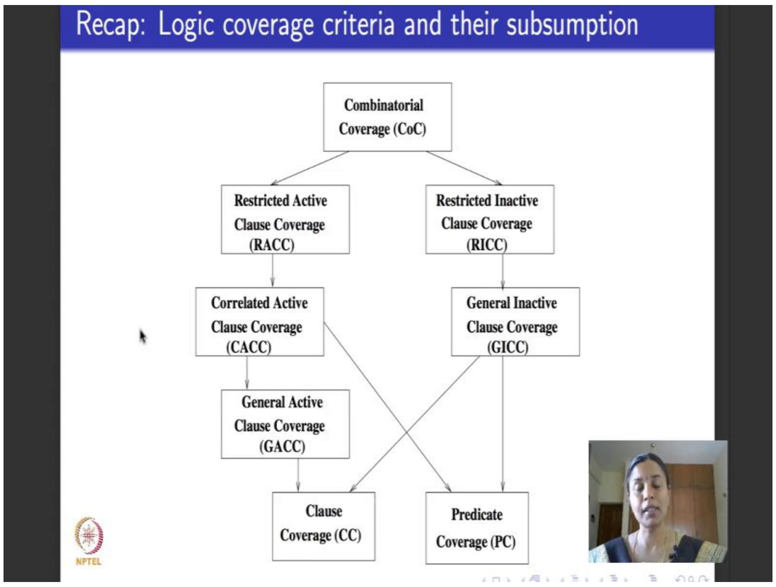
Software Testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture - 28
Logic Coverage Criteria: Applied to test specifications

Hello everyone. We are in week 6 and today I am going to do the fourth lecture of week 6. As always we will continue with logic. What I will do today after we saw 2 lectures on how to apply logic coverage criteria for source code, for a change we will look at specifications, we will see what is the role that logical plays us for a specification is concerned.

(Refer Slide Time: 00:39)

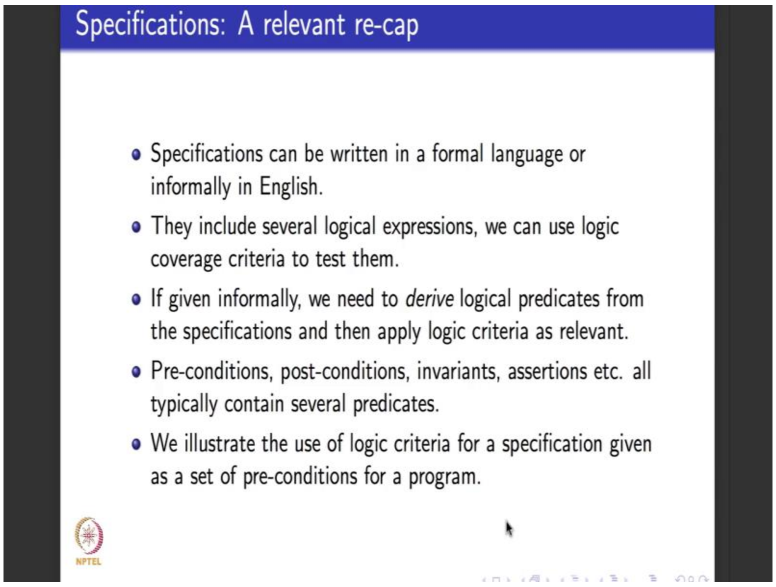


Recap: Logic coverage criteria and their subsumption

$$\begin{tikzcd}
 & \text{Combinatorial Coverage (CoC)} \arrow[dl] \arrow[dr] & \\
\text{Restricted Active Clause Coverage (RACC)} \arrow[d] & & \text{Restricted Inactive Clause Coverage (RICC)} \arrow[d] \\
\text{Correlated Active Clause Coverage (CACC)} \arrow[d] \arrow[drr] & & \text{General Inactive Clause Coverage (GICC)} \arrow[dl] \arrow[d] \\
\text{General Active Clause Coverage (GACC)} \arrow[d] & & \\
\text{Clause Coverage (CC)} & & \text{Predicate Coverage (PC)}
\end{tikzcd}$$

And we will see how to apply the coverage criteria that we learnt over specifications. So, this is same slide I have been showing for the past few lectures just to help you recap logic coverage criteria. At the bases we have clause and predicate coverage, the whole predicate becomes true once, false once and predicate coverage. Each clause is made true once, false once and predicate coverage. Combinatorial coverage tests for the entire truth table. Then we have active clause coverage criteria, three of them. The most popular is correlated active clause coverage here in the center then we have inactive clause coverage criteria 2 of them, and when a predicate has only one clause all these coverage criteria basically boil down to just predicate coverage criteria.

(Refer Slide Time: 01:29)



Specifications: A relevant re-cap

- Specifications can be written in a formal language or informally in English.
- They include several logical expressions, we can use logic coverage criteria to test them.
- If given informally, we need to *derive* logical predicates from the specifications and then apply logic criteria as relevant.
- Pre-conditions, post-conditions, invariants, assertions etc. all typically contain several predicates.
- We illustrate the use of logic criteria for a specification given as a set of pre-conditions for a program.

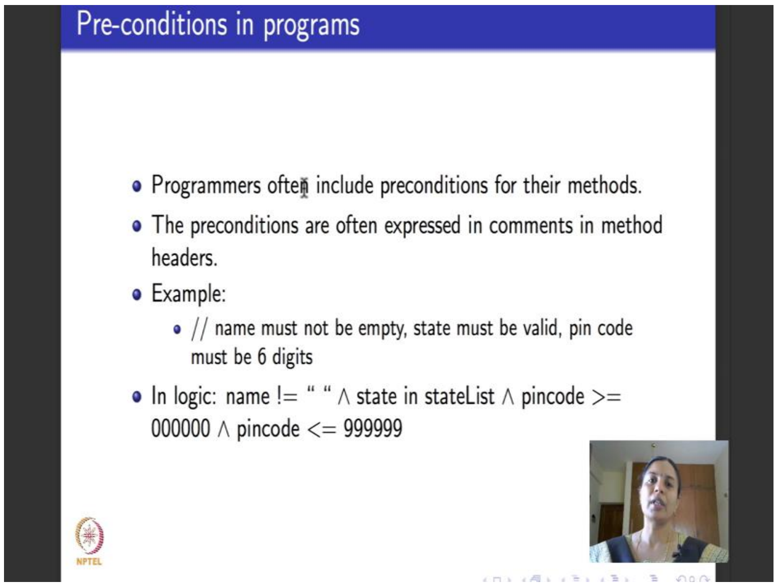
What we will do today is we look at specifications of requirements? It could be a part of requirement specification document, it could be a part of a design or an architecture document. We will see where do logic come into play in these kind of documents and how to apply the logical coverage criteria that we have learnt to generate test cases for specifications. So, typically where do specifications stay? You might have heard documents like software requirements specification or system requirement specifications they usually abbreviated as (SRS). Similarly you have functional requirements specifications (FRS) hardware requirement specifications (HRS). So, and these are always documents that are typically written in English. Somebody will write saying the server will respond to the client request within three milliseconds, the server will acknowledge every request by a client and so on. Or people use semi formal notations like use cases and so on too write these requirements.

Typically lot of these requirements include logical specifications like for example, they might write a specification which says that if the value of pin code is entered then it must be exactly a 6 digit number. So, this can be thought of as a formula or a condition that can be written in logic. Basically specifications involve conditions which talk about if this happens, this will happen, for this to happen something else must have happened. So, these conditions are basically nothing but logical predicates and if they are not given in the formal language of logic or in programming notation as testers is our duty to be able to read the English specification and derive logical predicates from them.

So, in today's example, we will look at English specifications and see how logical predicates can be derived from them and where do these kind of specifications come? These kind of specifications come typically as pre conditions, post conditions, invariants. Invariants, for example, it could be a property which says that at no point in time will if there is a critical region in a particular database or access to a database and in variant could say things like at no point in time 2 different processes have access to the critical regions. So, this can be thought of as an invariant describing the safety property of mutual exclusion.

Similarly, very common for programmers to write assertions to be able to debug their design or code. So you can find specifications there also. What we will do is we will take one of these examples, we will take either preconditions or post conditions. In this lecture I will take precondition as an example and I will show you how logic specifications are used to write these pre conditions and the coverage criteria that we learnt, how can we used to test that these pre conditions indeed hold.

(Refer Slide Time: 04:19)



Pre-conditions in programs

- Programmers often include preconditions for their methods.
- The preconditions are often expressed in comments in method headers.
- Example:
    - // name must not be empty, state must be valid, pin code must be 6 digits
- In logic: name != " " ∧ state in stateList ∧ pincode >= 000000 ∧ pincode <= 999999

So, where do preconditions occur in programs? Typically programmers include pre conditions for their methods in Java or if you writing in C you include pre conditions for your functions for the entire program before you begin the text of the program. Preconditions could be loosely documented they could just mean comments say you do this you do that and its assumed that the programmer would have written code that will take care of the input satisfying these preconditions. These specific conditions on what the input look like, these specify conditions on what they output should be and so on. So, here is a small example. This is an example of a precondition. Lets say we are working with the program that deals with some kind of addresses. So, it says that in the address assuming that it is a record containing fields like name, street name, house number locality, state and pin code, you might want certain preconditions saying that the name should not be an empty stream, nobody should enter in address where the name is empty.

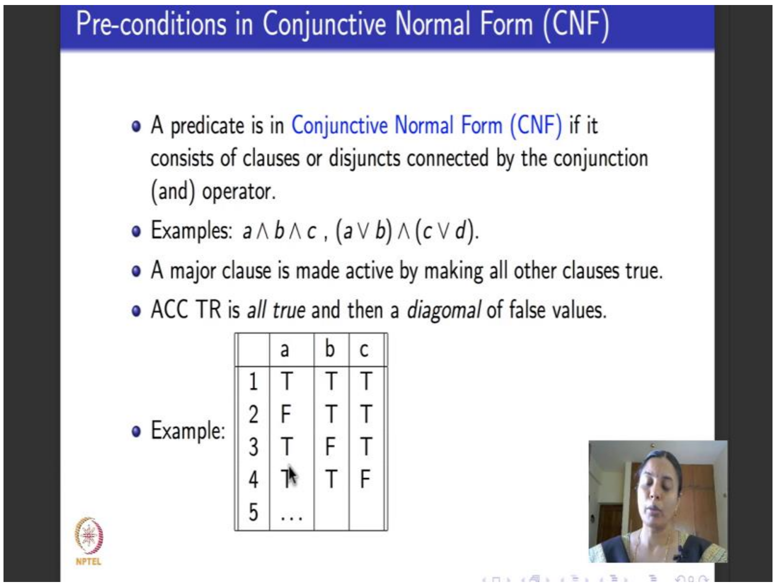
So, that is the first comment. The second comment says state must be valid. Like for example, if it is India then you cannot write any garbage name that you think as a state it must be one of this so many states that we have in India, it must be one of the valid states and lets say for an address the containing a pin-code in India, pin code cannot have a arbitrary number of digits pin code cannot just be written as a one digit number or as a 5 digit number, it should always be a 6 digit number. So, in any program that deals with a database that contains address, you might want to validate the data base by using the these three as example preconditions. They say name is not empty, state must be one of the predefined state because otherwise people might make spelling mistakes while entering.

So, you typically give a drop down menu where you let them choose the states, or pin and pin code must be exactly having 6 digits. So, typically they leave it at this. Somewhere in the code you will find parts that checks for these kind of things and give an exception when these things are not satisfied. But my goal is to be able to specify these preconditions as logical predicates and see if we can test them using the coverage criteria that we have learnt.

So, what do I write? I write it like this in logic I say name, as a variable name, should not be equal to, read this exclamation mark equal to as not equal to an empty string and state must be one in a list of states which I keep let us say in a list called state lists and pin code as a number should be greater than or equal to 0, as 6 different digits. You could even view it as a string because in some programming languages that could be treated as a number 0.

So, you will view it as a string containing 000000 and this order will be the lexicographic order and pin code should not be greater than or equal to this 999999. So, it is a number that has exactly 6 digits or the string that has exactly 6 digits where the preceding 0s do matter.

(Refer Slide Time: 07:21)



Pre-conditions in Conjunctive Normal Form (CNF)

- A predicate is in Conjunctive Normal Form (CNF) if it consists of clauses or disjuncts connected by the conjunction (and) operator.
- Examples: $a \wedge b \wedge c$, $(a \vee b) \wedge (c \vee d)$.
- A major clause is made active by making all other clauses true.
- ACC TR is *all true* and then a *diagomal* of false values.
- Example:

	a	b	c
1	T	T	T
2	F	T	T
3	T	F	T
4	T	T	F
5	...		

When we talk about pre conditions in specifications, if you say this example that we saw what is the connecting Λ operator in this precondition? If I view this as a predicate, I can say that this predicate has four clauses. This is the first clause which says name as non empty, this is the second clause which says state should be in the list of states, 3rd clause is pin code is greater or equal to this number fourth clause says pin code is less than or equal to this number. And these four clauses are connected by one operator which is the Λ operator. Typically you will realize that most of these pre conditions, post conditions typically are always connected by an Λ operator or by a V operator.

So, special kinds of predicates, there are normal forms which describe these kinds of predicates. So, a predicate is said to be in conjunctive normal form, the precondition that we saw here is in conjunctive normal form if it consists of clauses that are connected by an outermost conjunction operator or an Λ operator.

In conjunctive normal form. these clauses could be disjuncts themselves. In the sense that one particular clause within itself could contain V operator. Of course, for going back out condition of logic coverage criteria we would distinguish its 2 different clauses, because for us a clause cannot contain an V operator, but in conjunctive normal form that is allowed. So, here are 2 examples of formulae and conjunctive normal form. The first one a Λ b Λ c has three a disjuncts or clauses as we call it in this course: a, b and c and the outermost operator that connects these three disjuncts is an Λ operator.

In this case, the second example I have a V b Λ c V d for us it has four clauses, but if you see the outermost operator is still an Λ operator. Inside, the clauses could be connected by an V. So, if the clauses are connected by or this entire thing, a V b in the jargon of logic, is what is called a disjunct. In the sense that these clauses individual clauses inside could be connected by a disjunction operator or an V operator, but the outermost operator connecting them is an Λ. We also implicitly assume that inside there are no Λs.

For example, I should not have a Λ b Λ c Λ d. If I have that then I say a is one clause disjunct and b is another disjunct I separate it out. The outer most operator in a conjunctive normal form should be an Λ. So, it is a conjunct of disjuncts, that is how you call a conjunctive normal form formula as. Now, remember we want to be able to work with logic coverage criteria. Always it is easy to do predicate coverage and clause coverage, but to be able to do active clause coverage, if a predicate is in conjunctive normal form its fairly easy. What you will have to do is the following. Major clause that you want to focus and make the make it determine the predicate you make that active by making all other clauses true.

Let us take this example a Λ b Λ c. If suppose I want to make a as my major clause and I want a to determine the predicate p. So, I will make b Λ c completely true right, which are the minor clauses I will make them true. If b and c are true then the truth or falsity of the predicate is completely influenced by a, because when a becomes true, the predicate will be true and a becomes false the predicate will be false. b and c are true anyway so, they will not influence the predicate at all.

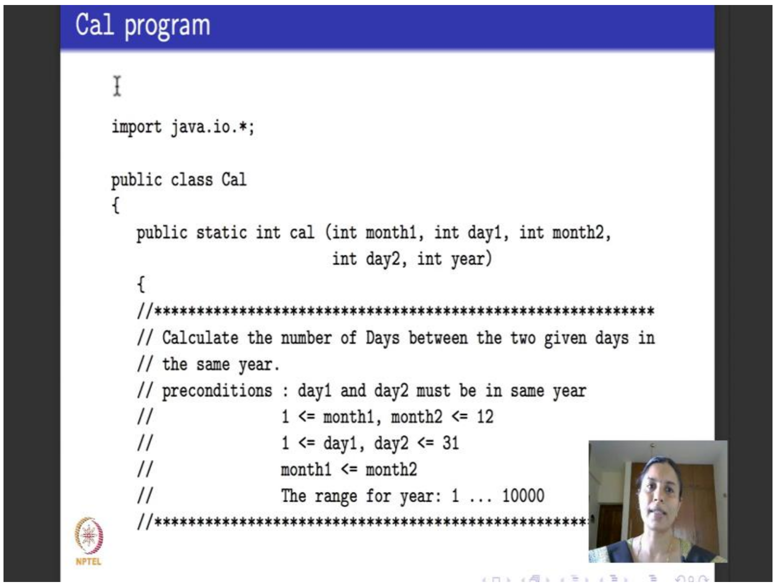
So, that is what this point. Here says it says major clause is made active, means active means made it made to determine the predicate by making all other clauses true. Then what is true truth what is a table containing the test requirements for active clause coverage look like? It look like this: it will have a row of all true’s and then it will have a diagonal of false values. So, what do we mean by diagonal of false values? In this case I am making the major clause. So, a is true once false once the rest of the minor clauses are all true with that is what I wanted to be for a to be determining p. In this case I am making the major clause b determine p. So, b is false once true once and the rest of the clauses will be true and so on. So, this is how ACC test requirement looks like when the predicate is given in conjunctive normal form. There is another popular normal form in logic called disjunctive normal form. It is the opposite of, somewhat the opposite of conjunctive normal form.

Here the outermost connective is an V. Inside various clauses could be connected by Λ its reverse the role of Λ and V are reversed in disjunctive normal form from conjunctive normal form. So, we say a predicate is in disjunctive normal form if it consists of clauses or conjuncts that are connected by a V operator. So, here are 2 examples: a V b V c there are three clauses connected by an V operator. So, we say this predicate is in disjunctive normal form. Here again there are four clauses: a Λ b is one clause, c Λ d is one clause. This I mean sorry a Λ b is one conjunct, c Λ d is one conjunct, together there are four clauses. These conjuncts inside can have Λs, but in disjunctive normal form the outermost operator is an V.

So, when do you say a formula is in disjunctive normal form? We say a formula is in disjunctive normal form if it is an V of Λs. So for this to satisfy various logic coverage criteria in particular to satisfy active coverage criteria what do you do? You make the major clause active or determine predicate by making all other minor clauses false. So, in this example suppose you make a, b and c both false then the truth value a will completely influence the truth or falsity of the predicate. If a is false predicate will be false, if a is true because b and c are false the predicate will become true just because of a. So, here again for active clause coverage the TR is quite easy. It has a row of all false and then a diagonal of true values. True values go along the diagonal by making each clause take true in turn. The rest of the table is typically filled with false values and you would realize that in this case it nicely happens that each clause takes turns to be the major clause and does determine the predicate.

So, these two are two standard normal forms in which many preconditions and post conditions are written and when we do logic coverage criteria for these preconditions and post conditions, we will apply these truth tables to be able to determine ACC test requirements for them.

(Refer Slide Time: 14:21)



Cal program

```
import java.io.*;

public class Cal
{
   public static int cal (int month1, int day1, int month2,
                          int day2, int year)
   {
   //***********************************************************
   // Calculate the number of Days between the two given days in
   // the same year.
   // preconditions : day1 and day2 must be in same year
   //                 1 <= month1, month2 <= 12
   //                 1 <= day1, day2 <= 31
   //                 month1 <= month2
   //                 The range for year: 1 ... 10000
   //*****************************************************
```

So, here is an example. This is an example of a calendar program, a simple program that calculates the number of days between the two given days in a year. Lets say one day is let us say second of February, the other day is let say 3rd of April. It says how many days are there between second February and 3rd April. The only condition is this the two dates 2nd February and 3rd April should be within this same year.

So, this program will obviously be manipulating data like a date right because it will ask you to enter 2 dates within the same year, and then it will calculate the number of days in between. Of course, it has to take smaller things like is the year a leap year. You know if it is a leap year then you know if February is included in the range of days to be calculated then it has to adjust the number of days and so on. So, this program has several preconditions. They are all listed to start with this comments here. So, it says day 1 and day 2 must be in the same year because that is what the program is meant to calculate if given across years this is not the program that will calculate.

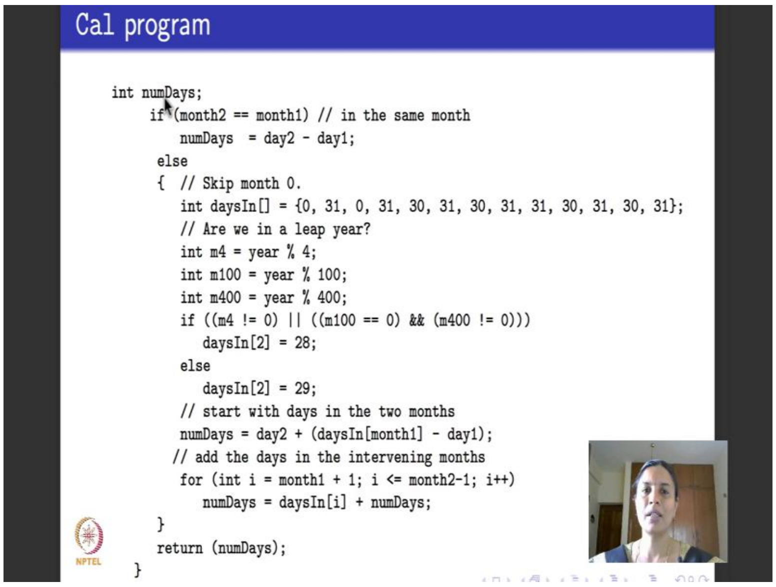
The second pre condition says that the number that you enter for month it should be a number between 1 and 12. Both month 1 and month 2, the two months that told you right February and April, both these months should be numbers between 1 and 12; and it says both the days should be between 1 and 31 and it says in addition, month 1 should be less than or equal to month 2. In the sense that I do not ask for the number of days from let say 25th of May to 28th of February because its within the same year the first date that I give should be before the second date that I give. So, a simple way of enforcing that is to say that I enter month 1 which is the part of date one and that should be less than or equal to month 2 which is a part of date 2. And then it says the range for year should be between one and ten thousand ,this is just another precondition.

Now, these are what are called preconditions. I hope it makes sense why they are needed because otherwise you could expect any kind of garbage as a input which we do not want. So, we write a whole set of pre conditions which says that please accept inputs only of these kinds. If the input is not of these kinds then you write appropriate exception handling statements which will make the program alert the user about the range or details being wrong. Now, one is to write these preconditions and the other is to check whether these preconditions themselves are complete. It is not very difficult to see that for this example that we have considered the pre conditions are not complete. In fact, they are reasonably inaccurate. Why is that so because here for example, it says for every day 1 as how is the date given date is given is day month and year and the condition for day 1 says that it is a number between 1 and 31.

So, let us say the month is a month of November we really do not have something like 31st November, where as these preconditions will let you enter something like 31st November, they do not specify that. Of course, in the program we will take care of the fact that does not happen, but these preconditions per se do not rule out that. Similarly, for a day in month like February, you can merrily go ahead and enter 29 February for a year that is not a leap year or for that matter you can enter 30th February and 31st February. These pre conditions let you do that. The other thing is the condition that we have asked is that both the days should be in the same year.

So, if this range for year is not really necessary data at all, but it is given just as a precautionary thing. On one side pre conditions are useful to rule out junk inputs, but on the other side preconditions can get very cumbersome if you have to get every detail right. So, pre conditions can be inaccurate, they need not be complete. So, this is how the Cal program starts.

(Refer Slide Time: 18:37)



Cal program

```
int numDays;
     if (month2 == month1) // in the same month
         numDays  = day2 - day1;
      else
      {  // Skip month 0.
         int daysIn[] = {0, 31, 0, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
         // Are we in a leap year?
         int m4 = year % 4;
         int m100 = year % 100;
         int m400 = year % 400;
         if ((m4 != 0) || ((m100 == 0) && (m400 != 0)))
            daysIn[2] = 28;
         else
            daysIn[2] = 29;
         // start with days in the two months
         numDays = day2 + (daysIn[month1] - day1);
        // add the days in the intervening months
         for (int i = month1 + 1; i <= month2-1; i++)
            numDays = daysIn[i] + numDays;
      }
      return (numDays);
   }
```

So, it takes input as month 1, day 1, month 2, day 2 and year. So, it says in this year this is the range from day 1 and month 1 to day 2 and month 2, you calculate the number of days. How does the program work? Here is how the program works it call this thing method called Cal, it has an internal variable called number of days which is the variable that is returned back to the main program. What it first says is the if it is in the same month if the 2 days are in the same month that is if months 2 is equal to month 1 the number of days is easy to calculate. You just do day 2 minus day 1 then you get the number of days. Otherwise what it tries to do is to first tries to populate the calendar of how many days are there in a month.

So, this is the calendar that it populates. So, it says month, skip month 0 means skip this entry 0, then goes on as January, February, March, April, May, June, July and so on. So, January has 31 days. For February it is just initially put it a 0, based on a whether it is a leap year or not we will populate it as 28 or 29. March has 31 days, April 30 days, may 31 days, June 30 days, and so, on till December.

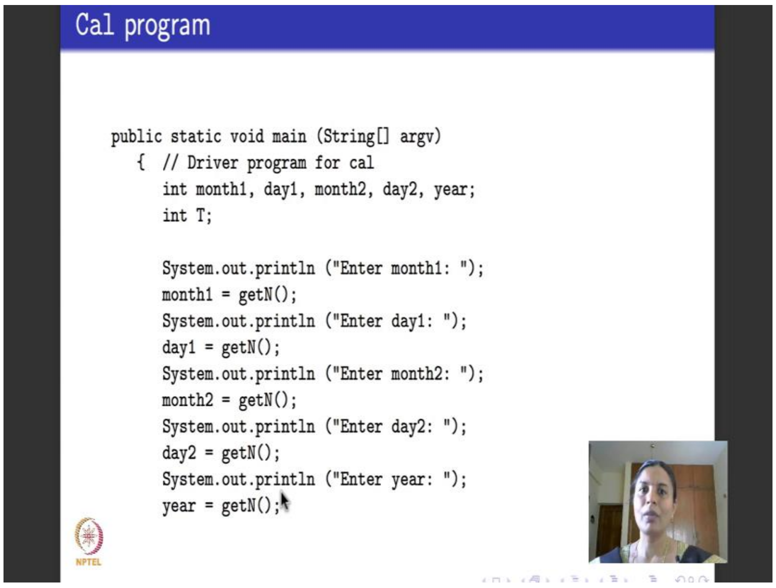
Now, the next piece that the code contains is to populate the date for February. So, it says, first checks whether are we in a leap year. So, year is already entered here as input, year is already entered here as input. So, it tries to divide a year by 4 by 100 and then by 400 and it computes the standard check for leap year right. If m does not, m 4 which is the year divided by 4 is not equal to 0 or m 100 which is the year divided by 100 equal to

0 and m 100 is not equal to 0 then it is not a leap year. So, the February which is this second; 0, 1, 2 the second the 3rd entry, but the second index entry in the list day days in is set to 28. If this condition is violated then it is indeed a leap year. So, you set the number of days in that month in the entry for February to be 29. So, after this you have populated this full array with the correct days for each month.

Now, what you do is you calculate the number of days in between the 2 days that was entered. First you calculate the days within the two months. Start with the days within the two months. So, you do number of days is day 2 plus days in month minus day 1, to this you add the days in the inter meaning intervening months right. So, let say I am asked to find out the number of days from second February to let say 25th May, I first start with the days in February and days in the month of may then add to that I add the number of days that I have in march and April that is what this segment of code is doing and for adding the number of days in the intervening month I start from the months after the month that I began and end in the month before the month that I ended and keep adding. Once you finally finish this you return number of days which tells you the number of days that exists in a month.

So, this is the total program for calculating the number of days in a month. This program had all these preconditions. These preconditions have to be brought into the program.

(Refer Slide Time: 21:41)

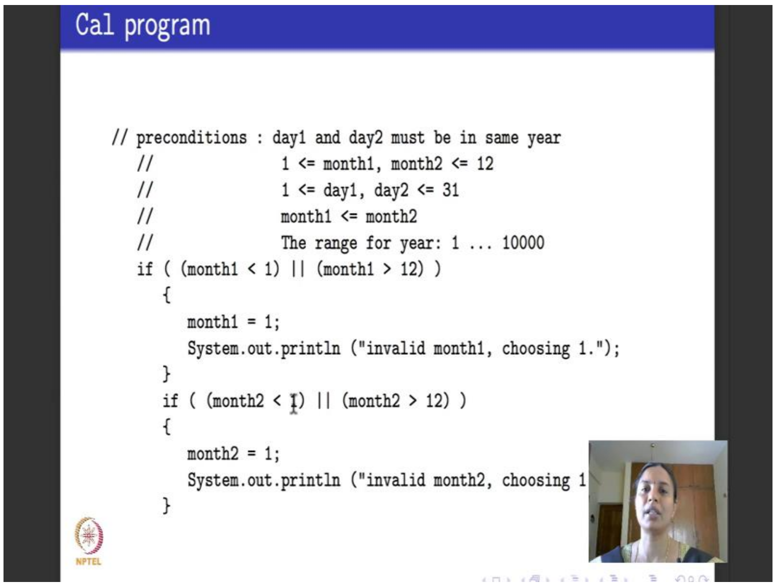


Cal program

```
public static void main (String[] argv)
   {  // Driver program for cal
      int month1, day1, month2, day2, year;
      int T;

      System.out.println ("Enter month1: ");
      month1 = getN();
      System.out.println ("Enter day1: ");
      day1 = getN();
      System.out.println ("Enter month2: ");
      month2 = getN();
      System.out.println ("Enter day2: ");
      day2 = getN();
      System.out.println ("Enter year: ");
      year = getN();
```

So, we will continue writing the code. We will write a main program that takes as all the inputs and calls the program for Cal. So, this is a driver program that calls this method Cal. So, what does the main program do? It initializes these things, it initializes a variable called t, then it gets the data: enter month1, day1, month 2, day2 and year, it gets all this data.

(Refer Slide Time: 22:05)



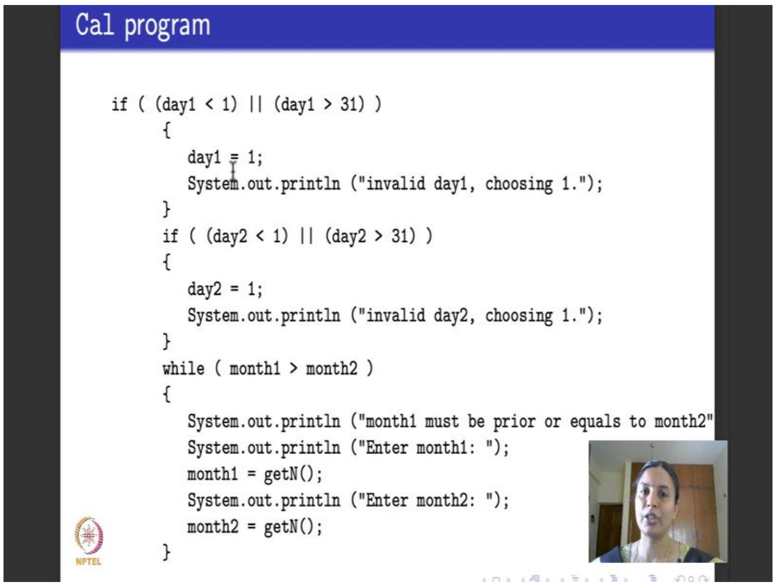
Cal program

```
// preconditions : day1 and day2 must be in same year
   //              1 <= month1, month2 <= 12
   //              1 <= day1, day2 <= 31
   //              month1 <= month2
   //              The range for year: 1 ... 10000
   if ( (month1 < 1) || (month1 > 12) )
      {
         month1 = 1;
         System.out.println ("invalid month1, choosing 1.");
      }
      if ( (month2 < 1) || (month2 > 12) )
      {
         month2 = 1;
         System.out.println ("invalid month2, choosing 1
      }
```

Now it has checks for these preconditions. I have just repeated the preconditions here just so that it is easy to refer to. What are the pre conditions ? They say that the day 1 and the day 2 must be in the same year, month 1 and month 2 should be numbers between 1 and 12 day, 1 and day 2 should be numbers between 1 and 31, month 1 should be less than or equal to month 2 and year should be a number between 1 and 10,000.

So, I write if statements for all these conditions. I say if month 1 is less than 1 or if month 1 is greater than 12 you throw an error. Similarly for month 2, if month 2 is less than 1 or if month 2 is greater than 12 you throw another error.

(Refer Slide Time: 22:43)

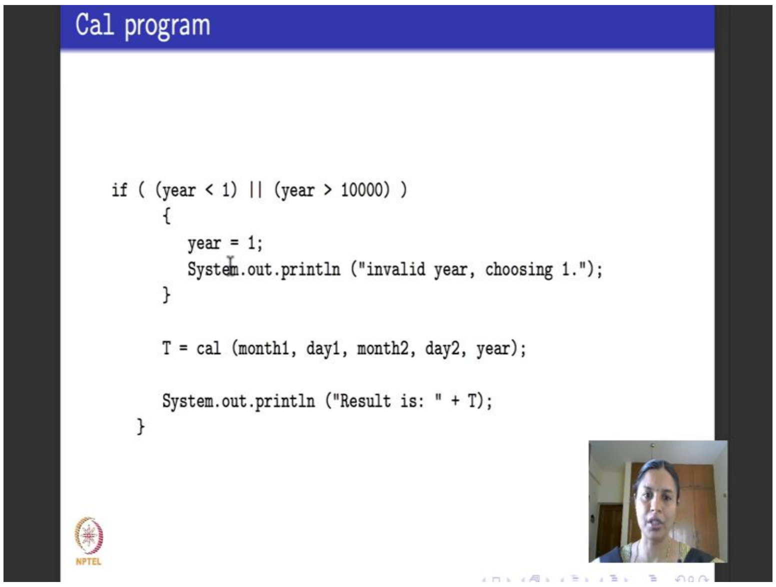


Cal program

```
if ( (day1 < 1) || (day1 > 31) )
      {
         day1 = 1;
         System.out.println ("invalid day1, choosing 1.");
      }
      if ( (day2 < 1) || (day2 > 31) )
      {
         day2 = 1;
         System.out.println ("invalid day2, choosing 1.");
      }
      while ( month1 > month2 )
      {
         System.out.println ("month1 must be prior or equals to month2"
         System.out.println ("Enter month1: ");
         month1 = getN();
         System.out.println ("Enter month2: ");
         month2 = getN();
      }
```

And go on with this, write conditions for day. If day 1 is less than 1 or day 1 is greater than 31 throw an error; if day 2 is less than 1 or day 2 is greater than 31 throw an error; if month 1 is greater than month 2 ask them to enter correct details, that is not acceptable.

(Refer Slide Time: 23:04)



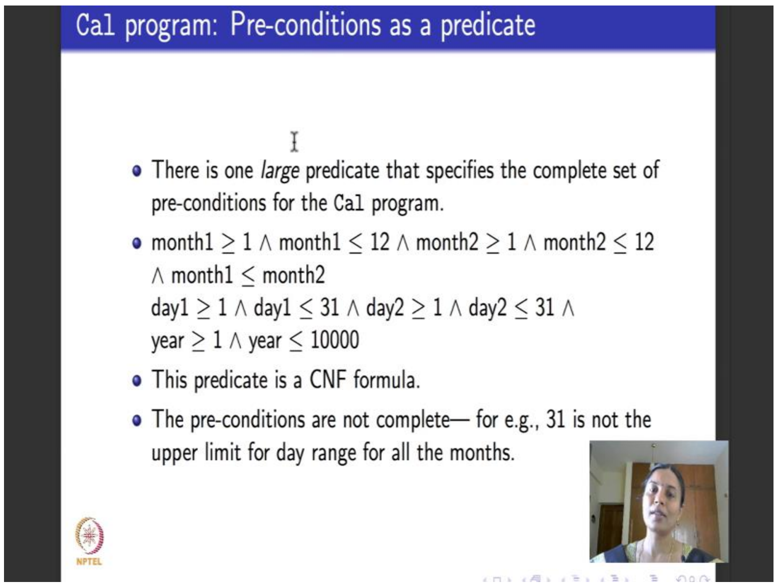
Cal program

```
if ( (year < 1) || (year > 10000) )
      {
         year = 1;
         System.out.println ("invalid year, choosing 1.");
      }

      T = cal (month1, day1, month2, day2, year);

      System.out.println ("Result is: " + T);
   }
```

Like this you go on building program snippets that begin to check for each of the pre conditions. This checks for years, and finally, it calls the method calendar with all these inputs that have been validated and runs the method calendar. Whatever data calendar returns, it will print that data. So, this is how are typical program looks like and pre conditions come as specifications that talk about what are the specifications or conditions related to inputs in the program. It is up to the programmer to be able to add these kind of conditions apart from the main functionality of the program to make sure that the inputs are well taken care of. Now if you go and see this program we have so many predicates here. There is this one, here this if statement, there is this if statement here in which are another predicate for month there are 2 predicates for days there is one more predicate for months there is one more predicate for year.

So, all these predicate mean that we can apply logic coverage criteria. So, what do we do, we collect them all.

(Refer Slide Time: 24:03)



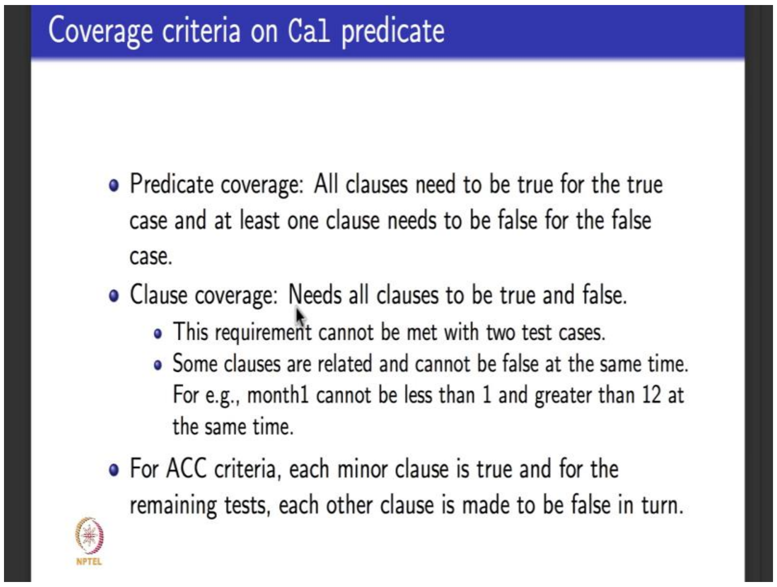
Cal program: Pre-conditions as a predicate

- There is one *large* predicate that specifies the complete set of pre-conditions for the Cal program.
- $month1 \geq 1 \wedge month1 \leq 12 \wedge month2 \geq 1 \wedge month2 \leq 12 \wedge month1 \leq month2$
  $day1 \geq 1 \wedge day1 \leq 31 \wedge day2 \geq 1 \wedge day2 \leq 31 \wedge$
  $year \geq 1 \wedge year \leq 10000$
- This predicate is a CNF formula.
- The pre-conditions are not complete— for e.g., 31 is not the upper limit for day range for all the months.

And we say these are all the constituent preconditions for this program. So, there is one large predicate that specifies the complete set of pre conditions for calendar program. That is what I have written out here as taken from the code. Tts again very easy to read. It says month 1 should be a number between 1 and 12, month 2 should be a number between 1 and 12, month 1 should be less than or equal to month 2, day 1 should be a number between 1 and 31, day 2 should be a number between 1 and 31, year should be within this range.

If you now look at these predicate, it is very easy to see that it s a formula in conjunctive normal form and this is what I told you earlier. These predicates has pre conditions still are not complete. That is a different problem we will worry about it later. For example, 31 is not the upper limit for the day of all months. This particular bit is taken care of in the code, but it need not be, the goal for us is not to check inaccuracy or incompleteness of predicates. The goal for us to check is that the predicative that is written does it make sense.

(Refer Slide Time: 25:08)



Coverage criteria on Cal predicate

- Predicate coverage: All clauses need to be true for the true case and at least one clause needs to be false for the false case.
- Clause coverage: Needs all clauses to be true and false.
  - This requirement cannot be met with two test cases.
  - Some clauses are related and cannot be false at the same time. For e.g., month1 cannot be less than 1 and greater than 12 at the same time.
- For ACC criteria, each minor clause is true and for the remaining tests, each other clause is made to be false in turn.

So, here is a predicate in conjunctive normal form. I told you how to generate test cases TRs for predicate in conjunctive normal form for the various active clause criteria. So, let us before we move on to active clause coverage criteria, let us look at a predicate coverage criteria. Predicate coverage criteria for such a predicate is quite easy: make all clauses true once, then the whole predicate will be true. Every clause has to be true because this is an And. Even if one clause is false the predicate will not be true and to make the predicate false you make any one arbitrary clause, at least one clause false the whole predicate will become false. You could make more than one clause false also.

So, predicate coverage is easy to do what about clause coverage? Clause coverage is a simple TR, as per definition says that each clause should be made true once and false once. But there is a catch here some clauses are dependent on each other. So, I cannot make each clause true once and false once. In fact, when we look at logical formulae corresponding to specifications will realize that while achieving clause coverage or while achieving active clause coverage, we will always have this condition this problem. In several cases, there will be interdependency between the clauses. It could be the case that 2 clauses simultaneously cannot be true, 2 clauses simultaneously cannot be false.

So, given all these extra conditions on the clauses it could be very well the case that the test requirement can become infeasible or the fact that its infeasible could indicate an error in the specifications. Both are valuable information. Like for example, in this case suppose I want to achieve clause coverage, then some clauses are related as I told you and they cannot be false at the same time. For example, month 1 cannot be less than 1 and greater than 12 at the same time.

So, we have to be careful when we deal with clause coverage. In this case it does not mean that the pre precondition has an error, but in certain other cases it could mean to the precondition has an error. So, we when we want to do clause coverage based testing for such predicates, we always exercise a bit of caution because sometimes it may not be practically possible to get test cases for clause coverage. For ACC criteria I told you how to generate right. For conjunctive normal form you have a first row of all true then you have a diagonal of false values, as many as the number of clauses that you have. So that is the test requirement for ACC criteria very easy to get.

So that hopefully would have given you a good idea of how to work with specifications that come as preconditions and how to test for logical predicates based on that. Similarly, you can do for post conditions for invariants for assertions and wherever they are not available or given in English as a tester it is up to us to be able to derive write the logical specifications for them and generate test cases.

Thank you.