Implementation list asignment 2

CLASS MODE	LECTURE
Notes	☐
Property	
SLIDES	
TOPIC	

Assignment 2

Specification of the list is a specification of a sorted list → only types E on which order has been defined can be elements of the list.

```
interface ListInterface <E extends Comparable<E>> {
  ...
}
```

→ The class that implements the list knows ever element, whatever the type E, contains a method compareTo() to sort the elements.

▼ 17: Return types in specifications

1) Part 1

- Example

- Suppose we have an ADT for a type X, specification in an interface Xinterface and implementation in class X. The X-ADT contains an operation ADT.
- In assignment 1, we would have created

```
X concat (X rhs);

with the corresponding method concat() in the class X

X concat (X rhs) {
...
}
```

-However, in specification we cannot know anything about the implementation (what is inside the class) and the program → the name of this class may not be known

-Correct code → However, since users will work with X-objects, this would force users to cast result from type Xinterface to type X.

```
Xinterface concat (Xinterface RHS);

Xinterface concat (Xinterface rhs( { // the interface is allowed to know its own name
...
}
```

Correct and most efficient code → Allowed because of polumorphism. Every object that is instance of class X has the type Xinterface (because class X implements class Xinterface).

```
X concat (Xinterface rhs) {
...
}
```

Always use name of interface if talking about type of obejct

Part II

- Example

```
Identifier ident = new Identifier();
ident.init("a").add("b").add("c").size(); // init makes it possible to use add
```

For identifier-ADT this is possible if operations like init() and add() (usally declared as void methods), are changed into methods that return the Identifier-object instead.

Old situation

```
void add (char c) {
  // code
}
```

New situation

```
Identifier add (char c) {
  // code
  return this;
}
```

▼ 18: Generic ADT's 1 (basics)

▼ Change interface NumberStackInterface to StringStackInterface → Use typeparameters

```
interface NumberStackInterface {
  Elements: objects of type Number
  Structure: linear
  Number pop ();
  Number top ();
  NumberStackInterface push (Number n);
}
```

```
interface StringStackInterface {
  Elements: objects of type String
  Structure: linear
  Number pop ();
  Number top ();
  NumberStackInterface push (String s);
}
```

```
interface StackInterface<E> { // it is unknown what E is when writing ADT
  Elements: objects of type E
  Structure: linear
  Number pop ();
  Number top ();
  NumberStackInterface push (E e);
}
```

- Stack that can contain anything

```
class Stack<E> implements StackInterface<E> {
```

```
NumberStack push (number e) {
...
}

becomes

Stack<E> push (E e) {
...
}

However, you cannot just replace
Number[] numberArray with E[] stackArray because E is not known yet, array cannot be made yet

Correct solution:

Object[] row2 = new Object[...]
```

- Replacing constructors

```
Stack (...) { // Create constructor without type parameter
...
}

Stack <E> stack = new Stack <E> (...) // Write type parameter
```

See Canvas for demo program

```
Stack<String> stringStack = new Stack<String>();
stringStack.add("abc");
```

```
Stack<Integer> intStack = new Stack<Integer>();
intStack.add(15);
```

- Easy example of making method with value parameters so you don't need to write 2 different methods

```
int a, b, n;

int powerAN {
  int result = 1;
  for (int i = 0; i < n; i++) {
    result *= a;
  }
  return result;
}
```

```
int a, b, n;

int powerBN {
  int result = 1;
  for (int i = 0; i < a; i++) {
    result *= b;
  }
  return result;
}
```

```
int power (int base, int exponent) {
  int result = 1;
  for (int i = 0; i < exponent; i++) {
    result *= base;
  }
  return result;
}
```

example call:

```
int c = power(a,n)
```

- 19: Exceptions 3
  - All exceptions are subclasses of the class Exception
  - Catching any exception (NullPointer & ArrayIndex) with try-catch statement:

```
try {
  ...
} catch (Exception e) {
  ...
}
```

- To make sure program only catches our written exceptions, only throw and catch exceptions that are an instance of the class below:

```
class APException extends Exception {

  APException (String s) { // essential constructor!!!, otherwise Java fills in default constructor from superclass
    super(s);
  }
}
```

▼ 20: Interfaces 3: Purpose, CompareTo<E>

- Interfaces used to specify common properies/types for classes
- Example
  Different type of bank accounts (classes), that all implement the same interface. The interface contains all similar operations.

  ```
  interface Account {
    Account withdrawal (double amount);
    Account deposit (double amount);
    double balans();
  }
  ```

- Example
  Type of different things, but they all implement the same interface because they have c**ommon properties**, e.g. interface Value
- ComparableTo interface (often used)

```
interface Comparable<E> {
  int compareTo(E e);
}
```

```
a.compareTo(b)

returns:
a before b --> negative int
a b equal in ordering --> 0
a after b --> positive int
```

```
"abc".compareTo("xyz") < 0
"abc".compareTo("abc") = 0
"xyz".compareTo("abc") < 0
```

▼ 21: Modifiers 2: Final, Abstract

final

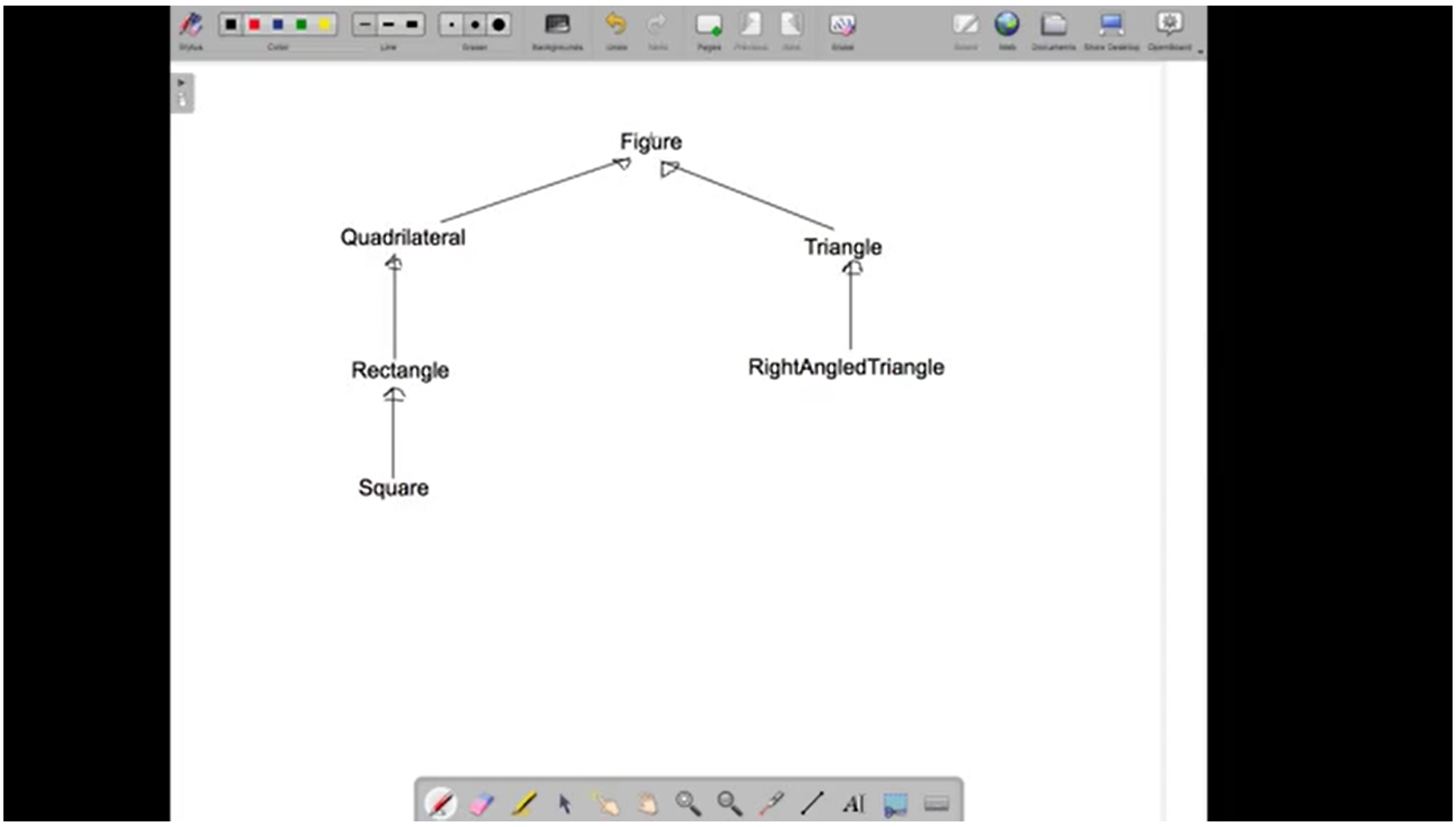
```
final <variable> --> makes variable a constant
final <method> --> method cannot be overriden (use in subclass)
final <class> --> class cannot have subclasses
```

- Abstract

```
abstract <method> --> method does not have a body but instead a semicolon. Write down methods that do not have a body yet.
abstract <class> --? class cannot be instantiated. Not possible to create objects that are instances of an abstract class.
```

```
abstract void m2 ();
```

- If a class contains an abstract method, it won't compile unless the class is declared abstract too
- A class can be declared abstract even if it does not contain abstract methods
- Where to use abstract classes? → Make figure and quadrilateral abstract




```
abstract class Figure {
  ...
  abstract double area ();
  abstract double circumference ();
  ...
}
```

```
 class Quadrilateral extends Figure {
  ...
  abstract double area ();
  abstract double circumference ();
  ...
}
```

```
abstract class Rectangle extends Quadrilateral {
  ...
  double area () { // forced to use this method, trick: return 0.0
    ...
  }
  double circumference () {
  ...
  }
}
```

```
Interface I {
  int MAX = 100;

  void m ();
}

is short version of

(public) interface I { // every interface is public
  public static final int MAX = 100;

  (public abstract) void m ();
}

This won't compile

class C implements I {
  // nothing
}

as m() is not implemented, but

abstract class C implements I {
  // nothing
}

will compile
```

```
Interface I {
  void m();
}

class C implements I {
}

does not compile
```

- Force using certain methods in certain classes
  1. System of abstract classes
  In is-A relation (see figure above)
  2. Interface

▼ 22: Generic ADT's: Bounds

- Bounds (on type parameters) limit the allowed types for the type parameter

```
class X<E> {
  ...
}

the possible types for E can be limited with a bound:

class X<E extends "bound"> {
  ...
}
```

- The "bound" is either a class or interface
- With bounded type parameter only types that also have the type of the "bound" are allowed

```
class Y<E> {
  // empty
}
class X1<E> {
  // empty
}
class X2<E> extends Y<E>{ // types of type parameter either Y or subclasses of Y
  // empty
}
class X3Y<E> extends Comparable<E> { // only types if there is an order defined of these types, so if they implement comparable
  // empty
}
```

▼ 23: Assignment operator

- Well known, the assignment statement

```
<variable> = <expression>;
```

- An assignment is also an expression, as the "="-symbol is an operator.
- The value returned by the expression is the value assigned to the variable.
- Operators like "+" and "-" are evaluated left to right

```
1 + 2 + 3 --> 3 + 3 --> 6
```

- The assignment operator is evaluated right to left

```
int a;
int b;

a = b = 23 --> a = 23 --> 23 // "=" is not an assignment but an expression using the assignment operator

After the evaluation, both a and b have value 23
```

▼ 24: Lists: Singly linked lists

- Lists: dynamic sequential linear datastructures
    - Dynamic: number of elements can grow and shrink
    - Sequential: there are only operations to go to the first/last/prior/next element —> going to the i-th element is expensive
    - Linear datastructures from introductory course was array. Arrays are static random acces linear datastructures. They are
      - Static: number of elements is fixed
      - Random acces: there is an operation to go to the i-th element (with index)
    - List contains 2 variables
      1)
      2) Reference to object of the list
    - Nodes = elements of the list = objects that contain elements
    - Last element points to null
    - Variable list (that contains list) points to first element of list
    - Example of making list

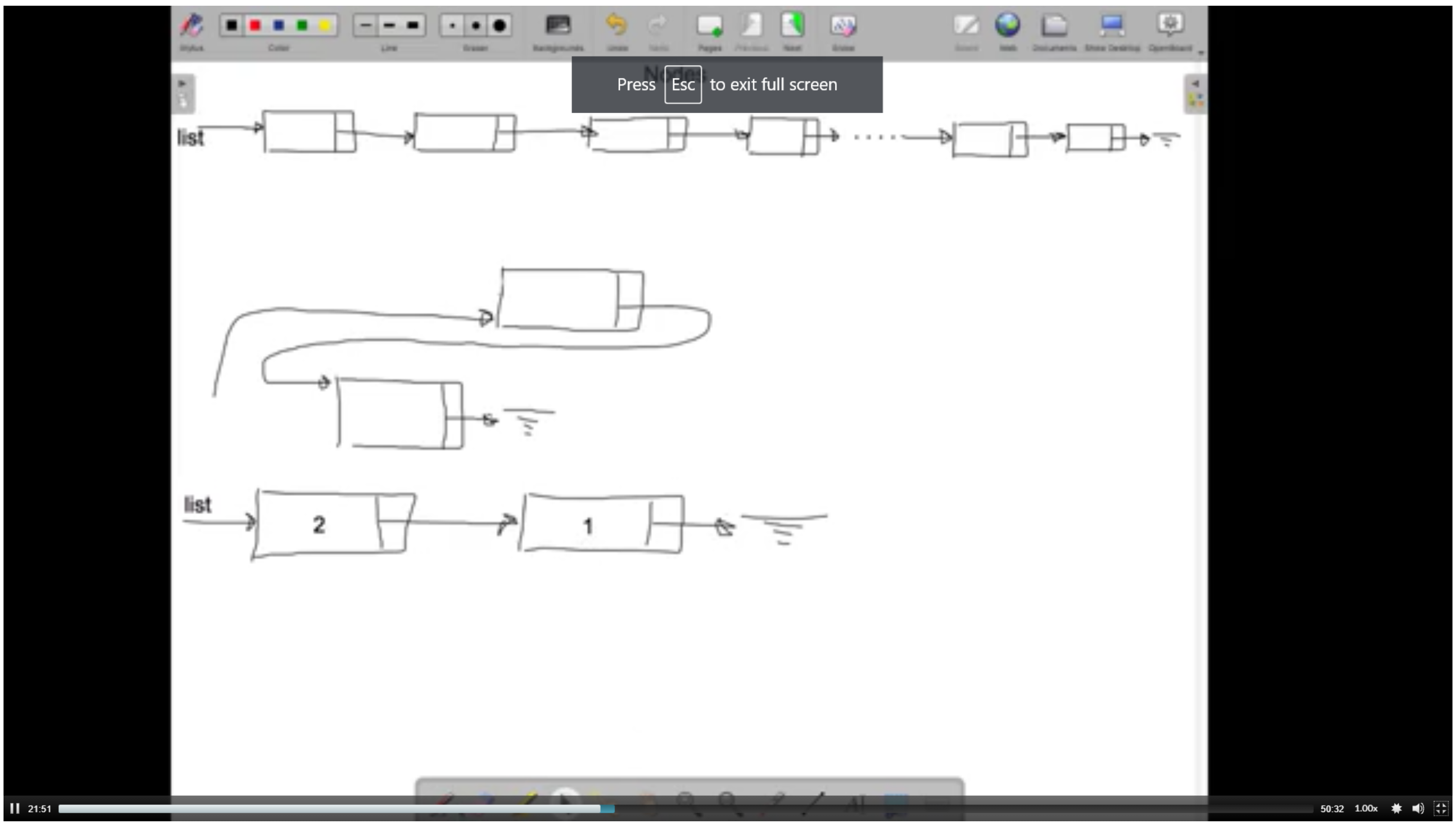
```
class Node {

int data;
Node next; // next is a reference, not an object.

Node (int i, Node next) { // constructor 1
  data = i;
  this.next = next;
}

Node (int i) { // constructor 2
  this(i, null);
}

Node () {
  this(0, null); // constructor 3
}
```

- Overloading: multiple constructors
  → Allowed here because different parameters

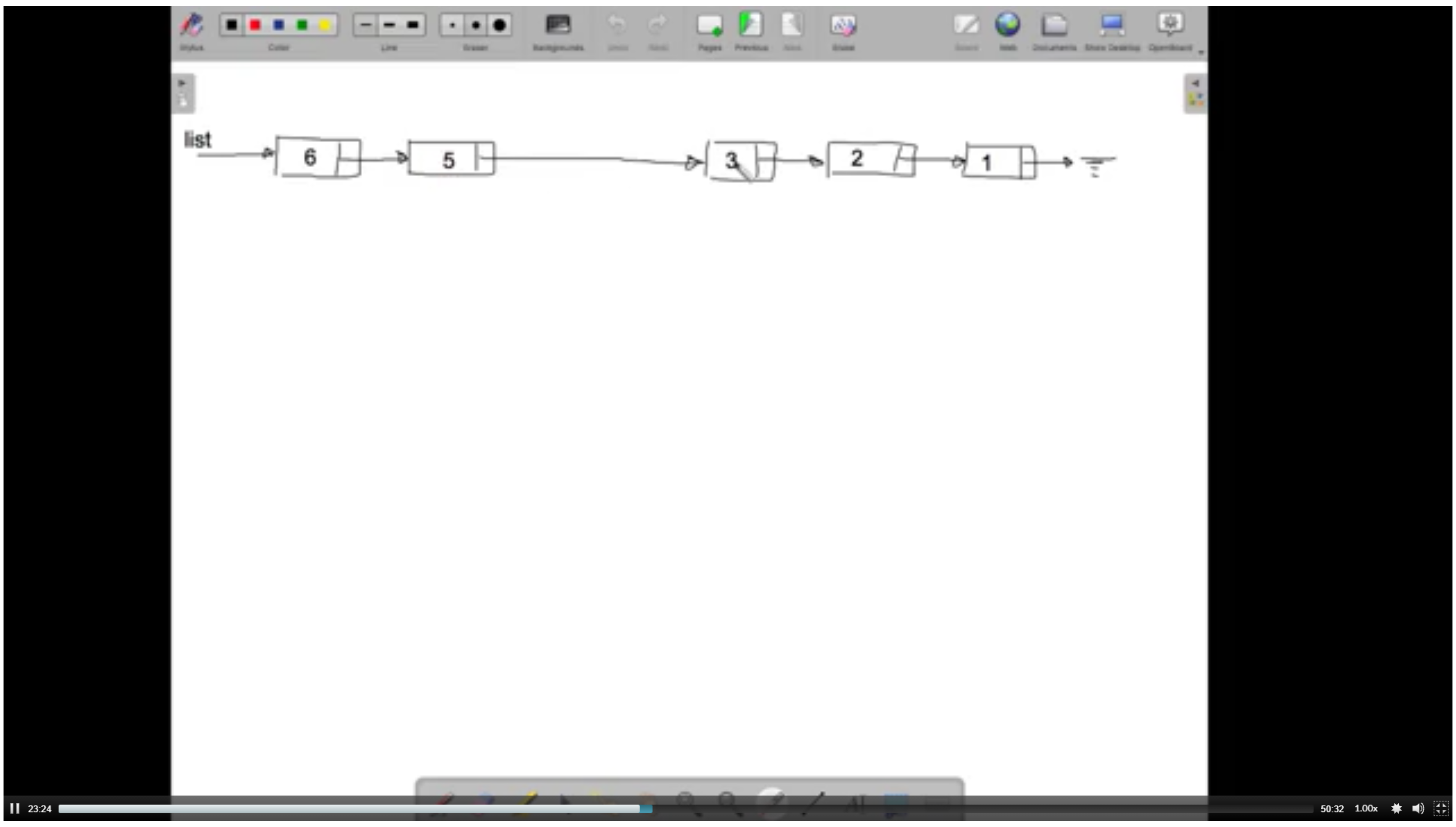
```
Node list;

// 1) Make the empty list:
list = null;

// 2) Add 1 node with value 1 in the list
list = new Node(1); // list points to new node and this new node points to null
```

```
// 3) Add 1 node with value 2 in the list
list = new Node(2, list);

// 4) Add node 4 after node 5. Find node with value 4
Node n = list;
while (n.data != 5) {
  n = n.next;
}

n.next = new Node (4, n.next); // Create new node such that list: 6->5->4->3->2->1

// 5) Add node 0 after the last
Node n =  list;
while (n.next != null) {
  n = n.next;
}

n.next = new Node (0); // bc last node points to null

// 6) Remove the 6 (1st element). We do this by refering list to node 5. If 6 has no reference, it is removed.
list = list.next;

// 7) Remove the 3 (element in the middle). We want help reference to node before the one we want to remove (here: node 4).
Node n = list;
while (n.next.data != 3) {
  n = n.next;
}

n.next = n.next.next;

// 8) Remove the last element without knowing its content
Node n = list;
while (n.next.next != null) {
  n = n.next;
}

n.next = null;

// 8) Remove the only element (so we assume the list consists of 1 node)
list = null;
```

Step 3

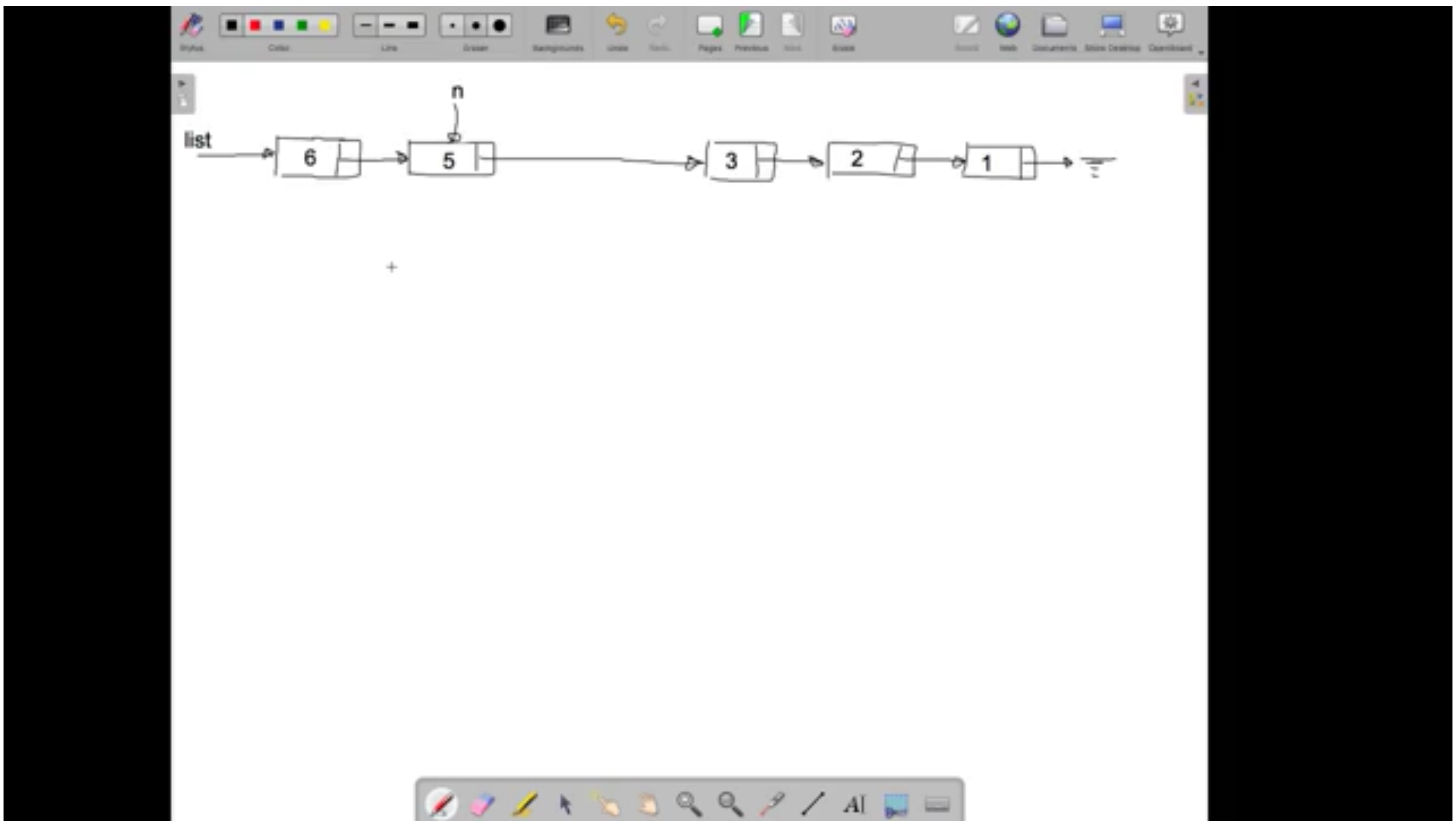


Step 4: Assume we repeated the process, and now have the following result

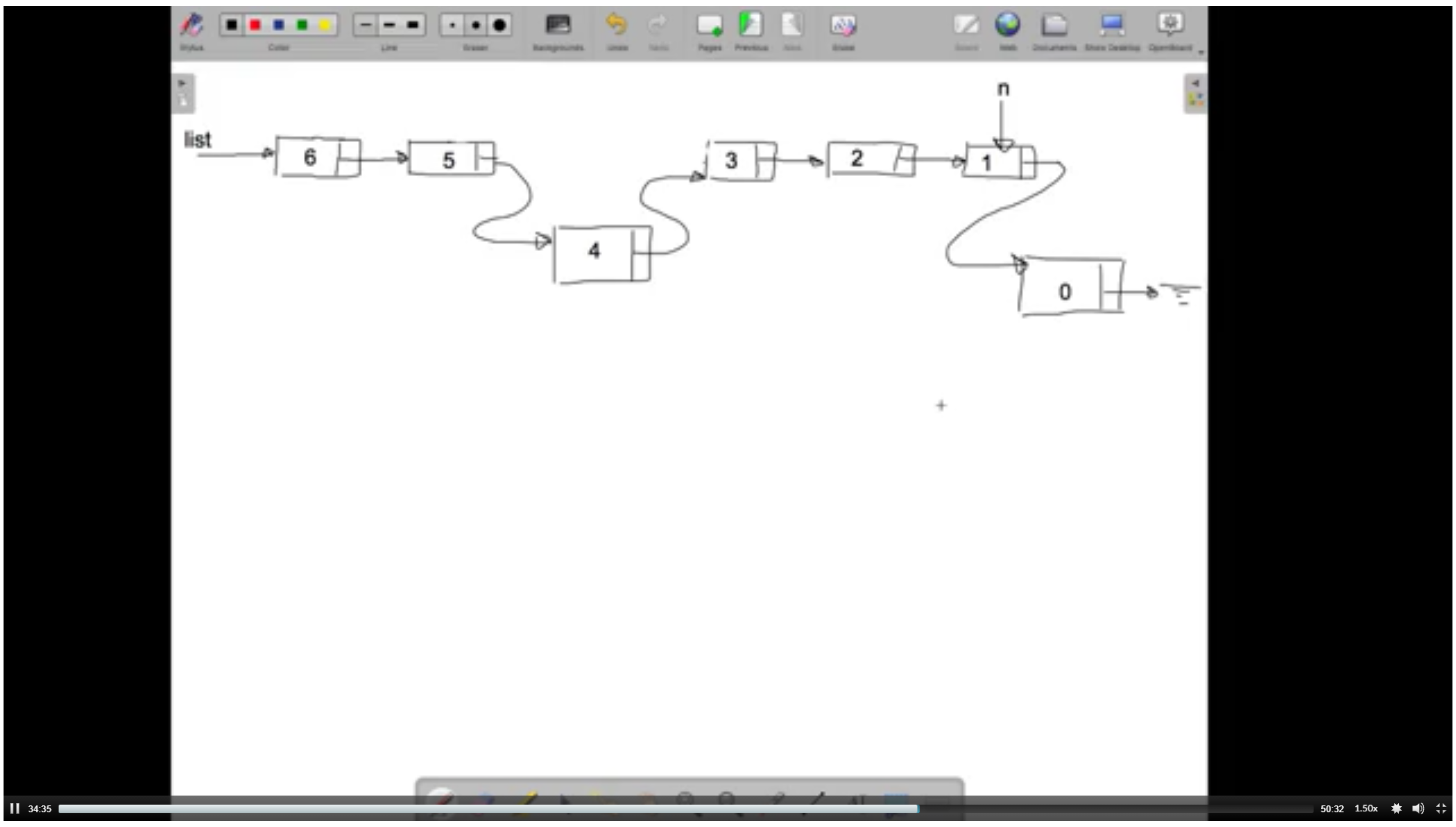


How can we add the 4 after the 5?

- Use n (help reference) to first element
- Move this to 2nd element, 3rd elements etc till it points to node with value 5



Step 5

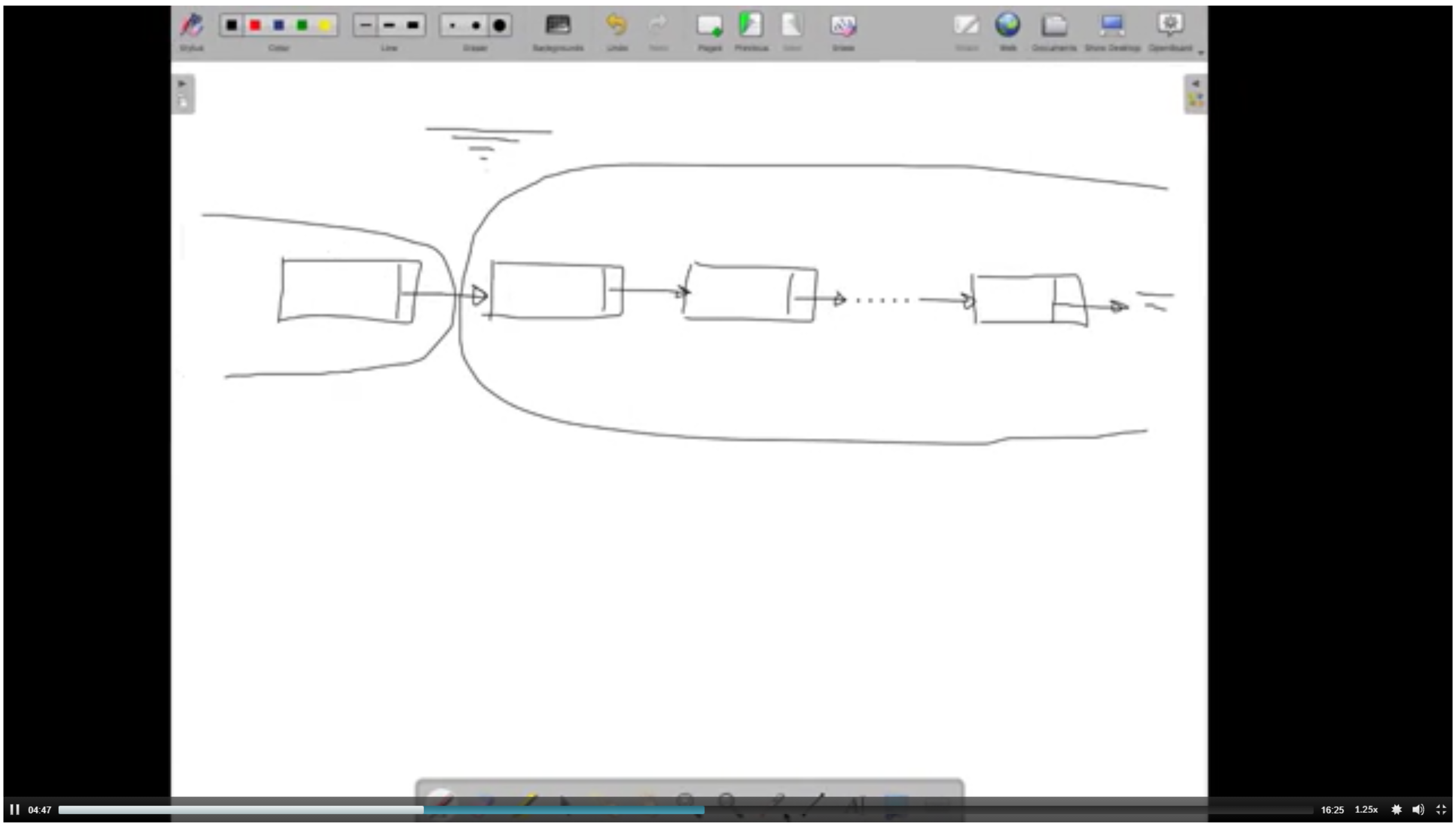


- Check if the next methods are true when you make a list:
  1. Add/remove to the empty list
  2. Add/remove a first element
  3. Add/remove a last element
  4. Add/remove an element "in the middle"

▼ 25: Lists (a list is a recursive datastructure)

- Every list is
  1) Empty
  2) Contains 1 node that points to a list
- Example: Recursive method that counts the number of nodes

```
int numberOfNodes (Node l) {
  if (l == null) { // stop condition: simplest case
    return 0;
  }

  return 1 + numberOfElements(l.next); // so new list that skips old 1st element
}

boolean contains (Node l, int i) {
  if (l == null) {
    return false; // bc if no nodes, there is no node with content i
  }

  return l.data == i || contains(l.next, i); // || means or
}
```



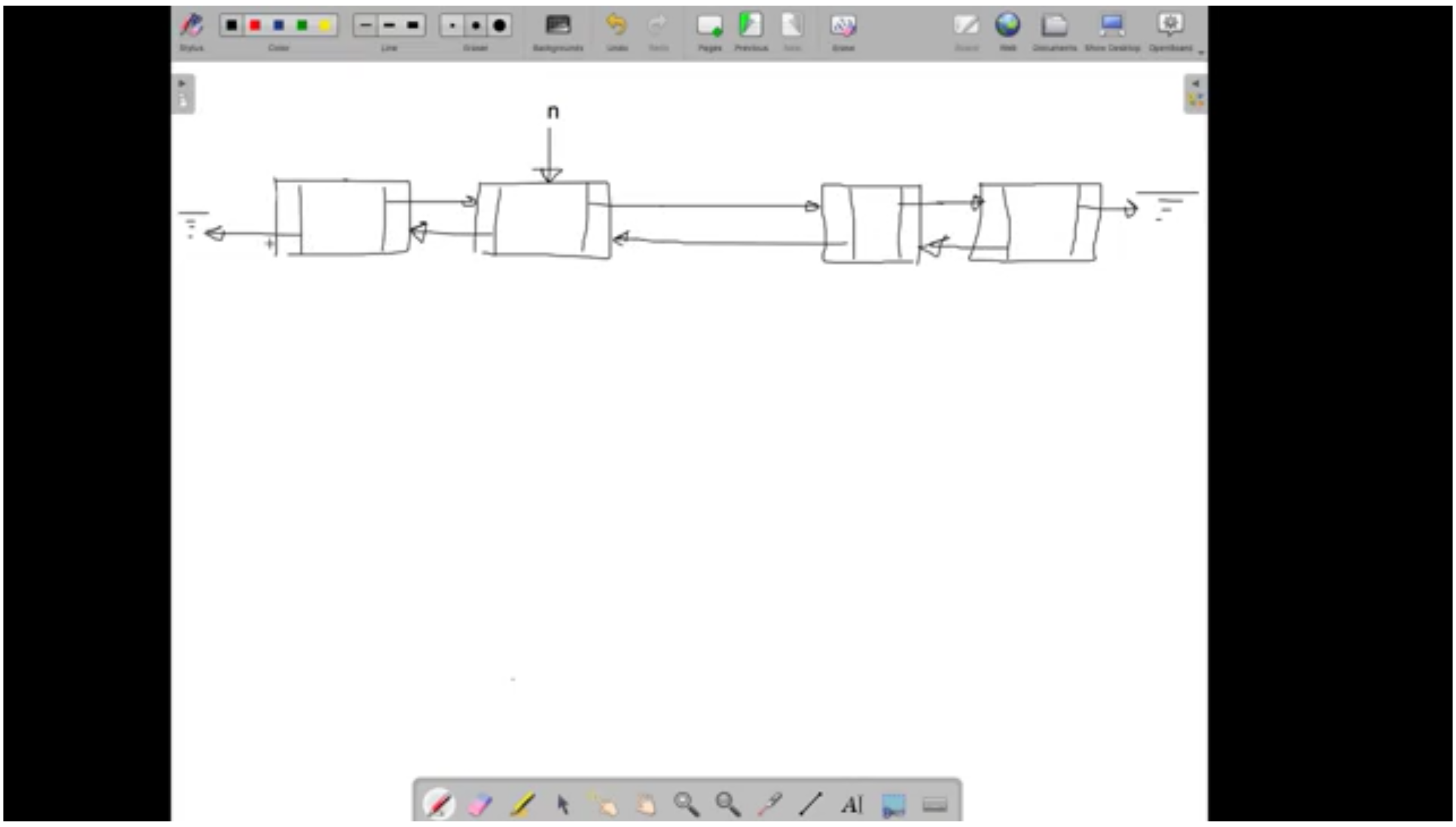
```
Find maximum in a list
Math.max(2,3) == 3;
```

```
Recursively
int maximum (node l) {
  if (l == null) {
    return Integer.MIN_VALUE; // is most negative value in Java
  }

  return Math.max(l.data, maximum(l.next));
```

```
In contrast,
0+1+2+3+4+5 --> 0 elements gives 0 without changing result
1*2*3 --> 0 element gives 1 without changing result
max(-1,2,3) --> element -inifinity without changing result
```

▼ 26: Lists: Doubly linked lists, List class

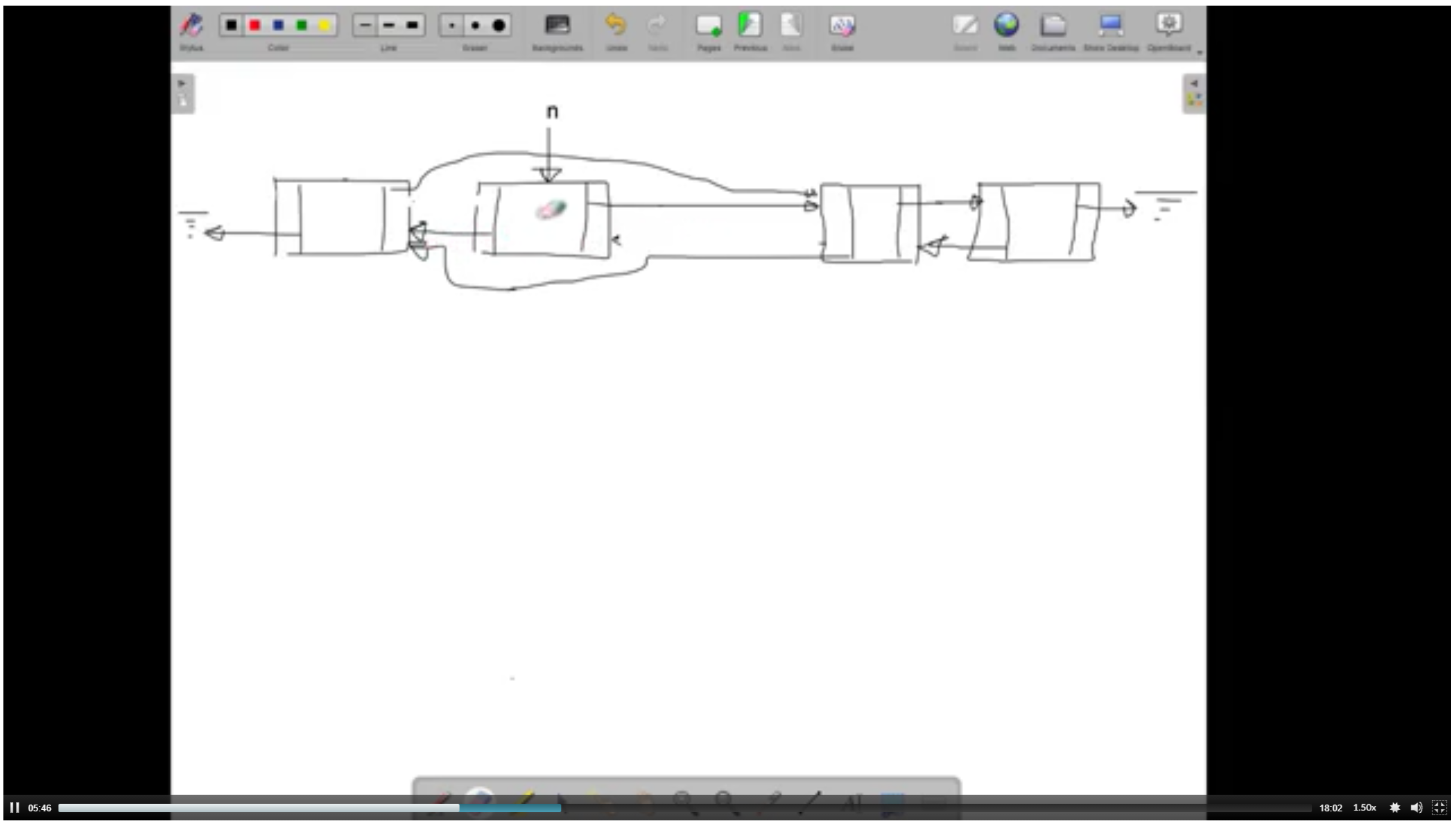
- Has 2 references: to next and prior



- Easier to find node that is needed
- Nodes from doubly linked list:

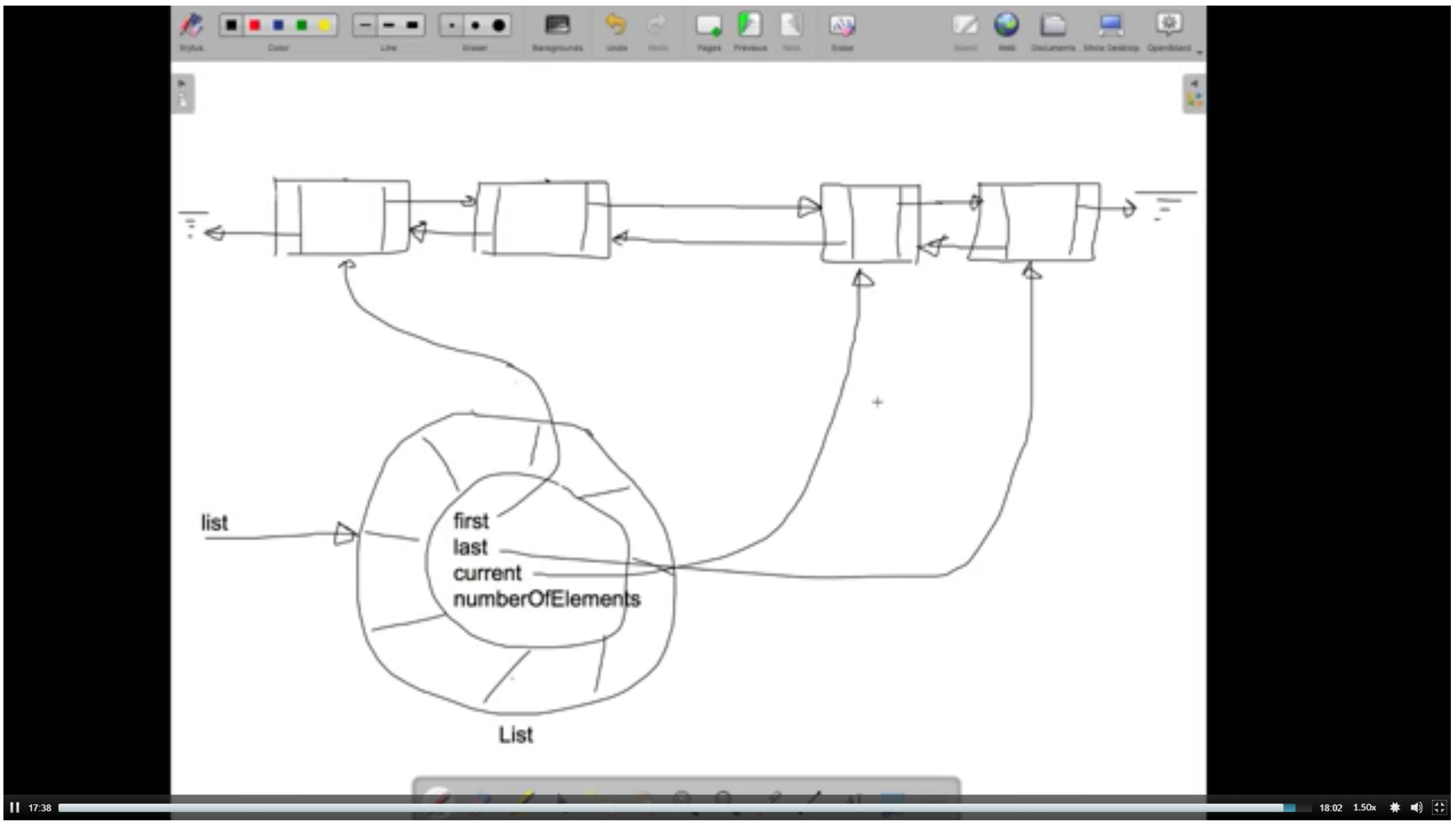
```
class Node {

int data;
Node prior;
      next; // next is a reference, not an object.

Node (int i, Node next) { // constructor 1
  data = i;
  this.prior = prior;
  this.next = next;
}

Node (int i) { // constructor 2
  this(i, null, null);
}

Node () {
  this(0, null, null); // constructor 3
}
```

1) Remove node where n points to




```
// 1) Remove the node where n points to --> We need to change the nodes that are incoming to the node we want
n.prior.next = n.next;
n.next.prior = n.prior;

// 2) Add a node value 7 after node n
n.next = n.next.prior = New node (7, n, n.next);

// n.next.prior = n.next = New node (7, n, n.next); DOES NOT WORK because we want the original value of n
```

- List object is part of list class ≠ node class



▼ 27: Inner classes

- Define a non static class inside another class → see skeleton of ListClass
- Why?
  1. Because this inner class will only be used in 1 class
  2. Because it is a logical structure
  3. Because inner classes can use private members from class they are defined in
- Because Node is an inner class inside class List, it can use the type parameter E → easier code than if Node would be an external class