frame::getcount(); // 0
  frames = f2.getcount();     // 0

  f1 = new(3,4);
  f2 = new(5,6);
  frames = f2.getcount();     // 2
end
```

Resolution operator access

Handle access

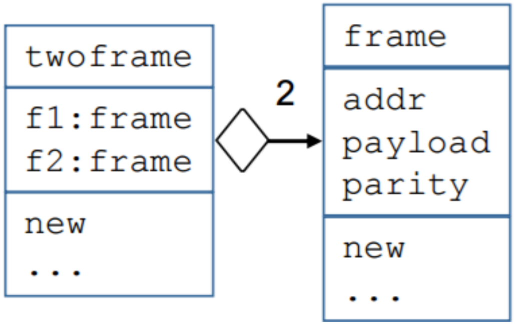
Outline

- Classes
- Static Properties and Methods
- **Aggregate Classes**
- Inheritance
- Polymorphism
- Randomization

Aggregate Classes

- A class attribute can be an instance of another class
- The constructors of the class attributes must be called explicitly.
- Instance handles must be chained to reach into hierarchy

Aggregate Class Example




```
class frame;
  logic [4:0] addr;
  logic [7:0] payload;
  bit parity;

  function new(input int add, dat);
    addr = add;
    payload = dat;
    genpar();
    ...
  endfunction
...
endclass
```

```
class twoframe;
  frame f1;
  frame f2;

  function new(input int basea, d1, d2);
    f1 = new(basea, d1);
    f2 = new(basea+1, d2);
  endfunction
...
```

```
twoframe tf1 = new(2,3,4);
initial begin
  tf1.f2.addr = 4;
  $display("base %h", tf1.f1.addr);
...
```

STUDENTS-HUB.com

Outline

- Classes
- Static Properties and Methods
- Aggregate Classes
- **Inheritance**
- Polymorphism
- Randomization

Inheritance (1)

- A class extends another class using the keyword **`extends`**.
- Only single inheritance is allowed, i.e., each subclass has only one parent
- The subclass inherits all the members of the parent class
  - It can add more members
  - It can re-declare (override) parent members
- Parent members are accessed as if they were members of the subclass
- The parent's constructor is automatically called by the subclass constructor
  - As the first line of the of the subclass's constructor

Inheritance (2)

Base Class

- base functions
- base data

Derived Class

- base functions
- base data
- more functions
- more data

```
class Base
   ... // base functions
   ... // base data
endclass
```

```
class Derived extends Base
   ... // more functions
   ... // more data
endclass
```

Inheritance (3)

- **`Super`** keyword allows the subclass to access the parent members
- You can only pass arguments one level at a time. **`super.super.new()`** is NOT allowed

Simple Inheritance Example

parent

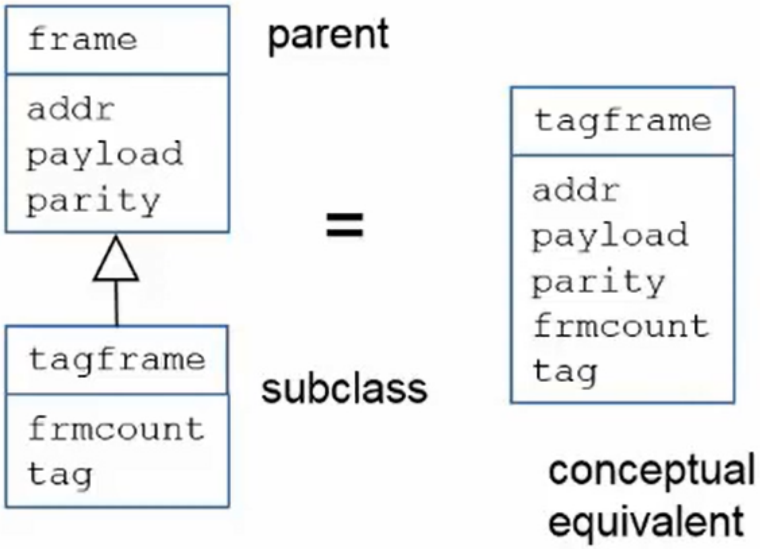
subclass

```
class frame;
  logic [4:0] addr;
  logic [7:0] payload;
  bit parity;
  ...
endclass

class tagframe extends frame;
  static int frmcount;
  int tag;
  ...
endclass

tagframe one = new();

initial begin
  one.addr = 0;
  ...
```



Inheritance with Constructors Example

parent

```
class frame;
  logic [4:0] addr;
  logic [7:0] payload;
  bit parity;

  function new(input int add, dat);
    addr = add;
    payload = dat;
    genpar();
  endfunction
...
endclass
```

subclass

```
class badtagframe extends frame;
  ...
  function new();
    super.new();
    frmcount++;
    tag = frmcount;
  endfunction
...
```

Automatically inserted

Error

subclass

```
class goodtagframe extends frame;
  ...
  function new(input int add, dat);
    super.new(add, dat);
    frmcount++;
    tag = frmcount;
  endfunction
...
```

First line explicit call overwrites implicit call

Polymorphism (I)

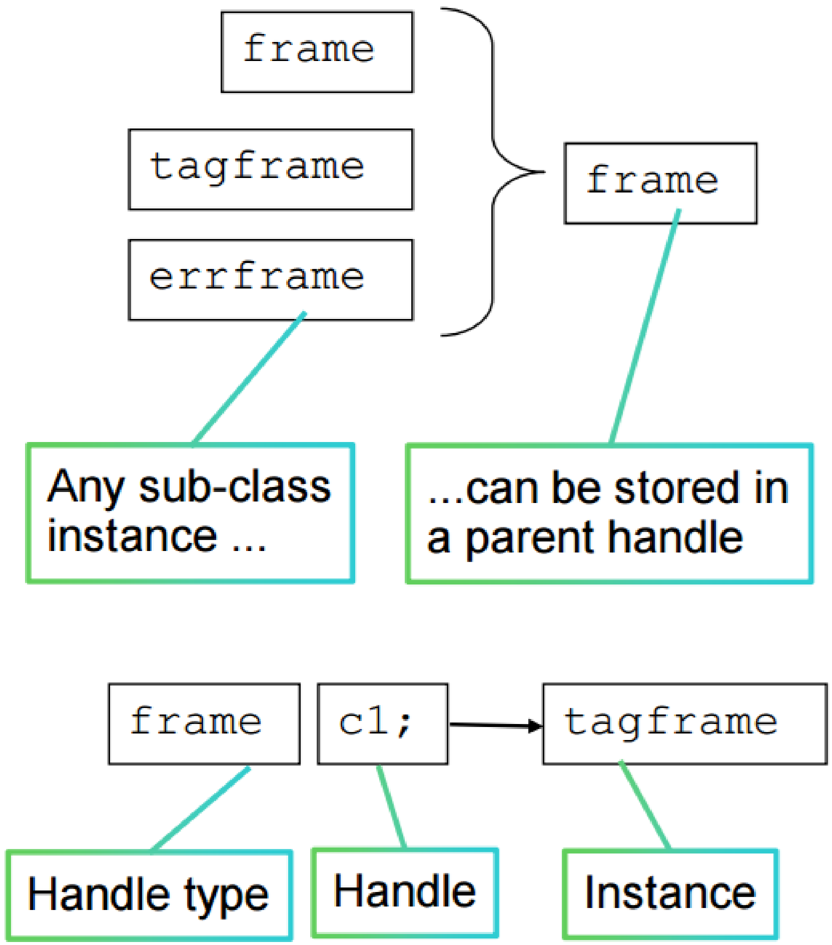
- Polymorphism allows the use of a variable of the superclass (even if it is an abstract class) type to hold subclass objects and to reference the methods of those subclasses directly from the superclass variable (handle).
- **`packets[1].send();`** shall invoke the **`send`** method associated with the **`TokenPacket`** class

```
BasePacket packets[100];

EtherPacket ep = new; // extends BasePacket
TokenPacket tp = new; // extends BasePacket

packets[0] = ep;
packets[1] = tp;
```

Polymorphism (II)



Virtual Methods

- A method of a class may be identified with the keyword **`virtual`**
- A virtual method shall override a method in all of its base classes
- A non-virtual method shall only override a method in that class and its descendants
- Virtual method overrides in subclasses shall have the same prototype of the function in the superclass including the arguments names.
- However, the return type of a virtual function shall be either a matching type or a derived class type

Virtual Methods Example

```
class BasePacket;
   int A = 1;
   int B = 2;
   function void printA;
      $display("BasePacket::A is %d", A);
   endfunction : printA
   virtual function void printB;
      $display("BasePacket::B is %d", B);
   endfunction : printB
endclass : BasePacket
```

```
class My_Packet extends BasePacket;
   int A = 3;
   int B = 4;
   function void printA;
      $display("My_Packet::A is %d", A);
   endfunction: printA
   virtual function void printB;
      $display("My_Packet::B is %d", B);
   endfunction : printB
endclass : My_Packet
```

```
BasePacket P1 = new;
My_Packet P2 = new;

 initial begin
    P1.printA;
    P1.printB;
    P1 = P2;
    P1.printA;
    P1.printB;
    P2.printA;
    P2.printB;
 end
```

Abstract Classes & Pure Virtual Methods (1)

- A base class may be characterized as being abstract by identifying it with the keyword **`virtual`**
- We cannot construct objects directly from an abstract class
- The abstract class constructor may only be called indirectly through the chaining of constructor calls originating in an extended non-abstract object.
- A pure virtual is a method in an abstract class that is declared as a prototype only without providing an implementation
- The pure virtual method shall be indicated with the keyword **`pure`** together without an implementation
- An extended subclass may provide an implementation by overriding the pure virtual method with a virtual method having a method body.

Abstract Classes & Pure Virtual Methods (2)

- Abstract classes may be extended to further abstract classes
- **But,** all pure virtual methods shall have overridden implementations in order to be extended by a non-abstract class
- Any class may be extended into an abstract class, and may provide additional or overridden pure virtual methods.

Abstract Class Example

```
virtual class BasePacket;
   pure virtual function integer send(bit[31:0] data); // No implementation
endclass

class EtherPacket extends BasePacket;
   virtual function integer send(bit[31:0] data);
      // body of the function
      ...
   endfunction
endclass
```

Outline

- Classes
- Static Properties and Methods
- Aggregate Classes
- Inheritance
- Polymorphism
- **Randomization**

Randomization

- Class attributes can be defined as random using **rand** and **randc**
- **rand**: random with uniform distribution
- **randc** : random-cyclic randomly iterates through all values without repetition
  - When an iteration is complete, a new random iteration automatically starts

```
class randclass;
  rand  bit[1:0] p1;
  randc bit[1:0] p2;
endclass
```

Each value has an equal probability (1/4)

`p1 rand` example output: `01 11 00 10 01 11 01 10 11 00 01 11`

Close repetition is common

`p2 randc` example output: `01 11 00 10 | 00 11 10 01 | 10 00 11 10`

Each iteration cycles through all values without repetition

Close repetition across iterations is possible

`randomize()` Function

- `randomize()` function randomizes the object's random attributes.
- Every class has a built-in `randomize()` virtual method.
- You cannot re-declare this method
- It returns **1** on success, **0** otherwise.

```
class randclass;
  rand  bit[1:0] p1;
  randc bit[1:0] p2;
endclass
randclass myrand = new();
int ok;
 initial begin
  ok = myrand.randomize();
  if ( !myrand.randomize() )
     $display ("myrand randomize failure");
end
...
```

p1 is a random variable
p2 is a random-cyclic variable

Randomizes all random variables in a class instance

Can check return value

`pre_randomize()` and `post_randomize()`

- `randomize()` automatically calls two "callback" functions:
  - `pre_randomize()` before randomization.
  - `post_randomize()` after successful randomization.
- If defined, these methods are automatically called on randomization.
- The `pre/post_randomize` declarations must match the prototypes shown, i.e., they must be void functions with no arguments.

```
function void pre_randomize();
   ...
endfunction
```

```
function void post_randomize();
   ...
endfunction
```

`pre_randomize()` and `post_randomize()` Example

```
class randclass;
  rand  bit[1:0] p1;
  randc bit[1:0] p2;
        bit[1:0] parity;
  function void post_randomize();
    parity = p1 ^ p2;
  endfunction
endclass

randclass myrand = new();
int ok;

initial begin
  ok =  myrand.randomize();
...
```

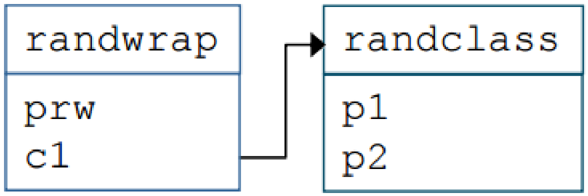
Define `post_randomize`

`randomize` automatically calls `post_randomize`

Randomization in Aggregate Classes

- Randomize can operate hierarchically on aggregate classes
  - The class instance property must be declared as rand.
  - Otherwise, that instance is skipped for randomization.

Randomization in Aggregate Classes: Example



c1 must be rand for its p1 and p2 properties to be randomized

```
class randclass;
  rand  bit[1:0] p1;
  randc bit[1:0] p2;
endclass

class randwrap;
  rand int prw;

  rand randclass c1;

  function new();
    c1 = new();
  endfunction
endclass

randwrap mywrap = new();
int ok;

initial begin
  ok =  mywrap.randomize();
...
```

STUDENTS-HUB.com

In-Line Random Variable Control with `randomize()`

- Specific class variables can be randomized by passing them as arguments.
  - This allows nonrandom (state) properties to be randomized (not cyclic)

```
class randclass;
  rand  bit [0:1] p1;
  randc bit [1:0] p2;
        bit [1:0] s1, s2;
endclass

randclass myrand = new;
int ok;

initial begin
  ok = myrand.randomize();
  ok = myrand.randomize(p1);
  ok = myrand.randomize(p1,s1);
  ok = myrand.randomize(s1,s2);
end
```

`p1, p2` are random properties

`s1, s2` are state properties

randomize `p1` and `p2` only;
`s1` and `s2` not randomized

randomize `p1` only;
others unchanged

randomize `p1` and `s1` only (others unchanged)

randomize `s1` and `s2` only (others unchanged)

Controlling Randomization: `rand_mode()`

- Every random attribute has an enable switch **`rand_mode`**
  - Enabled by default (1)
  - If disabled (0), the attribute will not be randomized
- Mode can be written with task **`rand_mode`** and read with function **`rand_mode`**
- Called off a random property, the task changes the mode of that property.
- Called off an instance, the task changes the mode or all random properties of the instance.
- Only random attributes have **`rand_mode`**
  - Calling method off a non-random attribute generates a compile error

```
function int rand_mode();
```

```
task rand_mode(bit on_off);
```

rand_mode() Example

```
class randclass;
  rand  bit[1:0] p1;
  randc bit[1:0] p2;
        bit[1:0] s1, s2;
endclass

randclass myrand = new;

int ok, state;
initial begin

  myrand.rand_mode(0);
  myrand.p2.rand_mode(1);

  state = myrand.p2.rand_mode();

  ok    = myrand.randomize();

  state = myrand.s1.rand_mode();
end
```

p1, p2 are random properties

s1, s2 are state properties

Disable randomization of all random variables of myrand

Re-enable p2 randomization

Read p2 mode (1)

Only p2 randomized

Error – s1 is not a random variable

Constraint Blocks

- Constraints restrict the random data generation to exclude values or change the probability distribution

- Constraints can be embedded in classes using constraint blocks

- You declare a constraint block as a class member with the **`constraint`** keyword, followed by an identifier, followed by a list of constraint items enclosed within curly braces {}

- A block can contain any number of any form of constraints

Constraint Blocks Example

```
class randclass;
  rand bit [1:0] p1;
  rand bit [7:0] p3;

  constraint c1 { p1 != 2'b00; }
  constraint c3 { p3 >= 64; p3 <= 192; }
endclass

randclass myrand = new;

int ok;
initial begin
  ok = myrand.randomize();
  ...
end
```

Constrains `p1` to the values `01,10,11`

; after each constraint expression but not after constraint block

Upper and lower bound constraints on `p3`

Randomize `p1` using constraint block `c1`, and `p3` using constraint block `c3`

Constraint Block Inheritance

- Constraint blocks are class members and are inherited just like any other members

```
class randclass;
  rand bit  [1:0] p1;
  constraint not0 {p1 != 2'b00;}
endclass

class rcx1 extends randclass;
  constraint not3 {p1 != 2'b11;}
endclass

class rcx2 extends randclass;
  constraint not0 {p1 != 2'b01;}
endclass

class rcx3 extends randclass;
  constraint not0 {}
  constraint not1 {p1 != 2'b01;}
endclass
...
rcx1 myrand = new;
initial begin
  ok = myrand.randomize();
  ...
```

`not0` constraint in base class

`rcx1` adds constraint `not3` to constraint `not0` from `randclass`

`rcx2` overrides constraint `not0`

`rcx3` removes constraint `not0` and defines new constraint `not1`

`p1` is `01` or `10`

inside Operator

- The **inside** operator is particularly useful in constraint expressions.
  - The operator can also be negated to generate a value outside of a set.

```
class randclass;
  rand bit[7:0] p3;
  constraint c1 {p3 inside {3, 7, [11:20]};}
endclass

randclass myrand = new;

int ok;
initial begin
  ok = myrand.randomize();
  ...
end
```

c1 constrains p3 to the set 3,7,11-20

c2 constrains p3 to outside the set 1,7,10-255

```
class not_inside;
  rand bit[7:0] p3;
  constraint c2 { !( p3 inside {1, 7, [10:255]} ); }
endclass
```

Weight Distributions

- You can change distribution by defining weights for values using the operator `dist` operator .
  - Default value is 1
- `:=` operator assigns weight to the item or every value in the range
- `:/` operator divides the weight by the number of values in the range, i.e., for a range of n values and a weight of w, the weight of each individual value is w/n.
- Fractional weights are possible
- Negative weights are not possible

Weight Distributions Examples

```
constraint c1 { p4 dist { [101:200]:=200 }; }
```

101 to 200 each gets a weight of 200

```
constraint c2 { p4 dist { [101:200]:/200 }; }
```

101 to 200 each gets a weight of 2 (200/100=2)

```
class randclass;
  rand int p4;
  constraint c3 {p4 dist {7:=5, [11:20]:=3, [26:30]:/1};}
endclass
```

7 has a weight of 5

11-20 each has a weight of 3

26-30 each has a weight of 1/5

Conditional Constraints

- Implication, using `->` operator
- `if … else`

```
class randclass1;
  rand int p3;
  bit mode;
  constraint c1
  {
    mode == 1 -> p3 < 100;
    mode == 0 -> p3 > 10000;
  }
endclass
```

```
class randclass2;
  rand int p4;
  bit mode;
  constraint c2
  {
    if (mode == 1)
      p4 < 100;
    else
      p4 > 10000;
  }
endclass
```

`else` is optional

Iterative Constraints

- You can use a loop to apply separate constraints to each array element

- Can affect performance for large arrays or complex constraints.

```
class randclass2;

  rand logic [3:0] arr[7:0];

  constraint c1 { foreach (arr[i])
                    (i <= 4) -> arr[i] <= i; }

  constraint c2 { foreach (arr[i])
                    (i > 4) -> arr[i] >= i; }
endclass
```

Iterative constraints

Constraints

```
arr[0] <= 0;
arr[1] <= 1;
arr[2] <= 2;
arr[3] <= 3;
arr[4] <= 4;
arr[5] >= 5;
arr[6] >= 6;
arr[7] >= 7;
```

`Constraint_mode()`

- Every constraint block has an enable switch called **`constraint_mode`**.
  - Enabled by default (1).
  - If disabled (0), the constraint block will not be used.
- Mode can be written with task **`constraint_mode`**.

```
task constraint_mode(bit on_off);
```

- Mode can be read with function **`constraint_mode`**.

```
function int constraint_mode();
```

- Only constraint blocks have **`constraint_mode`**.

Constraint_mode() Example

```
class randclass;
  rand bit [1:0] p1;
  constraint blue  {p1 != 2'b00;}
  constraint green {p1 != 2'b11;}
endclass
 randclass myrand = new;

int state, ok;
initial begin

  myrand.constraint_mode(0);
  myrand.blue.constraint_mode(1);
  state = myrand.green.constraint_mode();
  ok = myrand.randomize();

end
```

`blue` constrains `p1` to not `00`

`green` constrains `p1` to not `11`

Disable all constraints for `myrand`

Re-enable `blue` constraint

Read `green` mode (0)

`green` constraint disabled: `p1` will be `01`, `10`, `11`

Randomization Procedure and Its Effects

- Randomization proceeds as follows:
  - All **`randc`** properties randomized simultaneously.
  - Then all **`rand`** properties randomized simultaneously.
  - Then constraints are checked.
  - Cycle iterates until a solution is found or the random space is exhausted.
- You can only order **`rand`** variables of integral types.

```
class randclass;
  rand bit[2:0] vect;
  rand bit mode;
  constraint c1
    { mode -> vect == 0; }
endclass
```

Applying `solve…before`

```
class randclass_ordered;
  rand bit[2:0] vect;
  rand bit mode;
  constraint c2
    { mode -> vect == 0;
      solve mode before vect; }
endclass
```

Expectation: `mode` to be '`0`' one half the time

Result: `mode` is '`0`' one ninth of the time

Ordering constraint