

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

**Lecture - 42**  
**Mutation Testing**



Hello there, we are in week 9. Last lecture I did mutation testing for integration specifically focusing on design integration.

(Refer Slide Time: 00:21)

**Mutation testing for software artifacts: An overview**

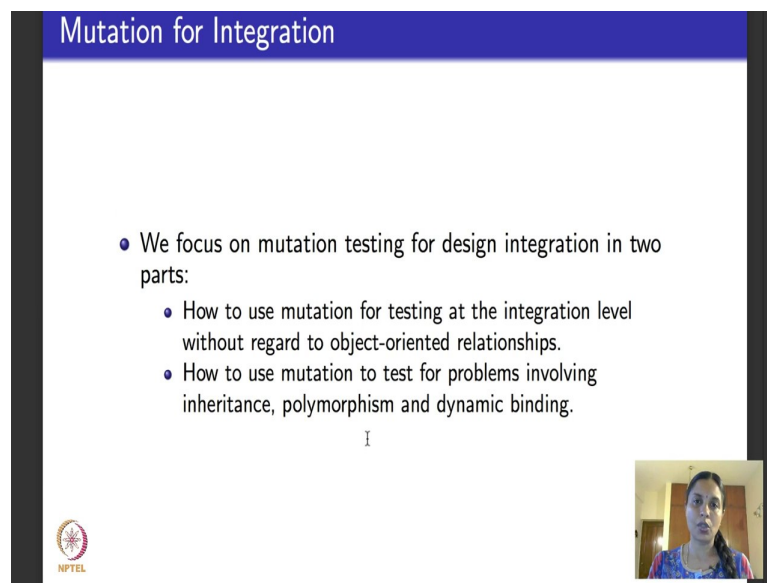
	For programs	Integration	Specifications	Input space
<b>BNF grammar</b>	<b>Programming languages</b>	–	Algebraic specifications	Input languages like XML
<b>Summary</b>	<b>Compilers</b>			Input space testing
<b>Mutation</b>	<b>Programs</b>	Programs	FSMs	Input languages like XML
<b>Summary</b>	<b>Mutates programs</b>	Tests integration	Model checking	Error checking

Focus of this lecture and next: Mutation for programs for integration testing.



So, this is what is the status of our lectures, we are inside mutation testing the ones marked in green are done, the one striked out we are not going to do, this we will be doing in the next two lectures. We are currently here, we are currently here in applying mutation testing for testing program with focus on integration.

(Refer Slide Time: 00:42)



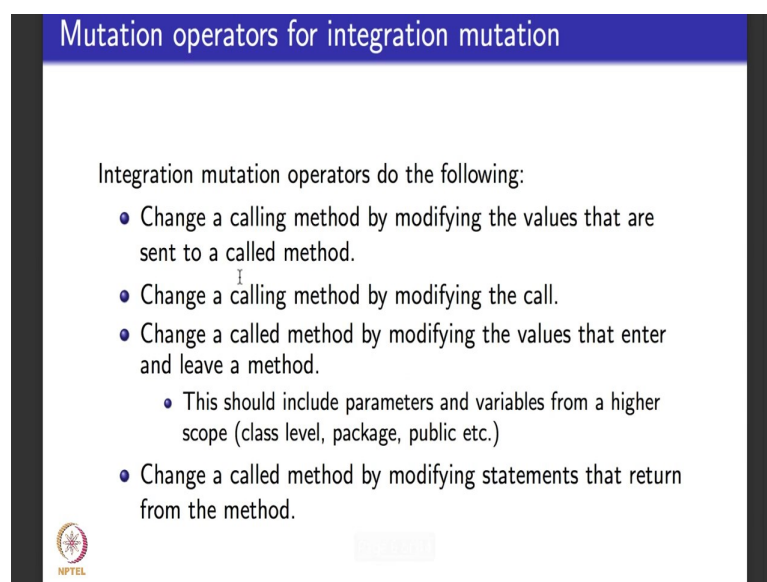
### Mutation for Integration

- We focus on mutation testing for design integration in two parts:
  - How to use mutation for testing at the integration level without regard to object-oriented relationships.
  - How to use mutation to test for problems involving inheritance, polymorphism and dynamic binding.

NPTEL

In the first part of this which I taught in the last lecture, I introduced you to generic mutation operators which talked about changing a calling method in more than one ways, change a called method and more than one ways.

(Refer Slide Time: 00:49)



### Mutation operators for integration mutation

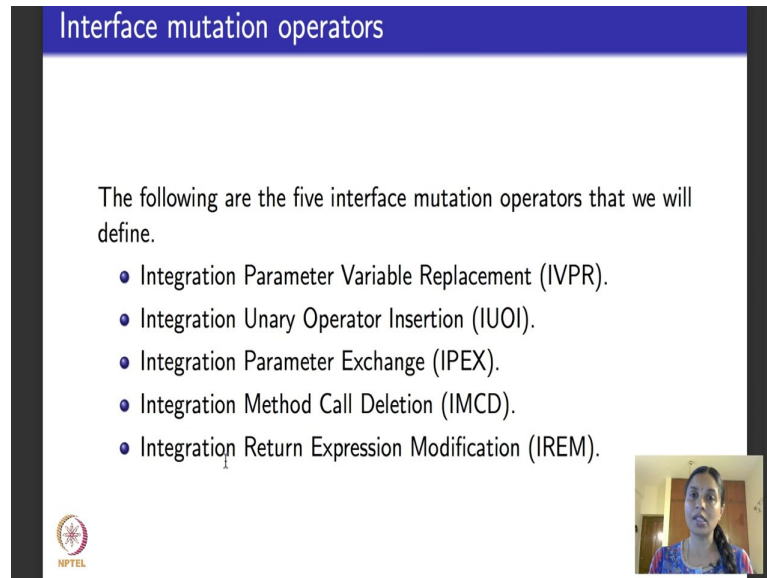
Integration mutation operators do the following:

- Change a calling method by modifying the values that are sent to a called method.
- Change a calling method by modifying the call.
- Change a called method by modifying the values that enter and leave a method.
  - This should include parameters and variables from a higher scope (class level, package, public etc.)
- Change a called method by modifying statements that return from the method.

NPTEL

We discussed these five different generic mutation operators that could be done for integration testing for almost every programming language that supports procedure calls or method calls.



(Refer Slide Time: 00:56)



### Interface mutation operators

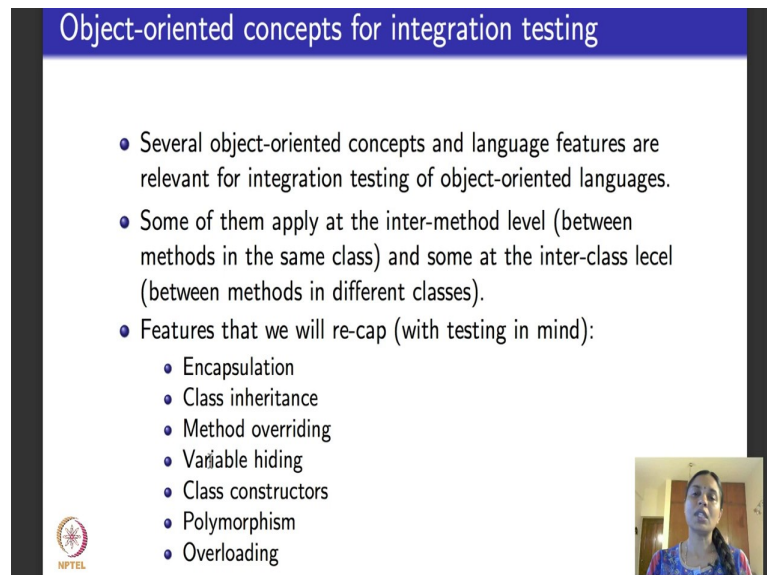
The following are the five interface mutation operators that we will define.

- Integration Parameter Variable Replacement (IVPR).
- Integration Unary Operator Insertion (IUOI).
- Integration Parameter Exchange (IPEX).
- Integration Method Call Deletion (IMCD).
- Integration Return Expression Modification (IREM).





Later on, in the last lecture after doing this what I told you was we said we will focus on object oriented integration testing because object oriented languages have picked up. Lot of web applications, enterprise applications are written in object oriented programming languages.

(Refer Slide Time: 01:11)



### Object-oriented concepts for integration testing

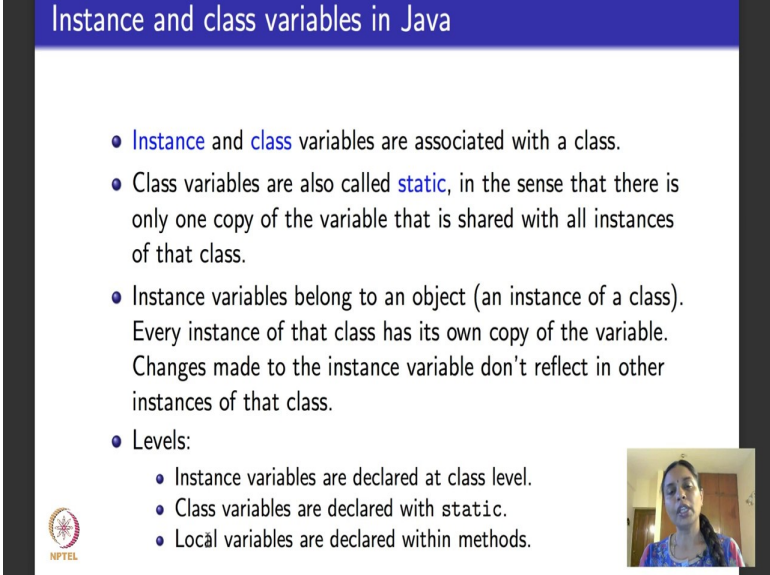
- Several object-oriented concepts and language features are relevant for integration testing of object-oriented languages.
- Some of them apply at the inter-method level (between methods in the same class) and some at the inter-class level (between methods in different classes).
- Features that we will re-cap (with testing in mind):
  - Encapsulation
  - Class inheritance
  - Method overriding
  - Variable hiding
  - Class constructors
  - Polymorphism
  - Overloading



So, we said will focus specifically on mutation operators that are available for object oriented programming languages especially Java. So, with that towards recapping what we need to understand mutation, we recap-ed the essential object oriented features in the

end of last lecture. So, I helped you to recap encapsulation, class inheritance, overriding methods, variable hiding, constructors, polymorphism and overloading.

(Refer Slide Time: 01:59)



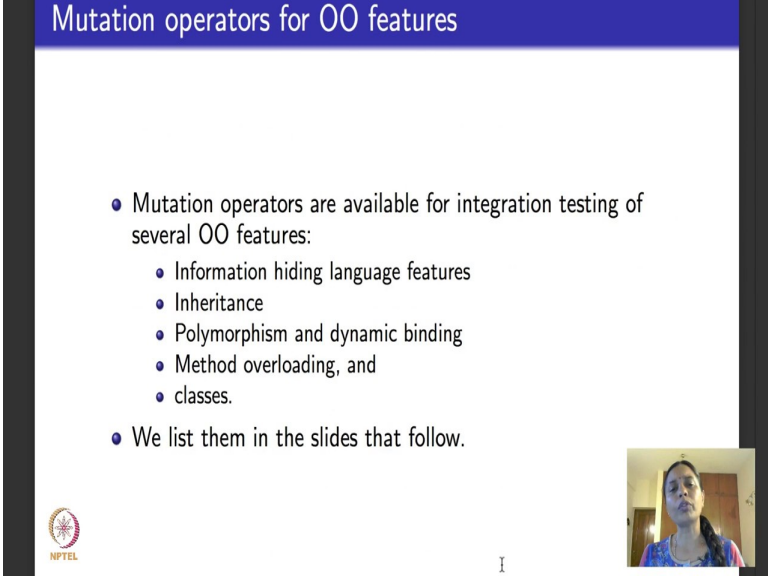
The slide is titled "Instance and class variables in Java". It contains a list of bullet points explaining the concepts of instance and class variables. In the bottom right corner, there is a small video inset showing a person speaking. The NPTEL logo is visible in the bottom left corner of the slide content area.

- Instance and class variables are associated with a class.
- Class variables are also called **static**, in the sense that there is only one copy of the variable that is shared with all instances of that class.
- Instance variables belong to an object (an instance of a class). Every instance of that class has its own copy of the variable. Changes made to the instance variable don't reflect in other instances of that class.
- Levels:
  - Instance variables are declared at class level.
  - Class variables are declared with **static**.
  - Local variables are declared within methods.

So, we did this last time. So, I will skip through these slides. We will directly move into looking at Java and what are the specific mutation operators that are available for focusing and doing integration testing for a language like Java. So, to do that one last concept that I would like to specifically recap related to Java is the notion of instance variables and class variables. So, instance and class variables, both are variables that are associated with a class in Java.

What is the difference, we will see the difference class variables are also called static variables, they are declared using the word static. They are static in the sense that there is only one copy of the variable that is shared with all the instances of the class. Instance variables on the other hand, they belong to an object or an instance of a class and every instance of the class has its own copy of the variable. Changes made to an instance variable do not reflect, suppose you have one copy of instance variable in one object, you make a change it does not reflect on all the other copies of instance variable. Like we had access levels for encapsulation instance variables where are they declared they are declared at class level. Class variables are declared with a keyword static as I told you and in addition to these two we also have local variables which are declared within methods in a class.

(Refer Slide Time: 03:13)



The slide has a blue header with the text "Mutation operators for OO features". Below the header, there is a bulleted list. The first bullet point states that mutation operators are available for integration testing of several OO features, followed by a sub-list of five features: information hiding language features, inheritance, polymorphism and dynamic binding, method overloading, and classes. The second main bullet point states that these features are listed in the following slides. In the bottom left corner, there is an NPTEL logo. In the bottom right corner, there is a small video inset showing a woman speaking.

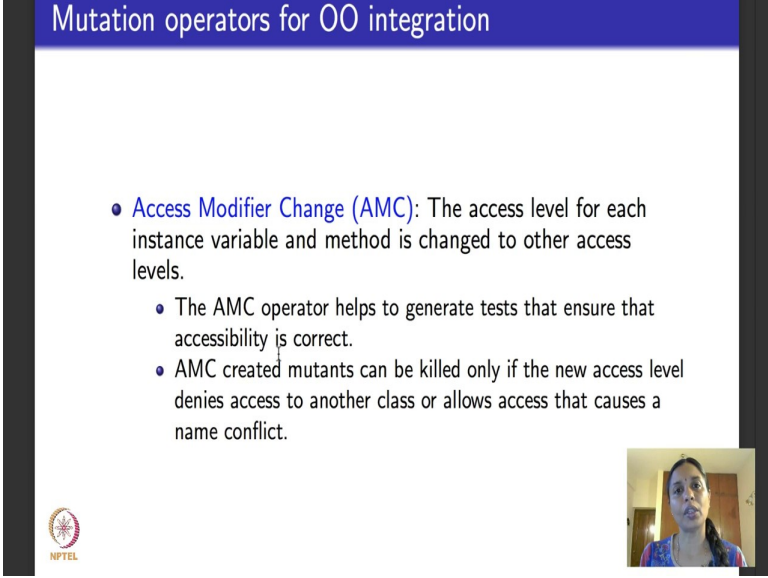
- Mutation operators are available for integration testing of several OO features:
  - Information hiding language features
  - Inheritance
  - Polymorphism and dynamic binding
  - Method overloading, and
  - classes.
- We list them in the slides that follow.

NPTEL

Now what we are going to see we are going to see mutation operators for doing integrations testing specific to object oriented features. Without loss of generality you can assume that we will be focusing on mutation operators for Java, but equally well applies to any other object for oriented programming that shares the same kind of object oriented features for integrating different pieces of code into one large code. So, the following are the generic features that we will be seeing. We will instantiate them with reference to Java, but let us say you see C++ and then it has this feature you could write you could as well use these mutation operators to be able to do integration testing for C++.

So, we will see information hiding related features, we will see inheritance related mutation operators, we will see mutation operator is related to polymorphism, dynamic binding, method overloading and classes. So, in the slides that follow, I will be listing about 20 different mutation operators. All of them are meant for testing integration of object oriented programs.

(Refer Slide Time: 04:21)



The slide has a blue header with the text "Mutation operators for OO integration". Below the header, there is a bulleted list. The first bullet point is "Access Modifier Change (AMC): The access level for each instance variable and method is changed to other access levels." followed by two sub-bullets: "The AMC operator helps to generate tests that ensure that accessibility is correct." and "AMC created mutants can be killed only if the new access level denies access to another class or allows access that causes a name conflict." In the bottom left corner, there is a circular logo with a star and the text "NPTEL". In the bottom right corner, there is a small video inset showing a woman speaking.

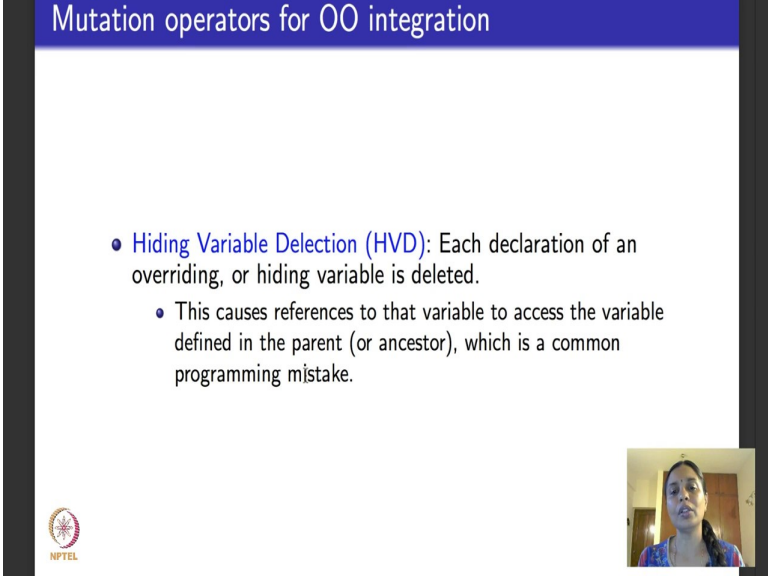
- **Access Modifier Change (AMC):** The access level for each instance variable and method is changed to other access levels.
  - The AMC operator helps to generate tests that ensure that accessibility is correct.
  - AMC created mutants can be killed only if the new access level denies access to another class or allows access that causes a name conflict.

So, for each of these operators, I will tell you what it does and then wherever applicable I will also give you one instance of what is the kind of error that it could be useful to detect while doing design integration. Obviously, when you test your program for design integration you are not going to be able to use all of these operators, this is meant to be used as an exhaustive list. What we are going to use is a select set, as and when we need based on what our focus for integration testing is.

So, the first operator that I am going to tell you is called access modifier change abbreviated as AMC. Do not worry too much about remembering these abbreviations. I just carry them forward because it is useful to illustrate them as you are going on. We really do not need to remember them for any reason at all. What is this operator do? This operator basically changes the access levels. What are the access levels that it changes, it changes the access levels for each instance variable and method.

What do we achieve by this? We achieve and get tests that ensure that accessibility is correctly done during integration. The access modifier change operator creates mutants that can be killed only if the new access level denies access to other classes or it is the opposite, it allows access to other classes which causes some kind of conflict in 'a'. That is when you can observe a change in the behaviour and then you say applying this mutation operator killed the mutant.

(Refer Slide Time: 05:50)



The slide has a blue header with the text "Mutation operators for OO integration". Below the header, there is a bulleted list. The first bullet point is "Hiding Variable Deletion (HVD): Each declaration of an overriding, or hiding variable is deleted." The second bullet point is "This causes references to that variable to access the variable defined in the parent (or ancestor), which is a common programming mistake." In the bottom left corner, there is a circular logo with a star and the text "NPTEL". In the bottom right corner, there is a small video inset showing a woman speaking.

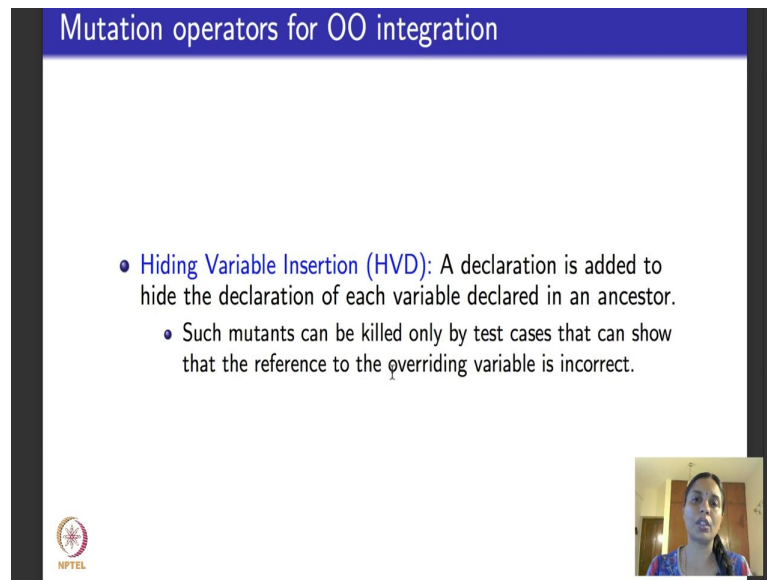
- **Hiding Variable Deletion (HVD):** Each declaration of an overriding, or hiding variable is deleted.
  - This causes references to that variable to access the variable defined in the parent (or ancestor), which is a common programming mistake.

So, the next mutation operator, in fact, all the mutation operators that we will be seeing will only do three kinds or four kinds of changes they will delete something, they will remove something, they will insert something, they will modify something.

So, the first one that we saw was a modification, the next one that we are going to see is deletion. What is this delete? This deletes what are called overriding or hiding variables. The operator is called hiding variable deletion. By just reading the name you should be able to figure out what it is supposed to do it cannot be too difficult. What is this mutation operator do? By hiding or I mean by deleting the overriding or hiding variable we cause references to that variable to access the variable defined in a parent or in any ancestor this could be a common programming mistake that can be caught during integration testing.

The next mutation operator that I am going to tell you specific to Java is an insert mutation operator.


(Refer Slide Time: 06:41)



Mutation operators for OO integration

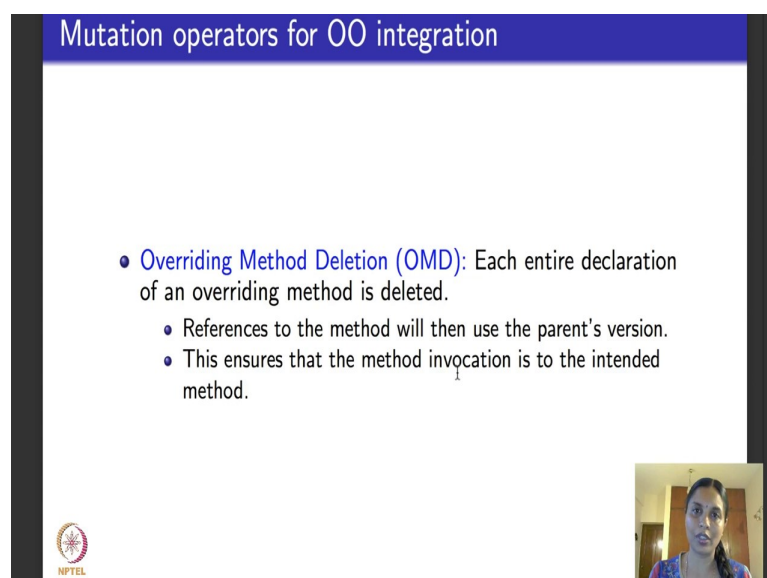
- **Hiding Variable Insertion (HVD):** A declaration is added to hide the declaration of each variable declared in an ancestor.
  - Such mutants can be killed only by test cases that can show that the reference to the overriding variable is incorrect.

NPTEL



In the previous one we showed you about deleting hiding or overriding variables. Here what we are going to do is, we add a new declaration. What are we going to do, what are the declaration do that declaration has got to do with hiding variables. So, declaration is added to hide the declaration of each variable declared in an ancestor. What does this mutant do? This mutant basically hides the variables that are declared in the ancestor and it tests whether the overriding reference variable, variable reference is correct or not. So, these mutants can be killed by test cases that show that the reference to the overriding variable is not correct.


(Refer Slide Time: 07:24)



Mutation operators for OO integration

- **Overriding Method Deletion (OMD):** Each entire declaration of an overriding method is deleted.
  - References to the method will then use the parent's version.
  - This ensures that the method invocation is to the intended method.

NPTEL

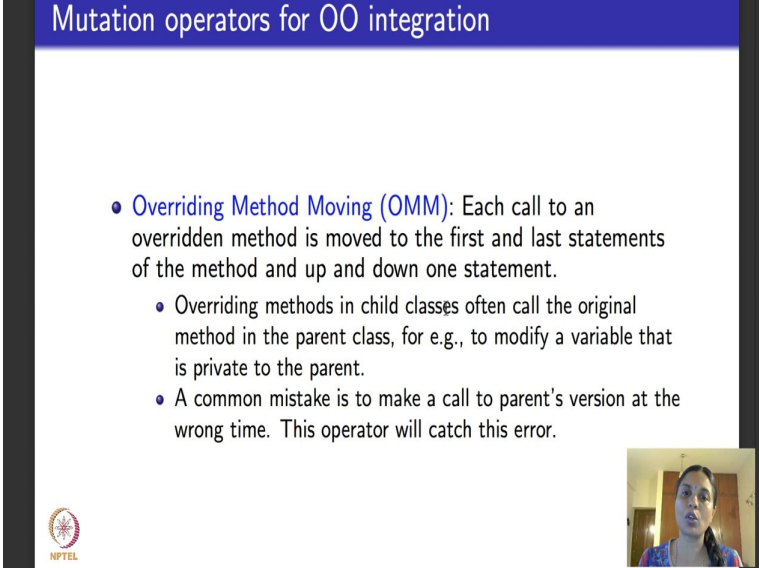




So, the next mutation operator in our long list of mutation operators that we are going to see, what does it do, it is a deletion mutation operator. Instead of working with variables it works with methods. It is called overriding method deletion abbreviated as OMD, what does it do? It removes the entire declaration of the overriding method. Please note we are not removing the method, we are removing the declaration of the method.

What will then happen? The references to this method that happen in a class will then use the parent's version right. So, if this happens then it ensures that the method invocation is indeed to the correct method and not to the wrong one. By deleting if there is a problem you will be able to find out because the method itself is deleted. So, if there was a correct reference there will be an issue, you will be able to kill the mutant and observe an error. But if there was an incorrect method, by deleting this method reference deleting the method declaration you will not be able to observe a change so that means, you will not be able to kill a mutant. The fact that you cannot kill a mutant reveals an error in this case.

(Refer Slide Time: 08:29)



The slide is titled "Mutation operators for OO integration" in a blue header. It contains a bulleted list describing the Overriding Method Moving (OMM) operator. In the bottom right corner, there is a small video inset showing a woman speaking. The NPTEL logo is in the bottom left corner.

- **Overriding Method Moving (OMM):** Each call to an overridden method is moved to the first and last statements of the method and up and down one statement.
  - Overriding methods in child classes often call the original method in the parent class, for e.g., to modify a variable that is private to the parent.
  - A common mistake is to make a call to parent's version at the wrong time. This operator will catch this error.

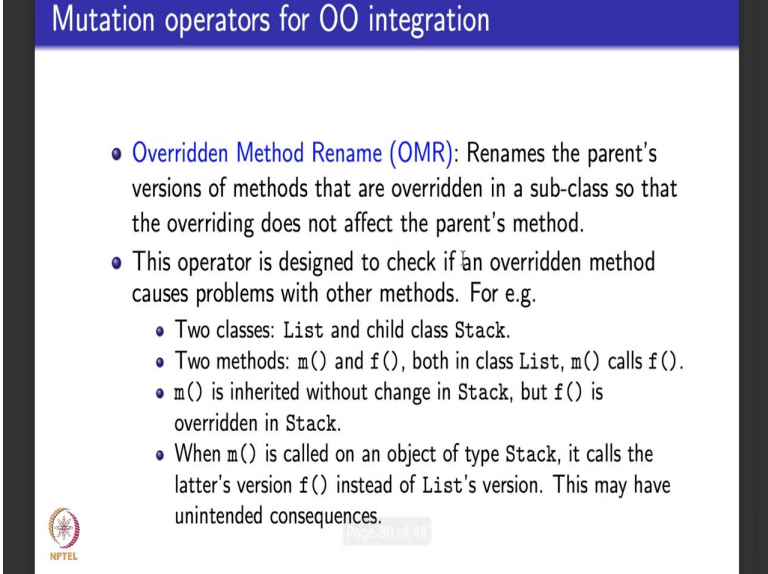
The next operators called overriding method moving abbreviated as OMM. What is it do? Each call to an overridden method is moved to the first and to the last statements of the method.

And wherever it is in the program, it is moved up by one statement, its moved down by one statement. The previous one we deleted the declaration of the method. Here were not

deleting it, instead there is a particular line in which the method is declared. What we do in the declaration of the call, we move its location within the program source code. We move it to the first statement, we move it to be the last statement, we move it one statement up, we move it one statement down.


What will this test? This will test the following. Typically overriding methods in child classes they often call the original method in the parent class for some reason. For example, they could call it to modify a variable that is private to the parent. There could be other reasons also. One common mistake is to make the call to the parents version at the wrong time. You called too early, maybe by the time you called it was too late to use the variable. So, moving around the call the statement of the call, we will be able to correctly detect whether it was called at the right time. So, killing such a mutant will detect errors related to where the call is made. If it is made at an inappropriate place it will be obviously detected. So, the next operator also deals with overridden methods. So, I will go back, we deleted overriding methods, we moved overriding methods.

(Refer Slide Time: 10:09)



**Mutation operators for OO integration**

- **Overridden Method Rename (OMR):** Renames the parent's versions of methods that are overridden in a sub-class so that the overriding does not affect the parent's method.
- This operator is designed to check if an overridden method causes problems with other methods. For e.g.
  - Two classes: List and child class Stack.
  - Two methods: `m()` and `f()`, both in class List, `m()` calls `f()`.
  - `m()` is inherited without change in Stack, but `f()` is overridden in Stack.
  - When `m()` is called on an object of type Stack, it calls the latter's version `f()` instead of List's version. This may have unintended consequences.

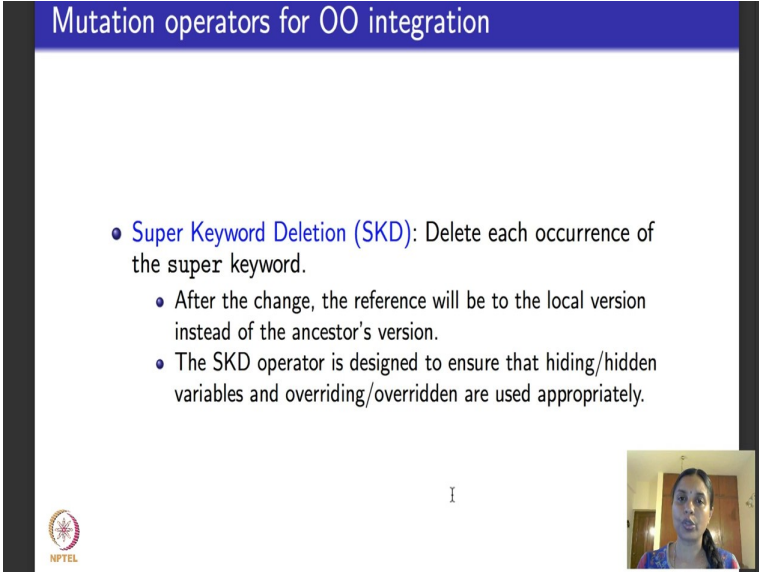
 NPTEL

Now, we are going to rename overriding methods, overridden method rename abbreviated as OMR. What does this do? It renames the parent's versions of the methods that are overridden in a subclass so that overriding does not affect the parent's method. What is this operator designed to check for? This mutation operator basically checks if an overridden method actually causes problems with other methods. So, here is an

illustrative example. Let us say there are two classes called list and stack. List is the parent class, list has a child class called stack. Maybe list is a list operations and stack is one of the ways of implementing a list, whatever it is. So, there are two classes: list, parent class and its child class called stack.

And there are two methods, a method called m a method called f. Both of them are present in the class list and in the class list method m calls the method f. In addition method m and f are also present in the class stack. How are they present in the class stack? m is inherited without a change in the class stack, but f is overridden in the class stack. So, now, when I rename one of them what will happen? When m is called on an object of let us say type stack, it calls the stack's version of f instead of the list's version. This kind of problems I can detect by using this mutation operator, I hope this is clear.

(Refer Slide Time: 11:40)



The slide is titled "Mutation operators for OO integration" in a blue header. It contains a bulleted list with the following text:

- **Super Keyword Deletion (SKD):** Delete each occurrence of the super keyword.
  - After the change, the reference will be to the local version instead of the ancestor's version.
  - The SKD operator is designed to ensure that hiding/hidden variables and overriding/overridden are used appropriately.

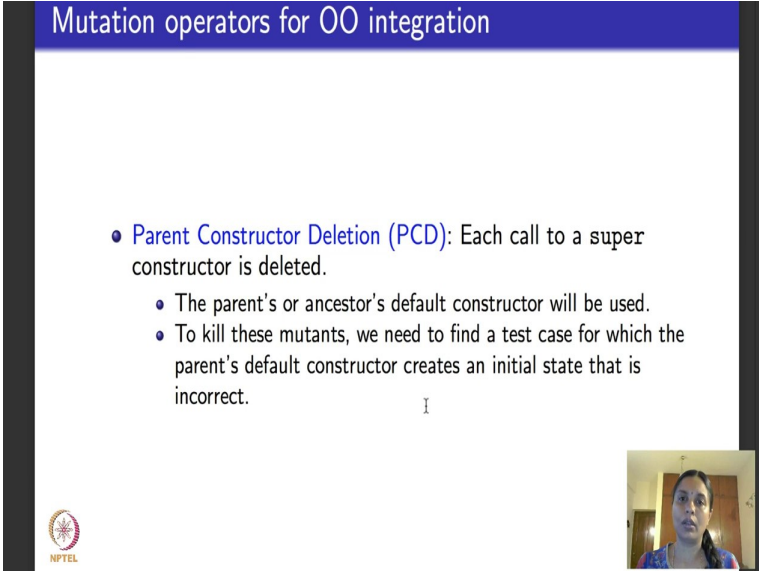
In the bottom left corner is the NPTEL logo, and in the bottom right corner is a small video inset showing a woman speaking.

Moving on the next mutation operator we are going to see relates to this key word called super. If you remember we used super, why did we use it? We used it to refer to appropriate variables in an ancestor right, in an ancestor class. That is the context in which I had introduced this when I told you about object oriented features in the last lecture. So, super keyword deletion, what does it do? It just simply deletes the super keyword, which means what? After deletion the reference will be to the local version of the variable instead of the ancestor's version. So, without, so let us say if you had super

dot x, you delete the word super and directly refer to x. So, the x that I am referring to now is in the local version and not in the ancestor's version.

So, what kind of errors can this reveal? This can reveal errors that ensure that hiding or hidden variables or overriding or overridden methods or variables are used appropriately. So, if it was meant to be actually from the ancestors class, not the local version, then deleting it will use the local version and reveal a potential error.


(Refer Slide Time: 12:48)



Mutation operators for OO integration

- **Parent Constructor Deletion (PCD):** Each call to a super constructor is deleted.
  - The parent's or ancestor's default constructor will be used.
  - To kill these mutants, we need to find a test case for which the parent's default constructor creates an initial state that is incorrect.

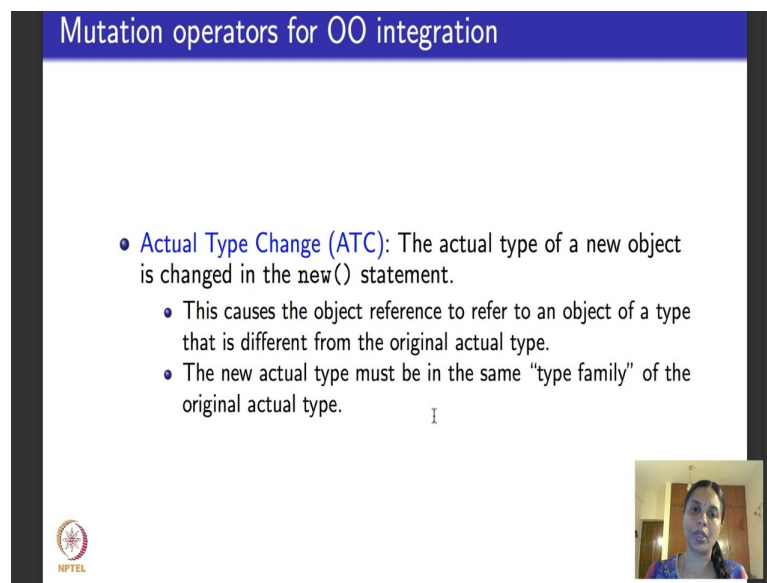
NPTEL



So, the next one next mutation operator is called parent constructor deletion. So, it deals with constructor. So, each call to the super constructor is deleted as the mutation operator describes. What happens in this case? In this case after deletion, the parents or the ancestors default constructor would be used.

Now to kill such mutants which delete the super constructor operator, what do we need to do? We need to find a test case for which the parents default constructor creates an initial state that is incorrect because that is when I will be able to observe an error about using the parents or an ancestors constructor. Otherwise I will not be able to observe any error.

(Refer Slide Time: 13:31)



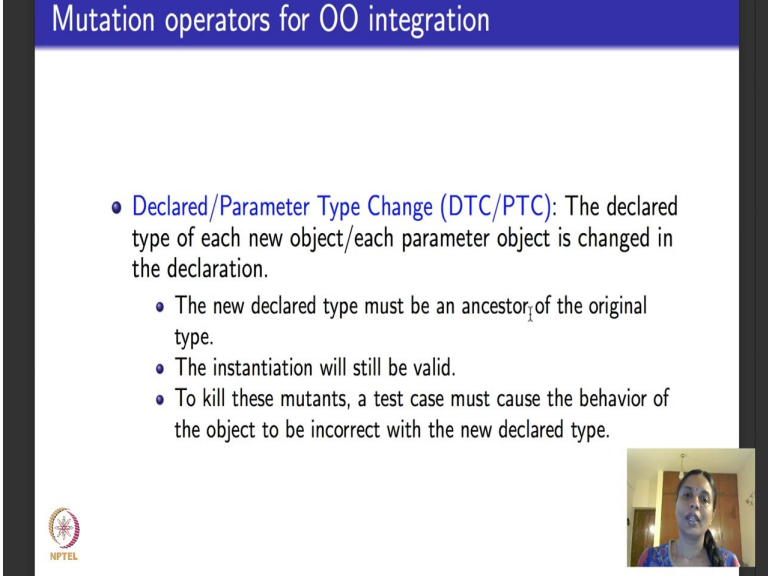
The slide is titled "Mutation operators for OO integration" in a blue header. It contains a bulleted list defining "Actual Type Change (ATC)". The list states that the actual type of a new object is changed in the `new()` statement, and provides two sub-points: that this causes the object reference to refer to an object of a different type, and that the new type must be in the same "type family" as the original. In the bottom right corner, there is a small video inset showing a woman speaking. The NPTEL logo is in the bottom left corner.

- **Actual Type Change (ATC):** The actual type of a new object is changed in the `new()` statement.
  - This causes the object reference to refer to an object of a type that is different from the original actual type.
  - The new actual type must be in the same "type family" of the original actual type.

So, the next mutation operator changes types. Whose type that is changed, it changes the actual type of a new object, where does it change? It uses this new statement that is available in Java to be able to change the type of the new object.

What kind of problem this causes? This, if you do this then it will cause the object reference to refer to an object of type that is different from the original actual type. Is that clear, because I have changed it now in the new statement. So, now what will happen, the new actual type that it is acquiring must be the same, in the same type family of the original actual type,. I need to ensure this because you remember the good old thing about mutation. After mutating a program, the program must compile it should not be syntactically illegal. So, type change, if you do, then the type change better be consistent, so that the mutation can be applied through a ground string to get a new program that can still combine and then we can worry about whether we can strongly kill it or weakly kill it and so on.

(Refer Slide Time: 14:31)



The slide has a blue header with the text "Mutation operators for OO integration". Below the header, there is a bulleted list. The first bullet point is "Declared/Parameter Type Change (DTC/PTC): The declared type of each new object/each parameter object is changed in the declaration." It has three sub-bullets: "The new declared type must be an ancestor of the original type.", "The instantiation will still be valid.", and "To kill these mutants, a test case must cause the behavior of the object to be incorrect with the new declared type." 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 person speaking.

- **Declared/Parameter Type Change (DTC/PTC):** The declared type of each new object/each parameter object is changed in the declaration.
  - The new declared type must be an ancestor of the original type.
  - The instantiation will still be valid.
  - To kill these mutants, a test case must cause the behavior of the object to be incorrect with the new declared type.

So, the next operator in our long list is what is called declared or parameter type change. This one slide actually contains two mutation operators. Both the mutation operators are about changing a type. One changes the type of a declared type, one changes the type of a parameter. So, the declared type of each new object or each parameter object type is changed in the declaration. So, what will happen? The new declared type must be an ancestor of the original type and the instantiation will still be valid because that is how we do it.

To kill such mutants for where the declared type of a object or the parameter is changed, what should a test case do? It must cause the behaviour of an object to be incorrect with reference to the new change that I have made to the type. So, if that happens then I know that there is a problem with my original declaration or maybe my original declaration was fine. So, that is the integration feature that I test.

(Refer Slide Time: 15:28)

The slide is titled "Mutation operators for OO integration" in a blue header. It contains a bulleted list explaining the Reference Type Change (RTC) operator. The list includes a definition of RTC and two examples. In the bottom right corner, there is a small video inset of a woman speaking. The NPTEL logo is in the bottom left corner.

- **Reference Type Change (RTC):** This right side objects of assignment statements are changed to refer to objects of a compatible type.
  - For e.g., if an Integer is assigned to a reference of type Object, the assignment may be changed to that of a String.
  - Since both are descended from Object, both can be assigned interchangeably.

So, the next one that we are going to see is also a type change, but this is a type change called reference type change. What does this do? Here the reference is changed, which means the right side objects of assignment statements are changed to refer to objects of a compatible type. They reference right hand side of the assignment statement or the reference statement, the type of that is changed. Not the entire reference, but only its type.



For example, if an integer is assigned to a reference of type object then you could make a type change and change it to that of a string. In general it may not be allowed, integer changing to string there will be big problems related to compiling, but here it will work fine because both have descended from object, so both can be assigned interchangeably. And this will again detect any errors that relates to changing of types and the types of references.

Moving on, the next operator that we are going to deal with is related to overloading methods. In fact, we will work with overloading method change here, overloading method deletion here, two operators.

(Refer Slide Time: 16:24)

Mutation operators for OO integration

- **Overloading Method Change (OMC):** For each pair of methods that have the same name, the bodies are interchanged.
  - This ensures that overloaded methods are invoked appropriately.





So, what is overloading method change do? For each pair of methods that have the same name the bodies are interchanged, which means the code that corresponds to these methods, the line entire code that corresponds to these methods, they are swapped, they are interchanged. Then what is this will test, what sort of error this will test? This will test any error that relates to overloading methods being invoked properly If they are not invoked correctly at the same time then one is invoked instead of the other you will be able to kill the mutant and detect any errors related to proper invocation of overloaded methods.

(Refer Slide Time: 17:14)

Mutation operators for OO integration

- **Overloading Method Deletion (OMD):** Each overloaded method declaration is deleted, one at a time.
  - This operator ensures coverage of overloaded methods— all of them must be invoked at least once.
  - If the mutant still works correctly without the deleted method, there may be an error in invoking one of the overloading methods— the incorrect method may be invoked or an incorrect parameter type conversion has occurred.



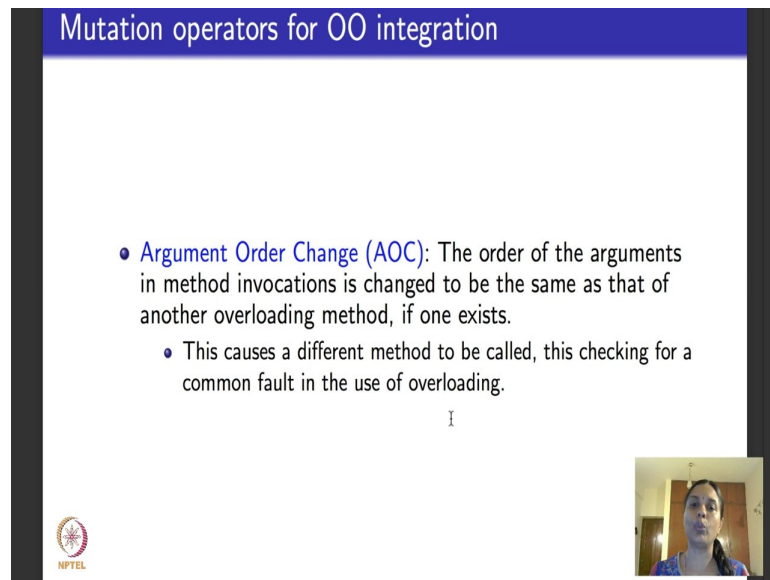


So, next operator also deals with overloading methods. Instead of changing them it deletes them. So, the operators called overloading method deletion abbreviated as OMD. What is this one do? Each overloaded method declaration is deleted. Please remember there is no point, suppose I have more than one overloaded methods in one shot I cannot apply this mutation operator and delete more than one. That will violate the problem of applying one mutation operator at a time to be able to mutate a ground string. This mutation operator can be applied to delete any overloaded method declaration but not more than one. If you apply it to more than one deletion then it will mean that you are deleting more than one at a time, that is not allowed. So, you delete each overloaded method declaration one at a time.

Choose which one to delete based on what you want to focus and test. What will this operator ensure? This operator will ensure coverage of overloaded methods. Why, because I delete then what will happen without it, then I test that. Then I put this one that I deleted back and delete something else then I test what happens without what second one that I deleted right. So, all the methods must be invoked once as and when they take turns to get deleted.

If a mutant still works correctly without the deleted method then it could be an error, it could be an error in invoking one of the overloading methods while doing integration. The incorrect method may be invoked or an incorrect parameter type conversion has happened. If nothing happens after deleting one particular method then maybe it was a piece of dead code and you could do without it right. So, if you get an equivalent mutant then it means that the method was dead code and you could do without it otherwise you will definitely be able to kill the mutant and observe one of these kind of errors.

(Refer Slide Time: 19:08)



The slide has a blue header with the text "Mutation operators for OO integration". Below the header, there is a bulleted list. The first bullet point is "Argument Order Change (AOC): The order of the arguments in method invocations is changed to be the same as that of another overloading method, if one exists." The second bullet point is "This causes a different method to be called, this checking for a common fault in the use of overloading." Below the list, there is a small "I" character. In the bottom right corner, there is a small video inset showing a woman speaking. In the bottom left corner, there is a small NPTEL logo.

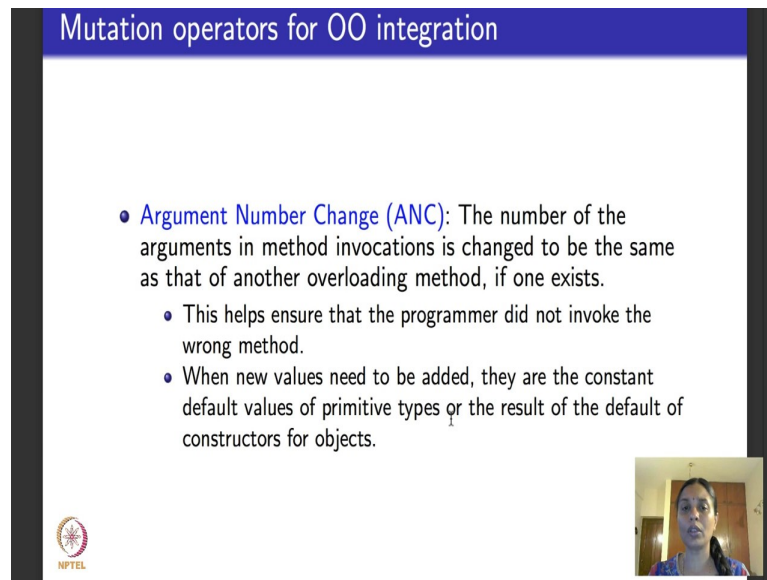
- **Argument Order Change (AOC):** The order of the arguments in method invocations is changed to be the same as that of another overloading method, if one exists.
  - This causes a different method to be called, this checking for a common fault in the use of overloading.

I

So, the next operator we will see changes the order in which arguments are passed during method invocation. It is called argument order change abbreviated as AOC, what does this do? The order of the arguments in a method invocation is changed to be the same as that of another overloading method if it exists. This causes a different method to be called because I am changing the arguments and hence it checks for common fault in the use of overloading. What happens? Because something could be called out of order, I have changed completely, so then, if there is an error because of the use of overloading I will be able to catch it.



So, the next mutation operator also deals with arguments.

(Refer Slide Time: 19:45)



**Mutation operators for OO integration**

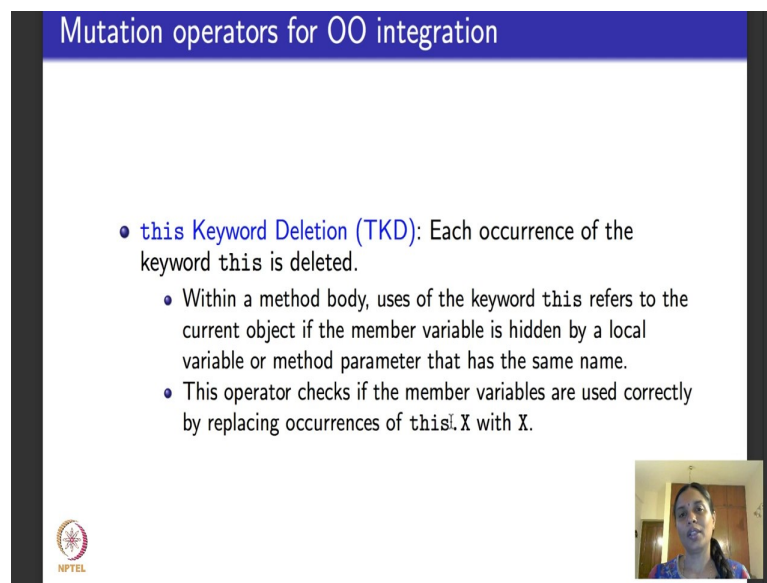
- **Argument Number Change (ANC):** The number of the arguments in method invocations is changed to be the same as that of another overloading method, if one exists.
  - This helps ensure that the programmer did not invoke the wrong method.
  - When new values need to be added, they are the constant default values of primitive types or the result of the default of constructors for objects.

But here we didn't change the order of arguments, here we change the number of arguments. Lets say if you had three, you could make that two, you could make that four that is what we mean change the number of arguments.

So, number of arguments in method invocations is change to be the same as that of another overloading method provided there is one such method that exists and you could replace it with that. What will this help? This will help to check that the programmer indeed invoked the correct method, he did not pick up the wrong method to invoke and when new values have to be added they are constant default values of primitive types or they are the result of default of constructors for object, is that clear.

(Refer Slide Time: 20:31)



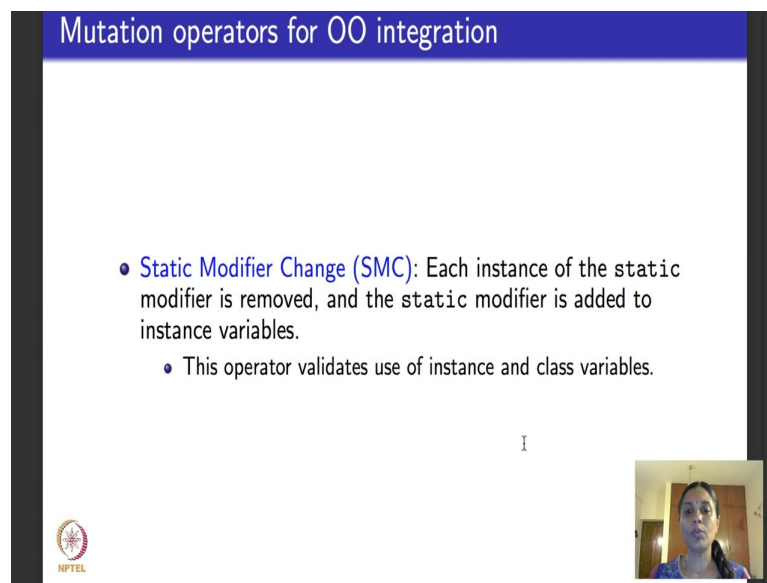
The slide has a blue header with the text "Mutation operators for OO integration". Below the header, there is a bulleted list. The first bullet point is "this Keyword Deletion (TKD): Each occurrence of the keyword this is deleted." followed by two indented bullet points. The first indented bullet point says "Within a method body, uses of the keyword this refers to the current object if the member variable is hidden by a local variable or method parameter that has the same name." The second indented bullet point says "This operator checks if the member variables are used correctly by replacing occurrences of this.X with X." 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 woman with dark hair, wearing a blue top, speaking.

- **this Keyword Deletion (TKD):** Each occurrence of the keyword `this` is deleted.
  - Within a method body, uses of the keyword `this` refers to the current object if the member variable is hidden by a local variable or method parameter that has the same name.
  - This operator checks if the member variables are used correctly by replacing occurrences of `this.X` with `X`.

So, moving on the next operator that we are going to see, we still have a long way to go remember I told you 20 operators. So, were probably around 15. So, we still have a good number of operators to go.

So, the next operator that we are going to see deletes this keyword called `this`. You remember Java has this keyword called `this`. What is the keyword `this` do? Within a method body, when I use a keyword `this` it refers to the current object of the member variable is hidden by a local variable or a method parameter that has the same name. Now if I delete it, then what will happen it checks basically, when I apply this mutation the resulting program if I am able to kill it then I am basically checking if the member variables are used correctly by replacing every occurrence of `this dot x` with `x`. So, here unlike the earlier one, when I say this keyword deletion, you have to be careful you have to delete more than one keyword of `this` because if you keep one and retain one they could be problems. So, sometimes you might have to delete one of this one, but if there is more than one related one, you have to delete all of them.


(Refer Slide Time: 21:38)



Mutation operators for OO integration

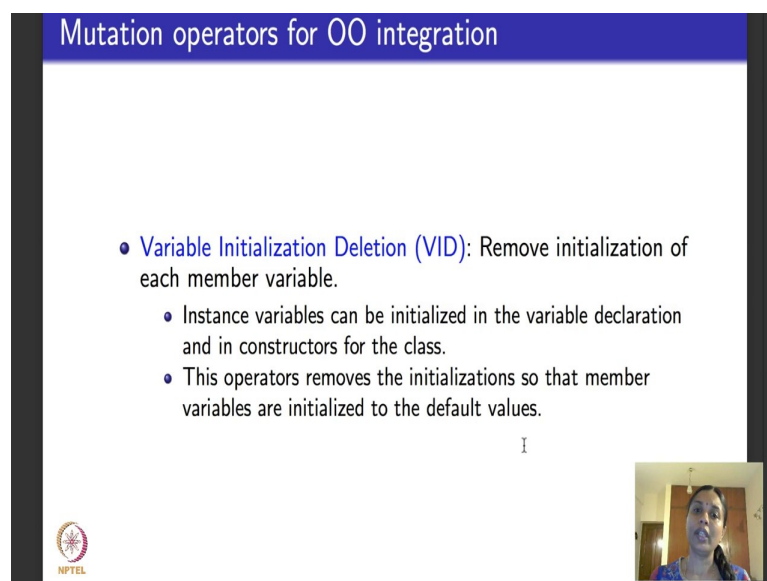
- **Static Modifier Change (SMC):** Each instance of the static modifier is removed, and the static modifier is added to instance variables.
  - This operator validates use of instance and class variables.

NPTEL



Then the next mutation operator is called static modifier change abbreviated as SMC. What happens here? Remember static variables, I told you right in the beginning instance variables and class variables of Java, class variables are static variables. So, each instance of a static modifier is removed and the static modifier is instead added to the instance variables. What will happen? This basically checks if the various instance variables and class variables that have been declared around a piece of code are declared correctly and are being used at the appropriate piece.


(Refer Slide Time: 22:13)



Mutation operators for OO integration

- **Variable Initialization Deletion (VID):** Remove initialization of each member variable.
  - Instance variables can be initialized in the variable declaration and in constructors for the class.
  - This operators removes the initializations so that member variables are initialized to the default values.

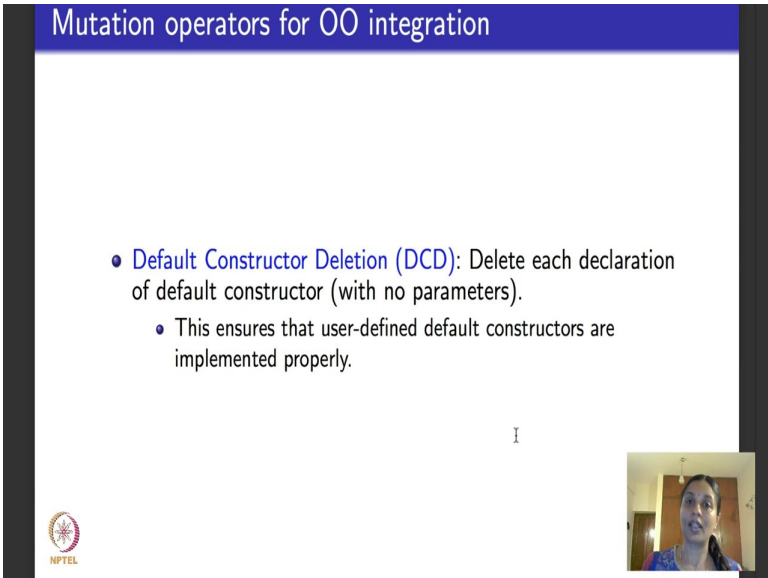
NPTEL



So, the next one variable initialization deletion VID, what does it do? It removes initializations of each member variable of a method or class. They are not initialized at all. If they are not initialized, what are the kind of errors that you could detect? Please remember instance variables can be initialized in the variable declaration or in the constructors for the class.

So, when I remove variables, this operator removes the initialization so that member variables are initialized to default values and it basically checks if the program will still run fine with reference to the initializations at the default values.

(Refer Slide Time: 22:49)



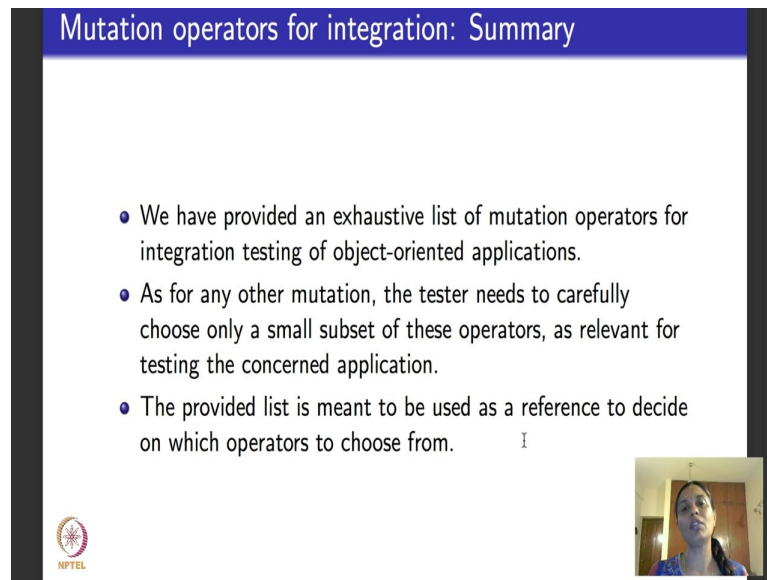
The slide is titled "Mutation operators for OO integration" in a blue header. It contains a bulleted list with two items. The first item is "Default Constructor Deletion (DCD): Delete each declaration of default constructor (with no parameters)." and the second item is "This ensures that user-defined default constructors are implemented properly." In the bottom left corner, there is an NPTEL logo. In the bottom right corner, there is a small video inset showing a woman speaking.

- **Default Constructor Deletion (DCD):** Delete each declaration of default constructor (with no parameters).
  - This ensures that user-defined default constructors are implemented properly.

NPTEL

So, the next, the last one, in our really long list of mutation operators what does it do, it is called a default constructor deletion DCD. It deletes each declaration of the default constructor with no parameters. What can this ensure? This ensures that the user defined default constructors are implemented properly. Suppose I delete and the delete, after deletion, the program still works fine which means it is an equivalent mutant then clearly I may not need it right. There is something wrong with my implementation. Suppose I delete an after deletion my program thus have a problem then I have implemented it correctly at least at the integration level.

(Refer Slide Time: 23:30)



The slide has a blue header with the text "Mutation operators for integration: Summary". Below the header, there are three bullet points in blue text. In the bottom right corner of the slide, there is a small rectangular video inset showing a woman with dark hair, wearing a blue top, speaking. In the bottom left corner of the slide, there is a small circular logo with a star and the text "NPTEL" below it.

- We have provided an exhaustive list of mutation operators for integration testing of object-oriented applications.
- As for any other mutation, the tester needs to carefully choose only a small subset of these operators, as relevant for testing the concerned application.
- The provided list is meant to be used as a reference to decide on which operators to choose from.

So, what did we do? We provided an exhaustive list. It must be tiring it must be a little confusing in fact, to go through such a big list. You might want to play this video once again to get the list right. But intuitively what it tells you, the list that we provided of twenty different mutation operators, it basically focuses on entities related to integrating methods, classes inheriting from each other, overloading, overriding, their access levels and it makes changes to all the places that focus on putting these together and tells you an exhaustive way of mutating to test this integration.

Obviously, when you have a piece of code that you want to test for design integration, you are not going to be able to consider all of these operators, not even half of them. Carefully based on the integration feature that you want to test, you will pick up one or two of these operators, maybe a few more and apply them to see the feature that you are going to test. So, the choice of the operator is based on the feature that you want to test and that is completely to be decided by you.

The focus of this lecture was to be able to provide an exhaustive listing to help you choose from this varied variety of choices that are available. So, I hope this exhaustive listing of mutation operators for object oriented integration testing helps you. You could use it when you are integrating java programs or C++ programs where all these features are available.

In the next lecture we will move on to using mutation testing for specifications, input grammars and XML models of inputs.

Thank you.