G51PRG:
Introduction to Programming
Second semester
Lecture 14

Natasha Alechina
School of Computer Science & IT
nza@cs.nott.ac.uk

Previous lecture

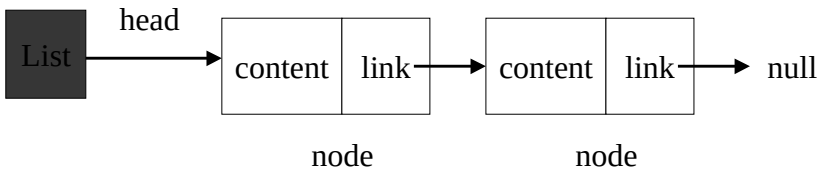
- Dynamic arrays and lists
- Implementing a dynamic array
- Implementing a very simple linked list in Java
- Inner classes

This lecture

- Iterators
- Synchronised data structures (back to ChatServer)
- Serialisation (saving objects)

Linked list

- A linked list consists of nodes .
- Each node has a contents fields which stores data (an Object) and a link field which says what the next item in the list is. A list has a head field which refers to the head of the list.



Linked list implementation

- Need a class Node to represent nodes in a list.
- Need a class List to store the head of the list and **`add(Object o), remove(Object o), get(int i)`** methods.

Node class (inner class of List)

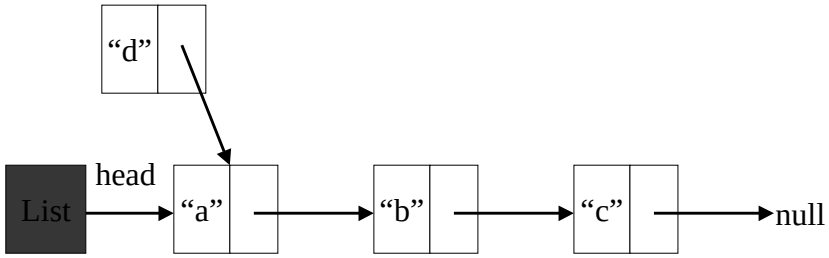
```
class Node {
  Object contents;
  Node link;

  public Node(Object o, Node next) {
   this.contents = o;
   this.link = next;
  }
}
```

List class

```
public class List{
  Node head;
  class Node {
  // Node class definition
  }
  // the rest of the List class...
  public void add(Object o){
   Node newHead = new Node (o, head);
   head = newHead;
  }
```

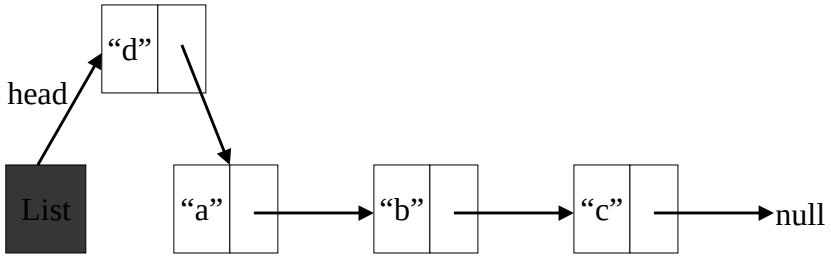

Adding a node with “d” in it:

- Create a new node and make it link to the head of the list



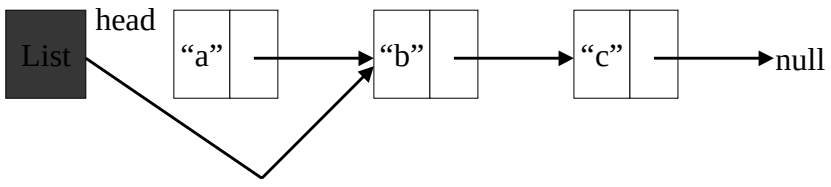
Adding a node continued

- Make it the new head:



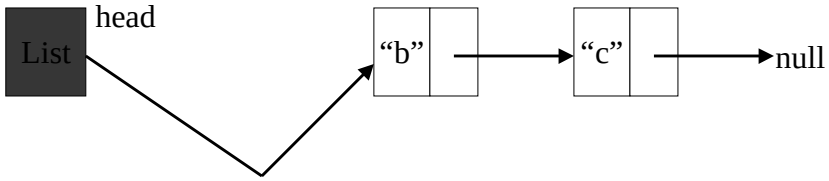
Removing a head

- To remove the head "a", make its successor the new head:



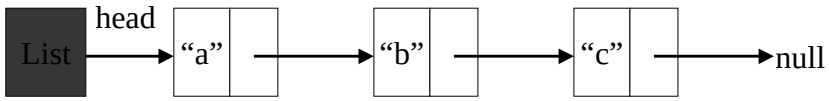
Removing a head continued

- This results in the following list:



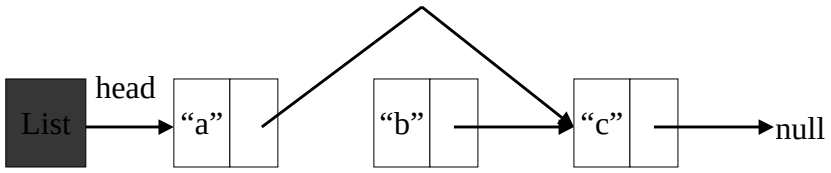
Removing a node in the middle

- To remove a node with “b”, make the previous node link to the “b”’s successor:



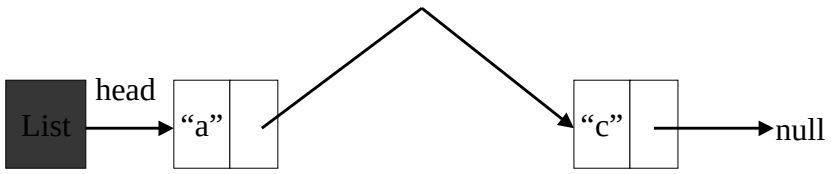
Removing a node in the middle

- To remove a node with “b”, make the previous node link to the “b”’s successor:



Removing a node in the middle

- This results in the following list:



Iterators

- Iterators are objects which know how to walk or iterate along the list.
- They have

`boolean hasNext()`

and

`Object next()` methods.

- We will implement **`get(int i)`** using an Iterator: get it to produce **`i`** items in order, and then return the **`i`**th item.

ListIterator class

- We only need a ListIterator because we have a List class.
- ListIterator may need access to (private) fields of a list (to **`head`** field in any case).
- This suggests that it is a good idea to make ListIterator an inner class of the List class.

Design of ListIterator class

```
class ListIterator {
// fields? Next node to return

boolean hasNext() {
// are there any more items in the list?
}

Object next() {
// return the next item we have not seen
}
```


Which fields

```
class ListIterator {
 Node currentnode; //node we are looking
  // at; what next() method will return

 public ListIterator() {
   currentnode = head; // List's field
 }
// note how we have access to the
// enveloping List object's field head
```

next()

```
public Object next() {
 if (currentnode == null) return null;
// actually better to throw an exception!
 Object x = currentnode.contents;
 currentnode = currentnode.link;
 return x;
}
```


hasNext()

```
public boolean hasNext() {
  return (currentnode != null);
}
```

In the List class ...

- Now we can add a method to the List class:

```
public ListIterator iterator() {
  return new ListIterator();
}
```


get(int i)

- Also in the List class...

```
public Object get(int i) {
 ListIterator li = iterator();
 if (i < 0) throw new
  ArrayIndexOutOfBoundsException();
// (better a new Exception class)
 for (int j = 0; j <= i-1; j++) {
   li.next(); // skip items at 0…i-1
 }
 return li.next();
}
```

Collections and concurrent access

- Quite often, several processes need to access the same list or array at the same time.
- This may potentially cause problems; we'll study them on an example of the ChatServer program from Java Gently.

Synchronised methods

- If a class A has a synchronised method **`amethod()`**, then any thread which needs to invoke this method on an instance x of A:

`x.amethod()`

has to obtain a ***lock*** on x. If another thread has the lock, the thread waits.

- When the thread has a lock, no other thread can invoke a synchronised method on x. (Can invoke unsynchronised methods).
- Similarly for synchronised static methods, only the lock is at the class level.

ChatServer

```
public class ChatServer {
 private static LinkedList clientList;
 static synchronized void
  broadcast(String s, String name) { … }
 static synchronized void remove(Socket
  client) { … }
 public static void main(String[] args) {
// listen for connections
// for each new client Socket,
  clientList.add(client)
// create a ChatHandler thread for the
  client
```


ChatHandler

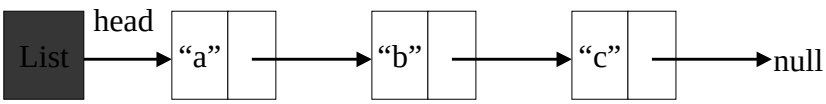
```
class ChatHandler {
 private Socket toClient;
// get the client's name
// loop reading messages from the
// client's input stream
ChatServer.broadcast(message, name);
// if the client types "BYE"
ChatServer.remove(toClient);
toClient.close();
```

Why use synchronized?

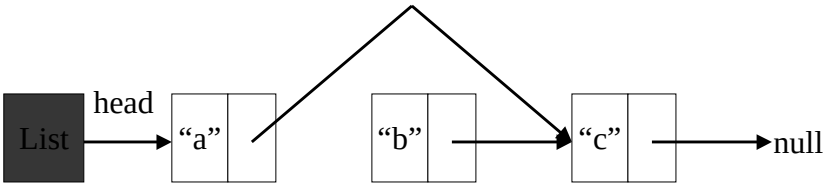
- It is possible that several clients simultaneously ask to be removed from the list or to broadcast a message.
- What may happen if several threads have a go at the list in an interleaved fashion?
- And is it a good idea to not lock the list while adding a client?

Race hazard with remove()

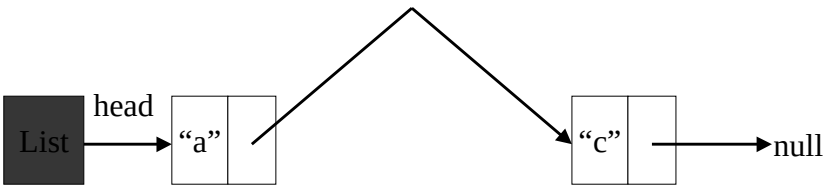
- Suppose that **`remove()`** is not synchronised, and two threads can try to remove clients from a list at the same time.
- Suppose the list is a LinkedList implemented as we did it in the last lecture.
- One thread is trying to remove “a” from the list, another thread is trying to remove “b” from the list.



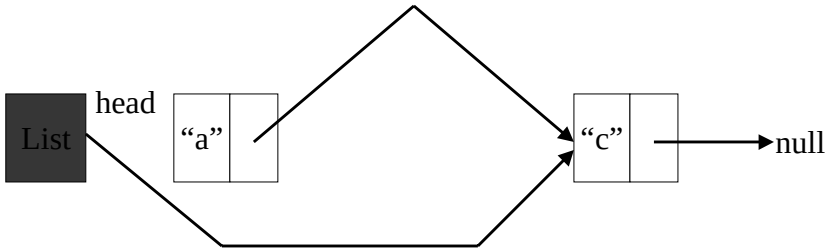
Ideally, to remove b



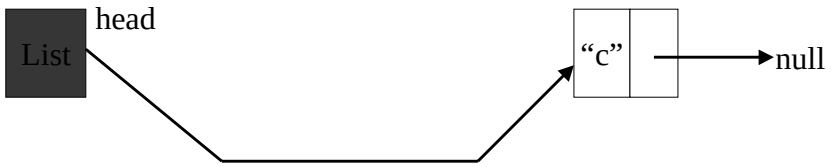
Resulting in:



And to remove a:



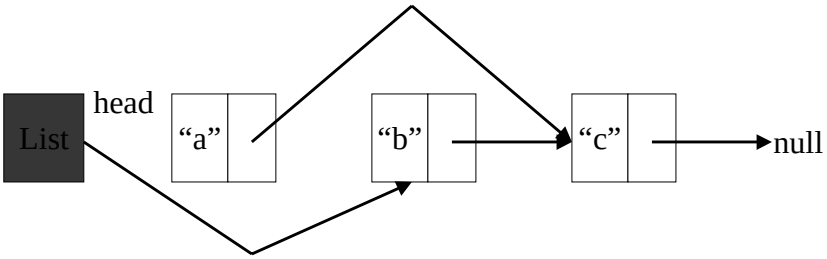
Resulting in:



With two threads racing:

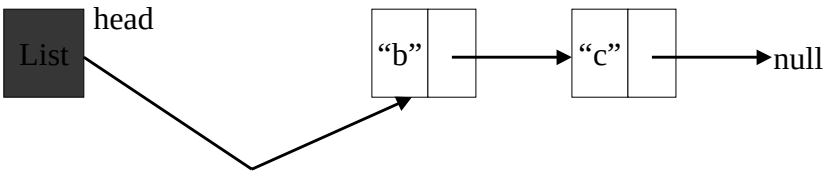
threadA: **`head = head.link`**

threadB: **`a.link = a.link.link`**



Resulting in:

threadB: remove of “b” is lost



Interaction of broadcast() and remove()

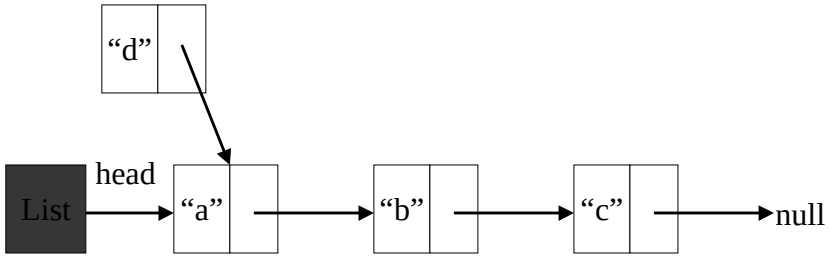
- If **`broadcast()`** and **`remove()`** are unsynchronised, the following bad thing can happen:
- thread1 is doing broadcast(), that is going through the list, opening a stream to each socket and sending a string;
- thread2 which is a handler for some client X called remove(), so client X's socket is bypassed in the list;
- thread1 already grabbed the reference to Client X's socket;
- thread2 closes the socket of X;
- thread1 tries to open a stream to a closed socket, exception is thrown and the program falls over.

Is unsyncronised add() safe?

- Now check if the Java Gently program is actually bullet-proof: can anything bad happen because a client may be added when a remove is in progress?
- Recall that ChatServer does not lock the list when it calls **`clientList.add(client)`**.

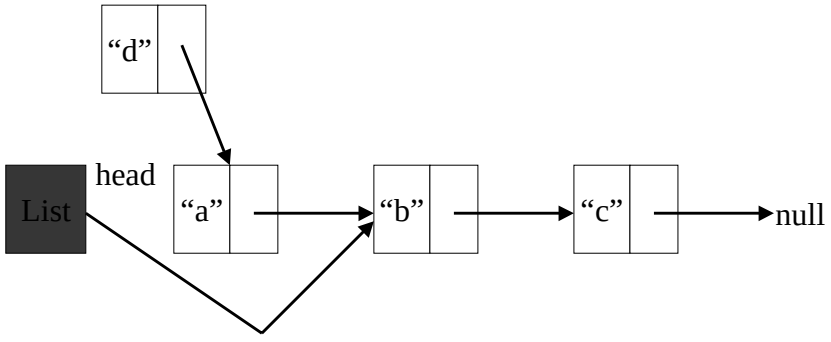
Adding “d” and removing “a”

- ThreadD links “d” to the head of the list:



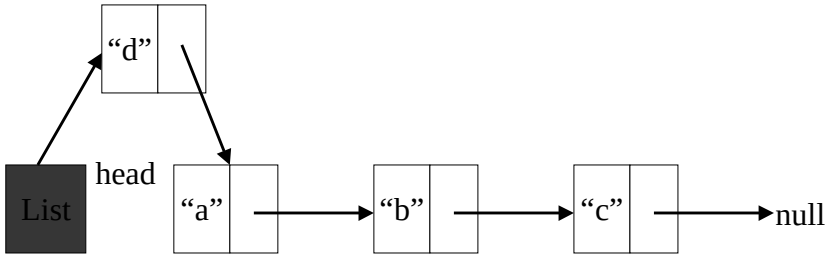
Adding “d” and removing “a”

- ThreadA reassigns the head to be “b”:



Adding “d” and removing “a”

- ThreadD completes insertion by making “d” the new head:



Result: lost update

- Java Gently implementation is not totally safe: ChatServer may be adding clients when some thread is trying to do remove().
- Since ChatServer does not bother to lock the list, the thread which is doing remove gets the lock and starts removing the client.
- This update may be lost because of ChatServer’s actions.
- Conclusion: add() method should be synchronised.
- The easiest is to use a synchronised collection for the clientList.

Synchronised collections

- Synchronised collections only allow their methods to be called by a single thread at a time.
- They are thread-safe: no undesirable race hazards happen.
- The only two examples in java.util:
  - **`Vector`**
  - **`Hashtable`**

Non synchronised collections

- All other Collections, including java.util.LinkedList.
- To get a synchronised collection, use ***synchronization wrappers*** from Collections class:

```
public static Collection
  synchronizedCollection(Collection c);
public static Set synchronizedSet(Set s);
public static List synchronizedList(List
  list);
etc.
```

Example

- Suppose **`oldlist`** is of type java.util.LinkedList:

```
List synclist =
  Collections.synchronizedList(oldlist);
```

- Now we have a synchronized list **`synclist`**.

Iterating over synchronised collection

- You still need to lock the list when you are iterating over it:

```
synchronized(synclist) {
   Iterator i = synclist.iterator();
   while (i.hasNext())
       System.out.println(i.next());
    }
```

- This is because iterator makes multiple calls into the collection (it is unlocked between the calls to next()).

Serialisation

- We know how to write characters and Strings into a file. What if we need to save an object (e.g. a list).
- Reading and writing objects (saving them to a file, or passing them to a different machine in distributed computing) requires representing them in a serialized form (as a sequence of bytes). Then they can be passed along an

`ObjectInputStream and ObjectOutputStream.`

- In order to be serializable, an object should implement Serializable interface (an empty marker interface).

How to serialize your object

- State that it implements io.Serializable interface.
- It is a marker interface; no methods go with it.

```
public class List implements Serializable
```

- If it has Object type fields, they should be instances of serializable classes (e.g. Node).

```
class Node implements Serializable
```

How to serialize your object 2

```
ObjectOutputStream out =
     new ObjectOutputStream (
       new FileOutputStream("listtest"));
 out.writeObject(mylist);
 out.close();
 ObjectInputStream in =
     new ObjectInputStream (
        new FileInputStream("listtest"));
 copy = (List) in.readObject();
```


Summary

- For iterators, see Sun tutorial

http://java.sun.com/docs/books/tutorial/collections/interfaces/collection.html

- For serialization, see Sun tutorial

http://java.sun.com/docs/books/tutorial/essential/io/serialization.html

- For synchronised collections, see Sun tutorial

http://java.sun.com/docs/books/tutorial/collections/implementations/wrapper.html and http://java.sun.com/docs/books/tutorial/essential/threads/multithreaded.html