EECS 2030
Advanced Object-Oriented Programming

S2023, Section A

Recursion (notes: Chapter 8)

Printing n of Something

suppose you want to implement a method that prints out n copies of a string

```
public static void printIt(String s, int n)
{
  for(int i = 0; i < n; i++) {
    System.out.print(s);
  }
}
```

A Different Solution

alternatively we can use the following algorithm:

if $n \leq 0$ **done**, otherwise/**else**

I. print the string once

II. print the string $(n - 1)$ more times

```
public static void printItToo(String s, int n) {
  if (n <= 0) {
    return;
  }
  else {
    System.out.print(s);
    printItToo(s, n - 1);    // method invokes itself
  }
}
```

Recursion

a method that calls itself is called a *recursive* method

a recursive method solves a problem by repeatedly reducing the problem so that a base case can be reached

```
printItToo("*", 5)
*printItToo ("*", 4)
**printItToo ("*", 3)
***printItToo ("*", 2)
****printItToo ("*", 1)
*****printItToo ("*", 0) base case
*****
```

Notice that the number of times the string is printed decreases after each recursive call to printItToo

Notice that the base case is eventually reached.

Base cases

a base case occurs when you know the answer to the problem that the method is trying to solve

```
public static void printItToo(String s, int n) {
  if (n == 2) {
    System.out.print(s);
    System.out.print(s);
  }
  else {
    System.out.print(s);
    printItToo(s, n - 1);
  }
}
```

Base cases

the base cases must be exhaustive

- they must cover every possible condition for which the method can return a solution

```
public static void printItToo(String s, int n) {
  if (n <= 0) {
    return;
  }
  else {
    System.out.print(s);
    printItToo(s, n - 1);
  }
}
```

Infinite Recursion

if the base case(s) is missing, or never reached, a recursive method will run forever (or until the computer runs out of resources)

```
public static void printItForever(String s, int n) {
  // missing base case; infinite recursion
  System.out.print(s);
  printItForever(s, n - 1);
}
```

```
printItForever("*", 1)
* printItForever("*", 0)
** printItForever("*", -1)
*** printItForever("*", -2) ...........
```

leads to Stack Overflow Exception

Similar situation: constructors

(From the course forum last year)

I was trying to do constructor chaining. And I found the code below caused StackOverFlowError:

```
public Vector3(double x, double y, double z) {
        this(new Vector3(x, y, z)); //constructor calls itself
                                    // which causes the stack overflow.
}
```

What happens during recursion

What Happens During Recursion

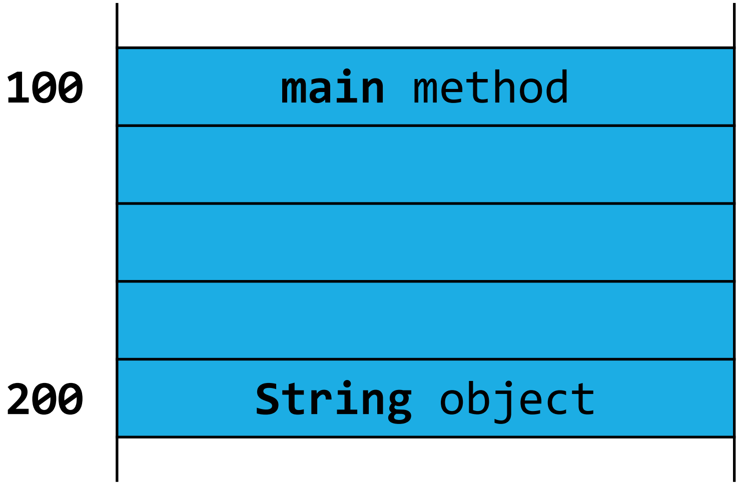
a simplified model of what happens during a recursive method invocation is the following:

- whenever a method is *invoked* that method runs in a *new* block of memory
  - when a method recursively invokes itself, a new block of memory is allocated for the newly invoked method to run in

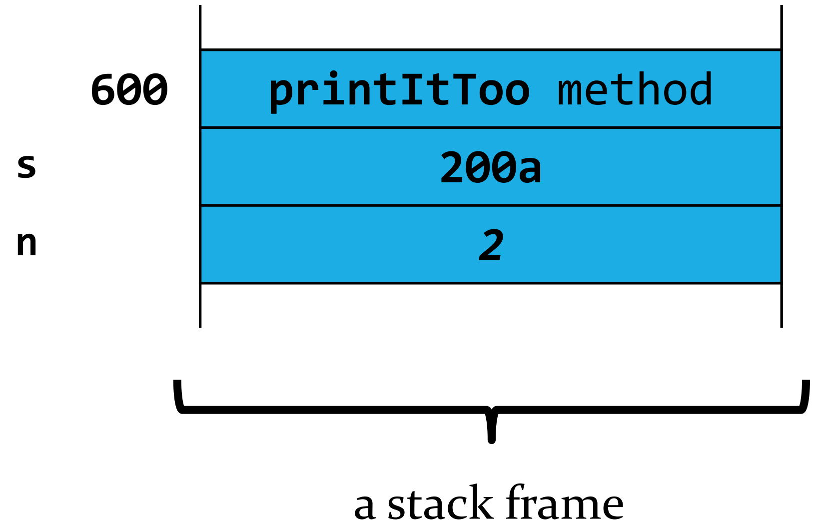
consider the **`printItToo`** method

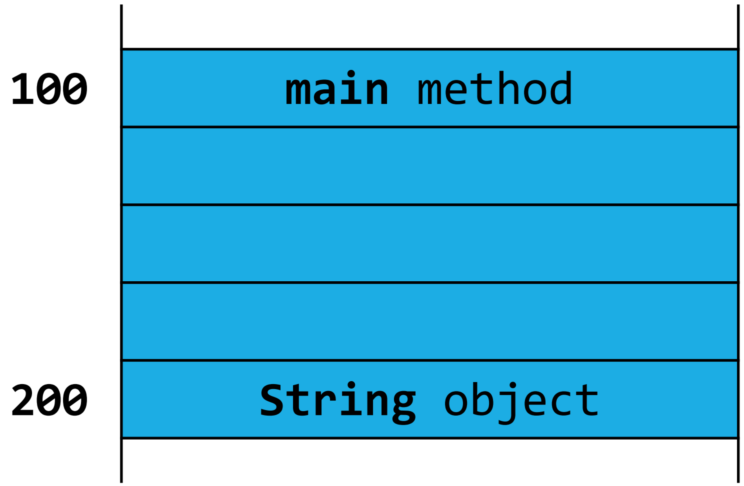
```java
public static void printItToo(String s, int n) {
  if (n <= 0) {
    return;
  }
  else {
    System.out.print(s);
    printItToo(s, n - 1);
  }
}
```

```
printItToo("*", 2);
```



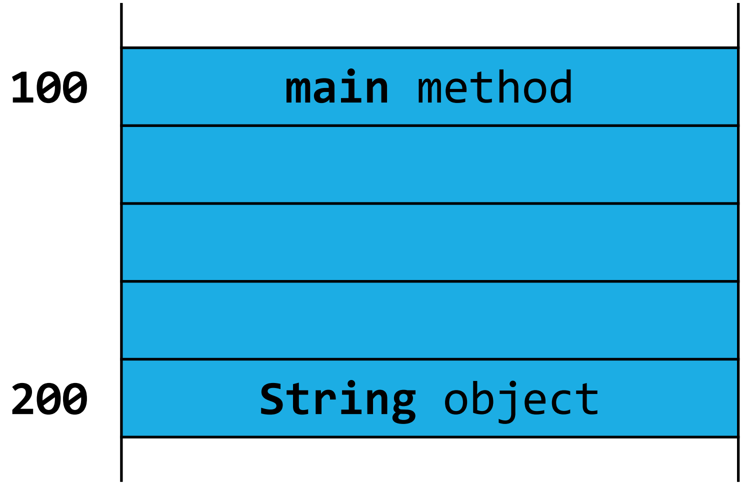
```
printItToo("*", 2);
```

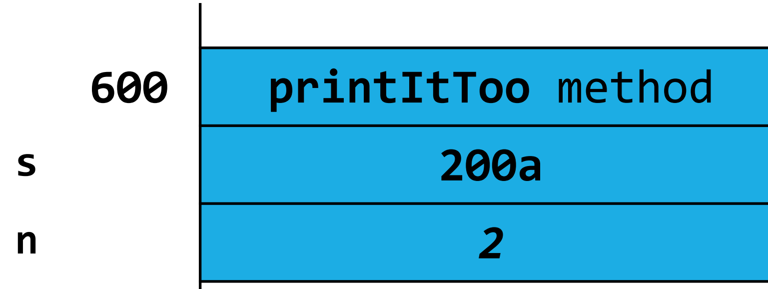




- methods occupy space in a region of memory called the *call stack*
- information regarding the state of the method is stored in a *stack frame*
- the stack frame includes information such as the method parameters, local variables of the method, where the return value of the method should be copied to, where control should flow to after the method completes, ...
- stack memory can be allocated and deallocated very quickly, but this speed is obtained by restricting the total amount of stack memory
- if you try to solve a large problem using recursion you can exceed the available amount of stack memory which causes your program to crash

```
printItToo("*", 2);
```



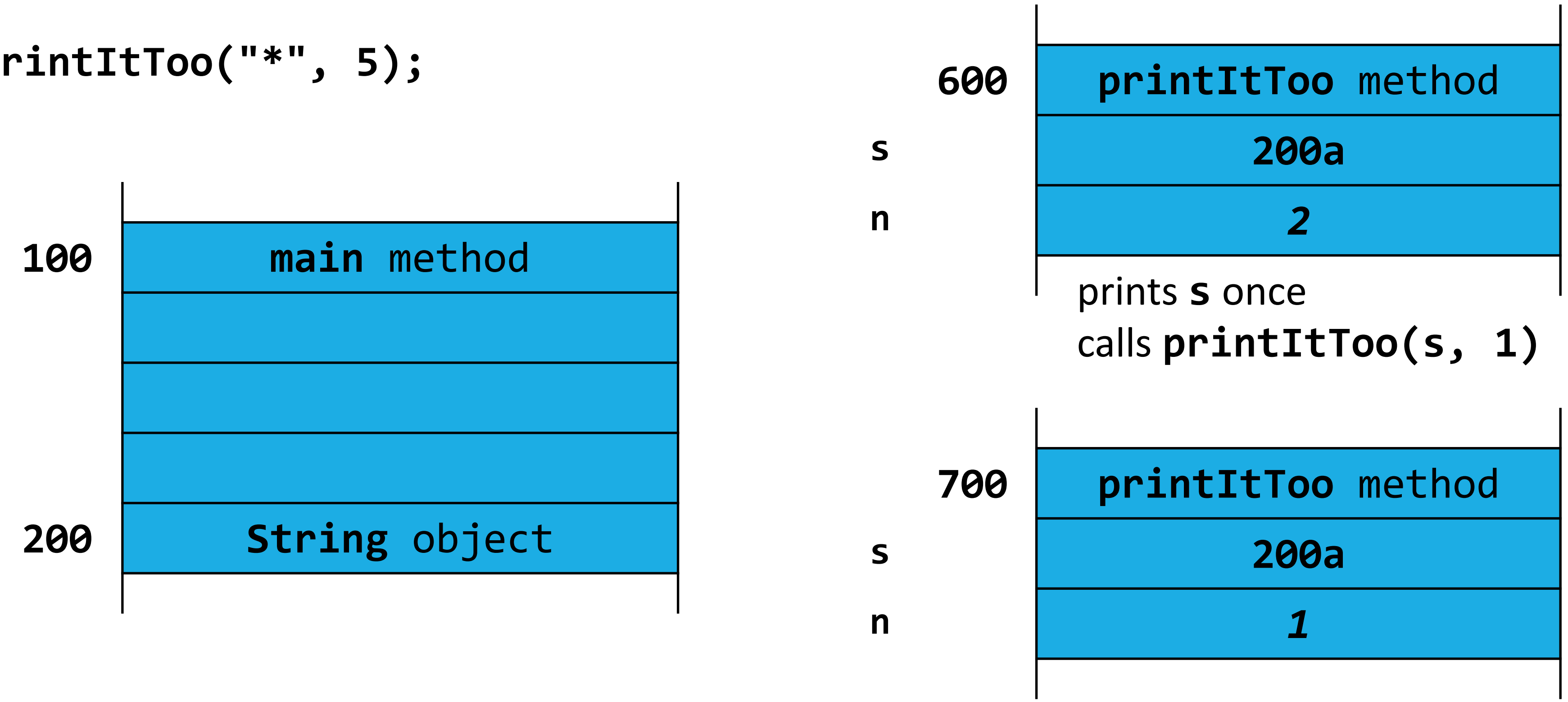


prints **s** once

calls **printItToo(s, 1)**

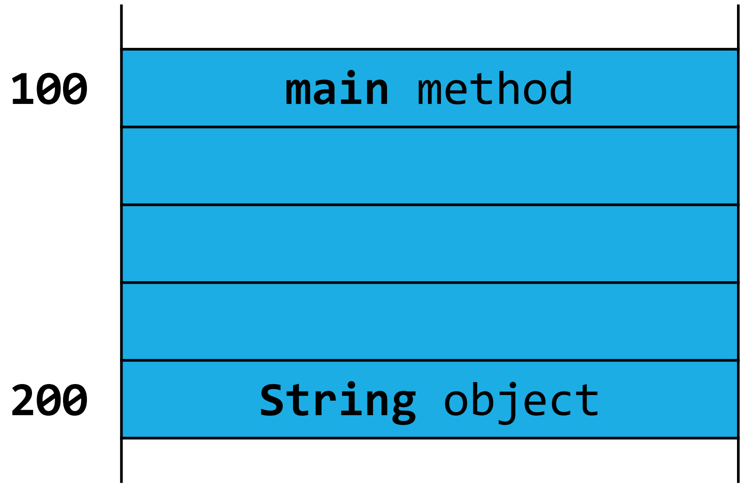
(this stack is “upside down”)

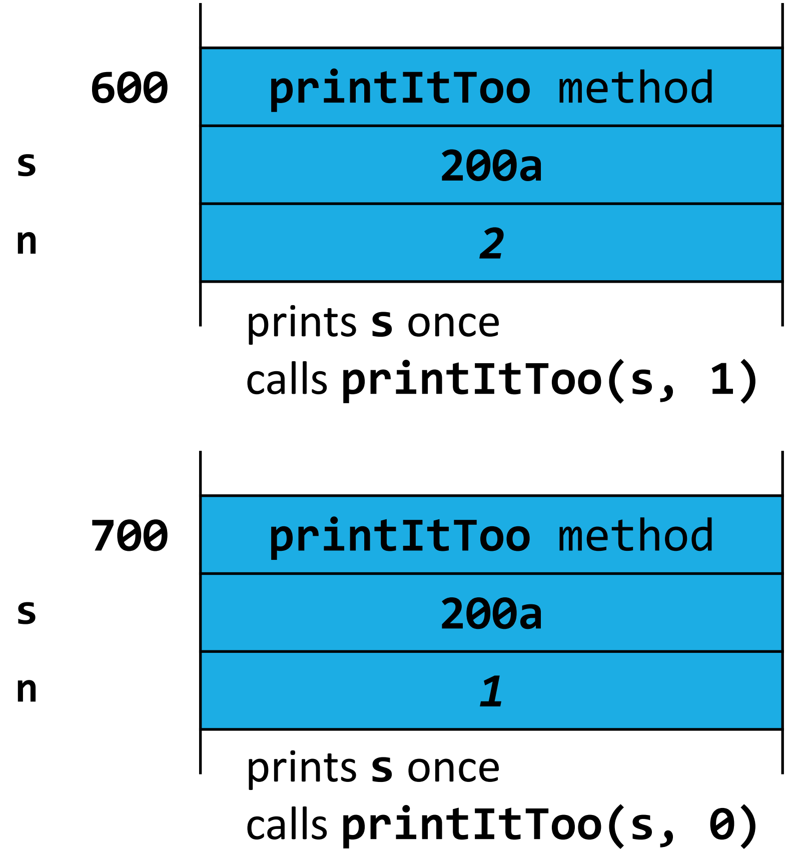
`printItToo("*", 5);`



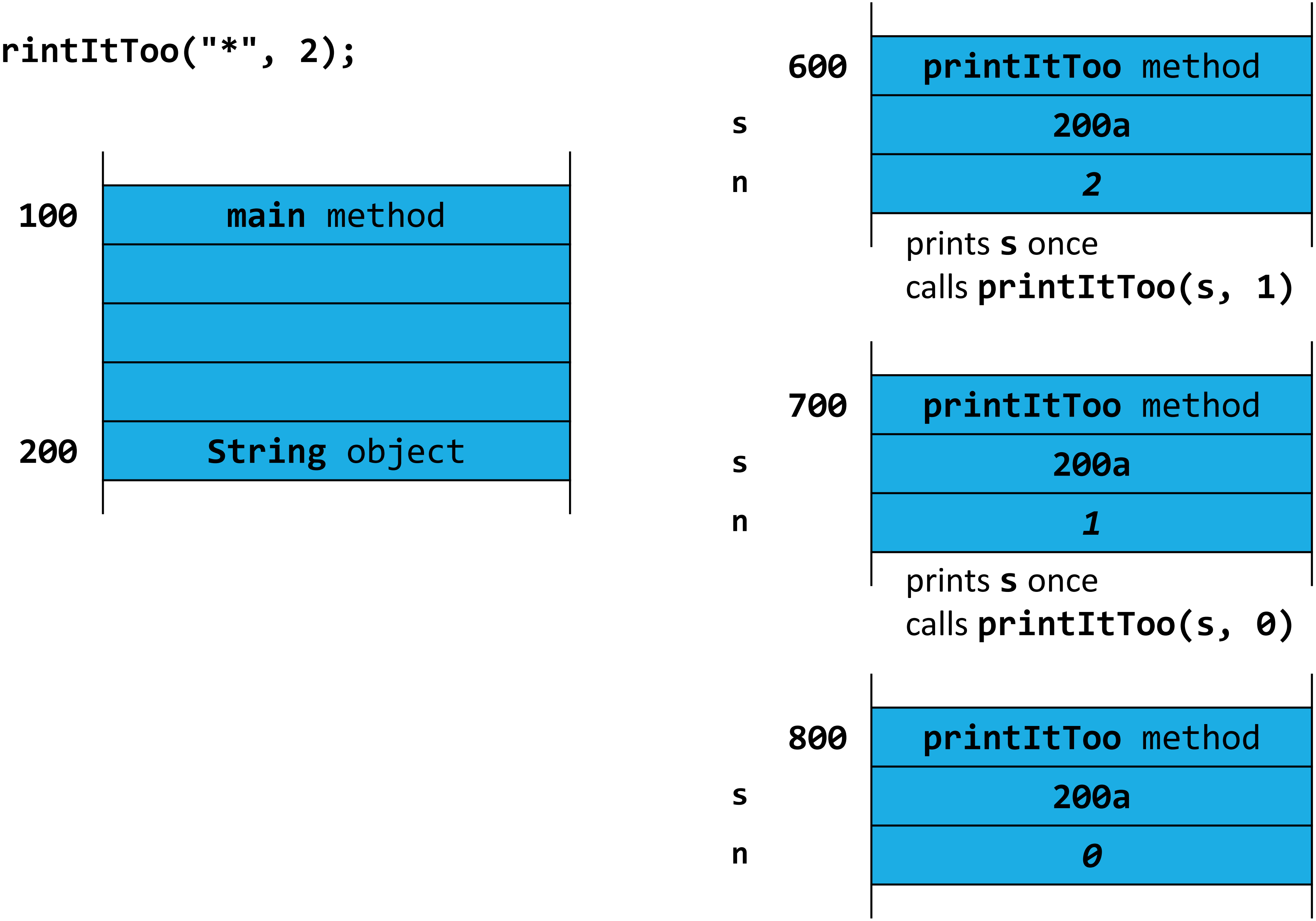
(this stack is “upside down”)

```
printItToo("*", 2);
```



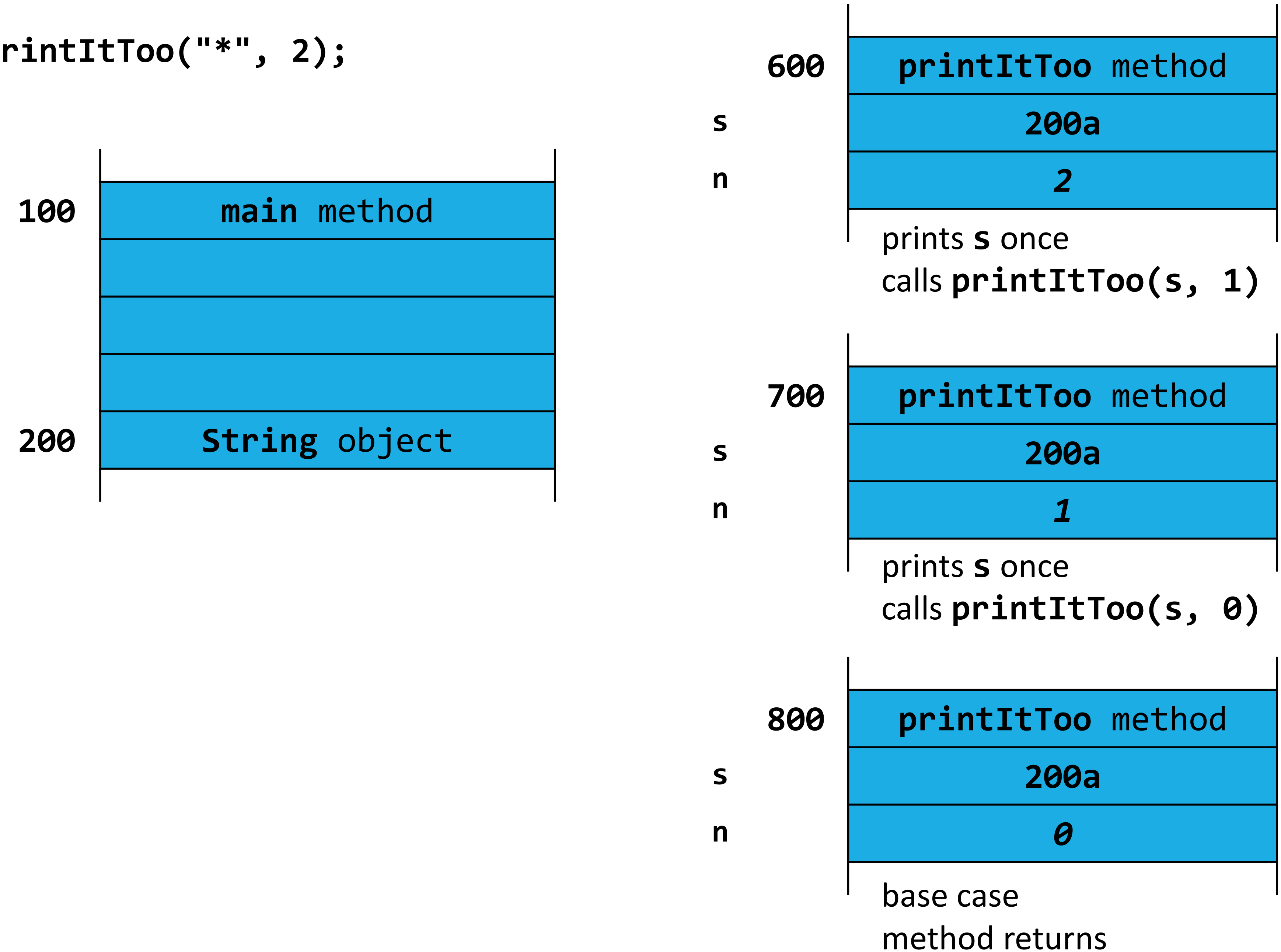


```
printItToo("*", 2);
```



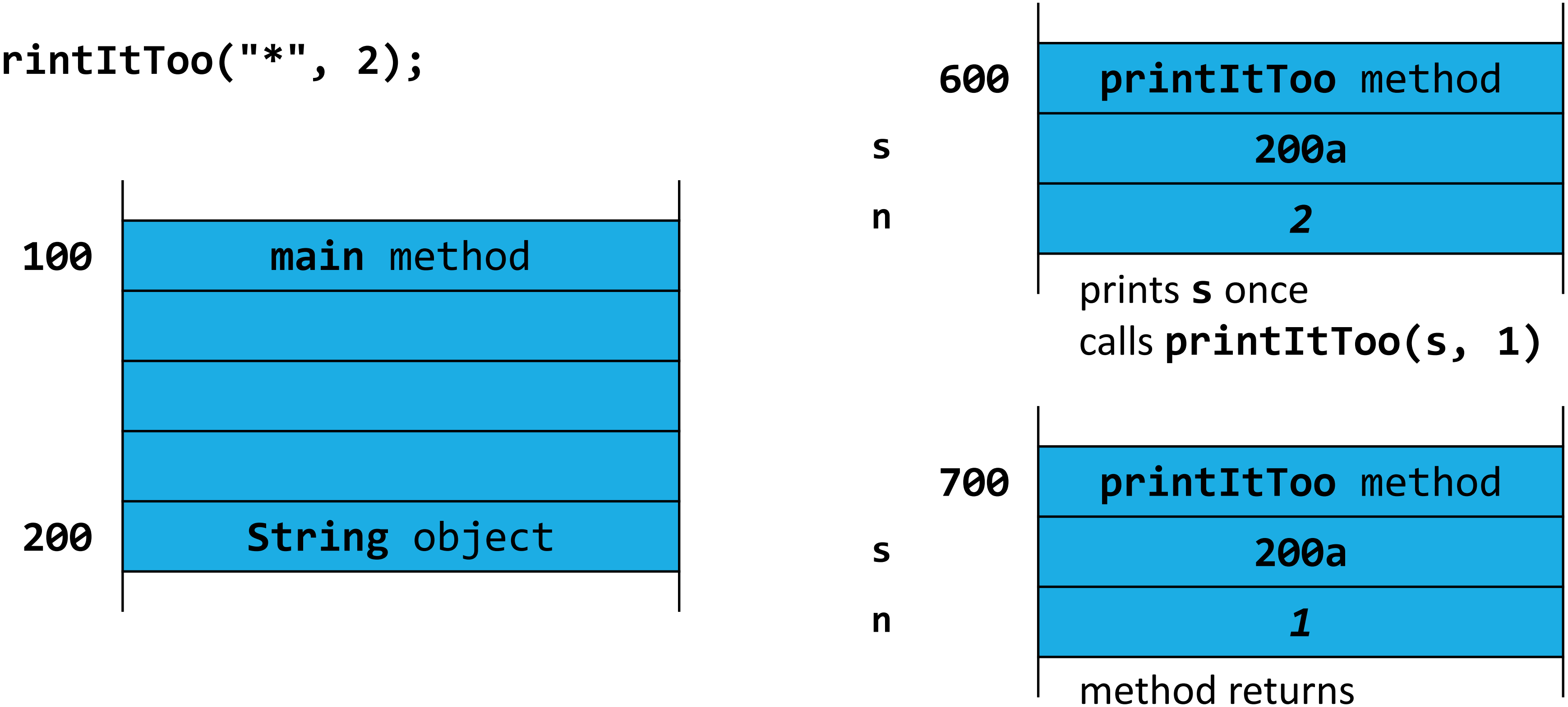
(this stack is “upside down”)

```
printItToo("*", 2);
```



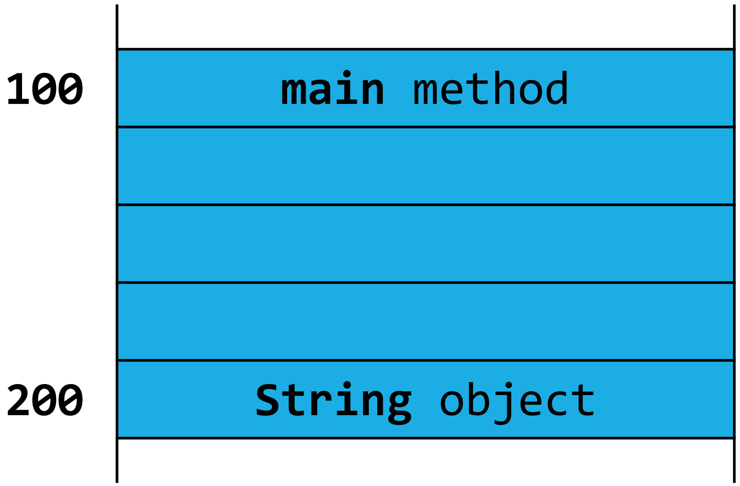
(this stack is “upside down”)

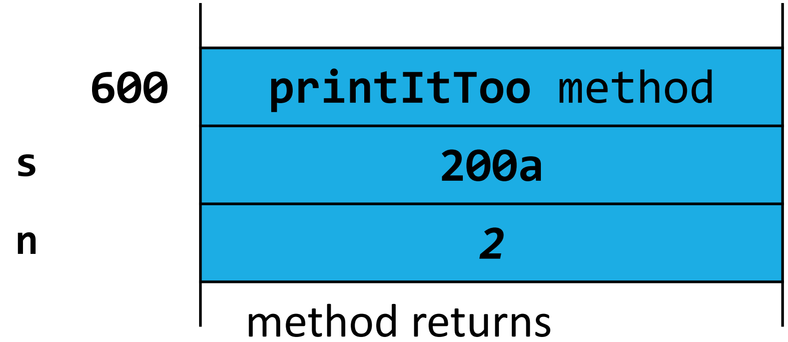
```
printItToo("*", 2);
```



(this stack is “upside down”)

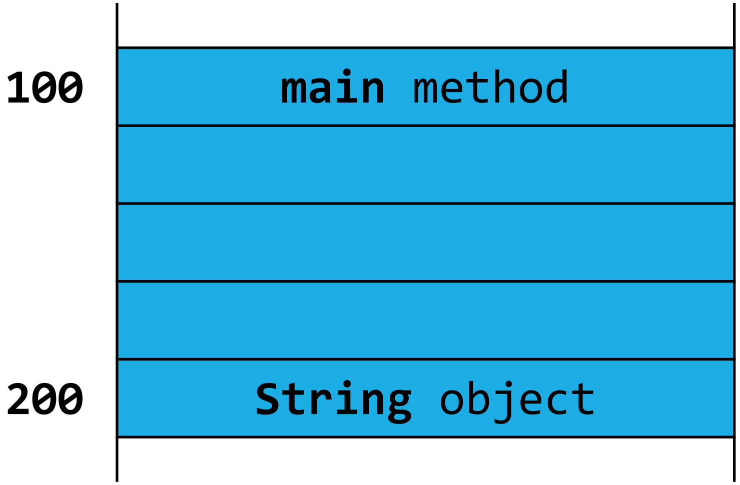
```
printItToo("*", 2);
```





(this stack is “upside down”)

```
printItToo("*", 2);
```



Recursion and collections

Recursion and Collections

Consider the problem of searching for an element in a list

Searching a list for a particular element can be performed by recursively examining the first element of the list

- if the first element is the element we are searching for then we can return true
- otherwise, we recursively search the sub-list starting at the next element

The **`List`** method **`subList`**

`List` has a very useful method named **`subList`**:

`List<E> subList(int fromIndex, int toIndex)`

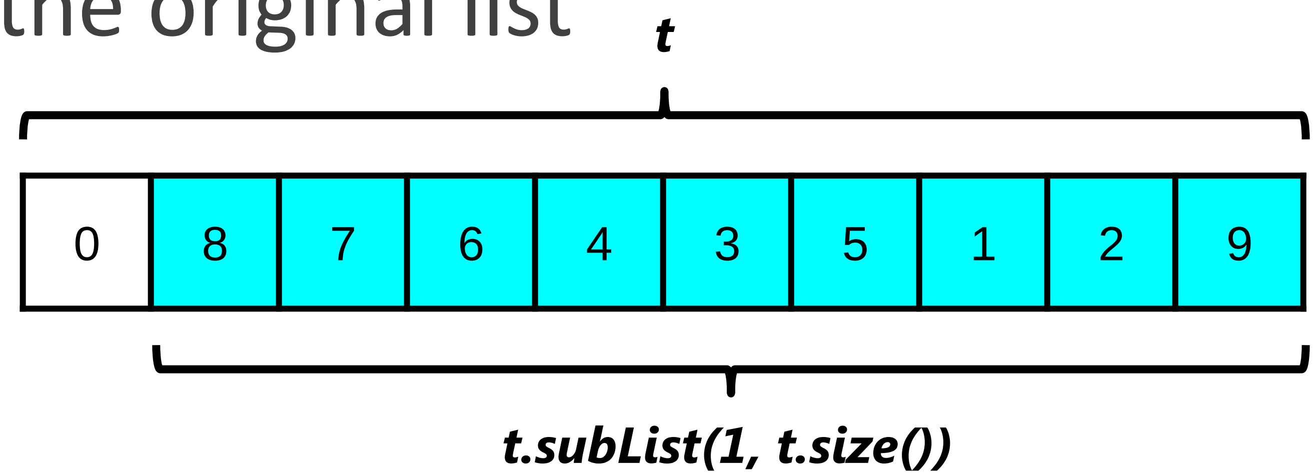
returns a List

Returns a view of the portion of this list between the specified **`fromIndex`**, inclusive, and **`toIndex`**, exclusive. (If **`fromIndex`** and **`toIndex`** are equal, the returned list is empty.) The returned list is backed by this list, so non-structural changes in the returned list are reflected in this list, and vice-versa. The returned list supports all of the optional list operations supported by this list.

http://docs.oracle.com/javase/7/docs/api/java/util/List.html#subList%28int,%20int%29

`subList` examples

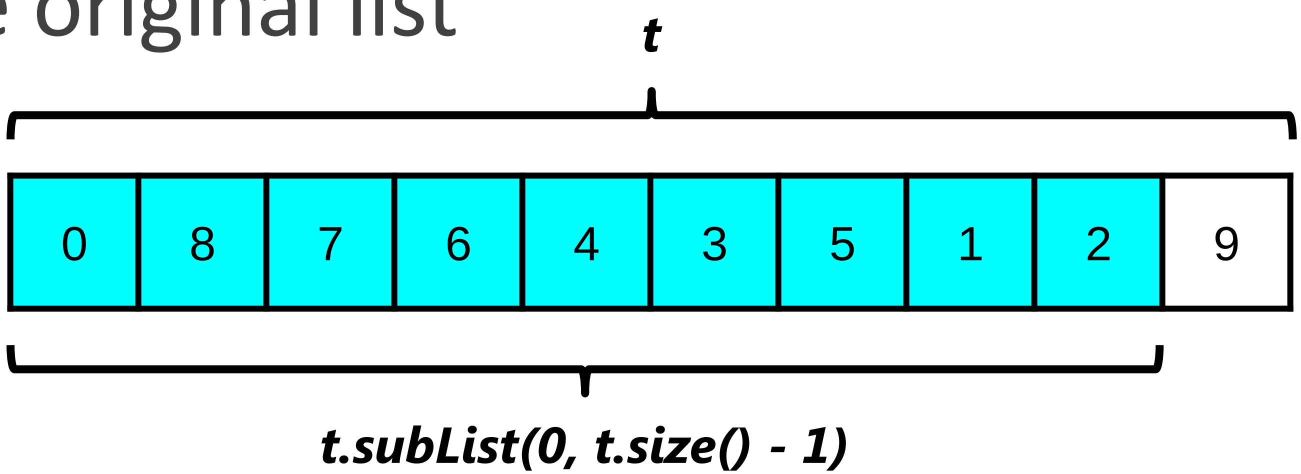
the sub-list excluding the first element of the original list




```
List<Integer> u = t.subList(1, t.size());
int first_u = u.get(0);                    // 8
int last_u = u.get(u.size() – 1);          // 9
```

`subList` examples

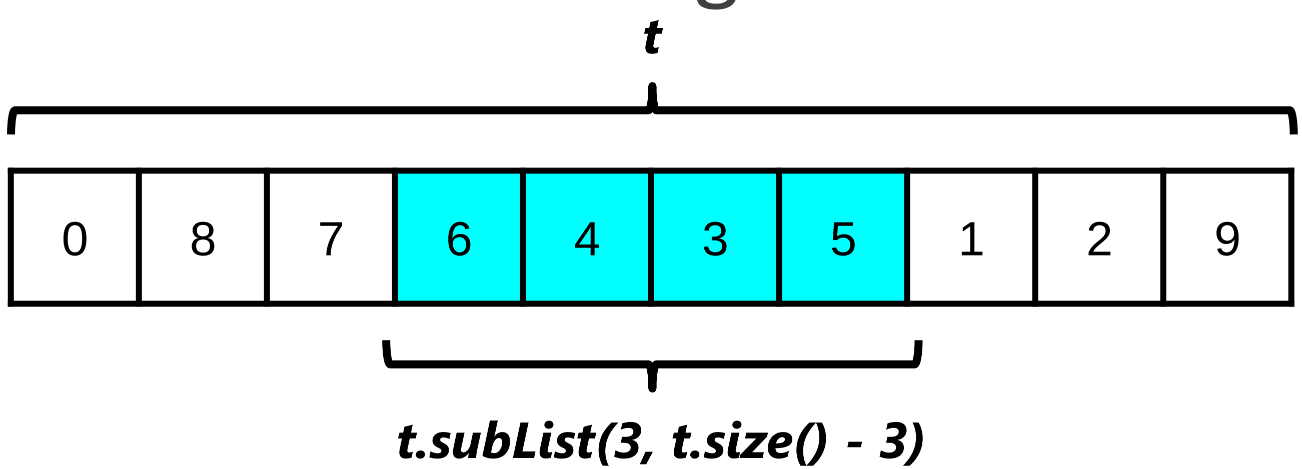
the sub-list excluding the last element of the original list




```
List<Integer> u = t.subList(0, t.size() - 1);
int first_u = u.get(0);                       // 0
int last_u = u.get(u.size() – 1);             // 2
```

`subList` examples

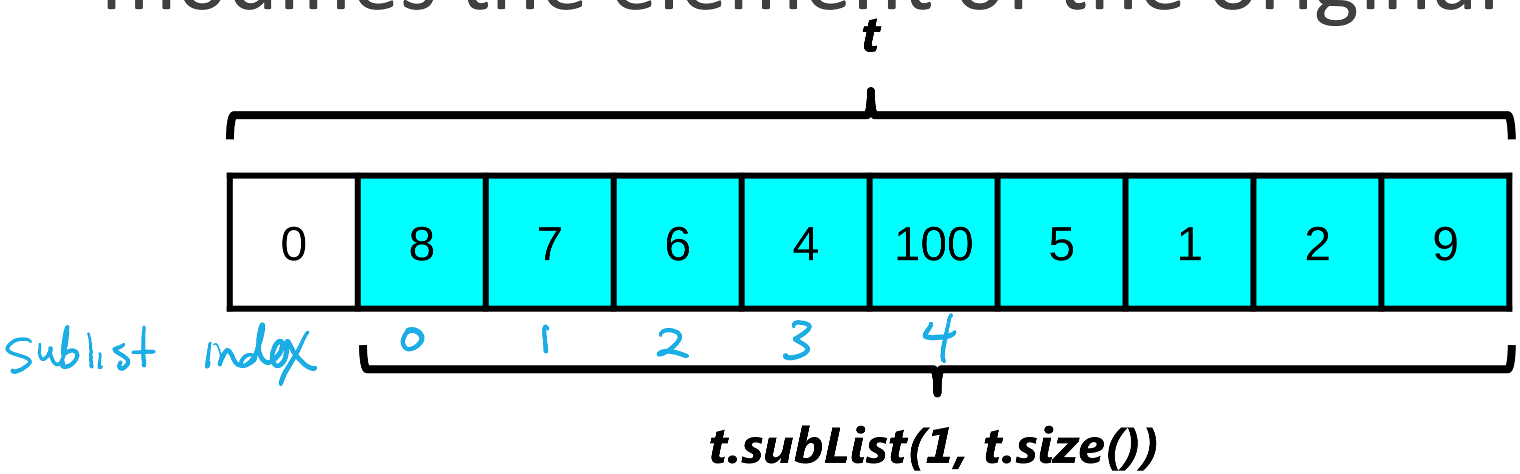
the sub-list excluding the first 3 and last 3 elements of the original list



List<Integer> u = t.subList(3, t.size() - 3);
int first_u = u.get(0); // 6
int last_u = u.get(u.size() – 1); // 5

`subList` examples

modifying an element using the sublist modifies the element of the original list




```
List<Integer> u = t.subList(1, t.size());
u.set(4, 100);                    // set element at index 4 of u
int val_in_t = t.get(5);          // 100
```

modifying the sublist modifies the original list

Recursively Search a List

algorithm for:

```
// returns true if elem is in t, false otherwise
boolean contains(String elem, List<String> t)
```

1. if **`t`** is empty
   return false
2. if the first element of **`t`** equals **`elem`**
   return true
3. else
   search the sublist starting at the second element for **`elem`**

Recursively Search a List

```
  contains("X", ["Z", "Q", "B", "X", "J"])

→ "X".equals("Z") == false
→ contains("X", ["Q", "B", "X", "J"])  recursive call

→ "X".equals("Q") == false
→ contains("X", ["B", "X", "J"])       recursive call

→ "X".equals("B") == false
→ contains("X", ["X", "J"])            recursive call

→ "X".equals("X") == true              done!
```

Recursively Search a List

base case(s)?

- recall that a base case occurs when the solution to the problem is known

```java
public class Recursion {

  public static boolean contains(String elem, List<String> t) {
    boolean result;
    if (t.size() == 0) {                    // base case
      result = false;
    }
    else if (t.get(0).equals(elem)) {       // base case
      result = true;
    }


  }
}
```

Recursively Search a List

- recursive call?
  - to help deduce the recursive call assume that the method does exactly what its API says it does
    - e.g., **`contains(elem, t)`** returns true if **`elem`** is in the list **`t`** and false otherwise
  - use the assumption to write the recursive call or calls

```java
public class Recursion {

  public static boolean contains(String element, List<String> t) {
    boolean result;
    if (t.size() == 0) {                         // base case
      result = false;
    }
    else if (t.get(0).equals(element)) {   // base case
      result = true;
    }
    else {                      // recursive call
      result = Recursion.contains(element, t.subList(1, t.size()));
    }
    return result;
  }
}
```

List size reduces by 1 every time!
No infinite recursion

```java
public class Recursion {

  public static boolean contains(String element, List<String> t) {
    boolean result;
    String first = t.get(0);                //any problem here?
    if (t.size() == 0) {
      result = false;
    }
    else if (first.equals(element)) {
      result = true;
    }
    else {
      result = Recursion.contains(element, t.subList(1, t.size()));
    }
    return result;
  }
}
```

Recursion and Collections

consider the problem of finding the smallest element in a list of integers

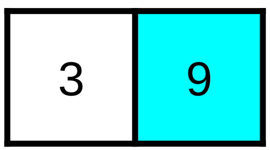
Minimum element in a list

what is the smallest element in the following list?

3

Minimum element in a list

what is the smallest element in the following list?



the smallest of the first element and the second element

Minimum element in a list

what is the smallest element in the following list?

3	?	?	?	?	?	?	?	?	?

the smallest of the first element and the *smallest element in the rest of the list*

Minimum element in a list

```java
/**
 * Returns the minimum element in a list.
 *
 * @param t a list of Integer
 * @return the minimum element in t
 */
public static int min(List<Integer> t)
```

Minimum element in a list

recursive algorithm for min

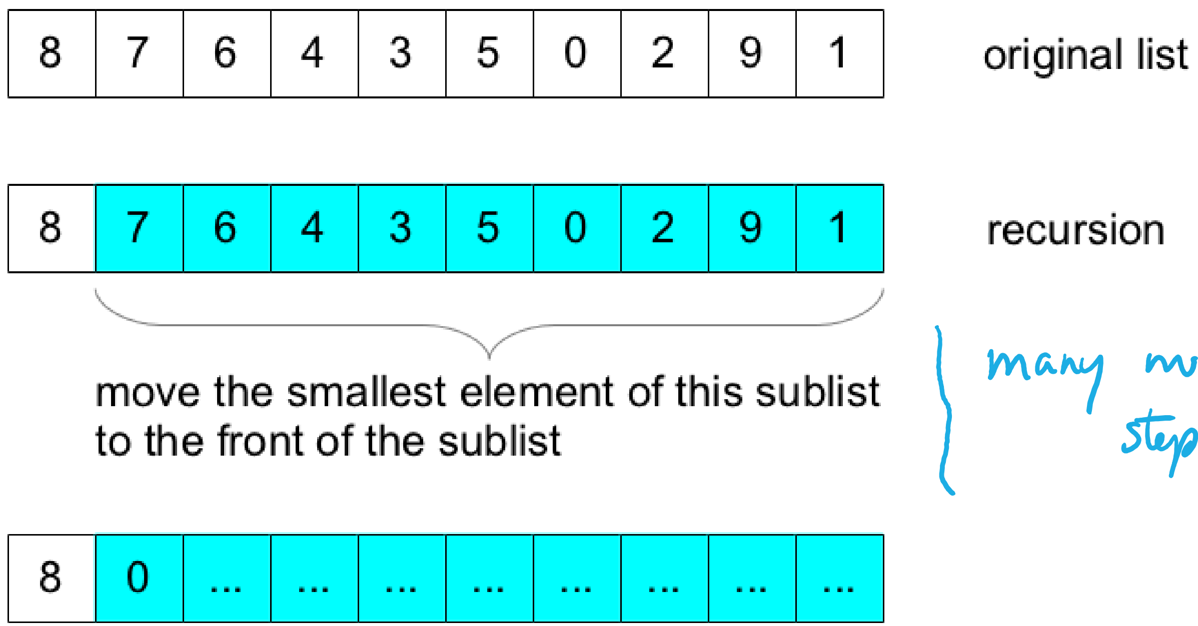
1. if **`t`** is empty
   ???
2. **`first`** = the first element of **`t`**
3. if **`t`** has one element
   return **`first`**
4. **`minInRest`** = minimum element in the sublist starting at the second element
5. return the minimum of **`first`** and **`minInRest`**

```java
public static int min(List<Integer> t) {
  if (t.isEmpty()) {
    throw new IllegalArgumentException();
  }
  Integer first = t.get(0);
  if (t.size() == 1) {
    return first;
  }
  Integer minInRest = min(t.subList(1, t.size()));
  if (first.compareTo(minInRest) < 0) { // "<=" ?
    return first;
  }
  return minInRest;
}
```

Recursion and Collections

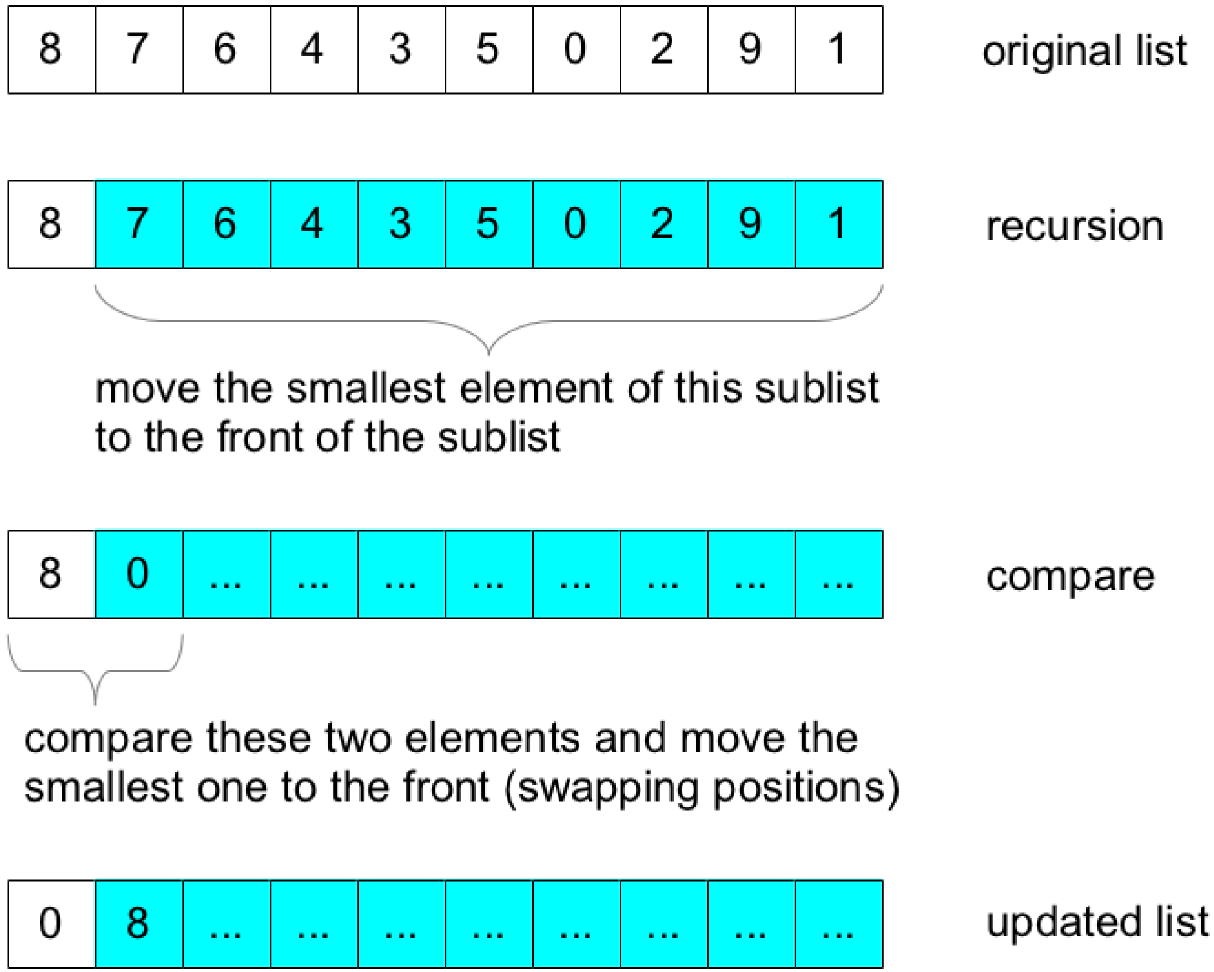
consider the problem of moving the smallest element in a list of integers to the front of the list

Recursively Move Smallest to Front



many missing steps

Recursively Move Smallest to Front



Recursively Move Smallest to Front

base case?

- recall that a base case occurs when the solution to the problem is known

Recursively Move Smallest to Front

```
public class Recursion {

  public static void minToFront(List<Integer> t) {
    if (t.size() == 1) {
      return;
    }

  }
}
```

Recursively Move Smallest to Front

```
public class Recursion {

  public static void minToFront(List<Integer> t) {
    if (t.size() < 2) { //what if empty???
      return;
    }

  }
}
```

Recursively Move Smallest to Front

- recursive call?
  - to help deduce the recursive call "**assume that the method does exactly what its API says it does**" (like before)
    - e.g., **`moveToFront(t)`** moves the smallest element in **`t`** to the front of **`t`**
    - use the **assumption** to write the recursive call or calls

Recursively Move Smallest to Front

```java
public class Recursion {

  public static void minToFront(List<Integer> t) {
    if (t.size() < 2) {
      return;
    }
    Recursion.minToFront(t.subList(1, t.size())); //just the first step!






  }
}
```

Recursively Move Smallest to Front

compare and update?

Recursively Move Smallest to Front

```
public class Recursion {

  public static void minToFront(List<Integer> t) {
    if (t.size() < 2) {
      return;
    }
    Recursion.minToFront(t.subList(1, t.size()));
    Integer first = t.get(0);
    Integer second = t.get(1);
    if (second.compareTo(first) < 0) {
      t.set(0, second);
      t.set(1, first);
    }
  }
}
```

don't do this :

```
if (t.get(1) < t.get(0)) {
    t.set(0, t.get(1));
    t.set(1, t.get(0));
}
```

Sorting the List

observe what happens if you repeat the process with the sublist made up of the second through last elements:

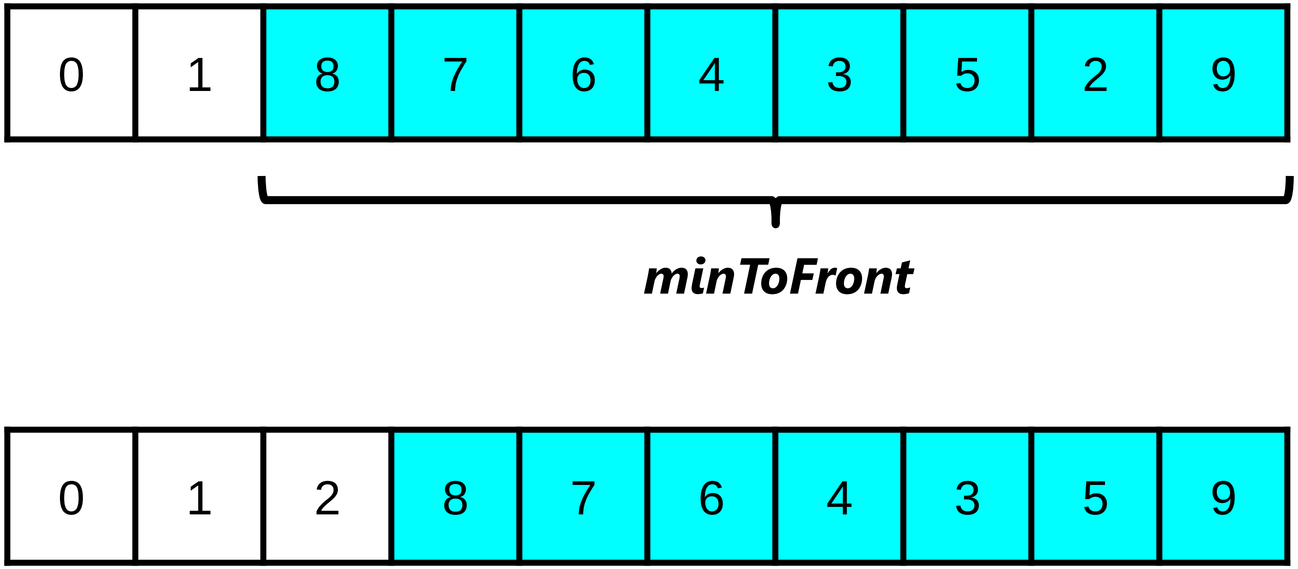
0	8	7	6	4	3	5	1	2	9

minToFront

0	1	8	7	6	4	3	5	2	9

Sorting the List

observe what happens if you repeat the process with the sublist made up of the third through last elements:



Sorting the List

observe what happens if you repeat the process with the sublist made up of the fourth through last elements:

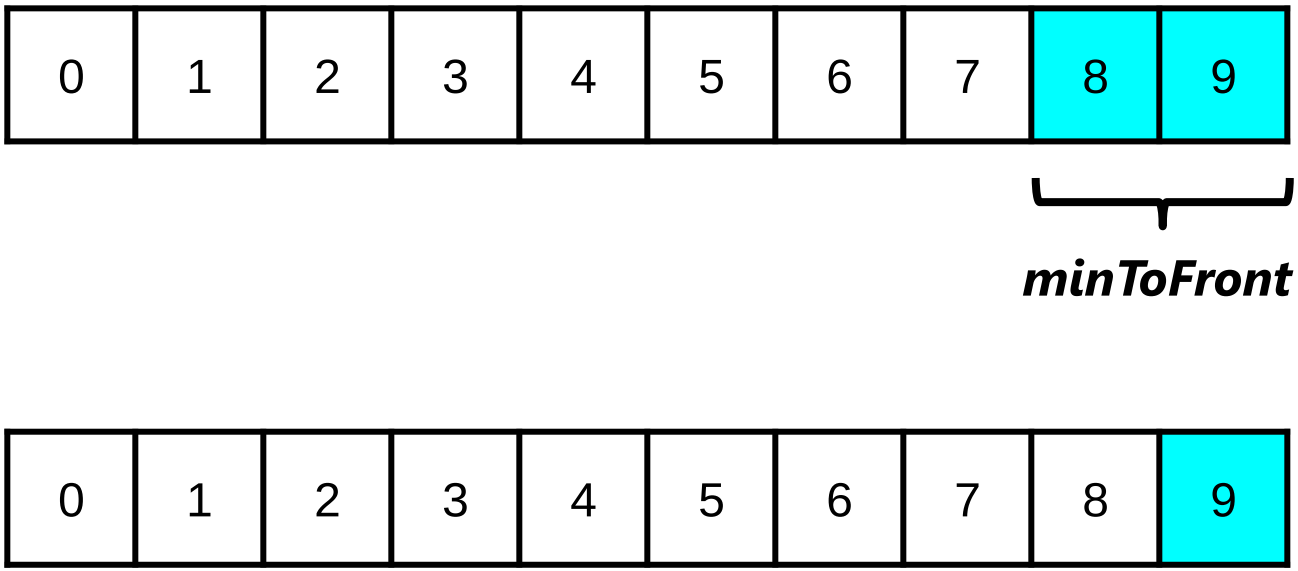
0	1	2	8	7	6	4	3	5	9

minToFront

0	1	2	3	8	7	6	4	5	9

Sorting the List

if you keep calling **`minToFront`** until you reach a sublist of size two, you will sort the original list:



this is the *selection sort* algorithm

Selection Sort

```
public class Recursion {

  // minToFront not shown

  public static void selectionSort(List<Integer> t) {
    if (t.size() > 1) {
      Recursion.minToFront(t);
      Recursion.selectionSort(t.subList(1, t.size()));
    }
  }


}
```

Stack Overflow?

Every method call requires JVM to keep track of the address of the code to return to (continue execution from)

This (and some other data) are stored in a stack

Stack size is implementation-dependent

- *Much* smaller than overall memory size
- Can be set: **“java –Xss256k YourProgram”**

Similar issues exist with other languages (C, C++…)

Thus, deep recursion is to be avoided