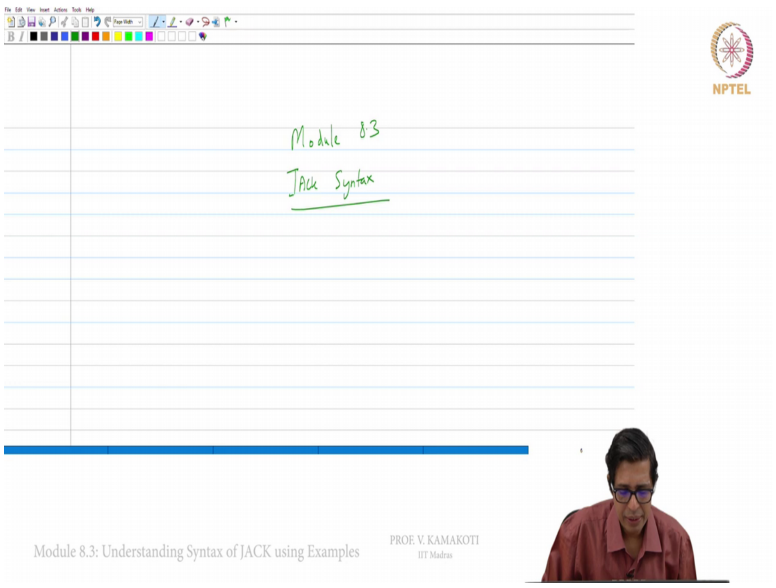
# Foundations to Computer Systems Design

**Prof. V. Kamakoti**
**Department of Computer Science and Engineering**
**Indian Institute of Technology Madras**

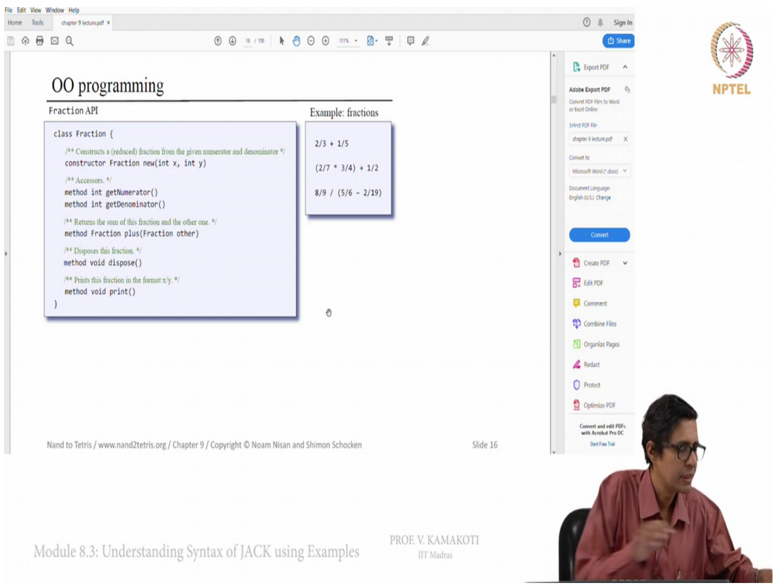
**Module 8.3**
**Understanding Syntax of Jack using Examples**

(Refer Slide Time: 0:17)



So welcome to module 8.3 where we are going to talk about the Jack syntax, syntax of jack, so when you are studying C you would have looked at syntax and semantics of the C programming language, similarly now we are looking at the syntax of the jack programming language and that will be part of this particular exercise.

(Refer Slide Time: 0:42)



OO programming

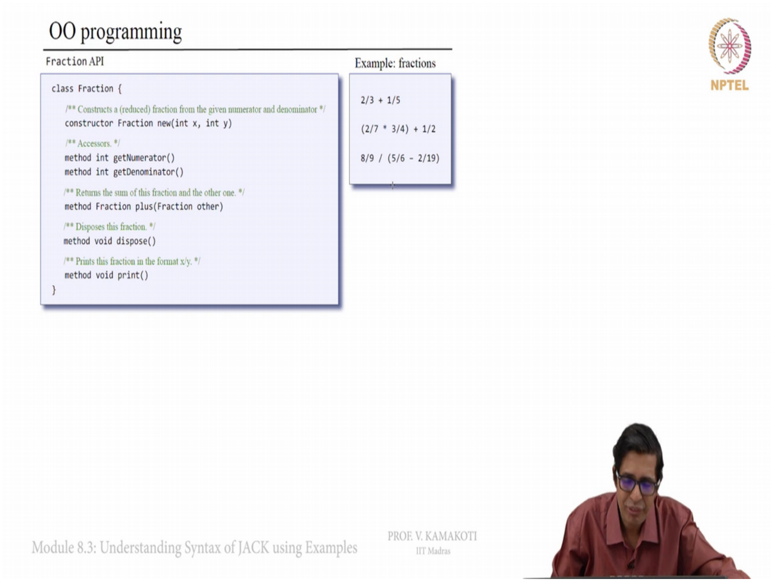
Fraction API

```
class Fraction {
    /** Constructs a (reduced) fraction from the given numerator and denominator */
    constructor Fraction new(int x, int y)
    /** Accessors. */
    method int getNumerator()
    method int getDenominator()
    /** Returns the sum of this fraction and the other one. */
    method Fraction plus(Fraction other)
    /** Disposes this fraction. */
    method void dispose()
    /** Prints this fraction in the format x/y. */
    method void print()
}
```

Example: fractions

```
2/3 + 1/5

(2/7 * 3/4) + 1/2

8/9 / (5/6 - 2/19)
```

Nand to Tetris / www.nand2tetris.org / Chapter 9 / Copyright © Noam Nisan and Shimon Schocken    Slide 16


## OO programming

Fraction API

```
class Fraction {
    /** Constructs a (reduced) fraction from the given numerator and denominator */
    constructor Fraction new(int x, int y)

    /** Accessors. */
    method int getNumerator()
    method int getDenominator()

    /** Returns the sum of this fraction and the other one. */
    method Fraction plus(Fraction other)

    /** Disposes this fraction. */
    method void dispose()

    /** Prints this fraction in the format x/y. */
    method void print()
}
```

Example: fractions

```
2/3 + 1/5

(2/7 * 3/4) + 1/2

8/9 / (5/6 - 2/19)
```

As mentioned earlier jack is an object oriented programming language, now we will try and you know basically explain jack through another example what we call as fractions right, so there are, so fraction is nothing but you know has a numerator and the denominator and always will be right to have a reduced fraction, for example I will not, if you give 2 x 10 I would like to represent it as 1 x 5, so 1 x 5 is the reduced form of 2 x 10 right.

So how do we, so I want to represent several fractions, so now will have one template for a fraction which you will instantiate many number of times to create several fractions right, now this is the fraction API, API stands for application programmer interface, somebody wants to use fraction how can the use it, this class that I am describing here tells the programmer, tells the application programmer or we can use it, that is why this classes also called as an application programmer interface right.

Now first class fraction okay, it is start with a bressers this is a syntax, now the first to think is whenever I describe in class I should describe a constructor function, a constructor function is a one which will construct that class, every time I am instantiating that class to make an object, the constructor function will execute first and it will create that object for you, the constructor function essentially, so if I have ten fractions, I need ten different memory locations to store the fraction, the constructor function one of the prime objective of the constructor function is to create some memory for this and you know basically create a memory for it and then store the values inside it right.

So there will be, once a moment I say class fraction there will be a constructor for this class fraction as you see here, I am moving the mouse on top of it and then there will be a new, so always every class xxx will have construct are xxx followed by new okay and this constructor function needs to integers int Y and int Y, one for the numerator and another one for the denominator okay, right so there is a constructor function.

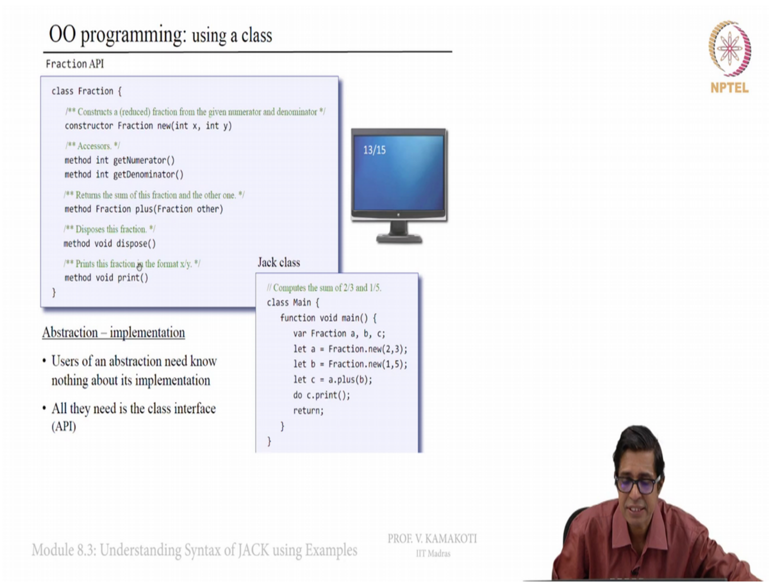
Then what are the other functions we need for the application programmer, so if I have a fraction, I will like to know what is a numerator? I would like to know what is the denominator of that fraction? So there are two methods define inside which will allow the application programmer to go and find the numerator and the denominator of the fraction that he has or she has defined, so there is a method call get numerator which will return an integer which is the numerator, this method call get denominator which will return an integer call the denominator, which is the denominator.

Then I will have another method which is basically, which will return a fraction, another method call plus which will take as input some other fraction and added to the current fraction, it will take a, so there is a fraction access X if I say plus is a method sorry, which will take another fraction and added to the current fraction okay and then when I create a fraction, when a create an object fraction sometime after that I may not needed, so I have to throw it out.

So there is always every class will have a dispose function, which will dispose this fraction, always it will have is dispose function, like how I have a new which is equivalent to getting a memory and storing for that particular object right, so whenever I create an object from a class I create a memory and store it, so that is like my lock and whenever I want to dispose this is like free right, so my lock, if you my lock particular pointer giving it lot of space, then you go and free that pointer once you do not use it.

So dispose is equivalent to free and of course I could have a print statement of this fraction right, so examples of fractions are 2 x 3, 1 x 5, 2 x 7, 3 x 4, 1 x 2, 8 x 9, 5 x 6, 2 x 19 and these are all different varieties of fractions that you see, but this fraction implies that it is the reduced fractions, so I not have 2 x 10 rather I have 1 x 5 right for example.

(Refer Slide Time: 6:14)



## OO programming: using a class

Fraction API

```
class Fraction {

    /** Constructs a (reduced) fraction from the given numerator and denominator */
    constructor Fraction new(int x, int y)

    /** Accessors. */
    method int getNumerator()
    method int getDenominator()

    /** Returns the sum of this fraction and the other one. */
    method Fraction plus(Fraction other)

    /** Disposes this fraction. */
    method void dispose()

    /** Prints this fraction in the format x/y. */
    method void print()
}
```

Jack class

```
// Computes the sum of 2/3 and 1/5.
class Main {
   function void main() {
      var Fraction a, b, c;
      let a = Fraction.new(2,3);
      let b = Fraction.new(1,5);
      let c = a.plus(b);
      do c.print();
      return;
   }
}
```

Abstraction – implementation

- Users of an abstraction need know nothing about its implementation
- All they need is the class interface (API)

Module 8.3: Understanding Syntax of JACK using Examples

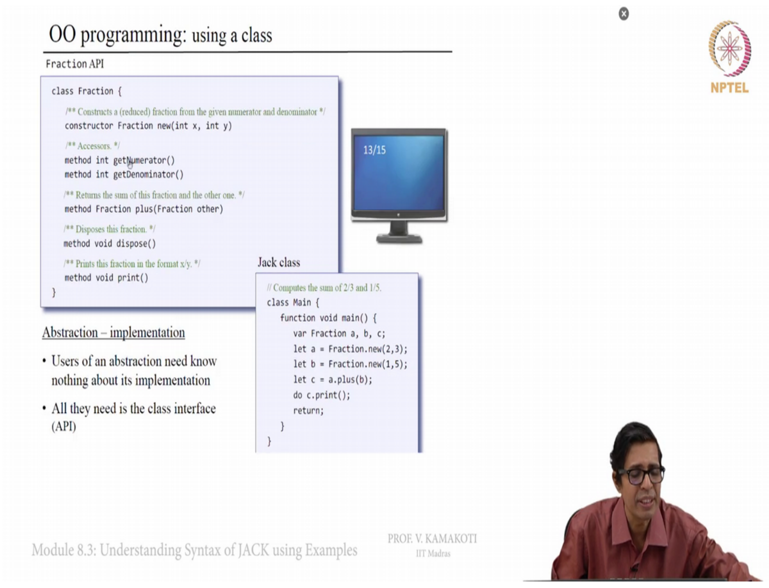
PROF. V. KAMAKOTI
IIT Madras

So this is the API for fraction so how does any application programmer used this fraction, so he creates a class call main, insight that function call main, small main, so main main, so then this is where fraction because you have a class A, B, C, so A, B and C are objects of the fraction, of the class fraction, so I created three objects of the class fraction here, now I say let A equals to fraction.new 2, 3 right, a equals to fraction.new, fraction.new will construct that fraction given, so A is of the type fraction and what is that fraction? Now I am initialising A fraction.new 2, 3, so A will be 2 x 3, similarly I am initialising let B equal to fraction.new 1, 5, so B will be 1 x 5.

Now so A is fraction, B is another fraction, so what is their inside A, there is a numerator and denominator 2, 3 here and then for A I have all these methods like get numerator, get denominator plus dispose print, similarly for B also I have all these, so for every fraction that I have defined, so I have the numerator, denominator in addition to this I do have all this application programming interface namely constructor get numerator etc.

Now C is equal to A.+, so I am using fraction A and in for fraction A there is a plus right, so that is why I say A.+ and that will take another fraction, other fraction as input that is B, so A. +B basically takes A as, in the A fraction there is a method plus to which I am inputting B and this will give me C, this is expected to give me A+B right and that will be C because that particular plus function returns a action that is why I say C, now this C, I print this C, so C.print and I return, so this is how the application programmer actually uses is fraction using the different routine methods that are given as part of your fraction class.

(Refer Slide Time: 9:21)



OO programming: using a class

Fraction API

```
class Fraction {
   /** Constructs a (reduced) fraction from the given numerator and denominator */
   constructor Fraction new(int x, int y)
   /** Accessors. */
   method int getNumerator()
   method int getDenominator()
   /** Returns the sum of this fraction and the other one. */
   method Fraction plus(Fraction other)
   /** Disposes this fraction. */
   method void dispose()
   /** Prints this fraction in the format x/y. */
   method void print()
}
```

Jack class

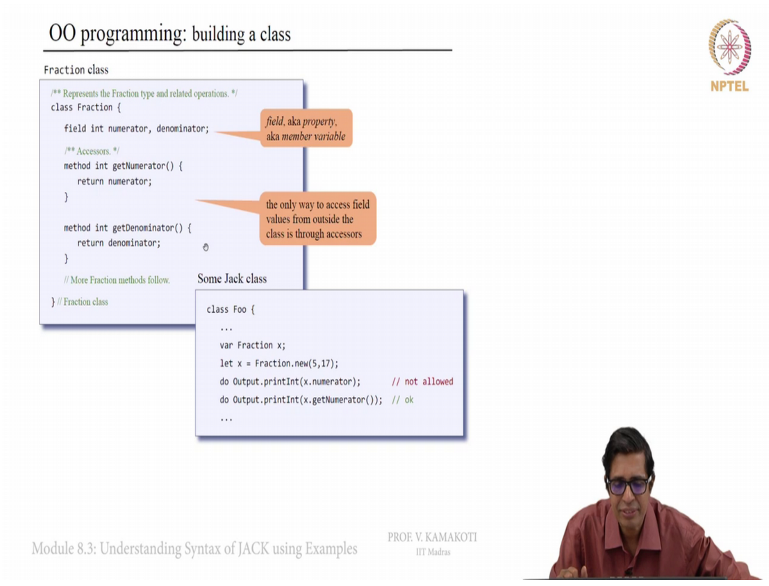
```
// Computes the sum of 2/3 and 1/5.
class Main {
   function void main() {
      var Fraction a, b, c;
      let a = Fraction.new(2,3);
      let b = Fraction.new(1,5);
      let c = a.plus(b);
      do c.print();
      return;
   }
}
```

Abstraction – implementation

- Users of an abstraction need know nothing about its implementation
- All they need is the class interface (API)

Module 8.3: Understanding Syntax of JACK using Examples

PROF. V. KAMAKOTI
IIT Madras



OO programming: building a class

Fraction class

```
/** Represents the Fraction type and related operations. */
class Fraction {
   field int numerator, denominator;
   /** Accessors. */
   method int getNumerator() {
      return numerator;
   }

   method int getDenominator() {
      return denominator;
   }
   // More Fraction methods follow.
} // Fraction class
```

*field*, aka *property*, aka *member variable*

the only way to access field values from outside the class is through accessors

Some Jack class

```
class Foo {
   ...
   var Fraction x;
   let x = Fraction.new(5,17);
   do Output.printInt(x.numerator);        // not allowed
   do Output.printInt(x.getNumerator());   // ok
   ...
```

Module 8.3: Understanding Syntax of JACK using Examples

PROF. V. KAMAKOTI
IIT Madras

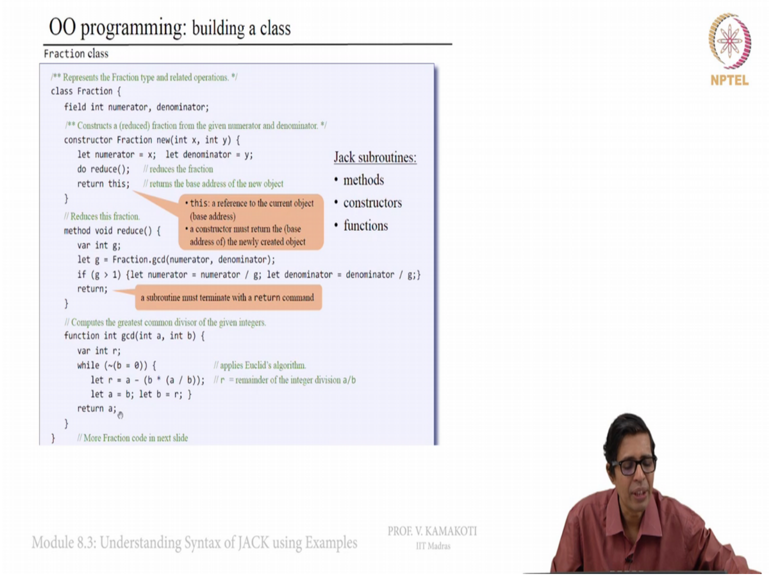
Now how do we populate this class fraction, so inside this class fraction I have two fields, so field is a keyword and both of them or integers, one is numerator another is the denominator right, so these are all variables inside and, so method int get numerator, so now these two are the fields, now I have to see the accessor functions, accessor means which will access the different fields here, method int to get numerator will return the numerator for me, method int get denominator return the denominator for me right.

So this is how I get numerator, get denominator okay right, so I can say, so I can define another class foo, whatever main or whatever and inside there is a, I am declining a fraction call X variable, let X equal to fraction.new 5, 17, so this will give me 5, 17, so 5 x 17, so the moment I say fraction.new this will, that numerator will become 5 and 17 will become the denominator for X, so X is a type of this class fraction in which it is numerator is 5 and denominator is 17.

Now if I want to access the numerator I cannot say x.numerator this fellow will not allow us, so any field I want access I have to go through this methods right, so I cannot directly access x.numerator like in the struck case, so if I want to instruct I could put, if it is a struct fraction int numerator I say x.numerator but in this case I cannot do this, so I have to say to output.print int x.get numerator, X is a object of this fraction associated with  X would be get numerator which will give excess numerator and they associated with X there be get denominator which will give you the denominator of X.

Similarly if you declared another fraction Y and I say get numerator it will be the denominator of y.get denominator, y.get numerator it will give me a numerator of Y right, so x.get numerator is correct, x.get numerator I cannot directly access, so this is basically the object orientation and what we call as the data hiring, so any data about the object that I have initiated I can only get through the accessor function, I cannot read it directly.

(Refer Slide Time: 12:17)



## OO programming: building a class

Fraction class

```
/** Represents the Fraction type and related operations. */
class Fraction {
   field int numerator, denominator;
   /** Constructs a (reduced) fraction from the given numerator and denominator. */
   constructor Fraction new(int x, int y) {
      let numerator = x;  let denominator = y;
      do reduce();     // reduces the fraction
      return this;     // returns the base address of the new object
   }
   // Reduces this fraction.
   method void reduce() {
      var int g;
      let g = Fraction.gcd(numerator, denominator);
      if (g > 1) {let numerator = numerator / g; let denominator = denominator / g;}
      return;
   }
   // Computes the greatest common divisor of the given integers.
   function int gcd(int a, int b) {
      var int r;
      while (~(b = 0)) {               // applies Euclid's algorithm.
         let r = a - (b * (a / b));   // r  = remainder of the integer division a/b
         let a = b; let b = r; }
      return a;
   }
}     // More Fraction code in next slide
```

- this: a reference to the current object (base address)
- a constructor must return the (base address of) the newly created object

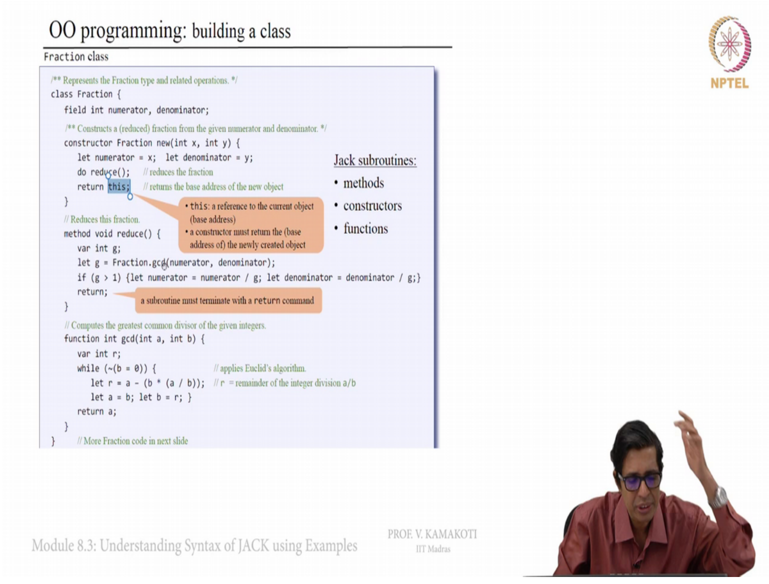
a subroutine must terminate with a return command

Jack subroutines:

- methods
- constructors
- functions

Module 8.3: Understanding Syntax of JACK using Examples

PROF. V. KAMAKOTI
IIT Madras



## OO programming: building a class

Fraction class

```
/** Represents the Fraction type and related operations. */
class Fraction {
   field int numerator, denominator;
   /** Constructs a (reduced) fraction from the given numerator and denominator. */
   constructor Fraction new(int x, int y) {
      let numerator = x;  let denominator = y;
      do reduce();     // reduces the fraction
      return this;     // returns the base address of the new object
   }
   // Reduces this fraction.
   method void reduce() {
      var int g;
      let g = Fraction.gcd(numerator, denominator);
      if (g > 1) {let numerator = numerator / g; let denominator = denominator / g;}
      return;
   }
   // Computes the greatest common divisor of the given integers.
   function int gcd(int a, int b) {
      var int r;
      while (~(b = 0)) {               // applies Euclid's algorithm.
         let r = a - (b * (a / b));    // r = remainder of the integer division a/b
         let a = b; let b = r; }
      return a;
   }
}      // More Fraction code in next slide
```

- this: a reference to the current object (base address)
- a constructor must return the (base address of) the newly created object

a subroutine must terminate with a return command

Jack subroutines:
- methods
- constructors
- functions

Module 8.3: Understanding Syntax of JACK using Examples

PROF. V. KAMAKOTI
IIT Madras

Right, so we are seeing, so different accessor function we have finished off get numerator, get denominator, now we have to see what you mean by constructor function fraction new int x, into y, so I give 5 and 17 as input, what will do? It will make let numerator equal to 5, let denominator equal to 7, so that X and Y gets to read particular field, so when I create a new fraction when a new object is created the numerator of that will become X and denominator Y but then so we do not say, suppose I give 5, 10 I do not wanted to be 5 eaten, I wanted to be actually you know 1 x 2, so we do reduce.

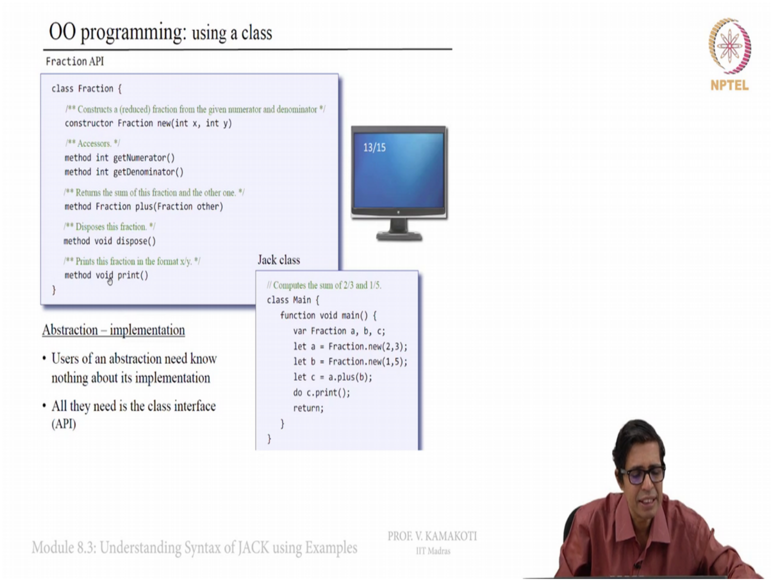
What is this reduce, do reduce? So do reduce is a function inside this fraction, so what will reduce do it will find the greatest common devisor between the numerator and the dominator and it will divided by that greatest common devisor so that after that division the greatest common devisor between the numerator and denominator will be 1 and that is the reduce fraction and for that we use what we call as the Euclidean algorithm  okay, void reduce will do G equal to fraction.gcd, greatest common devisor of numerator, denominator right and if G is greater than 1 let numerator equal to numerator by G, let denominator equal to denominator by G and it will return okay.

So essentially this will adjust the numerator and denominator to become equal, so how do you compute the gcd, for computing that, so this fraction, so it uses a function, see why it is a method, reduce is a method that uses a function call gcd, so gcd is a function and it takes in two inputs A, B and it applies the euclidean algorithm , I want you to go and use, I hope all of you know euclidean algorithm  for finding the greatest common devisor, if you do not know just go to Google and find out, it basically files a reminder of A by B and it makes A equal to B and B equal to R and it keeps out repeating so that is why this euclidean algorithm.

So this will return the gcd okay, now right and this is how it works, so now after I get back, if after I do the reaction your numerator and the denominator will be set as with the gcd as 1, so if I have 2 x 10 it will become 1 x 5 etc and then I say return this, what you mean by this? This actually in object orientation refers to the current object, so if I mean an object, I am modifying an object and I want to return to the calling function, so essentially I say return this object, so now you can slowly correlate, we have been using this and that as part of your R3 and R4 etc in your symbol table in your vm etc.

So this will be used for the coat and coat this will be used for this purpose as you see here, so this will return back the same object here, so always a subroutine must terminate with your return command and so this is what is happening here and this is how this goes.

(Refer Slide Time: 15:52)



## OO programming: using a class

Fraction API

```
class Fraction {
   /** Constructs a (reduced) fraction from the given numerator and denominator */
   constructor Fraction new(int x, int y)
   /** Accessors. */
   method int getNumerator()
   method int getDenominator()
   /** Returns the sum of this fraction and the other one. */
   method Fraction plus(Fraction other)
   /** Disposes this fraction. */
   method void dispose()
   /** Prints this fraction in the format x/y. */
   method void print()
}
```

Abstraction – implementation

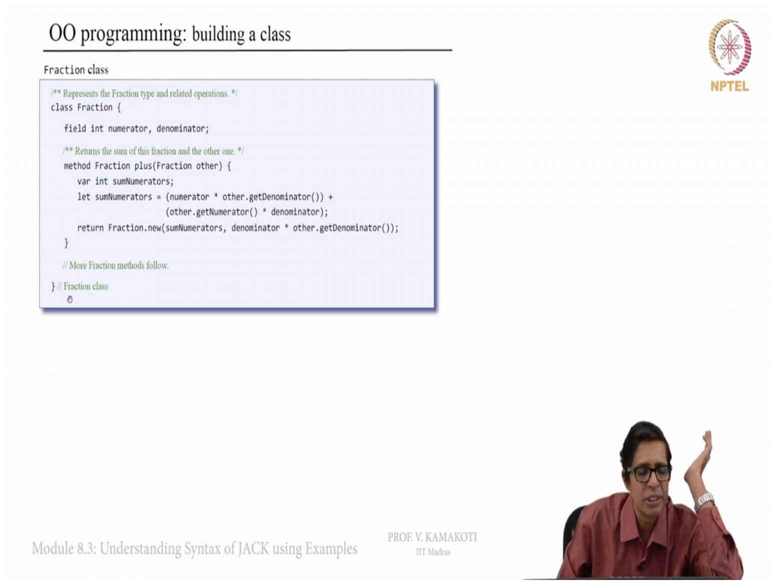
- Users of an abstraction need know nothing about its implementation
- All they need is the class interface (API)

Jack class

```
// Computes the sum of 2/3 and 1/5.
class Main {
   function void main() {
      var Fraction a, b, c;
      let a = Fraction.new(2,3);
      let b = Fraction.new(1,5);
      let c = a.plus(b);
      do c.print();
      return;
   }
}
```

Module 8.3: Understanding Syntax of JACK using Examples

PROF. V. KAMAKOTI
IIT Madras



## OO programming: building a class

Fraction class

```
/** Represents the Fraction type and related operations. */
class Fraction {

   field int numerator, denominator;

   /** Returns the sum of this fraction and the other one. */
   method Fraction plus(Fraction other) {
      var int sumNumerators;
      let sumNumerators = (numerator * other.getDenominator()) +
                          (other.getNumerator() * denominator);
      return Fraction.new(sumNumerators, denominator * other.getDenominator());
   }

   // More Fraction methods follow.

} // Fraction class
```

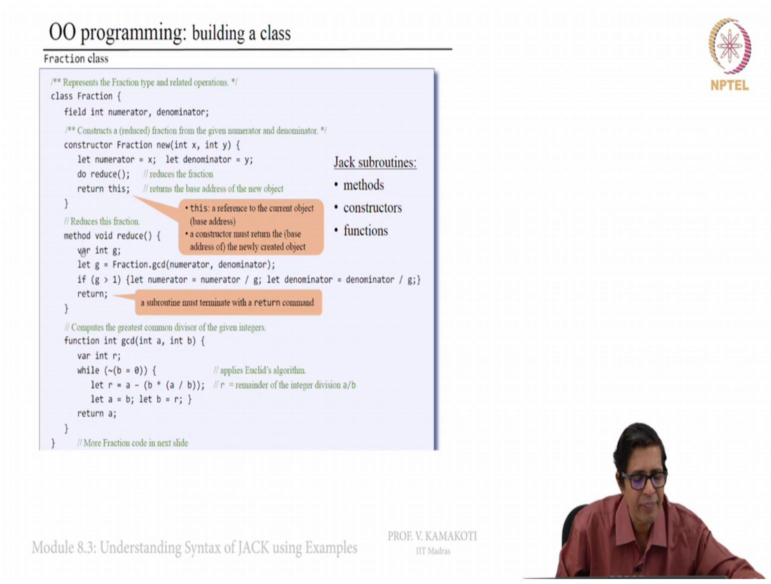
Module 8.3: Understanding Syntax of JACK using Examples

PROF. V. KAMAKOTI
IIT Madras

Right, so the last thing that we need to see in the fraction as we have seen is plus right, so this we have seen plus here, so basically the, so class fraction, so we have put all the other things here, so method fraction plus fraction other, some other fraction is input here and you have to submit up, so were not sum numerators let sum numerator equal to the numerator of the current fraction into the denominator of the other fraction plus the numerator of the other fraction that you get it by using other.get numerator into denominator of the current fraction.

If I want to do A by B plus C by D it is AD+ BC divided by BD right, so the numerator sum is current numerator into other fellows denominator plus other fellows numerator into current fellows denominator and return fraction.new create another fraction which is sum numerators whatever, denominator into others denominator, so this will be so this fraction that new in turn will again call a reduce and it will give you a reduce version of this and this is how this whole thing works.

(Refer Slide Time: 17:18)



## OO programming: building a class

Fraction class

```
/** Represents the Fraction type and related operations. */
class Fraction {
   field int numerator, denominator;
   /** Constructs a (reduced) fraction from the given numerator and denominator. */
   constructor Fraction new(int x, int y) {
      let numerator = x;  let denominator = y;
      do reduce();     // reduces the fraction
      return this;     // returns the base address of the new object
   }
   // Reduces this fraction.
   method void reduce() {
      var int g;
      let g = Fraction.gcd(numerator, denominator);
      if (g > 1) {let numerator = numerator / g; let denominator = denominator / g;}
      return;
   }
   // Computes the greatest common divisor of the given integers.
   function int gcd(int a, int b) {
      var int r;
      while (~(b = 0)) {              // applies Euclid's algorithm.
         let r = a - (b * (a / b));   // r = remainder of the integer division a/b
         let a = b; let b = r; }
      return a;
   }
}      // More Fraction code in next slide
```

- this: a reference to the current object (base address)
- a constructor must return the (base address of) the newly created object

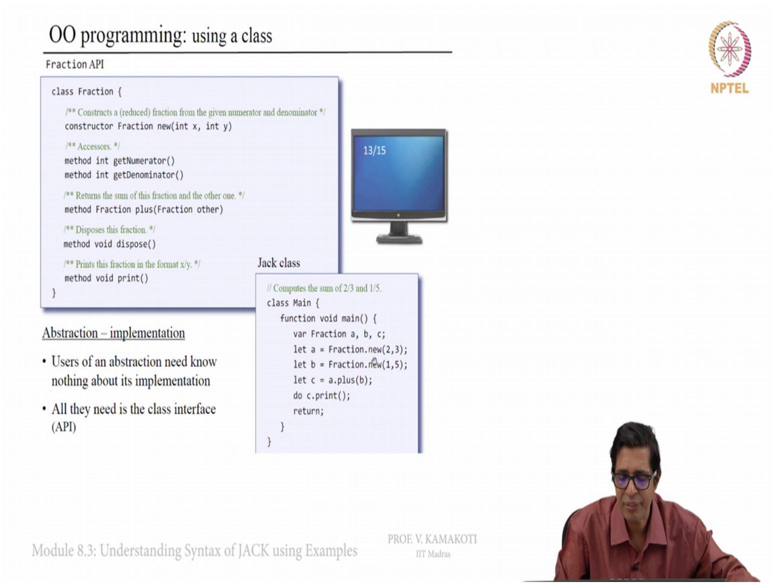
a subroutine must terminate with a return command

Jack subroutines:

- methods
- constructors
- functions

Module 8.3: Understanding Syntax of JACK using Examples

PROF. V. KAMAKOTI
IIT Madras



## OO programming: using a class

Fraction API

```
class Fraction {
    /** Constructs a (reduced) fraction from the given numerator and denominator */
    constructor Fraction new(int x, int y)

    /** Accessors. */
    method int getNumerator()
    method int getDenominator()

    /** Returns the sum of this fraction and the other one. */
    method Fraction plus(Fraction other)

    /** Disposes this fraction. */
    method void dispose()

    /** Prints this fraction in the format x/y. */
    method void print()
}
```

Jack class

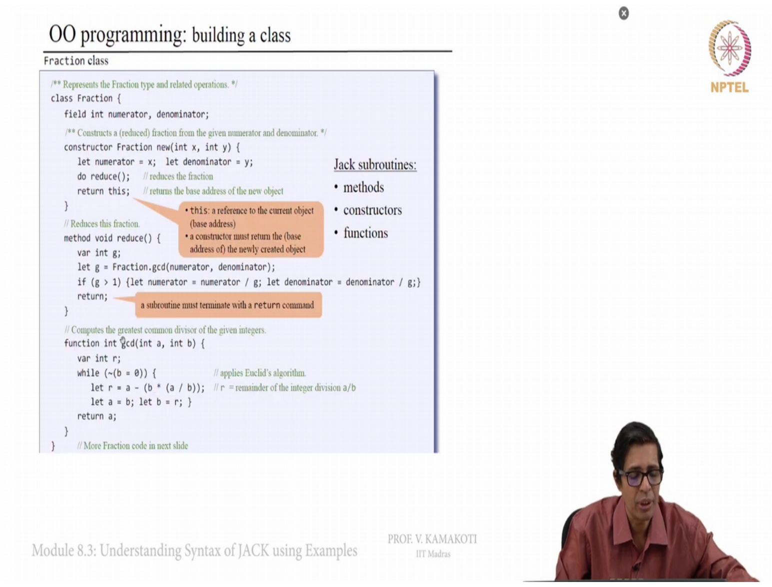
```
// Computes the sum of 2/3 and 1/5.
class Main {
    function void main() {
        var Fraction a, b, c;
        let a = Fraction.new(2,3);
        let b = Fraction.new(1,5);
        let c = a.plus(b);
        do c.print();
        return;
    }
}
```

Abstraction – implementation

- Users of an abstraction need know nothing about its implementation
- All they need is the class interface (API)

Module 8.3: Understanding Syntax of JACK using Examples

PROF. V. KAMAKOTI
IIT Madras



## OO programming: building a class

Fraction class

```
/** Represents the Fraction type and related operations. */
class Fraction {
    field int numerator, denominator;

    /** Constructs a (reduced) fraction from the given numerator and denominator. */
    constructor Fraction new(int x, int y) {
        let numerator = x;  let denominator = y;
        do reduce();     // reduces the fraction
        return this;     // returns the base address of the new object
    }

    // Reduces this fraction.
    method void reduce() {
        var int g;
        let g = Fraction.gcd(numerator, denominator);
        if (g > 1) {let numerator = numerator / g; let denominator = denominator / g;}
        return;
    }

    // Computes the greatest common divisor of the given integers.
    function int gcd(int a, int b) {
        var int r;
        while (~(b = 0)) {                  // applies Euclid's algorithm.
            let r = a - (b * (a / b));      // r = remainder of the integer division a/b
            let a = b; let b = r; }
        return a;
    }
}       // More Fraction code in next slide
```

- this: a reference to the current object (base address)
- a constructor must return the (base address of) the newly created object

a subroutine must terminate with a return command

Jack subroutines:

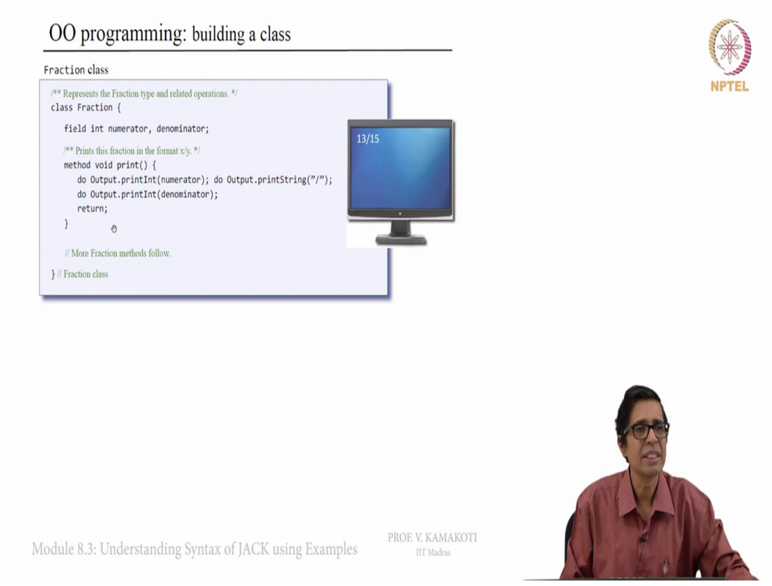
- methods
- constructors
- functions

Module 8.3: Understanding Syntax of JACK using Examples

PROF. V. KAMAKOTI
IIT Madras

So now we see that there were methods and there were functions, what is the difference between method and function? Methods are those which can be used by the external world, functions are those which cannot be use by the external world, for example somebody can say, say suppose I declare fraction A, so where fraction A I can say a.get denominator, a.get numerator, a.+, a.dispose, a.print etc but I can say a.gcd right because gcd is a function inside, I cannot say a.gcd, I can say a.reduce, I can say a.new, I can still say a.+ But I cannot say a.gcd because it is a function inside this.

(Refer Slide Time: 18:14)



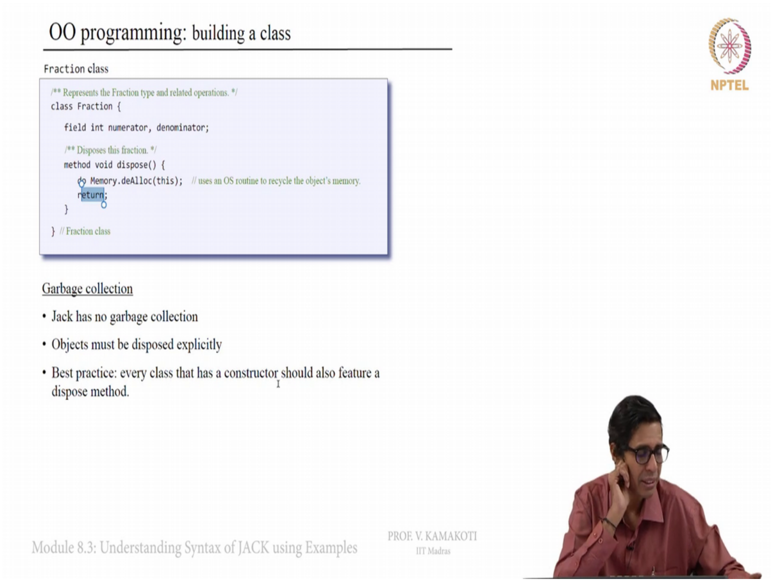
## OO programming: building a class

Fraction class

```
/** Represents the Fraction type and related operations. */
class Fraction {

   field int numerator, denominator;

   /** Prints this fraction in the format x/y. */
   method void print() {
      do Output.printInt(numerator); do Output.printString("/");
      do Output.printInt(denominator);
      return;
   }

   // More Fraction methods follow.

} // Fraction class
```

Now this is how I can, this is a fraction similarly method void to print, do output.print int numerator, do output.print string/ than output.print int denominator so if I give print a fraction a.print it will print the denominator divided by, numerator this/sign divided by denominator, so this is how it print, so this is the total overall how a fraction is being built.

(Refer Slide Time: 18:49)



## OO programming: building a class

Fraction class

```
/** Represents the Fraction type and related operations. */
class Fraction {

   field int numerator, denominator;

   /** Disposes this fraction. */
   method void dispose() {
      do Memory.deAlloc(this);   // uses an OS routine to recycle the object's memory.
      return;
   }

} // Fraction class
```

Garbage collection

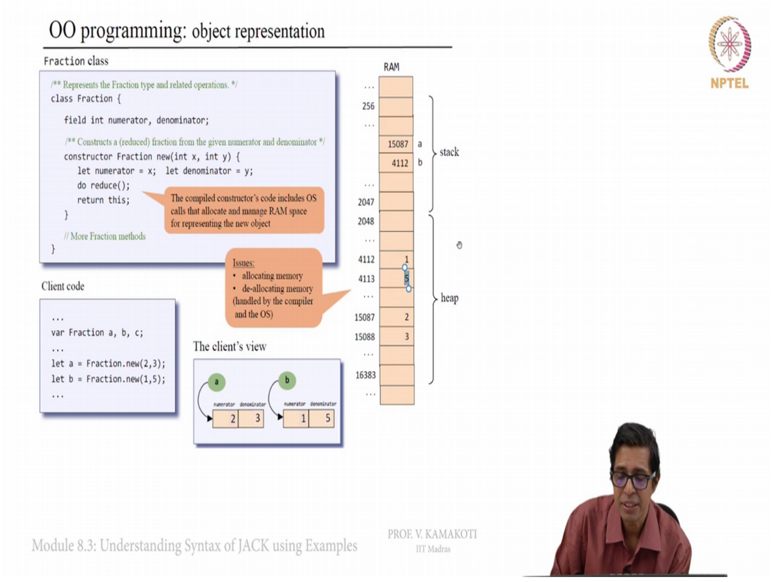
- Jack has no garbage collection
- Objects must be disposed explicitly
- Best practice: every class that has a constructor should also feature a dispose method.

Okay and very importantly jack has no garbage collections, so we are responsible for actually releasing whatever we do not need, so if I use a fraction and I do not need the fraction the first point where I do not need something better there is garbage collection, where the programming language or not, one programming practice that we need to follow rigorously is whenever we do not need something immediately free it out and that is very very important.

So here when we use this method void disposed, so what it does do memory.dalock this, this is the current object, so dalock this, so memory.dalock is equivalent to your free command and who provides you the code for this memory.dalock the voids routine actually provides this method and then you return back right, so best practices every class that has a constructor should also feature a dispose method and this is true, even when you are operating system does support garbage collection, garbage is something like you use my lock if for see people, if you have use my lock and you have never used free than that will be a garbage.

So actually the operating system actually does some garbage collection but in this case we do not to garbage collection, so it is a responsibility for us to free and in general a good programming practice is always to free of the memory whenever you do not need it right, that is very very important.

(Refer Slide Time: 20:27)



## OO programming: object representation

Fraction class

```
/** Represents the Fraction type and related operations. */
class Fraction {

   field int numerator, denominator;

   /** Constructs a (reduced) fraction from the given numerator and denominator */
   constructor Fraction new(int x, int y) {
      let numerator = x;  let denominator = y;
      do reduce();
      return this;
   }

   // More Fraction methods
}
```

The compiled constructor's code includes OS calls that allocate and manage RAM space for representing the new object

Client code

```
...
var Fraction a, b, c;
...
let a = Fraction.new(2,3);
let b = Fraction.new(1,5);
...
```

Issues:
- allocating memory
- de-allocating memory

(handled by the compiler and the OS)

The client's view

Module 8.3: Understanding Syntax of JACK using Examples

PROF. V. KAMAKOTI
IIT Madras

Okay, this is how the whole thing works right, there is a two view, there is a clients view and this is a system view, so whenever we start, this is something that we need to understand very quickly and very nicely, so when the client code is executing he say they have described three variables namely A, B, C as fraction, now we say let A equal to fraction.new 2, 3, let B equal to fraction.new 1,5, so from the clients point of view there is memory address called A which will point to two memory locations storing 2 and 3, there is another B which would point to two memory locations storing 1 and 5, 1 is for the numerator and another is for denominator.

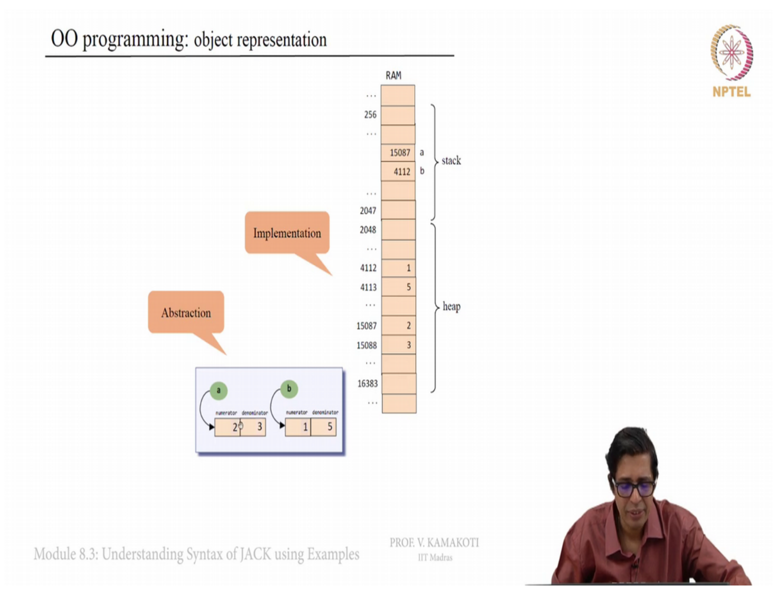
This is the clients view, but in real system what happens is in the stack whenever this classes is encountered let A and B are encountered, there is an address that is created on the heap, that is 15087 and 4112 for example and you go to 15087 there are two locations that are created along with 15087 because this A has two integers numerator and 87 and 88 is stored 2 and 3 for the numerator.

Similarly for B which is stored at 4112, 4112 and 4113 is stores 1 and 5 right and then, so the moment we say fraction, the moment the compiler encounters A, B and C immediately for A, so in this case A and B for example start with, so A and B are allocated a space in the stack and nothing is initialized here, so this things will not be initialized, the moment it is sees A, B, for A and B in the stack there are two spaces allocated.

Then when it executes like a equal to fraction.new of 2, 3 at that point it will go and find out in the heap where are there are two locations and that address it will put 15087 and you go to 15087 you will find the fields of the current fraction namely to and, the numerator and denominator, similarly the moment we execute let be equal to fraction.new of 1, 5, now B is already assigned this location on the stack whatever be this, inside that location we store 4112 and this 4112 is given by the operating system say it is okay, take it and there you go and store in 4112 1and 5 right.

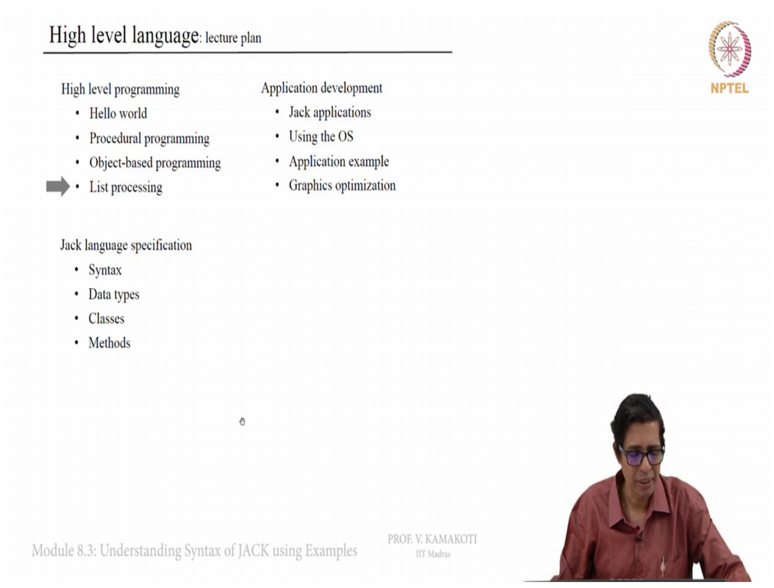
So when I mean by deallocating is memory when I go here, I want deallocate, so in the heap 4112, 4113, 15087, 15088 are both now nonempty, they are not free, the moment I say deallocate A now it will go there, it will find out. Okay 15087 and this is A has two objects inside that, so it will go and free out 15087, 15088 similarly 4112 and 4113 this will be freed out, so this is basically how the fraction works.

(Refer Slide Time: 24:20)



So this is the actual implementation, this is the abstraction of A and B and this is the implementation of A and B right,

(Refer Slide Time: 24:28)



High level language: lecture plan

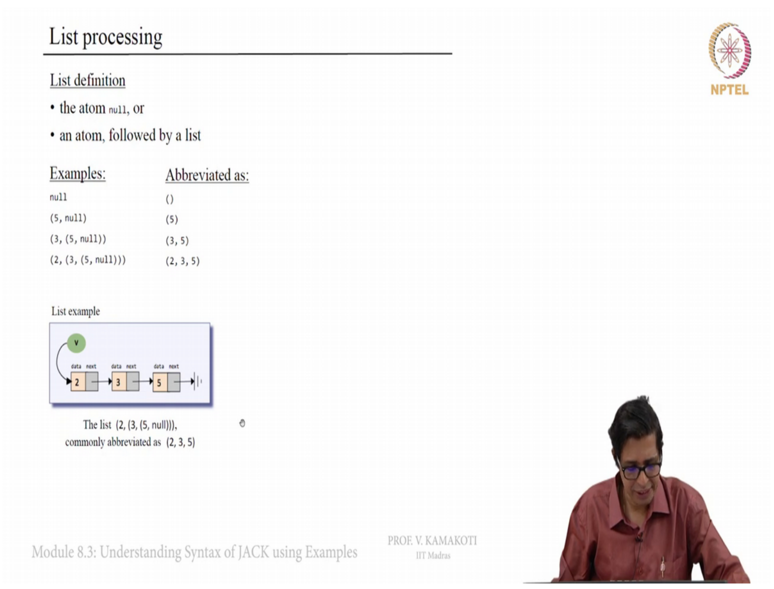
High level programming
- Hello world
- Procedural programming
- Object-based programming
- List processing

Jack language specification
- Syntax
- Data types
- Classes
- Methods

Application development
- Jack applications
- Using the OS
- Application example
- Graphics optimization

Module 8.3: Understanding Syntax of JACK using Examples

PROF. V. KAMAKOTI
IIT Madras



List processing

List definition
- the atom `null`, or
- an atom, followed by a list

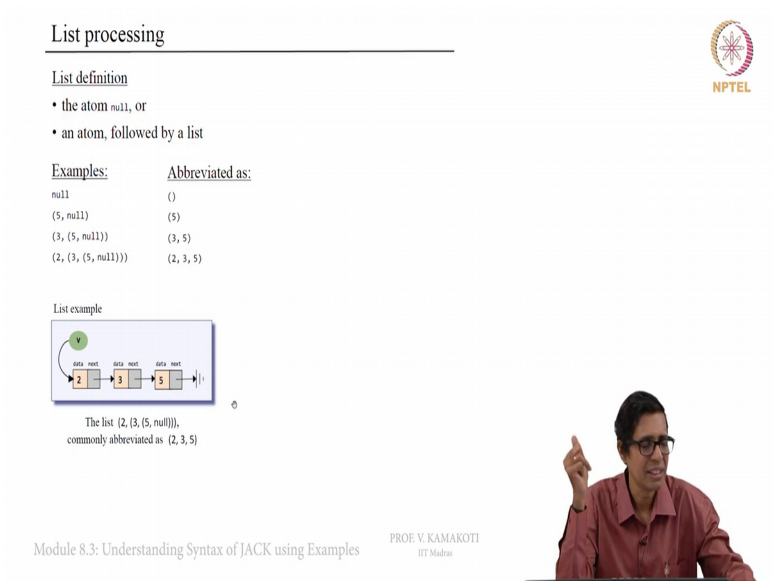
| Examples: | Abbreviated as: |
|---|---|
| `null` | `()` |
| `(5, null)` | `(5)` |
| `(3, (5, null))` | `(3, 5)` |
| `(2, (3, (5, null)))` | `(2, 3, 5)` |

List example

The list (2, (3, (5, null))), commonly abbreviated as (2, 3, 5)

Module 8.3: Understanding Syntax of JACK using Examples

PROF. V. KAMAKOTI
IIT Madras



## List processing

List definition

- the atom `null`, or
- an atom, followed by a list

| Examples: | Abbreviated as: |
|---|---|
| `null` | `()` |
| `(5, null)` | `(5)` |
| `(3, (5, null))` | `(3, 5)` |
| `(2, (3, (5, null)))` | `(2, 3, 5)` |

List example

The list (2, (3, (5, null))), commonly abbreviated as (2, 3, 5)

Module 8.3: Understanding Syntax of JACK using Examples

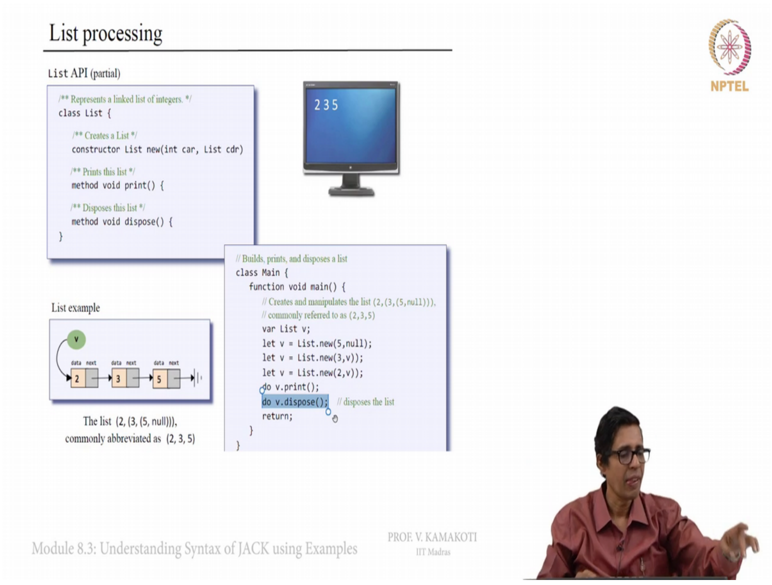
PROF. V. KAMAKOTI
IIT Madras

Right, now we will see the last one example here which is the list to processing, so list is nothing but an element of this form right, so you would have seen link list as a part of your C programming, but it is a self-referential structure, self-referential in this sense, it will store one element as you see here, it will store one element in this case two and it will point to another list of the same type, which will internals store three and it will point another list of the same type which will store five and this is null and there is always a header the list, so this is called a link list and there is an header to the list which you call as V, which points to the first element.

So initially this and null, initially it is null than we had 5 become 5, null than we added 3 and friend of the list, it became 3, 5, null, now 2, 3, 5, null, so list is a data structure in which it stores an element and it points to a similar structure, so 3, 5, is a list, 5 is a list, null is a list, if you take here, if you just look at this figure null is the list, now 5, null is another list, 3, 5, null is another list, 2, 3, 5, null is another list right.

So a list stores an every element of the list, stores an every entry of the list stores an element and a pointer to a list of the same type of it, so 2 this first location here, stores the element 2 and points to another list of the same type which in turns stores 3, 5, etc right, so this is the understanding of the link list, list is the self-referential structure, it refers to another list as a part of its definition.

(Refer Slide Time: 26:47)



## List processing

List API (partial)

```
/** Represents a linked list of integers. */
class List {

    /** Creates a List */
    constructor List new(int car, List cdr)

    /** Prints this list */
    method void print() {

    /** Disposes this list */
    method void dispose() {
}
```

```
// Builds, prints, and disposes a list
class Main {
   function void main() {
      // Creates and manipulates the list (2,(3,(5,null))),
      // commonly referred to as (2,3,5)
      var List v;
      let v = List.new(5,null);
      let v = List.new(3,v));
      let v = List.new(2,v));
      do v.print();
      do v.dispose();   // disposes the list
      return;
   }
}
```

List example

The list (2, (3, (5, null))),
commonly abbreviated as (2, 3, 5)

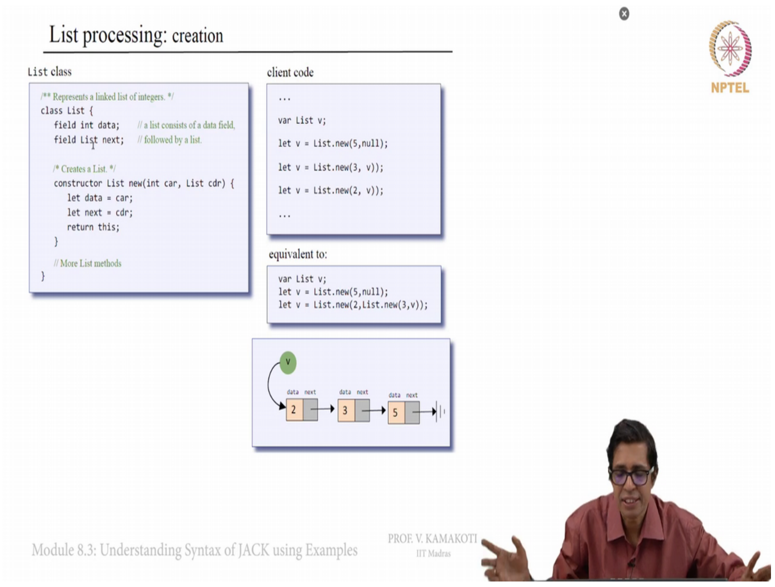
Module 8.3: Understanding Syntax of JACK using Examples

PROF. V. KAMAKOTI
IIT Madras

So how do you define, so this is a thing API for this list, so we can construct a list new which has an initial element and the next element would be null, we can print a list, we can dispose a list okay, so these are all the APIs, so how do you use main, main.main again a class main with their compromise, now where list V, so V is of the type list, V equals to list.new 5, null okay, so this is always, so V equal to list.new 3, V, V equal to list.new 2, V right, there are two extra parentheses here which may be neglected. Okay.

So first V will be 5, null, second it will be 3, 5, null, the third will be make it 2, 3, 5, null and then there is do v.print, can print the entire list starting from the beginning and then do v.dispose, we need to dispose the list also and return, so whenever I do.new I have to go and dispose, so this is how I maintain a list and declare a list, define a list, print a list, dispose a list etc okay.

(Refer Slide Time: 28:22)



## List processing: creation

List class

```
/** Represents a linked list of integers. */
class List {
   field int data;      // a list consists of a data field,
   field List next;     // followed by a list.

   /* Creates a List. */
   constructor List new(int car, List cdr) {
      let data = car;
      let next = cdr;
      return this;
   }

   // More List methods
}
```

client code

```
...

var List v;

let v = List.new(5,null);

let v = List.new(3, v));

let v = List.new(2, v));

...
```

equivalent to:

```
var List v;
let v = List.new(5,null);
let v = List.new(2,List.new(3,v));
```

Module 8.3: Understanding Syntax of JACK using Examples

PROF. V. KAMAKOTI
IIT Madras

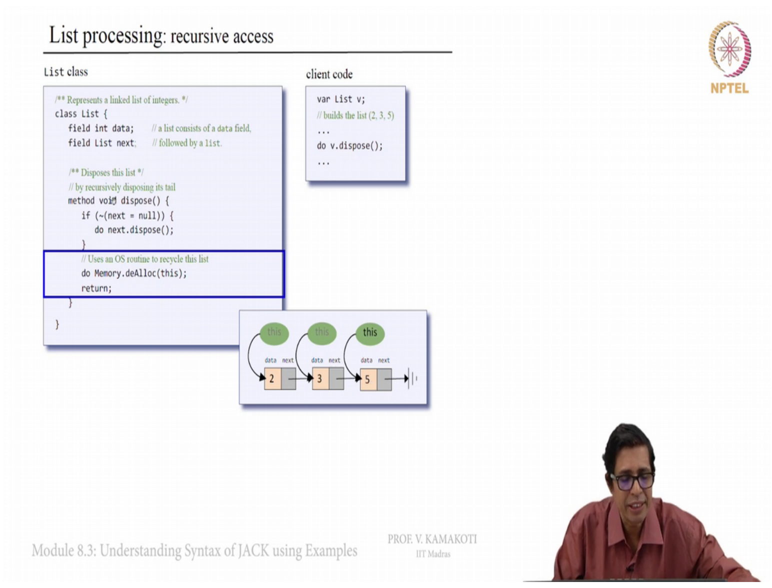
So how is the list, so will now start doing step-by-step, class list, field int data, field list next, so the client code will be VAR list V, V equal to list.new 5, null, V equal to list.new 3, V, V equal to list.new 2, V right, so this is how it goes, so this is the constructor, list new, int car, list cdr, so one will be an integer and another will be the other list and what will be the constructor function do, let it. I equal to car and next is equal to cdr and return this object.

So the first time where list V, the compiler will create one location on the stack with a junk their allocated to V, the moment I say V list.new 5, null, so this list.new will be called here, it will create a space for storing the car and the cdr and so what it will do your car is 5 and your cdr is null, so the next is null and it will return this means, return this object whatever you see here, return this particular object, okay, it will return this particular object.

So your V gets assigned to this particular object, what you mean by return this particular, it will return the address to this particularly, so V, now points to 5, null, now the next statement V equal to list.new 3, V, so the list.new will now find 3, list.new again, it will make the, it will create another data next which is different from the previous data next, it will create another data next in which the data will be 3 and the next will be V, so the next will now point the original V was pointing to this, so you are the next of this will now point to this.

Now since I making this V is equal to next of 3, V, this whole V will go, now the new V will point to this 3, so I have created this list now, now will see the next them again this will be right, so this is how the list can be constructed, so is actually V equal to list.new 5, null, let V equal to list.new, 2, list.new, 3, V, so if I do this, this is equivalent to creating, so I this V is 5, null then I create 3, 5 with this list.new, then I create 2, 3, 5 with null for this, so since a list points to itself is called self-referential structure.

(Refer Slide Time: 31:59)



## List processing: recursive access

List class

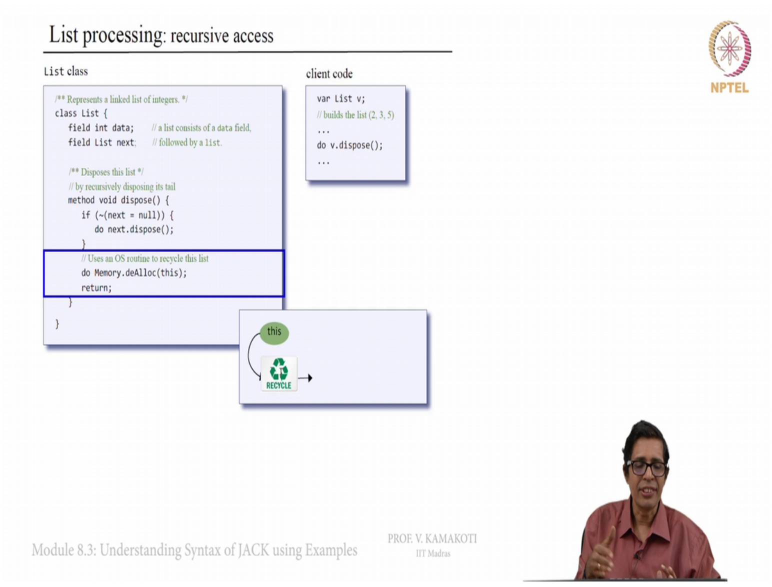
```
/** Represents a linked list of integers. */
class List {
   field int data;      // a list consists of a data field,
   field List next;     // followed by a list.

   /** Disposes this list */
   // by recursively disposing its tail
   method void dispose() {
      if (~(next = null)) {
         do next.dispose();
      }
      // Uses an OS routine to recycle this list
      do Memory.deAlloc(this);
      return;
   }
}
```

client code

```
var List v;
// builds the list (2, 3, 5)
...
do v.dispose();
...
```

Module 8.3: Understanding Syntax of JACK using Examples

PROF. V. KAMAKOTI
IIT Madras



## List processing: recursive access

List class

```
/** Represents a linked list of integers. */
class List {
   field int data;      // a list consists of a data field,
   field List next;     // followed by a list.

   /** Disposes this list */
   // by recursively disposing its tail
   method void dispose() {
      if (~(next = null)) {
         do next.dispose();
      }
      // Uses an OS routine to recycle this list
      do Memory.deAlloc(this);
      return;
   }
}
```

client code

```
var List v;
// builds the list (2, 3, 5)
...
do v.dispose();
...
```

Module 8.3: Understanding Syntax of JACK using Examples

PROF. V. KAMAKOTI
IIT Madras

Now let us say do v.print, so what we do, how do you do this v.print? a var list current, let current equal to this okay, so current will start with V, while current is not equal to null is not, not equal to null, to output.print int current.get data, get data will get me the integer for that and do.print car 32 v space, the ask key equivalent to 32 space and do current equal to current.get next, so you start with current as this, this means V that is as well. This becomes very useful, this means the current object and then current as this V.

Now while current is not equal to null you keep repeating print get data of V that will print and you print it to, then it will put a space, then you say current is equal to, current is this right, current and we assume, current is equal to, so this is this, so now current is this, while current is not equal to null, current.next, current is this, current is not equal to null, no current is pointing to something here, so we just print current.data, now current equal to current.get next, so I go there, then I print current.get data, followed by blank, now current equal to current.next, I print the next data 5 and then over here, current equal to null, yes it is null, so I get data right, so this is how 2, 3, 5 is right.

The last thing is dispose, dispose is a very interesting algorithm and this is recursive algorithm as you see, so method void dispose, so again method is used when somebody outside can use it with class, function is something which is used internally by the class like GCD in the case is a function while here, dispose is a method or dispose there also the case of fraction was a method, now let us see while next is not equal to null, do next.dispose, when next is equal to null and after doing this particular statement, this is the for statement, so this recursively calls dispose, dispose calls itself and after you come out of the recursive call to memory.allocate this and return and go okay, right.

Now how do we do this, now so I have V and I want to dispose this entire thing, so how do you do this, first v.dispose so this is this, immediately this means this is pointing to this, while if not next is equal to null is not truly are. Okay, because we have V this.next is not equal to null, do to next.dispose, so the moment I say next.dispose another function is called, so that will have its own activation record etc and that this will be pointing to 3 this one.

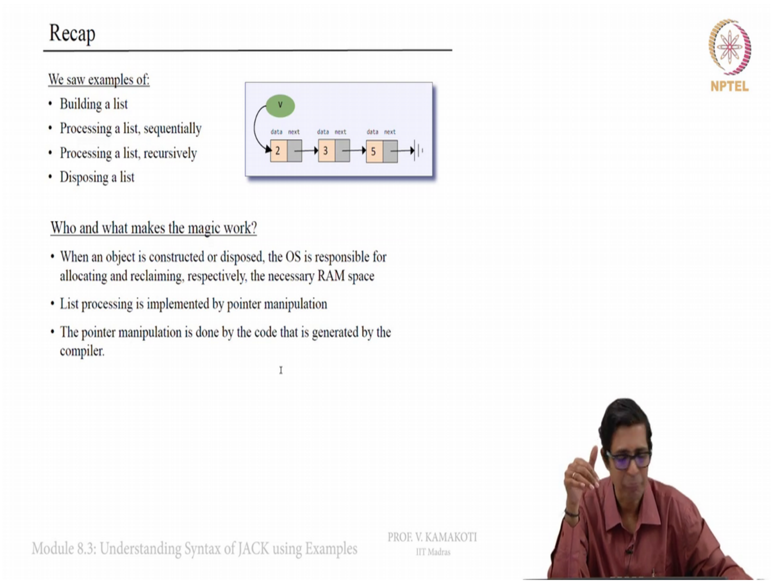
Now the next of this is not again null, so I will again call next.dispose and that, that this again, now that this will be pointing to 5, so always that this is a keyword which will point to the current object, now this.next is actually null, so I come to this part, so I do memory.deAlloc of this, so this memory gets deallocated and they returned, return to what the fellow who called me, it was this.next, this is the fellow who called me.

Now I go and after I, so where did he call, he called at this point, when I returned I have to go to this pointer right, so here I find do.memory dealloc of that this and that is going for recycle and after that I return to the previous call of dispose in which this was pointing to this and now I dispose this right.

So the dispose is a function which called another dispose which in turn call another dispose and every time I was going to its different list and every dispose had its own this right, so that this, so then we dispose the last one, then the next one, then the next want, so we went one full scan from that top of the list to the end of the list and while returning back we disposing them one after another and we came here but note that every call had its own this right.

So now that when we implemented the return and call and return functions of the VM, we were storing that this of every calling function and so that when it returns I go back to the this of that particular calling function and this basically explains you the notion of the keyword this okay, right.

(Refer Slide Time: 38:30)



Recap

We saw examples of:

- Building a list
- Processing a list, sequentially
- Processing a list, recursively
- Disposing a list

Who and what makes the magic work?

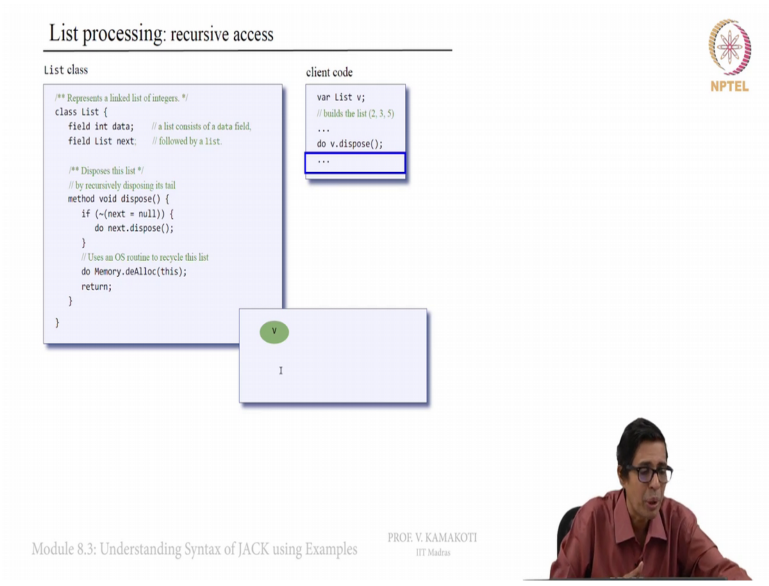
- When an object is constructed or disposed, the OS is responsible for allocating and reclaiming, respectively, the necessary RAM space
- List processing is implemented by pointer manipulation
- The pointer manipulation is done by the code that is generated by the compiler.

Module 8.3: Understanding Syntax of JACK using Examples

PROF. V. KAMAKOTI
IIT Madras

So what did we do so far we saw examples of building a list, processing a list sequentially, processing a list recursively, disposing a list and this magic actually works, so when an object is constructed or dispose, the OS is responsible for allocating and reclaiming, respectively, the list processing is implemented by pointer manipulation as you see here and the pointer manipulation is done by code that is generated by the complier.

(Refer Slide Time: 39:03)



## List processing: recursive access

List class

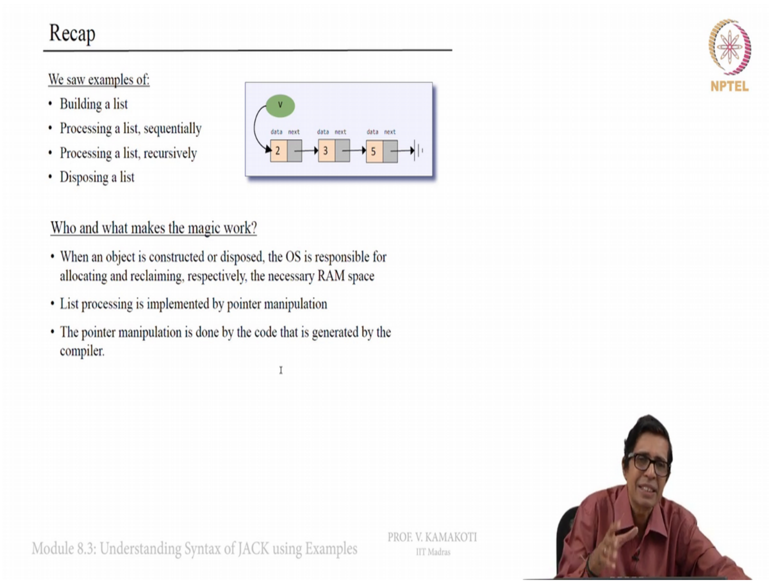
```
/** Represents a linked list of integers. */
class List {
   field int data;      // a list consists of a data field,
   field List next;     // followed by a list.

   /** Disposes this list */
   // by recursively disposing its tail
   method void dispose() {
      if (~(next = null)) {
         do next.dispose();
      }
      // Uses an OS routine to recycle this list
      do Memory.deAlloc(this);
      return;
   }

}
```

client code

```
var List v;
// builds the list (2, 3, 5)
...
do v.dispose();
...
```

Module 8.3: Understanding Syntax of JACK using Examples

PROF. V. KAMAKOTI
IIT Madras



## Recap

We saw examples of:

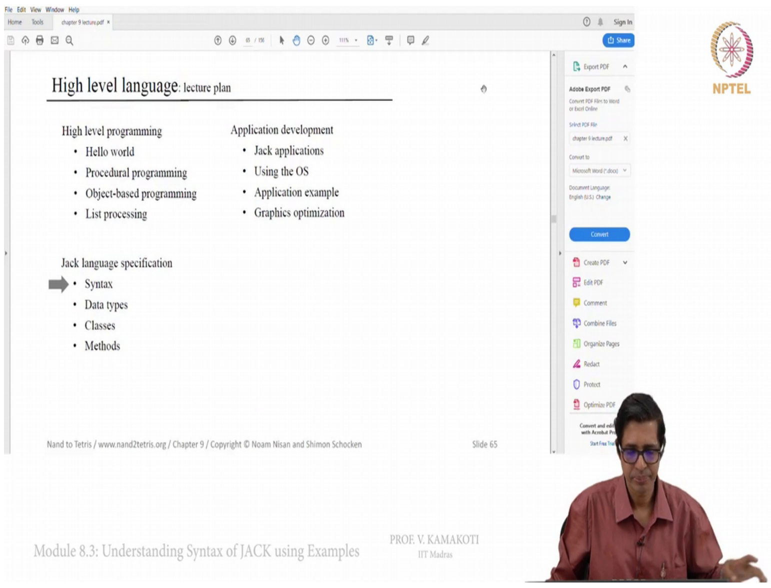
- Building a list
- Processing a list, sequentially
- Processing a list, recursively
- Disposing a list

Who and what makes the magic work?

- When an object is constructed or disposed, the OS is responsible for allocating and reclaiming, respectively, the necessary RAM space
- List processing is implemented by pointer manipulation
- The pointer manipulation is done by the code that is generated by the compiler.

Module 8.3: Understanding Syntax of JACK using Examples

PROF. V. KAMAKOTI
IIT Madras



## High level language: lecture plan

High level programming

- Hello world
- Procedural programming
- Object-based programming
- List processing

Application development

- Jack applications
- Using the OS
- Application example
- Graphics optimization

Jack language specification

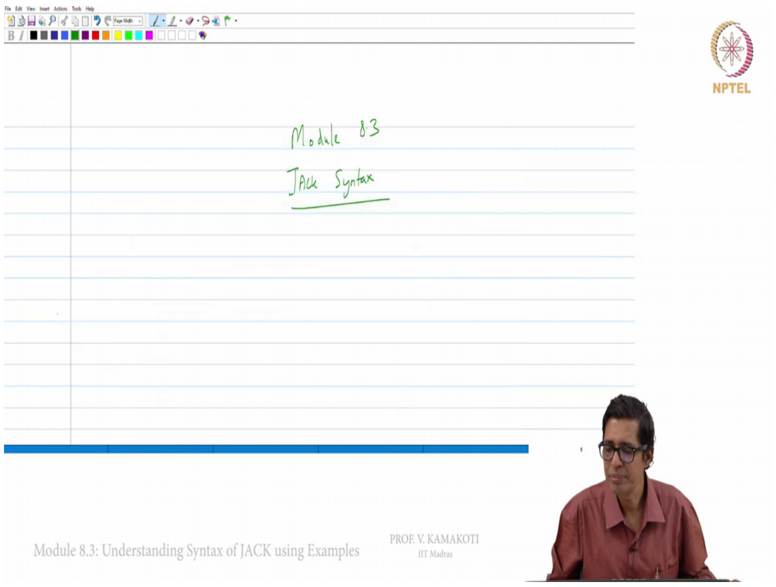
- ➡ Syntax
- Data types
- Classes
- Methods

Nand to Tetris / www.nand2tetris.org / Chapter 9 / Copyright © Noam Nisan and Shimon Schocken

Slide 65

Module 8.3: Understanding Syntax of JACK using Examples

PROF. V. KAMAKOTI
IIT Madras



So you write this code and the compiler has to compile it into a VM code which will manipulate this pointer, so we are going to see all those things, but as I said just programming language this is all we can use it right, so before we go into the syntax of the Jack language specification, we will now go to the next part of project 09 wherein we will show a couple of more examples before we start looking at the Jack language in more detail. Thank you.