

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

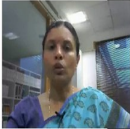

Lecture – 41
Mutation Testing

Welcome to week 9, we continue with mutation testing this week. This will be the first uploaded lecture for week 9 now. What are we going to do today? We will begin by recapping what we have planned to do for mutation testing. If you remember I have showed you this figure last week in one of my lectures, I say this is an overview of what we are going to do for mutation testing.

(Refer Slide Time: 00:21)

Mutation testing for software artifacts: An overview				
	For programs	Integration	Specifications	Input space
BNF grammar	Programming languages	–	Algebraic specifications	Input languages like XML
Summary	Compilers			Input space testing
Mutation	Programs	Programs	FSMs	Input languages like XML
Summary	I Mutates programs	Tests integration	Model checking	Error checking

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



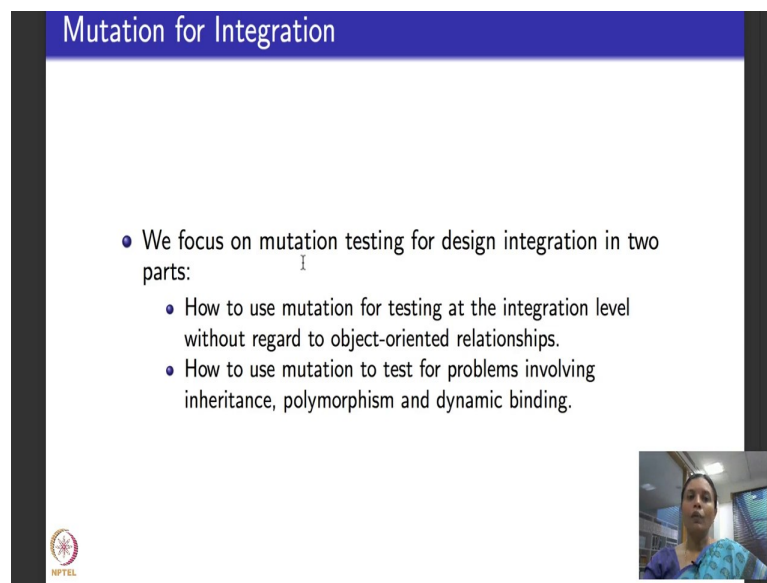
Two kinds of mutation testing, one that directly tests based on the BNF grammar, one that applies the mutation operator and tests that could be applied for programs that unit level for integration testing, for specifications and to mutate the input spaces and in programs you could apply it for any programming language, you could apply to test compilers and you could mutate program and test this is what we had looked last week. There is no known application of BNF grammar based testing for integration testing, but you can apply mutation operators to program to do integration testing. When it comes to specifications there are specific classes at specifications called algebraic specifications

for which mutation testing can be used, you can also apply to finite state machines, you can apply it to model checking tools like SMV and NuSMV.

In input space partitioning the main goal of mutation testing is to apply it to input description languages, we will see it for XML. So, this was an overview of all that we were going to do regard to mutation testing for software artifacts. What have we done already? The things that are colored in green here, we have completed BNF grammar for programming languages. I told you how to mutate by using the grammar, we also saw mutation operators for programs strongly killing, weakly killing mutants, ground strings and we saw an exhaustive list of mutation operators that you could apply for programs written in Java and C to mutate your program and do unit testing for that program. So, these we have already completed. The things that I have stricken out, things that as I told you we are not going to cover as a part of this course. So, testing of compilers is a very involved topic that based that is based on grammar testing, is not within the scope of this course. So, I am not going to do that.

Algebraic specification needs me to introduce the detail notion of algebraic specifications, and it does not have great practical applications. So, I decided to skip that also, and write in the beginning I told you this course does not cover model checking. So, we will not really do mutation operators for model checking. So, the pending ones are the ones that are in black to be done. What we are going to do in this lecture and into a good amount in the next lecture would be, how mutation testing is applied to programs to be able to do design integration testing. This is what I am going to tell you. So, this is the focus of the next couple of lectures including this lecture.

(Refer Slide Time: 03:10)



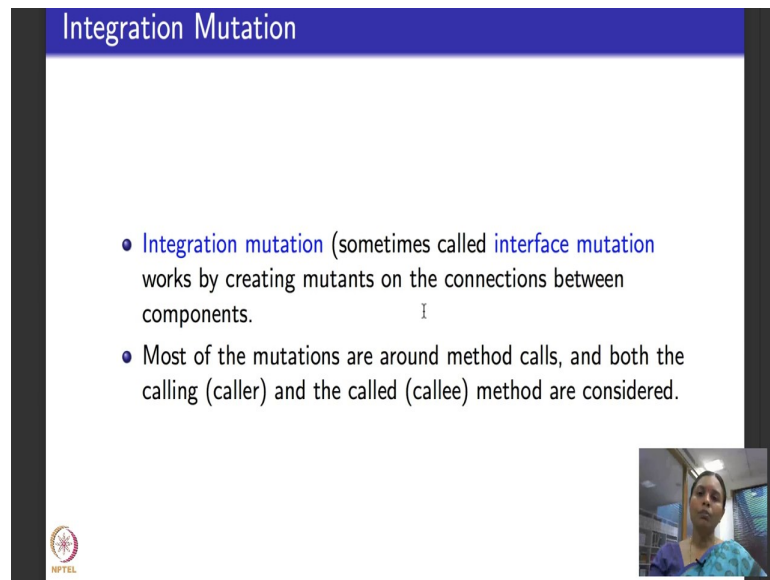
The slide has a blue header with the text "Mutation for Integration". Below the header, there is a bulleted list. The first bullet point is "We focus on mutation testing for design integration in two parts:". The second bullet point is "How to use mutation for testing at the integration level without regard to object-oriented relationships.". The third bullet point is "How to use mutation to test for problems involving inheritance, polymorphism and dynamic binding.". 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.

- 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.

So, when it comes to mutation programs at the level of integration testing, we have considered two parts of it. The first that we ask is how to use mutation for testing at integration level without focusing on object orientated relationships. Let us say I have a C program; the C program can be modular it could have several procedure that call each other, several functions that call each other, and I might want to test the interfaces for the procedure calls related to C. There will be no object oriented features at all, we will do that separately that is such that it applies to any procedures methods, called interfaces and could be used generically cutting across several different programming languages. Then we will consider specifically integration related to object oriented features, because that is very important than lot of new complications come when it comes to integrating modules from different classes modules from different methods.

So, we will look at a separate set of mutation operators that focus on object oriented integration testing. So, I will begin this lecture in this lecture we will do the first part I will tell you what are the inter mutation operators for mutation testing at integration level for generic programming language. You could use it for generic programming language like C, you could also use it for a programming language like Java, but without focusing on the object oriented relationships or any other programming language.

(Refer Slide Time: 04:35)



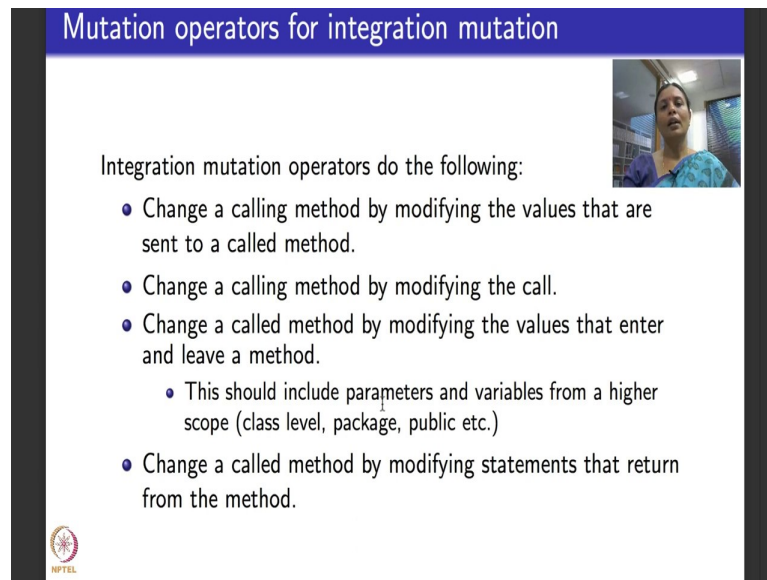
The slide is titled "Integration Mutation" in a blue header. It contains two bullet points: "• Integration mutation (sometimes called interface mutation) works by creating mutants on the connections between components." and "• Most of the mutations are around method calls, and both the calling (caller) and the called (callee) method are considered." In the bottom right corner, there is a small video inset showing a person. In the bottom left corner, there is a small circular logo with the text "NPTEL" below it.

- Integration mutation (sometimes called interface mutation) works by creating mutants on the connections between components.
- Most of the mutations are around method calls, and both the calling (caller) and the called (callee) method are considered.

So, integration mutation, integration testing or integration mutation is also called interface testing. If you remember in the module that I told you about design integration when I do integration testing. The focus is on interfaces also called interface testing, how does it work? It works by creating mutants. Where does it create mutants? It creates mutants on the interfaces or on the connections between the components, it does not mutate it within a method that we already saw last week.

The mutation that we want to focus on this week will focus on mutation on the calls are interfaces that are present between the functions and procedures and methods. Most of the mutations that we will do are around method calls or function calls, both the calling method and the called method are considered for mutations.


(Refer Slide Time: 05:24)



Mutation operators for integration mutation

Integration mutation operators do the following:

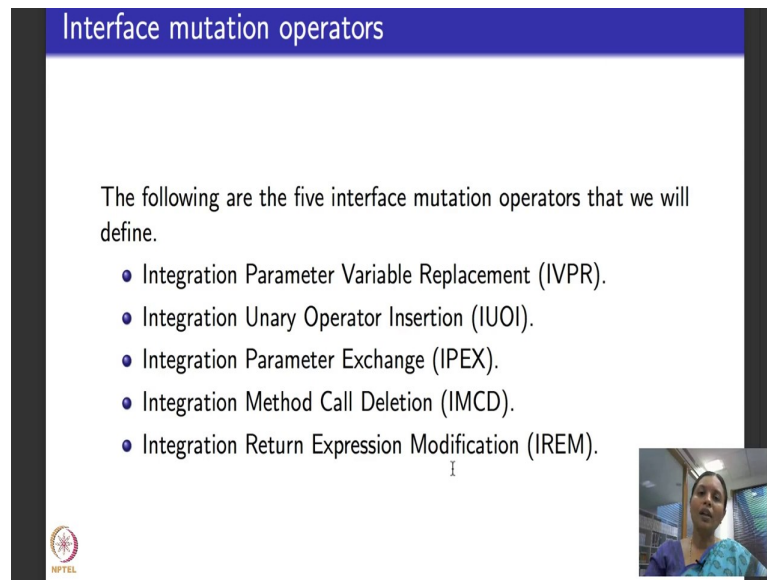
- Change a calling method by modifying the values that are sent to a called method.
- Change a calling method by modifying the call.
- Change a called method by modifying the values that enter and leave a method.
 - This should include parameters and variables from a higher scope (class level, package,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



So, generically listing these are what the integration mutation operators do. They first, this is the first to talk about the calling method, the last to talk about the called method. The first one says if they change a calling method by modifying the values that are sent to the called method. So you could change variable names, you could change expressions that are sent to the call method. The second mutation that again focuses on the calling methods, it is that is modifies the call itself, we will see examples of how to do this. The other two kinds of mutation focuses on the method that is being called, called method. It changes the called method by modifying the values that enter and leave the method. What does the called method return, what is out what is it enter, what does it leaves it.

So, typically it includes parameters and variables from highest scope also. For object oriented programming including variable set come from a class level, from package level public variables and so on. Of course, if you are in C it does not make sense, but if you are working with object oriented programs then you need to worry about this also. The last kind of mutation will change the call method by modifying statements that return from the method. So, here it modifies the values that enter and return from the method, it here it modifies the statement itself that return the method, only the return statements. We do not focus on modifying the internal statements when we do unit testing. We would have tested that method separately by modifying the internal statements, we don't have to consider that.


(Refer Slide Time: 06:54)



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).

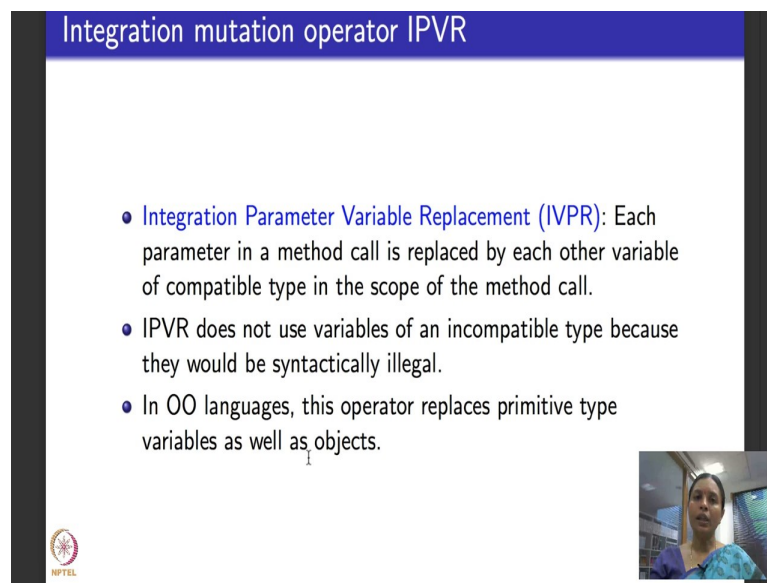
NPTEL



So, here are five generic interface mutation operators that we will be discussing. As I told you can apply it to any programming language that is modular, without it being object oriented it applies to any programming language. So, we will see them one at a time. The first one we will see is called integration parameter variable replacement. Do not worry too much about these abbreviations, I have given them because the book has them and sometimes it useful as a matter of convenience, but you do not have to remember the abbreviation and they are by no means important for us. So, if you see every operator begins by with the word integration. So, it says what do I do in integration. I replace a parameter variable that is the first mutation, the second one says in integration testing, I reply I insert a new unary operator. If you call a method with x comma y may be instead of calling a method x comma y, as arguments you will call it with minor x comma y that would be a unary operator insertion mutation.

The third one says I change the parameters that are called then doing the interface or integration. The fourth one says I delete the method call itself see what happens. It is a natural thing to do right, now maybe it is a redundant call. So, its mutation could delete the entire method call itself that last mutation says I modify the return expression returned by the called method.

(Refer Slide Time: 08:23)



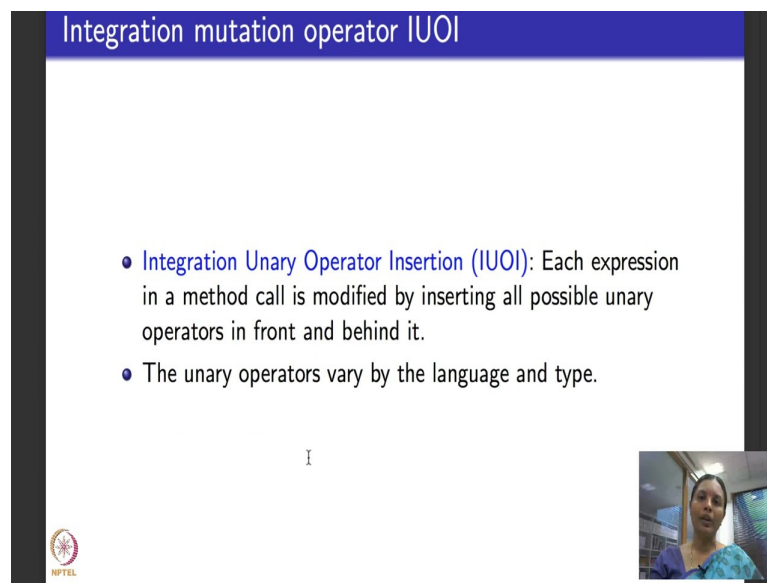
The slide has a blue header with the text "Integration mutation operator IPVR". Below the header, there are three bullet points in blue text. In the bottom right corner of the slide, there is a small rectangular video inset showing a person speaking. In the bottom left corner, there is a small circular logo with the text "NPTEL" below it.

- **Integration Parameter Variable Replacement (IPVR):** Each parameter in a method call is replaced by each other variable of compatible type in the scope of the method call.
- IPVR does not use variables of an incompatible type because they would be syntactically illegal.
- In OO languages, this operator replaces primitive type variables as well as objects.

So, we will go ahead and define one at a time, the first one integration mutation operator I V P R expands as integration parameter variable replacement. So, what does it say? If you read it out, it says there is a parameter which are the parameters with which the method is being called by the column method. This mutation says, take each parameter in the method call and replace it with each other variable of compatible type. So, let us say that is a call to a method called g with two parameters a and b, maybe you replace it with c and d such that type of c matches with the type of a the type of b matches with the type of d. Why is type matching important? It is important, the type should be compatible or matching. Remember it is important because the mutated program should be able to compile execute and run for the notion of killing.

So, I cannot replace it with any other variable, I replace the parameters with other variables that are of this same type of compatible type that are also present in the program in object oriented languages. When we apply integration parameter variable replacement, we also have to consider replacing primitive type variables and objects. So, in summary, what is this mutation operator says? It says that the caller is calling a calling procedure with some variables, those variables are called parameters go ahead and change these parameters to any other variable of a compatible type that is all it says.

(Refer Slide Time: 09:51)



The slide has a blue header with the text "Integration unary operator IUOI". Below the header, there are two bullet points:

- **Integration Unary Operator Insertion (IUOI):** Each expression in a method call is modified by inserting all possible unary operators in front and behind it.
- The unary operators vary by the language and type.

Below the bullet points, there is a small, faint letter 'I'. In the bottom right corner of the slide, there is a small video inset showing a person speaking.

The next mutation operator called as integration unary operator insertion, what is it do? It says there is a method call there is an expression in the method call you modify the expression by inserting all possible unary operators in front and in behind.



So, as I told you suppose there was a call to a method at some point called, let us say call to a method f , which had the three parameters as a , b and c , you could insert a unary operator in front of a to mutate I, t to make it minus a . So, instead of the method being called with parameters a , b and c , it will be called with parameters minus a , b and c . In turn you could mutate and change the sign of b , you could mutate the change and change the sign of c and so on. So, unary operators that are available vary by language and by type. So, whichever language you are using basically whatever is compatible in that language, feel free to pick up that unary operator and change the parameters by inserting a new unary operator in front of a reference variable or a parameter.

(Refer Slide Time: 10:53)

Integration mutation operator IPEX

- **Integration Parameter Exchange (IPEX):** Each parameter in a method call is exchanged with each parameter of compatible type in that method call.
- For e.g., if a method call is `max(a,b)`, a mutated method call of `max(b,a)` is created.

I





The next operation, third operation mutation operation is this, it says integration parameter exchange, which means what each parameter in a method call is exchanged with each parameter of compatible type in that method call. What is it mean, it says if the method call, for example, is something like this a comma b, then I exchange the parameters a and b and instead create a call which says max of b comma a. Maybe this will find a bug, this is a useful interface mutation operator.

(Refer Slide Time: 11:34)

Integration mutation operator IMCD

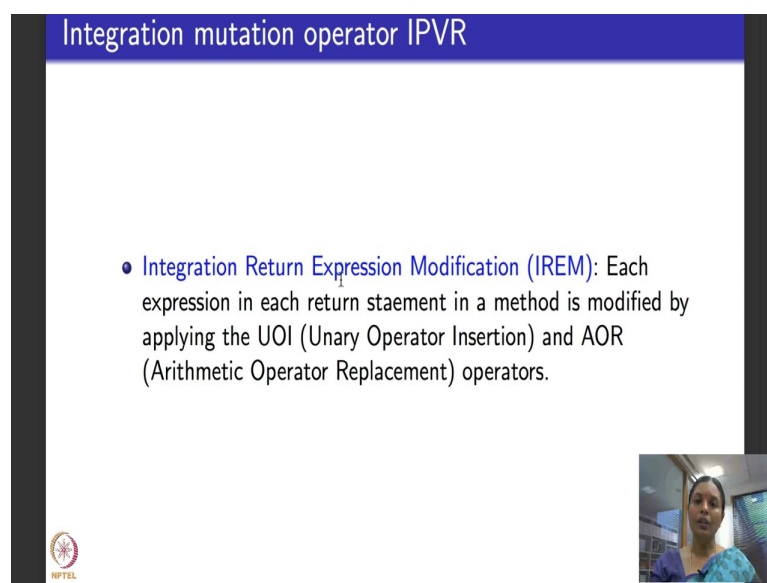
- **Integration Method Call Deletion (IMCD):** Each method call is deleted. If the method returns a value and it is used in an expression, the method call is replaced with an appropriate constant value.
- In Java, the default values should be used for methods that return values of primitive type. If a method returns an object, the method call should be replaced by a call to `new()` on the appropriate class.



The next one that we will see is what is called integrator integration method call deletion. So, what it says as I told you here, it is bold, it just deletes the entire statement that touches the method of the function call. So, if each call is deleted means, what if it is deleted, the mutated program should still compile and produce the value. So, it should be the method is actually originally called would have returned the value. So, when you delete the method call there will be no returning of value, and the program execution not compiling will stop. So, you cannot just delete and hope everything will be fine.



Obviously the program is not going to compile for you to see if there is any error. So, when you delete you also do the following. If the method returns a value and is used in an expression the method call is replaced with appropriate constant value. Let us say there was a method call which was returning an integer value which was used in an expression at a subsequent point in the program in the main calling program. So, what you do? You go ahead and replace wherever it was used. This method call was used with a simple assignment of some constant integer value. So, that the main program continues to compile and execute without any problem. When I specifically apply this method called deletion in programming language like Java I must be careful, the default value should be used for methods that return values of primitive types also. The method returns an object then the method call should be replaced by a call to the special new method within the appropriate class otherwise will run into trouble.

(Refer Slide Time: 13:04)



Integration mutation operator IPVR

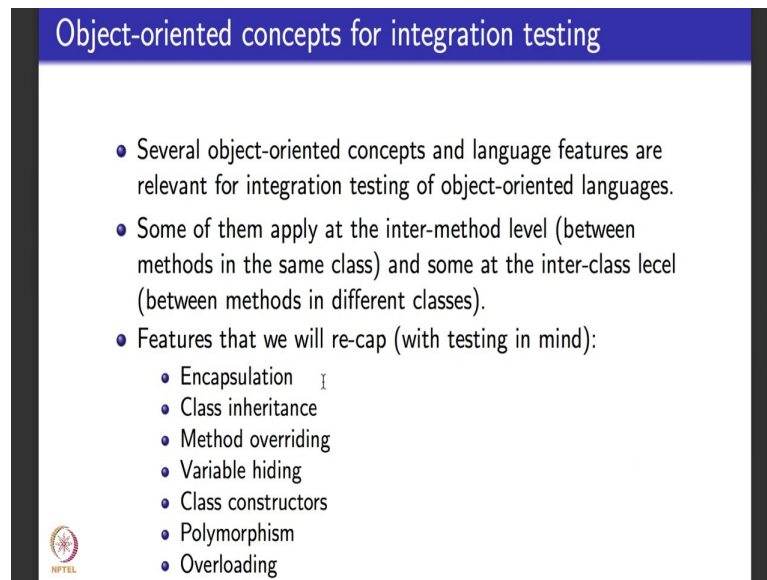
- **Integration Return Expression Modification (IREM):** Each expression in each return statement in a method is modified by applying the UOI (Unary Operator Insertion) and AOR (Arithmetic Operator Replacement) operators.

So, the final genetic mutation operator that we will see is what is called integration return expression modification, what does it say? It says each expression in a return statement in a method is modified by using any of these operators. If you remember we had seen these two operators in the last lecture. Last week's lectures, when I had explained to you about method unit level integration operators with same category of operators one after the other, we had specifically seen this unary operator insertion which inserts the unary operator. Similar to the one we saw here in this slide is in such unary operators for parameters, this could insert unary operator in any expression and then there was arithmetic operator replacement which let us you replace with the plus with a minus plus with a star minus with a star, star with a plus and so on. So, you could use any of these mutation operators and change the expression that was involved in the callee method returning the value to the called method. As I told you, do not make any other changes to the internals of the callee method; only the return part of the expression is changed because our focus is only on interfaces.

Just to summarize what I set out by giving our five generic interface mutation operators, which could be applied for any programming language, the first in computation operator says you check and change or replace the parameters that are called during the call, you could change the parameter by inserting a negation or a unary operator you could exchange the parameters, you could delete the method call, but replace it with a dummy stuff so that the main program continuous to run or you could change the return expression modification by using any of the method level mutation operators. All these 5 mutation operators can be applied to any programming language, when you want to focus and test interfaces.

(Refer Slide Time: 15:02)



The slide has a blue header with the title "Object-oriented concepts for integration testing". Below the header, there is a white box containing a bulleted list. The list starts with three main points, and the third point has a sub-list of seven items. In the bottom left corner of the white box, there is a small circular logo with the text "NPTEL" below it.

- 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

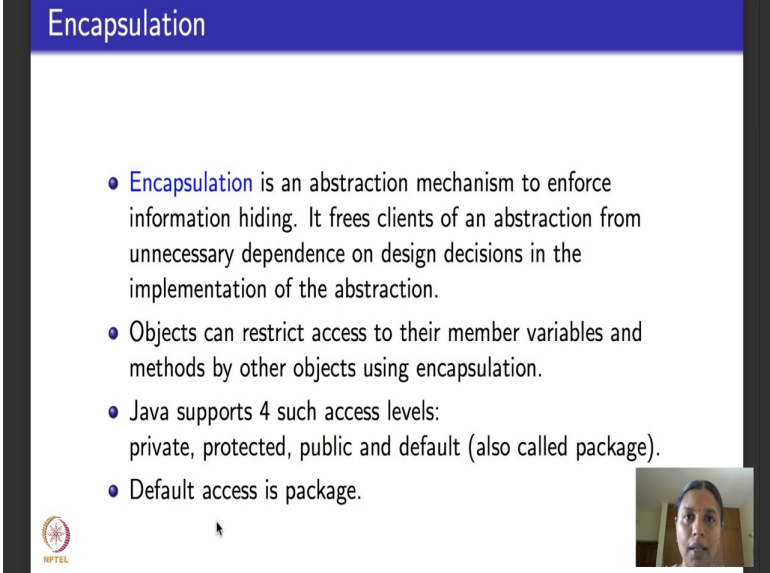
So, towards moving onto using mutation testing specifically for object oriented languages, first point to be noted as I told you is that for object oriented languages the kind of integration that happens is very different from imperative languages like C and all that. So, it helps to spend some time understanding what are the various concepts underlying object oriented programming language that helps us to integrate software components that are developed for these languages. So, with that in mind I would like to recap from the point of view of testing the following object oriented concepts, and before we move on and do that.

When it comes to integration testing for object oriented languages we should remember that there could be two levels, one is you have developed individual methods and you want to put together the methods in the same class and test how they interact. The other is you have written more code you developed a lot of code for several different classes and you want to put together the code of these classes, and test the interactions between the methods that come from different classes. Whatever it is the concept that we will be recollecting today, will help us to do both these kinds of integration tests.

So, what are the features that we will recap? Please note that this not meant to be an exhaustive introduction to these features, we are only going to give a brief overview of the listed features here from the point of view of our use in testing. So, will use it from here for mutation testing, and later when I do exclusive modules and object oriented

testing we will revisit some of these features. So, I will tell you what encapsulation is, class inheritance, method overloading method overriding, hiding of variables class constructor and the notion of polymorphism.

(Refer Slide Time: 16:56)



Encapsulation



- **Encapsulation** is an abstraction mechanism to enforce information hiding. It frees clients of an abstraction from unnecessary dependence on design decisions in the implementation of the abstraction.
- Objects can restrict access to their member variables and methods by other objects using encapsulation.
- Java supports 4 such access levels: private, protected, public and default (also called package).
- Default access is package.

So, the first one in our list is encapsulation, what is encapsulation? Remember the center of object oriented languages is to do abstraction. Encapsulation is one of the extraction mechanisms, that several object oriented languages offer. Encapsulation and abstraction that enforces information hiding; the word encapsulate says you close something you hide something. It frees the clients of an abstraction from unnecessary dependence on some design decisions that you do not want to happen at that level of abstraction. Sometimes at an abstract level you want to not know what certain decisions and a refined level of the abstraction you want to know about the design decisions. Encapsulation helps you to do that. True encapsulation objects can restrict that access to member variables and methods by other objects using encapsulation.

Java support four different access levels, other programming languages that support object oriented features might have other access levels. The four access levels that we will discuss with reference to Java in mind are private, protected, public and default. Default is also called package because default is also called package, the default access if not specified is package.

(Refer Slide Time: 18:13)

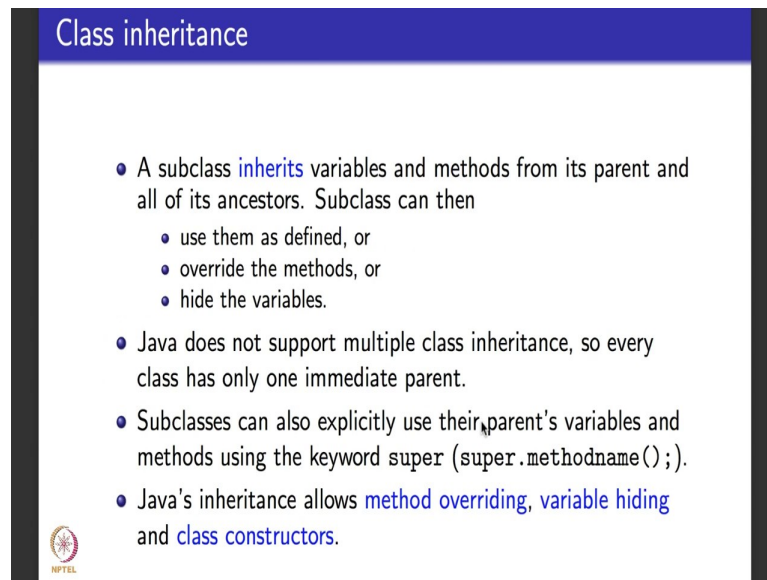
Encapsulation: Access levels in Java				
Specifier	Same class	Different class same package	Different package subclass	Different package non-subclass
private	Y	n	n	n
package	Y	Y	n	n
protected	Y	Y	Y	n
public	Y	Y	Y	Y



So, how do these access levels look like? For example, if a particular entity is private let us say a variable is private, then here are all the access level read capital, yes capital Y as yes, n as, no small n as no. So, if a particular variable is private then within the same class is available for access, from different class same package in different package subclass in different package, in a different class it is not available. If a variable is declared as package, which is the default declaration is available within the same class, in different class, but within the same package, but it is not available in a different package, in different a subclass variable.


If a variable is protected then its yes for class, yes for different class and same package, yes for different package and no for different package non subclass public access is available through out there is no restriction the variable or entity can be seen everywhere. So, this is what encapsulation deals with.

(Refer Slide Time: 19:15)



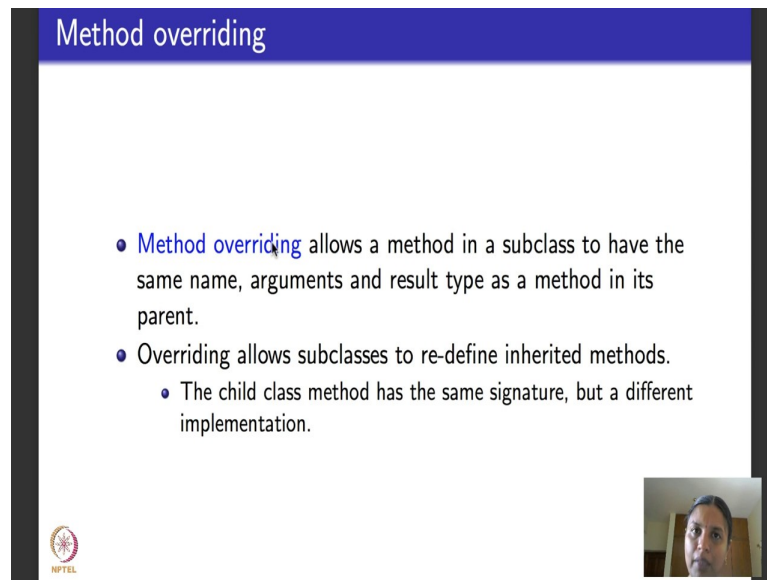
Class inheritance

- A subclass **inherits** variables and methods from its parent and all of its ancestors. Subclass can then
 - use them as defined, or
 - override the methods, or
 - hide the variables.
- Java does not support multiple class inheritance, so every class has only one immediate parent.
- Subclasses can also explicitly use their parent's variables and methods using the keyword `super` (`super.methodname();`).
- Java's inheritance allows **method overriding**, **variable hiding** and **class constructors**.



The next concept that we want to look at is what is called class inheritance. What is class inheritance? Let us talk about. We say one class inherits from another class then the class that it inherits from is called a parent class, and the other class is called a subclass or a child class. Parent of parent could be an ancestor. So, we see a subclass inherits. What are the things that it can inherit? It can inherit variable, it can inherit methods, it can inherit these two entities, variables and entities from its methods parent, or it can inherit from its parent's parent which will be an ancestor and moving on parent's parent's parent, which will be another ancestor and so on. Then after inheriting what can the sub class do with the variable and methods? It can use them as they are defined, it can overwrite the methods redo them or it can hide the methods. We will see overriding and hiding very soon now. Java does not support multiple class inheritance. So, every class has only one immediate parent. Now the sub classes that inherit can explicitly use their parent variables and methods without changing or overriding by using the word `super`. So, the prefix whatever variable or method name with `super`, like this, `super dot method name` then you can use it as it is.


(Refer Slide Time: 20:46)



Method overriding

- **Method overriding** allows a method in a subclass to have the same name, arguments and result type as a method in its parent.
- Overriding allows subclasses to re-define inherited methods.
 - The child class method has the same signature, but a different implementation.

NPTEL





Java's inheritance allows us to do, as I told you here, override methods, hide the variables can also use class constructors. So, we will look at method overriding first. What does method overriding do? Method overriding allows a method in a subclass to have the same name, same arguments and same result type as a method in its parents, but inside, the code that the method implements could be different. So, for all practical purposes its name is also same, arguments is also same, written type is also same, but whatever it computes or whatever is core functionality is, that could be different. Overriding what does it do? It allows subclasses to redefine inherited methods; child class method can have the same signature meaning same name arguments and results, but a different implementation that is what I told you just now.

(Refer Slide Time: 21:27)

Variable hiding

- **Variable hiding** is achieved by defining a variable in a child class that has the same name and type of an inherited variable.
- This has the effect of hiding the inherited variable from the child class.




Variable hiding: what is that? Variable hiding is achieved by defining a variable in a child class that has the same name and the type of an inherited variable. So, which means what? You have a variable from your parent class, you declare another variable with exactly the same name, with exactly the same type. So, it is as good as saying I am not considering the variable that I have inherited from the parent class or in other words I am hiding the variable that I have inherited from the parent class.

(Refer Slide Time: 21:58)

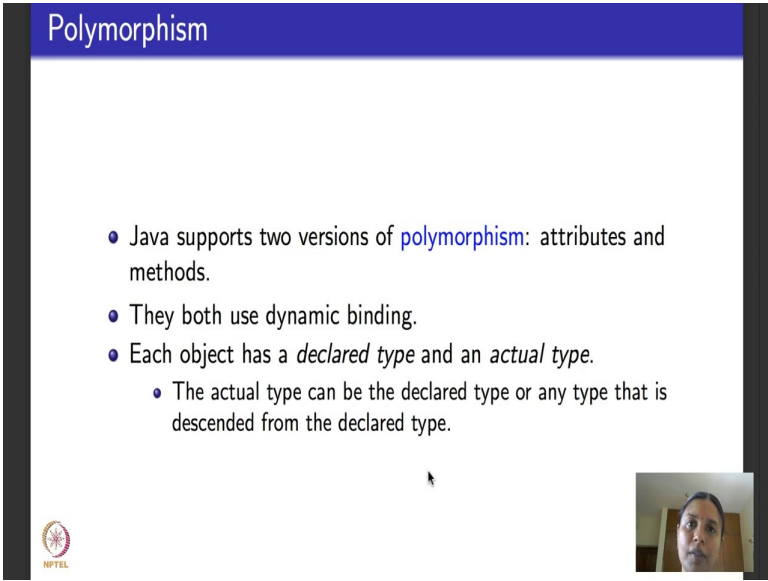
Class constructors

- A **constructor** in a class is a special type of subroutine called to create an object.
- It prepares the new object for use, often accepting arguments that the constructor uses to set required member variables.
- They are not inherited in the same way methods are.
- To use a constructor, we must explicitly call it using the **super** keyword. The call must be the first statement in the derived class constructor and the parameter list must match the parameters in the argument list of the parent constructor.



What is the class constructor? A class constructor is a class in a class, it is a special type of subroutine that is called to create an object. It prepares the new object for use of accepting arguments that the constructor uses to set the required member variables of that object that is being created. These objects are not inherited the same way methods are constructors are very specific features; to use a constructor is explicitly call it again using the super keyword that we saw here using this super keyword.

(Refer Slide Time: 22:41)

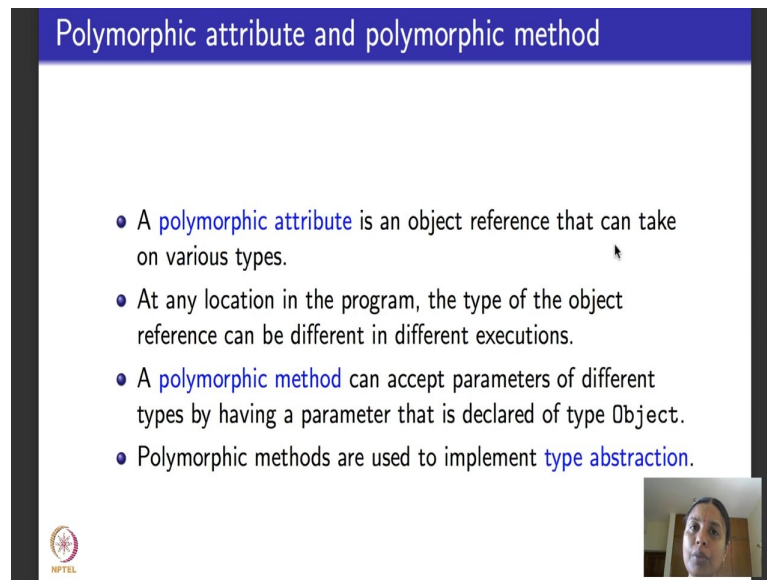


The slide is titled "Polymorphism" in a blue header. It contains a bulleted list of points about Java's polymorphism. In the bottom right corner, there is a small video inset showing a person's face. In the bottom left corner, there is a small logo for NPTEL.

- Java supports two versions of **polymorphism**: attributes and methods.
- They both use dynamic binding.
- Each object has a *declared type* and an *actual type*.
 - The actual type can be the declared type or any type that is descended from the declared type.

The call must be the first statement that derived class constructor and the parameter list must exactly match the parameters in the argument list of the parent constructor. What is polymorphism? Java supports two versions of polymorphism: attributes and methods both of them use dynamic binding. So, what did you mean by that? An object actually in Java has a declared type and actual type. Actual type can be the declared type or it can be any type that is descended from the declared type polymorphic attributes and polymorphic methods as I told you here right two types of polymorphism attributes and methods.

(Refer Slide Time: 23:09)



The slide has a blue header with the text "Polymorphic attribute and polymorphic method". Below the header, there are four bullet points:

- A **polymorphic attribute** is an object reference that can take on various types.
- At any location in the program, the type of the object reference can be different in different executions.
- A **polymorphic method** can accept parameters of different types by having a parameter that is declared of type Object.
- Polymorphic methods are used to implement **type abstraction**.

In the bottom left corner, there is a small circular logo with the text "NPTEL". In the bottom right corner, there is a small video inset showing a person's face.

So, what do they do? Polymorphic attribute is an object reference that can take on various types that is why the word poly, poly means it can be of several types at any point and time in a given piece of code.



At any location in a program the type of object reference can be different in different executions of the same program for methods. The method is called polymorphic, if it can accept parameters of different types, having a parameter that is declared of type object. What are polymorphic methods used for? For polymorphic methods are used to implement type abstraction please remember that is also something called overriding methods which is basically another method in a child class that has the same name argument and type this is different the polymorphic method can accept parameters of different types by having a parameter that is declared of type object these two are different.

The next concept in our list is overloading, overloading is the use of same name for different constructors or methods in the same class.

(Refer Slide Time: 24:08)

Overloading

- **Overloading** is the use of the same name for different constructors or methods in the same class.
- They must have different **signatures**, or lists of arguments.
- Note: Overloading is different from overriding.
 - The former occurs with two methods in the same class, whereas overriding occurs between a class and one of its descendants.





So, the it again intuitively in the English sense of the term overloading, we use the same name again and again. So, you over load the name they must have different signatures or lists of argument unlike overriding. Overloading is different from overriding why overloading a class with two methods in the same class whereas, override occurs between a class and one of its descendants.

(Refer Slide Time: 24:32)

Instance and class variables in Java

- **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.



The next concept is instance and class variables, instance and class variables are associated with a class. Class variables in Java called static variables in the sense that

there is only one copy of the variable that is shared with all instances of that class. Unlike class variables that are static, instance variables belong to an object and instance of a class, every instance can have its own copy of the variable that way it is not static. Changes made to a particular instance of an instance variable do not reflect in other instances of that class. So, instance variables are typically declared at class levels, class variables are also declared at class level, but with the keyword static. Local variables are declared within methods.

So, this was a brief overview of some object oriented concepts that I wanted to tell you. So, we recapped all these concepts that are listed in the slides. This by no means is meant to be an exhaustive introduction. So, please do not consider this as an exhaustive introduction, we just briefly recap concepts as we wanted for testing specifically mutation testing related to object oriented integration. Later when we do object oriented applications testing also will come and recap these concepts feel free to pick up any good book on object oriented programming, and you will get a more detailed thorough introduction to these concepts.

I hope this brief overview was helpful, we will continue with object oriented mutation testing for integration in the next lecture.

Thank you.