

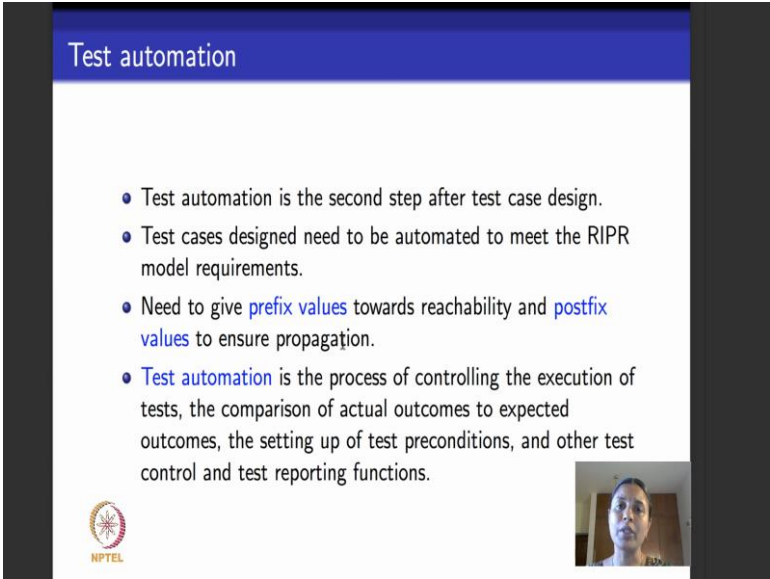
Software Testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture – 04
Software Test Automation: JUnit as an example

Hello everyone. We will do the last module of the first week. In today's module I would like to concentrate on the second step of testing. If you remember in the previous modules we saw that there were four steps involved in software testing: your design test cases, then you automate test case design- that is you make the test design test case ready for execution, followed by the third step which is the test execution process, and finally when the test execution results are recorded you evaluate the test.

So, the second step in these four processes is that of test case automation. And the focus of this lecture is to understand what is test case automation and what exactly goes into it.

(Refer Slide Time: 00:54)



The slide is titled "Test automation" in a blue header. It contains a list of four bullet points:

- Test automation is the second step after test case design.
- Test cases designed need to be automated to meet the RIPR model requirements.
- Need to give **prefix values** towards reachability and **postfix values** to ensure propagation.
- **Test automation** is the process of controlling the execution of tests, the comparison of actual outcomes to expected outcomes, the setting up of test preconditions, and other test control and test reporting functions.

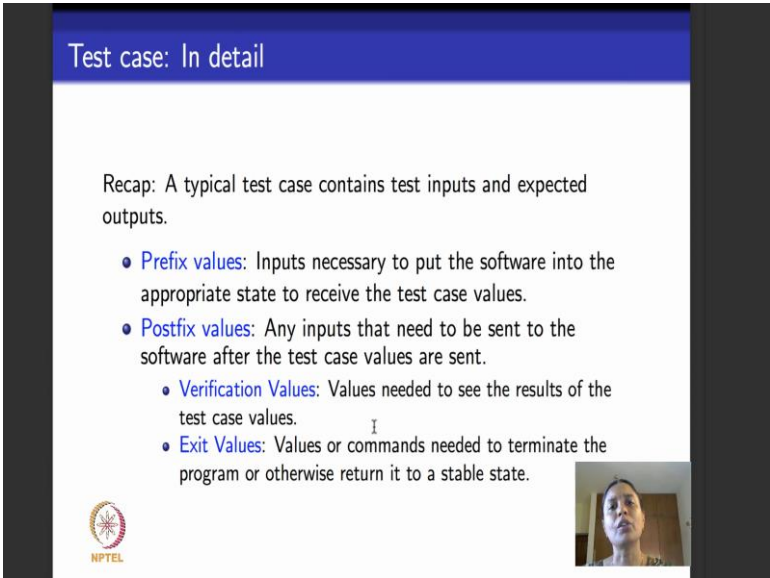
In the bottom left corner, there is a small circular logo with the text "NPTEL" below it. In the bottom right corner, there is a small video inset showing a man speaking.

In the previous module we saw that testability of a software, in turn, translates into observability and controllability, and these are measured using the RIPR model; R for reachability, I for infection, P for propagation and the final for R for revealing. So, how do I ensure that reachability propagation and revealing are done? I ensure that these three are done in the step of test case automation which is what we will look at in this module. So, how do we do that? We have to give what are called prefix values to ensure

reachability of a particular piece of code, and then once that test case design exercises that piece of code we have to give postfix values to the test case design to ensure that if there is an error that gets propagated outside.

So, all these things are what are called test automation. To formally define test automation it is the process of controlling the execution of tests, and actually ensuring the reachability is done by giving prefix values of preconditions. Then, you execute the test and compare the actual out come to the expected outcome and then you know report the results of your test case execution.

(Refer Slide Time: 02:11)



The slide is titled "Test case: In detail" in a blue header. Below the header, it says "Recap: A typical test case contains test inputs and expected outputs." followed by a bulleted list:

- **Prefix values:** Inputs necessary to put the software into the appropriate state to receive the test case values.
- **Postfix values:** Any inputs that need to be sent to the software after the test case values are sent.
 - **Verification Values:** Values needed to see the results of the test case values.
 - **Exit Values:** Values or commands needed to terminate the program or otherwise return it to a stable state.

In the bottom left corner is the NPTEL logo. In the bottom right corner is a small video inset showing a person's face.

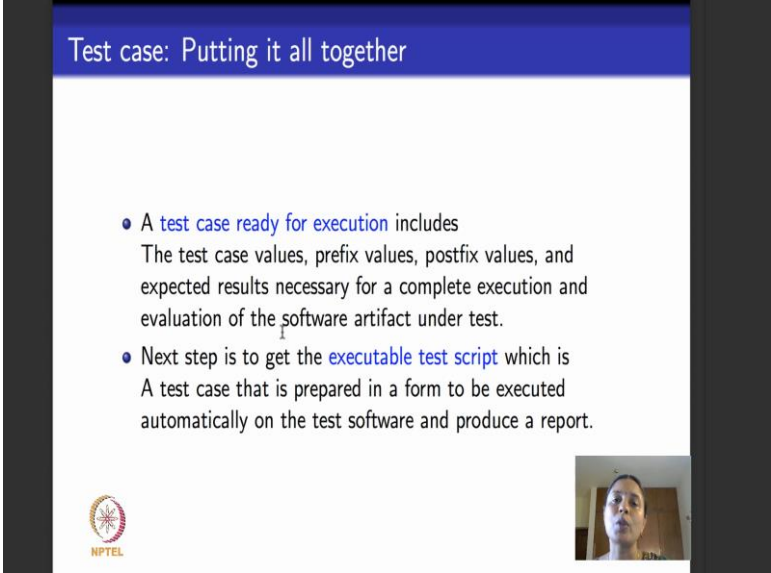
So, we will recap what a test cases and then see what are the add ones that we have to do to a test case to make it ready as an executable test script. If you remember from the first lecture what was the test case; test case basically contains test inputs and expected out puts. And if the expected outputs match the actual output after execution then you say that the test cases passed otherwise it is failed. This is a raw test case that has been designed.

Now, to make it into an executable test script the process of test automation has to add prefix values and postfix values to this test case. So, what are prefix values? They are basically inputs that are necessary to put the software or the software artifact into an appropriate state so has to be able to receive the actual test case for execution. And after the test case is executed, postfix values come into picture. What a postfix values? They

have values that are necessary so that the results of execution are sent to the software as an observable value by an external user.

Postfix values intern bifurcated into two categories: verification values and exit values. What are verification values? Verification values basically tell you that an exception has occurred this test case has passed or it has failed and it is a value that clearly tells you what is the result of test case execution. And what are exit values? Exit values are basically values or pieces of code that are needed to make sure that after the test case is executive the code appropriately finishes its execution fully and exits so, as to retain and reveal the error state, if there was any present during execution.

(Refer Slide Time: 04:04)

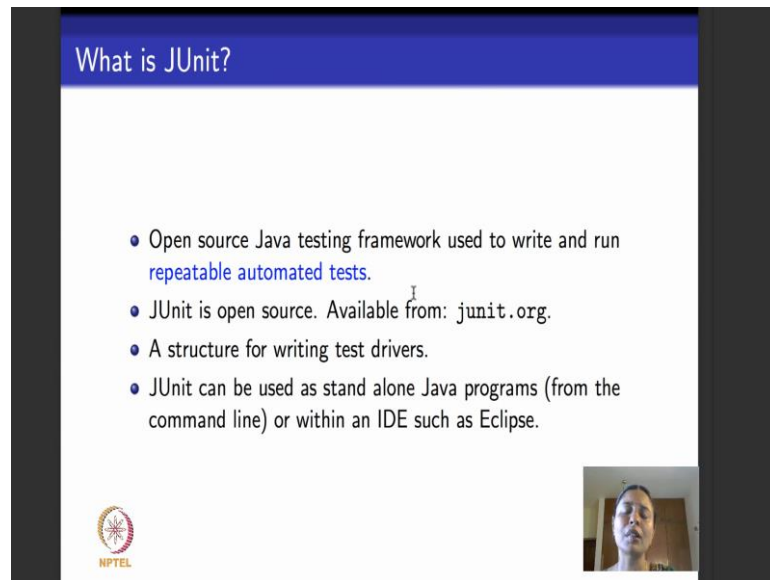


The slide has a blue header with the text "Test case: Putting it all together". Below the header, there are two bullet points. The first bullet point is "A test case ready for execution includes" followed by "The test case values, prefix values, postfix values, and expected results necessary for a complete execution and evaluation of the software artifact under test." The second bullet point is "Next step is to get the executable test script which is" followed by "A test case that is prepared in a form to be executed automatically on the test software and produce a report." In the bottom left corner, there is a logo for NPTEL. In the bottom right corner, there is a small video inset showing a person's face.

- A test case ready for execution includes
The test case values, prefix values, postfix values, and expected results necessary for a complete execution and evaluation of the software artifact under test.
- Next step is to get the executable test script which is
A test case that is prepared in a form to be executed automatically on the test software and produce a report.

So, now if you take the raw test case as it was designed and make it ready as a test script to be executed what are the summary of things that it has. It has the actual test case values, the prefix values, the postfix values, and as I told you it also has the expected outputs. So, when we put it all together and get it ready, a final result at the end of the test automation step should be to be able to get an executable test script, which is a test script that contains all these values and can be executed directly on the piece of code that it to being tested on.

(Refer Slide Time: 04:45)



What is JUnit?

- Open source Java testing framework used to write and run **repeatable automated tests**.
- JUnit is open source. Available from: junit.org.
- A structure for writing test drivers.
- JUnit can be used as stand alone Java programs (from the command line) or within an IDE such as Eclipse.

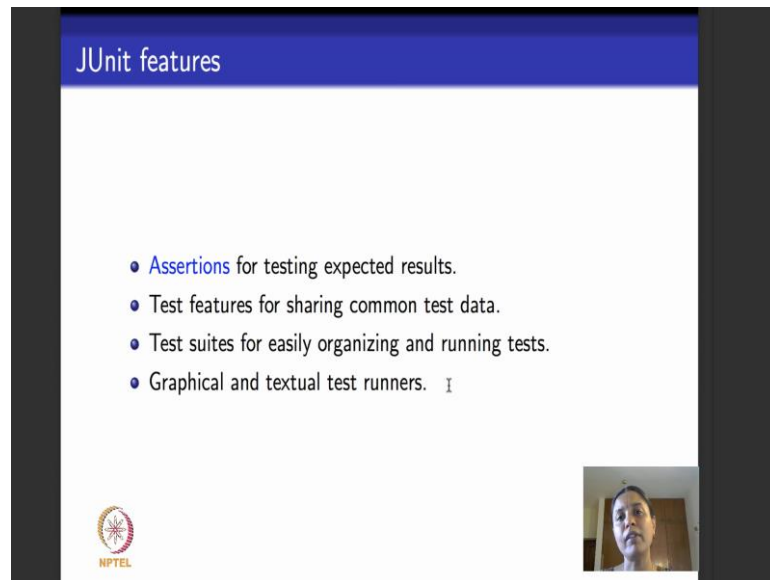
NPTEL

So, I will explain in detail how to give prefix values, how to give postfix values, and what happens in the process of test automation by using this open source tool; very good open source tool called JUnit as an example. Later in the part of the course when we see several examples of Java programs or C programs you could use the JUnit framework to be able to experiment with the test cases that you have designed as a part of assignments or other lectures that we will see in the course.

This module is not meant to be an exhaustive introduction of JUnit, but it is more meant to be used to interpret JUnit as a test automation tool, as a framework to understand how test automation works. In that process you will also learn some of the commands of JUnit which will be useful when you run your own experiments with JUnit at later points in the course. So, what is JUnit? It is an open source testing tool very popularly used in the industry. You can download it from [Junit.org](http://junit.org). And what are the things that it supports? It is very good for writing and executing test scripts.

So, once you design a test case you can automate it using JUnit and you can execute it and evaluate the results also using JUnit. It can be used as a standalone entity on Java programs or it can be used within an IDE like Eclipse.

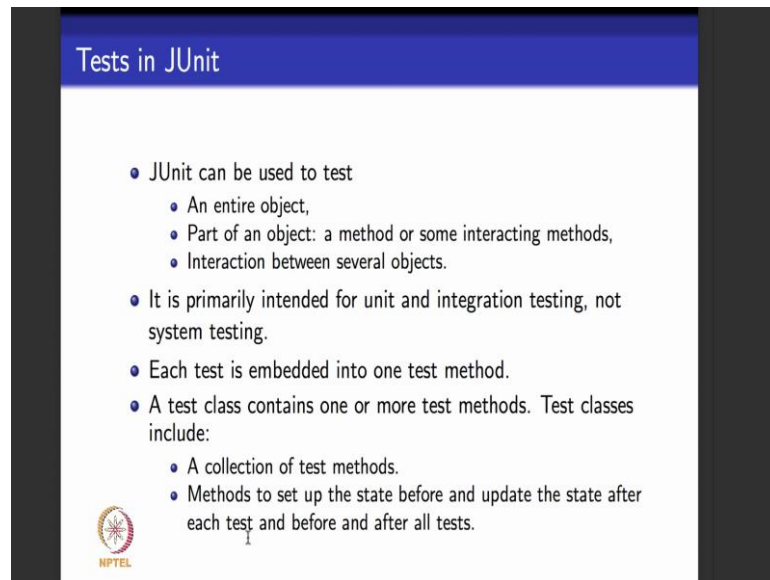
(Refer Slide Time: 06:13)



So, what are the high level features of JUnit? JUnit supports what are called assertions. How are assertions used? Assertions are basically statements that will always return true or false. You can view an assertion as if it returns true then everything is fine, the test case is passed. If an assertion returns false then there is an error that has been found and you can use this assertion to be able to reveal the error to the tester. So, we will assertions, how to use assertions through examples in JUnit.

JUnit also has test features for sharing common test data. Let us say two people are writing a common piece of code and they want to be able to share the test cases that they are writing then you can use JUnit to be able to do that. And JUnit also has suites for easily organizing, running, executing and observing tests. And of course, like many other tools it has a textual interface and it also has graphical interface.

(Refer Slide Time: 07:11)



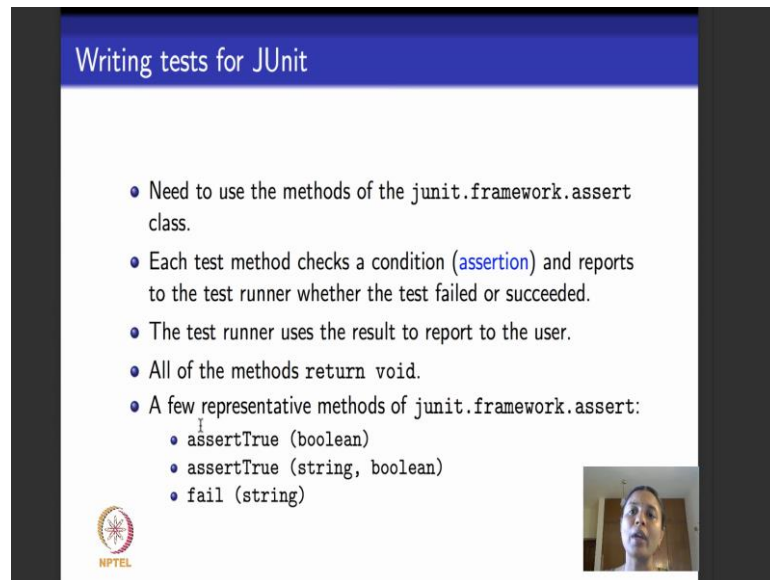
The slide is titled "Tests in JUnit" in a blue header bar. Below the header, there is a bulleted list of points. The first point states that JUnit can be used to test, followed by three sub-points: "An entire object," "Part of an object: a method or some interacting methods," and "Interaction between several objects." The second main point states that JUnit is primarily intended for unit and integration testing, not system testing. The third main point states that each test is embedded into one test method. The fourth main point states that a test class contains one or more test methods. Below this, it says "Test classes include:" followed by two sub-points: "A collection of test methods." and "Methods to set up the state before and update the state after each test and before and after all tests." In the bottom left corner, there is a circular logo with a star and the text "NPTEL" below it.

- JUnit can be used to test
 - An entire object,
 - Part of an object: a method or some interacting methods,
 - Interaction between several objects.
- It is primarily intended for unit and integration testing, not system testing.
- Each test is embedded into one test method.
- A test class contains one or more test methods. Test classes include:
 - A collection of test methods.
 - Methods to set up the state before and update the state after each test and before and after all tests.

So, what can be JUnit be used for testing? The main thing is JUnit used for testing Java programs. So, it can be used as a very good unit testing tool or it can be used as a very good integration testing tool. It obviously cannot be used as a system testing tool. In system testing if you remember what I had told you, we take the inputs put the system is a part of the input server, connected to the database and let the outputs be observed as commands and so on.

JUnit being a testing tool for Java cannot be used for system testing, but can very well be used for unit testing and integration testing phases. So, it can be used to test an entire object, it can be used to choose just one method a certain interacting methods within an object it is up to you. So, JUnit has what are called test methods and each test is embedded into one test method, then it has a test class that contains one or more test methods. Test class, apart from this, can also have method that I used to setup the software to the state before and update the state after each test. So, they are these methods can also be used to do the prefix and postfix values that I had telling you about. And then there are code test methods which actually contain the test cases that have to be executed.

(Refer Slide Time: 08:34)



Writing tests for JUnit

- Need to use the methods of the `junit.framework.assert` class.
- Each test method checks a condition (**assertion**) and reports to the test runner whether the test failed or succeeded.
- The test runner uses the result to report to the user.
- All of the methods return `void`.
- A few representative methods of `junit.framework.assert`:
 - `assertTrue (boolean)`
 - `assertTrue (string, boolean)`
 - `fail (string)`

NPTEL

NPTEL

So, how do I write tests for JUnit? The first thing that we will understand while writing test using JUnit is to be able to use assertions. As I told you a little while ago what is assertion used for; assert is like a debug but in the context of testing. So, I say assert something if this test case passes, assert something if this test case fails. Assertion is always a Boolean expression. The value inside an assert always evaluate to true or false.

The implicit understanding while writing assertions is that if it evaluates to true then everything is fine your test cases passed so you can move on. But if an assertion evaluates to false then it indicates or it might indicate an error state. So, you will typically put a print statement there, saying what is gone wrong and you throw an exception or you print an appropriate warning and so on. So each test method that is used inside JUnit basically checks for a condition which is nothing but the assertion. And reports to the main test runner method about whether the test is passed or failed.

The test runner method uses the result of this assertion failing of passing to be able to report the result to the end user. Here are some examples of how asserts run. So, it is present in JUnit or framework or assert. So, you can do something like

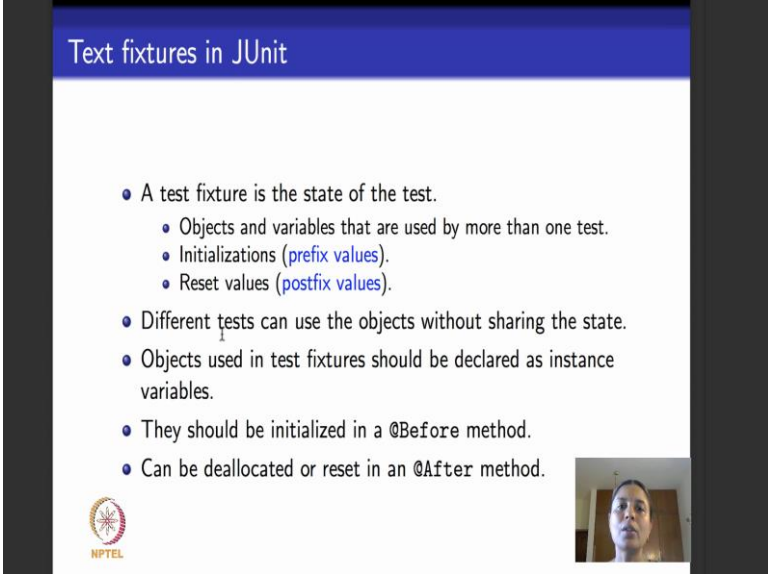
`assert true Boolean.`

So, this just says that if this Boolean predicate inside this asserts true evaluates to true then you just quietly say that it is true. If it evaluates to false maybe he will give a warning and terminate if it require.

The second asserts true test two arguments it takes a string and it takes a Boolean predicate. The idea here is that if the Boolean predicate evaluates to be true then you do not do anything. And if the Boolean predicate evaluates to be false then you print the string. So, it can be used as a way of warning the user.

A third assertion is what is called fail assertion which basically says only if it fails, if a particular thing fails then you output string as a warning to the user. So, when we see examples of code that we write with JUnit, I will show you examples of how to use all these assertions.

(Refer Slide Time: 10:58)

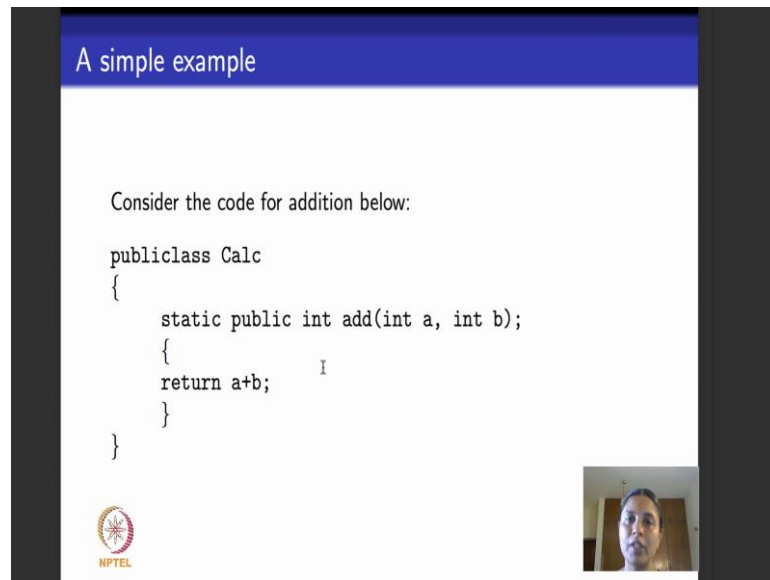


The slide is titled "Text fixtures in JUnit" in a blue header. It contains a bulleted list of points about test fixtures. At the bottom left is the NPTEL logo, and at the bottom right is a small video feed of a person.

- A test fixture is the state of the test.
 - Objects and variables that are used by more than one test.
 - Initializations (*prefix values*).
 - Reset values (*postfix values*).
- Different tests can use the objects without sharing the state.
- Objects used in test fixtures should be declared as instance variables.
- They should be initialized in a `@Before` method.
- Can be deallocated or reset in an `@After` method.

So, now how to write a test fixture? Assertion is a core component which tells you when a test cases passed or failed. Now we have to write still prefix values, postfix values, how do you do that. So, how do I do that? Prefix values are initialized in what is called before method; 'at before' method and postfix values are given in a method called 'at after' method- the names are very intuitive and very easy to remember.

(Refer Slide Time: 11:27)



A simple example

Consider the code for addition below:

```
public class Calc
{
    static public int add(int a, int b);
    {
        return a+b;
    }
}
```

NPTEL

So, I will first walk you through a couple of examples. We will start with the simple code for addition as an example, then I will look at another example which returns the minimum element in the list, and tell you how to write prefix values, how to write postfix values, how to give the test cases, and how to write assertions that will output the result of the test case passing or failing.

So, we will start with an example of a code that does addition. So, here is an example of a code that does addition there is nothing complicated to it, it takes two integers a and b and it returns the sum of a and b which is a plus b. Now suppose you are given the task of testing this code. So, let us recap and understand what a test case is.

So, test case should give inputs and it should give expected outputs. What are inputs to this piece of code? Inputs are one value for a and one value for b, and expected output is the actual sum of a and b. Now what you have to do is a part of the automation you say that you please take these inputs and compare it to the expected output. If the actual output matches the expected output fine, everything is working fine, but if the actual output differs from the expected output then you use assertions to be able to flag the fact that the test case execution has failed.


(Refer Slide Time: 12:50)

A simple example

```
import org.junit.Test;
import static org.junit.Assert.*;

public class CalcTest
{
    @Test public void testAdd()
    {
        assertTrue ("Calc sum incorrect",
            5 == Calc.add (2, 3));
    }
}
```

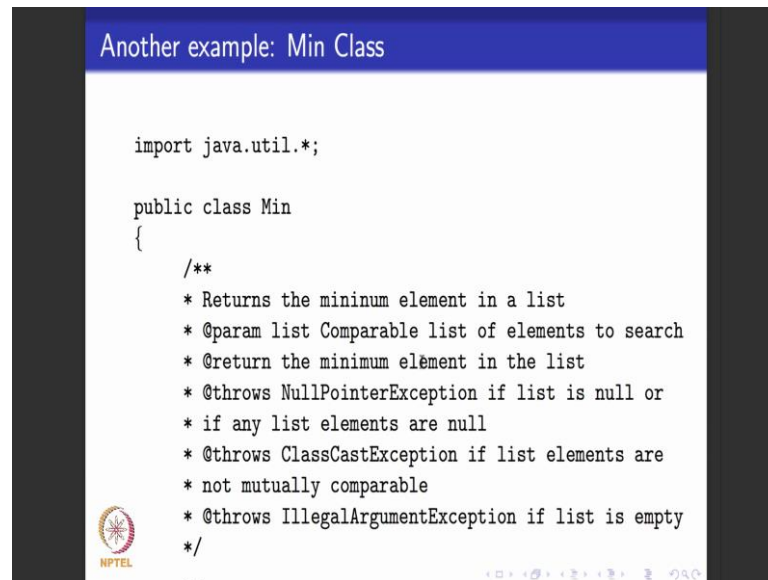
- 2 and 3 are test inputs, 5 is the expected output.
- The string Calc sum incorrect is printed if assert fails, if the actual output is different from the expected output.



So, how will you do it in JUnit? This is how you do it in JUnit. So, these has standard things that you have to import, every time you will write any program any test method using JUnit for test case automation and execution. So, you import what is called JUnit or test and you import this library of assertions. So, I am writing something called the test for the calc method. How do I do? I write these two statements the first statement says that- you assert true and it gives a string, and the second statement actually runs the addition code that we saw in this slide with two actual test inputs--- 2 and 3. And if 2 and 3 are the test inputs assuming that addition works correct what is the expected output? The expected output should be 5. So, it compares it to 5.

So, what it says is that you run this add program on inputs 2 and 3 and check if the result that is outputted by the add program on these inputs is actually equal to 5. If it is actually equal to 5 then you exit, test cases passed. But if it is not equal to 5, if the calc program actually has an error; in this case it does not have an error because it just a simple program that returns a plus b. But assuming that it does have an error you write it for some other program, then this assert true will take over and it will output this string. So, it will say that calc sum is incorrect. So, it will say that there is a error somewhere in the code.

(Refer Slide Time: 14:22)



So, now we look at another example, slightly longer than that simple toy addition example. So, this is an example of a piece of code, Java code, that written the minimal element in a list. So, this code is very well debugged; in the sense that it takes care of all exception conditions, it takes care of what if the list is empty there is nothing to written, what if the list has entities that cannot be compared at all. For example, suppose the list has a string and a number, the list has a string and a Boolean value--- there of different types they cannot be compared.

So, in all these cases this code is meant through several different kinds of exceptions. So, these are what are called a part of debugging. Whether developer himself takes care of all the corner cases like wrong inputs, wrong data types etcetera which need not be taken care of by a tester. So, this code is well written to be able to take care of all the exceptional cases. Of course, the tester's job would be able to test for all these features also as we will see through examples, but we will first see what the code does and what are the various exceptions that it takes care of.

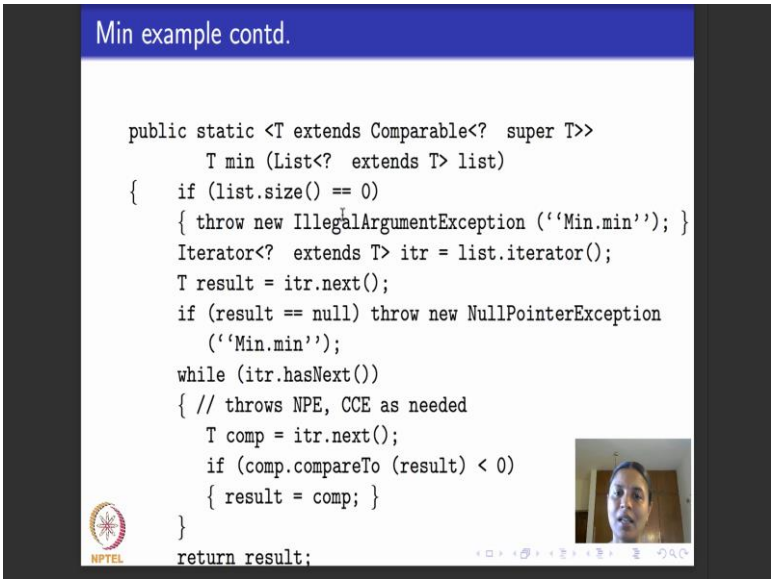
So, this class is called min and as I told you it returns the minimum element in a list and it takes as input comparable list of elements. What I said is that all of them should be comparable, we should all be a list of numbers or they should be a list of strings that I can compare them with respect to the lexicographic ordering. They should not be

incomparable types. And what is the main functionality of this class min? It supposed to return the minimum element in the list.

As I told you this class min is also supposed to take care of exceptions, like wrong inputs, incomparable inputs and so on. So, it has several exceptions. The first exception that it has is what is called a null pointer exception. It throws the null pointer exception if the list is empty or if any of the elements of the list are empty. And the second kind of exception that it throws is what is called as ClassCastException which it throws if the list elements are not comparable to each other. As I told you one is a string and the other is a Boolean constant how do you compare them, so you have to throw an exception. It also throws an illegal argument exception if the list that is passed to it is empty, there are no elements to compare.

So, I have split this code across two slides because otherwise it would not be readable. So, we will move on and look at the rest of the code in the next slide.

(Refer Slide Time: 16:54)



Min example contd.

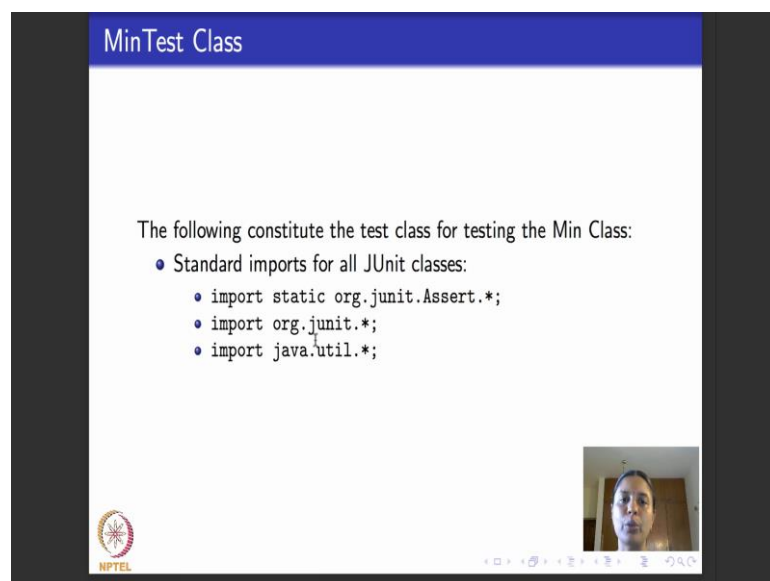
```
public static <T extends Comparable<? super T>>
    T min (List<? extends T> list)
{
    if (list.size() == 0)
    { throw new IllegalArgumentException ("Min.min"); }
    Iterator<? extends T> itr = list.iterator();
    T result = itr.next();
    if (result == null) throw new NullPointerException
        ("Min.min");
    while (itr.hasNext())
    { // throws NPE, CCE as needed
        T comp = itr.next();
        if (comp.compareTo (result) < 0)
        { result = comp; }
    }
    return result;
}
```

So, what is the main code look like? This is how the main code looks like, I just glanced through it because I do not want to read the code line by line, is just a standard written Java program that takes a list called T. And then if the list is empty then it throws an illegal argument exception otherwise it repeatedly compares it to the next element in the list and returns the result in this value called result; it returns the minimum element. So, this is a piece of Java code. So, just to recap what its main functionalities are, it takes a

list of comparable elements as its argument and then returns the minimum element in the list.

Suppose the list does not have comparable elements, the list is empty; the list has other kinds of problems this code does exception handling very well. So, now our job is to be able to design test cases for this minimum program and see how you can use the second step which is the test automation step that is the focus of this lecture to be able to test this program.

(Refer Slide Time: 18:02)



The slide is titled "MinTest Class" in a blue header. The main content area is white and contains the following text:

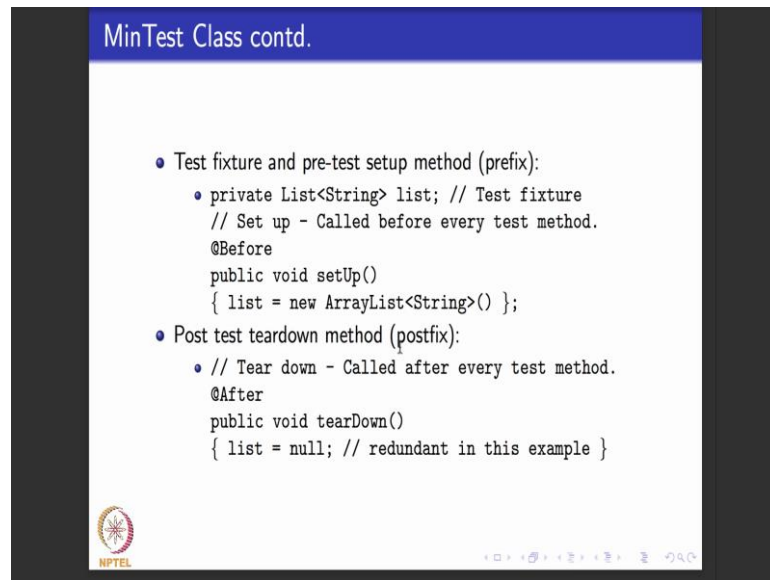
The following constitute the test class for testing the Min Class:

- Standard imports for all JUnit classes:
 - `import static org.junit.Assert.*;`
 - `import org.junit.*;`
 - `import java.util.*;`

In the bottom right corner, there is a small video feed of a person. In the bottom left corner, there is a logo for NPTEL.

So, how do I do? The first step as I told you is because we want to be able to use the tool JUnit we have to import all the standard classes; we have to import JUnit assert, we have to import JUnit you have to import the util library. After this the next job is to be able to give prefix value and postfix values to the code.

(Refer Slide Time: 18:25)



The slide is titled "MinTest Class contd." and contains the following Java code:

```
• Test fixture and pre-test setup method (prefix):
  • private List<String> list; // Test fixture
    // Set up - Called before every test method.
    @Before
    public void setUp()
    { list = new ArrayList<String>(); };

• Post test teardown method (postfix):
  • // Tear down - Called after every test method.
    @After
    public void tearDown()
    { list = null; // redundant in this example }
```

The slide also features the NPTEL logo in the bottom left corner and navigation icons in the bottom right corner.

How is that done? Prefix values is given by this kind of, as I told you right, by this act before thing. So, what you do is that you give a test fixture and a pre test setup method. So, you have it like this--- you give it a list which is a test fixture and then you set it up which is called before the main test method. The main test method actually has the test cases for execution.

So, you say that this is the thing and I pass a new array for this. And after this in the main test case the main test method what we called containing the test cases, and post this I have what is called teardown method which gives the postfix values which basically in this case is not relevant because there is nothing much to do, but assuming that you had a fairly large piece of code postfix teardown method will actually do the rest of the execution to be able to see the output as being produced by the code. In this case because it is a small example we will directly see the output. So, there is not much postfix activity to be done here.

(Refer Slide Time: 19:29)

Min Test Cases: NullPointerException

A test case that uses the fail assertion.

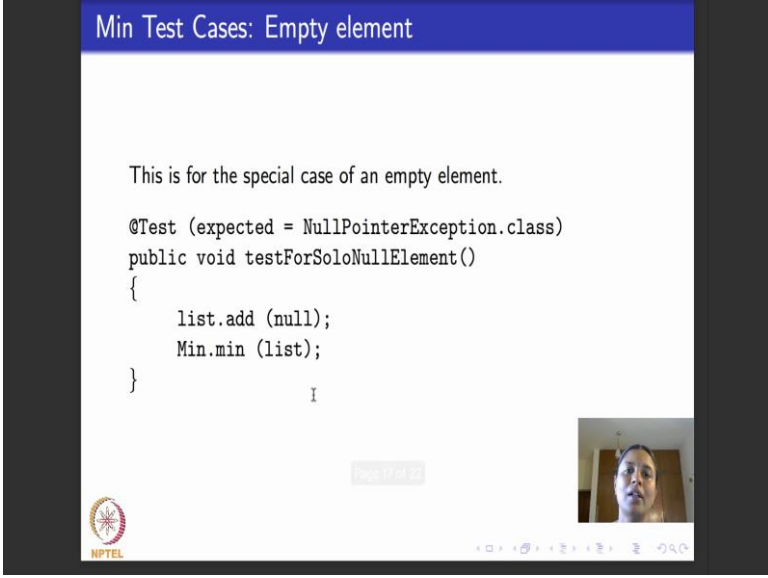
```
@Test public void testForNullList()
{
    list = null;
    try {
        Min.min (list);
    } catch (NullPointerException e)
    {
        return;
    }
    fail (NullPointerException expected);
}
```

NPTEL

So, here is an example of a test case that uses the fail assertion. If you remember I had told you there are three kinds of assertions. So, if you give me a few seconds I will go back to that slide. Remember there were three kinds of assertions I used took different kinds of assert true which basically return a warning if this Boolean string that is passed to it. If this Boolean predicate that is passed it is returns false or it returns string if the Boolean predicate that is passed to it is return false. And then third kind of assertion will returned this string if it fails.

So, here for this min example we will use the fail assertion and here is a test case that uses the fail assertion. What is this test case method called it is called test for null list, it is a void method and it passes an empty list and then its tries to see if the code actually throws a null pointer exception. And if it does not throw a null point exception using this try and catch, it will use the fail assert to say that it had actually expected a null pointer exception which did not happen. So, there is an error in the code. In our code this will not come because this error is handled.

(Refer Slide Time: 20:49)



Min Test Cases: Empty element

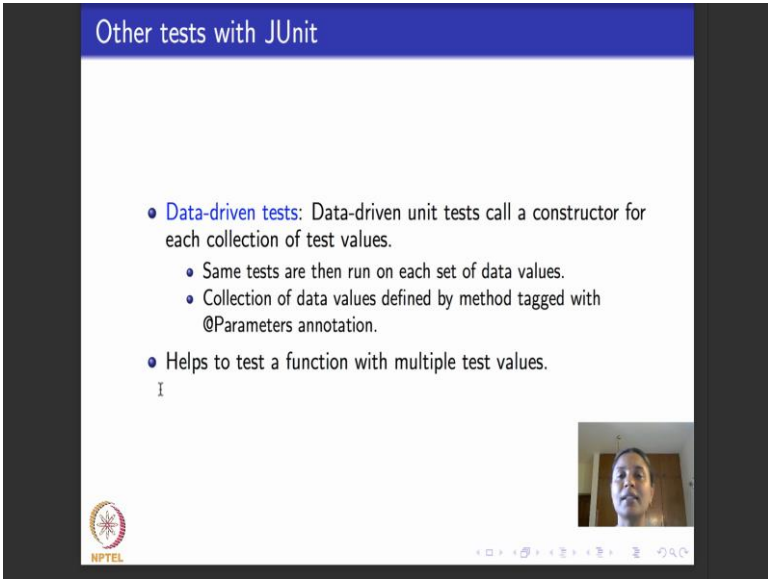
This is for the special case of an empty element.

```
@Test (expected = NullPointerException.class)
public void testForSoloNullElement()
{
    list.add (null);
    Min.min (list);
}
```

The slide includes a small video inset of a person in the bottom right corner and an NPTEL logo in the bottom left corner.

So, here is another example of a test case that tests our min code for the special case of an empty element. So, what it says is that I tried to add an empty element in the list and I tried to compute its minimum. And in this case because the code is well written this is also taken care of, it will return find and there is no error in this code even for this kind of test case.

(Refer Slide Time: 21:20)



Other tests with JUnit

- **Data-driven tests:** Data-driven unit tests call a constructor for each collection of test values.
 - Same tests are then run on each set of data values.
 - Collection of data values defined by method tagged with @Parameters annotation.
- Helps to test a function with multiple test values.

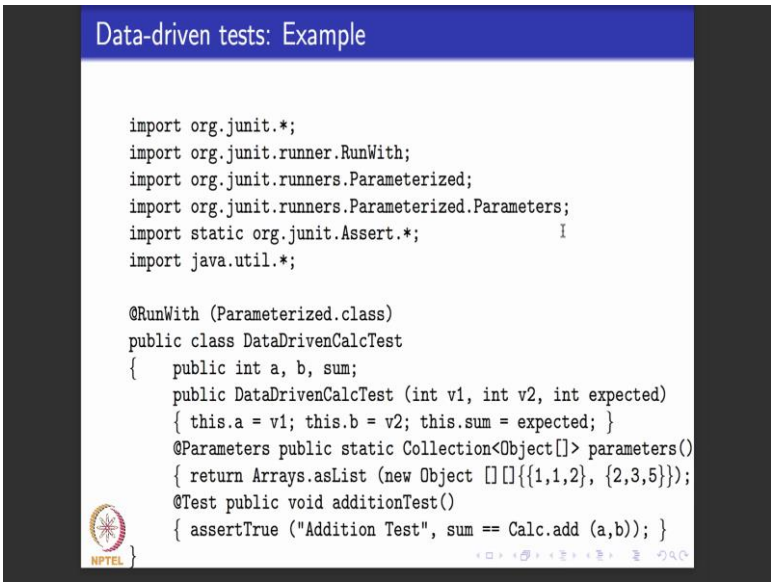
The slide includes a small video inset of a person in the bottom right corner and an NPTEL logo in the bottom left corner.

Now, what are the other things that we can do with JUnit. We saw two examples this is with testing for two exception cases. Of course, you can go on testing this minimum code

for several other exception cases, because the code handles several other an exception handling mechanism. But suppose I have to test the main functionality of the code. What is the main functionality of the code? The main functionality of the codes to be able to give a list of elements that are comparable and check if it actually returns the minimum value from the list; so for that I need to be able to give data, I need to be able to pass a list of compatible elements. How does that happen? How does one do it JUnit?

For giving data to test with JUnit we have a constructor for them. So, that constructor what it does is that you can passed several different data to it and the same tests are run for each set of data values and the collection of these data values are defined by a method tag with a @parameter. So, it basically helps to test a function with multiple text value. So, I can test a minimum function with several different lists and check for each of these lists does not return the minimum. I can check the add function that other toy example that we looked at with several different arguments a and b and check in each of these case does it actually return the sum of a and b.

(Refer Slide Time: 22:53)



```
import org.junit.*;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameters;
import static org.junit.Assert.*;
import java.util.*;

@RunWith (Parameterized.class)
public class DataDrivenCalcTest
{
    public int a, b, sum;
    public DataDrivenCalcTest (int v1, int v2, int expected)
    { this.a = v1; this.b = v2; this.sum = expected; }
    @Parameters public static Collection<Object[]> parameters()
    { return Arrays.asList (new Object [] [] {{1,1,2}, {2,3,5}}); }
    @Test public void additionTest()
    { assertTrue ("Addition Test", sum == Calc.add (a,b)); }
}
```

So, I will go back to the add example and show you how to use this @parameters with the constructor to be able to provide data along with prefix, postfix and assert. So, here is the complete JUnit program for the same. So, import as I told you all these various classes, now what I do is I have to pass data. Now I am going back to the add examples. So, I have to always first two integers a and b to it and check if the sum of a and b is

actually returned by the addition code. So, I use this constructor and then I write this particular class that does the main testing.

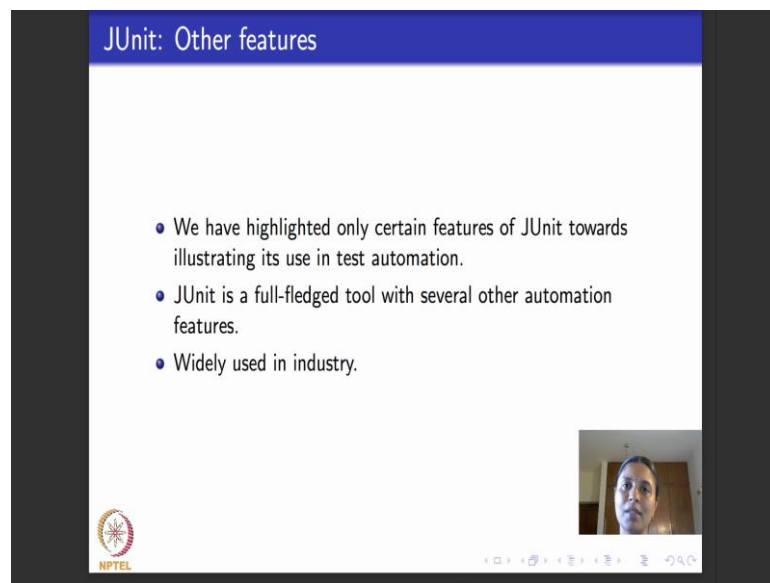
So, how does this work? If you see this dot a is equal to v 1 and this dot b is equal to v 2 these are the variables that are used to pass the actual parameters. And then the expected result is stored in this dot sum. And if you go here there is an array which the first value set of test case values that it passes are a is 1, b is 1, and expected result is 2. The second set of test cases that it passes are a is 2 b is 3 and the expected result is 5. In this example I have just given two so that I can explain it you, but you can pass an array of as many test cases as you want along with their expected outcome.

So, what it will do is; it will go back and execute this particular method. It will execute this particular method, and in any case for whichever test case if this assertion fails then it will output that this sum is incorrect. How it does is it will do it for each of these test case values and the expected output. And it will finally exit without giving any assertion violations if all of them pass. So, for our particular example the calculator addition was correct code so it will pass for all these examples.

So, similarly for the minimum element in a list also you can put it all together, right, check it for various null pointer, empty list and other kinds of exceptions, and after passing all these exceptions actually use this constructors class to be able to pass several different list of values and their minimum element and check whether the code actually returns the minimum element from this list; because that piece of test code in JUnit is fairly long to write it would not fit into even 2-3 slides, I have not given that as an example. What I can do is I will be putting it along with my notes feel free to look at it.

So, hopefully at the end of this exercise you would have understood how to give prefix values to a test case and how to actually give the test case values write asserts that will indicate whether the test case is passed or failed. And if needed, how to make postfix values in the codes such that the assert failure or assert pass actually reflexes output in the code.

(Refer Slide Time: 26:01)

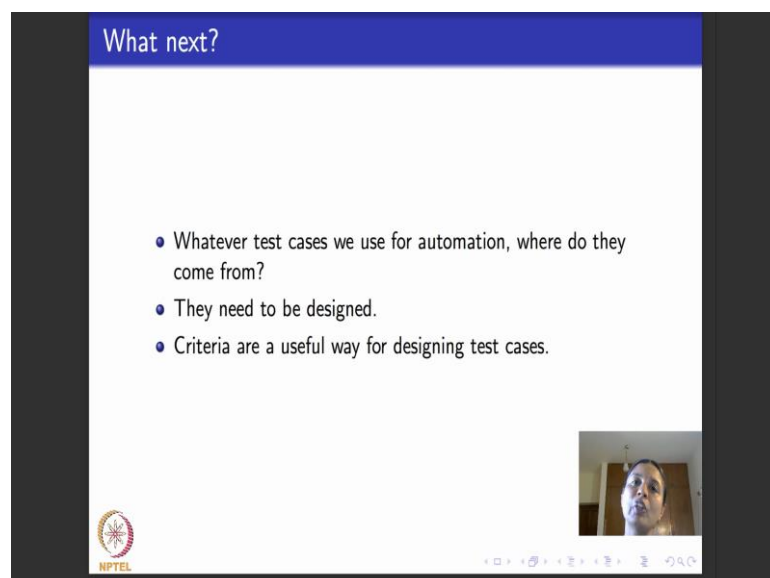


The slide has a blue header with the text "JUnit: Other features". The main content area is white and contains a bulleted list of three points. In the bottom right corner, there is a small video inset showing a man speaking. The NPTEL logo is in the bottom left corner, and navigation icons are in the bottom right corner.

- We have highlighted only certain features of JUnit towards illustrating its use in test automation.
- JUnit is a full-fledged tool with several other automation features.
- Widely used in industry.

To understand these we used JUnit. JUnit is a fairly extensive tool as I told you in the beginning; the purpose of this module was not to be able to teach you exhaustive features of the JUnit, but to be able to teach you how to use JUnit for test automation. So, feel free to go download JUnit and explore all its other features and try it out on your own Java programs to be able to see if you can test them or not.

(Refer Slide Time: 26:27)



The slide has a blue header with the text "What next?". The main content area is white and contains a bulleted list of three points. In the bottom right corner, there is a small video inset showing a man speaking. The NPTEL logo is in the bottom left corner, and navigation icons are in the bottom right corner.

- Whatever test cases we use for automation, where do they come from?
- They need to be designed.
- Criteria are a useful way for designing test cases.

So, where are we going on from this? So now, we saw how to automate and once you automate and execute JUnit also has an execution framework. As I told you it is fully

command driven, fully automated. So, there is nothing much to discuss about it, I will not be discussing about that in detail. After that you actually observed by using assertions the result of failures.

Now if you go back right to the first step; that is test case design it actually tells you what are the test cases that you pass on to a tool like JUnit for automating. How do you design test cases, how does one go about giving effective test cases without giving blind test cases and not hoping to find any errors. So, for this we go back to the problem of test case design. So, we will look at criteria based test case design, we model software using different mathematical model structure as I told you graphs, logical expressions, sets and so on. And teach you how to design test cases based on each of these. So, that will be the focus of my lectures beginning next week onwards.

Thank you.