

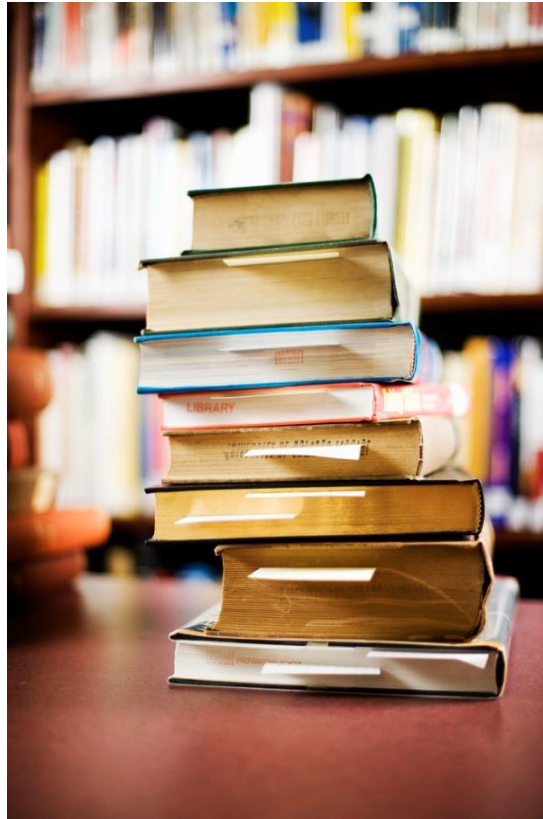
# More Data Structures (Part 1)

Stacks

# Stack

---

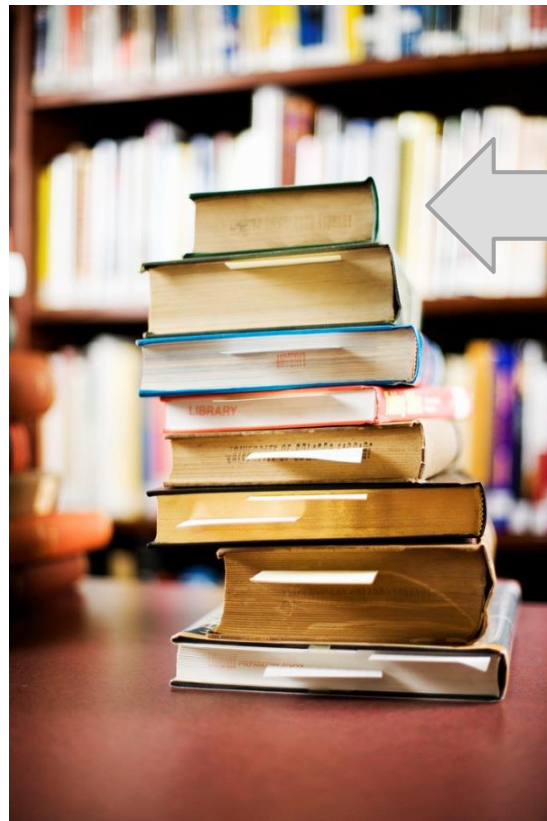
- ▶ examples of stacks



# Top of Stack

---

- ▶ top of the stack



# Stack Operations

---

- ▶ classically, stacks only support two operations
  1. push
    - ▶ add to the top of the stack
  2. pop
    - ▶ remove from the top of the stack
    - ▶ throws an exception if there is nothing on the stack

# Stack Optional Operations

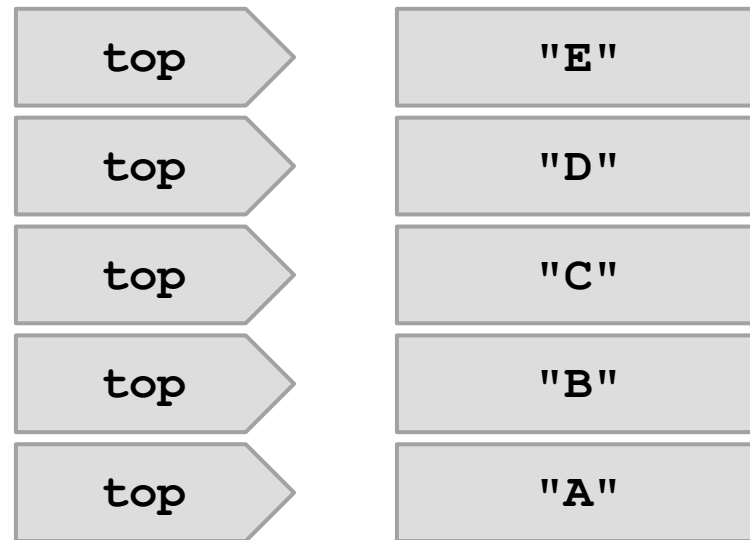
---

- ▶ optional operations
  1. size
    - ▶ number of elements in the stack
  2. isEmpty
    - ▶ is the stack empty?
  3. peek
    - ▶ get the top element (without removing it)
  4. search
    - ▶ find the position of the element in the stack
  5. isFull
    - ▶ is the stack full? (for stacks with finite capacity)
  6. capacity
    - ▶ total number of elements the stack can hold (for stacks with finite capacity)

# Push

---

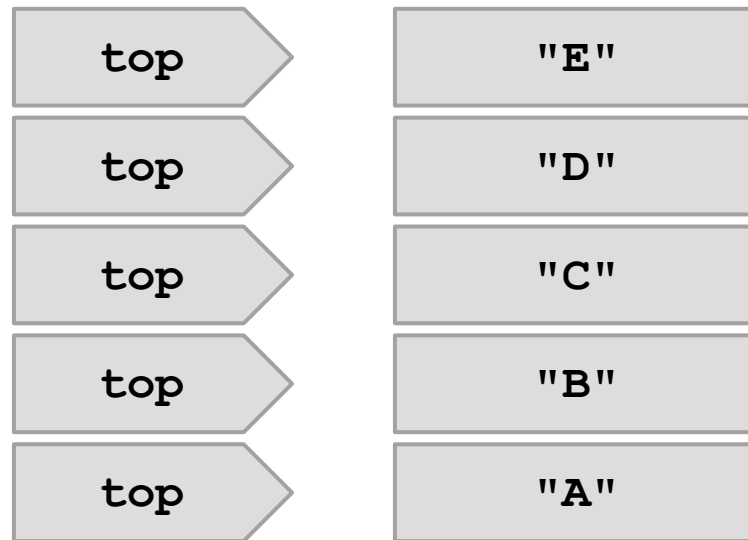
1. `st.push("A")`
2. `st.push("B")`
3. `st.push("C")`
4. `st.push("D")`
5. `st.push("E")`



# Pop

---

1. `String s = st.pop()`
2. `s = st.pop()`
3. `s = st.pop()`
4. `s = st.pop()`
5. `s = st.pop()`



# LIFO

---

- ▶ stack is a Last-In-First-Out (LIFO) data structure
  - ▶ the last element pushed onto the stack is the first element that can be accessed from the stack



# Applications

---

- ▶ stacks are used widely in computer science and computer engineering
  - ▶ undo/redo
  - ▶ widely used in parsing
  - ▶ a call stack is used to store information about the active methods in a Java program
  - ▶ convert a recursive method into a non-recursive one

# Example: Reversing a sequence

---

- ▶ a silly and usually inefficient way to reverse a sequence is to use a stack

# Don't do this

---

```
public static <E> List<E> reverse(List<E> t) {
    List<E> result = new ArrayList<E>();
    Stack<E> st = new Stack<E>();
    for (E e : t) {
        st.push(e);
    }
    while (!st.isEmpty()) {
        result.add(st.pop());
    }
    return result;
}
```

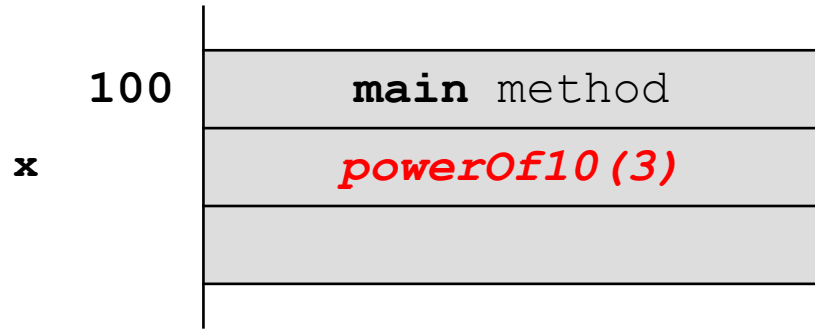
# Converting a Recursive Method

---

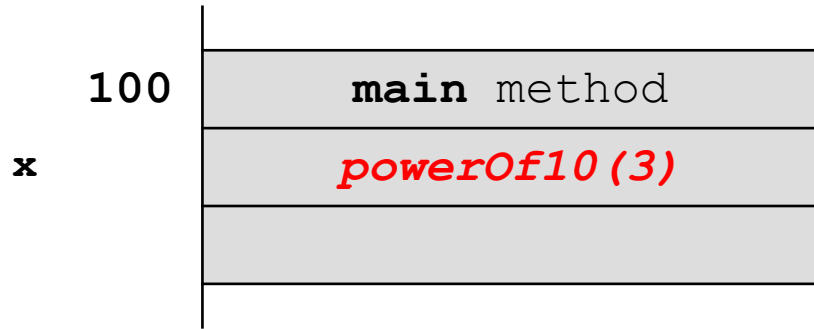
- ▶ a stack can be used to convert a recursive method to a non-recursive method
- ▶ the key to understanding how to do this lies in the memory diagram for a recursive memory
  - ▶ the following slides are from the Day 25 lecture

```
public static double powerOf10(int n) {  
    double result;  
    if (n < 0) {  
        result = 1.0 / powerOf10(-n);  
    }  
    else if (n == 0) {  
        result = 1.0;  
    }  
    else {  
        result = 10 * powerOf10(n - 1);  
    }  
    return result;  
}
```

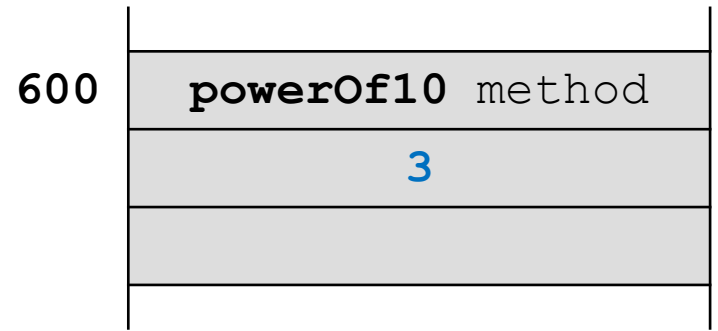
```
double x = Recursion.powerOf10(3);
```



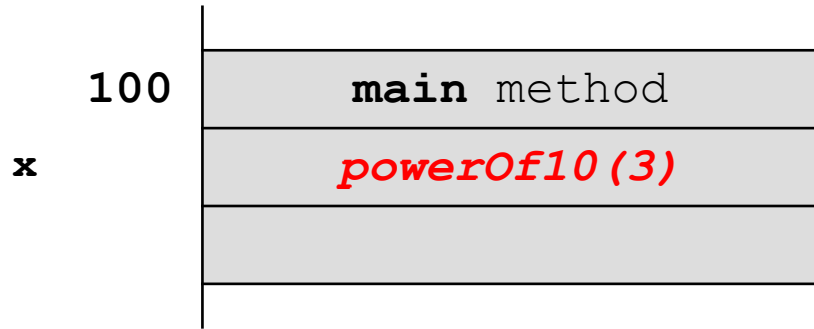
```
double x = Recursion.powerOf10(3);
```



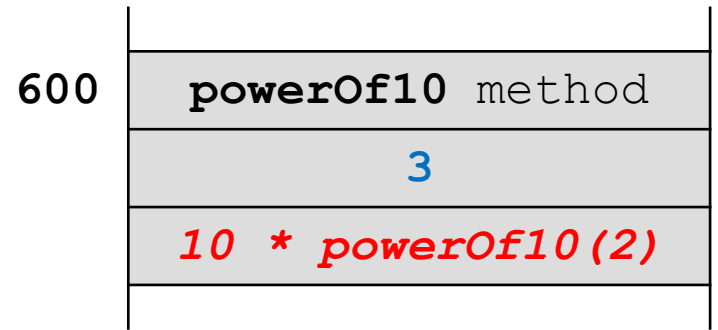
n  
result



```
double x = Recursion.powerOf10(3);
```

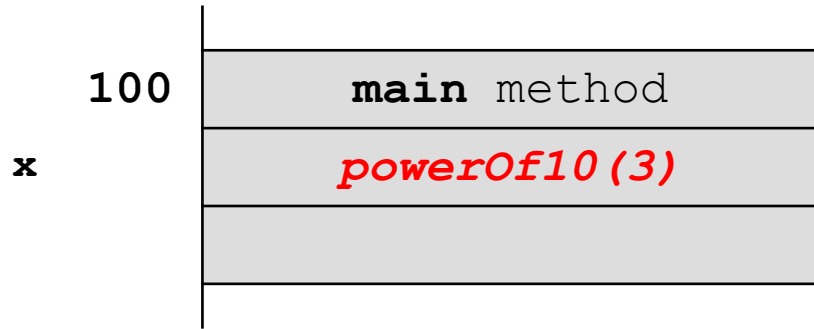


n  
result





```
double x = Recursion.powerOf10(3);
```



result

n

600

powerOf10 method

3

*10 \* powerOf10(2)*

750

powerOf10 method

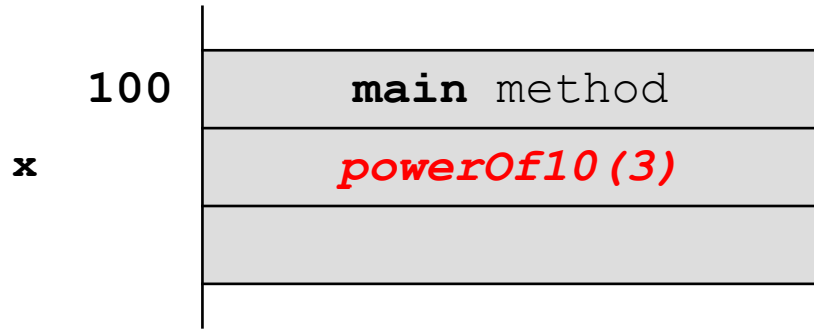
2

result

n



```
double x = Recursion.powerOf10(3);
```



result

n

600

powerOf10 method

3

$10 * \text{powerOf10}(2)$

750

powerOf10 method

2

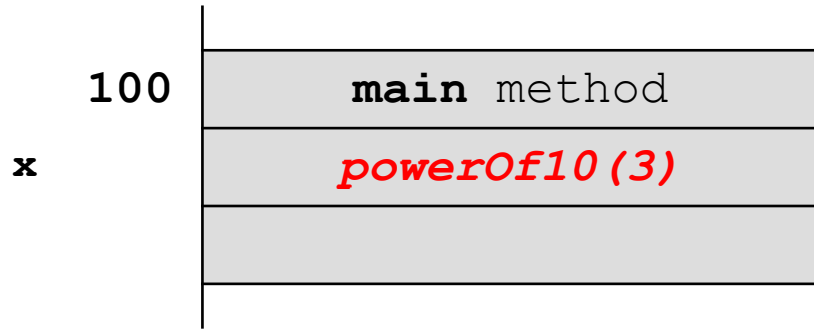
$10 * \text{powerOf10}(1)$

result

n



```
double x = Recursion.powerOf10(3);
```



result

n

600

powerOf10 method

3

$10 * \text{powerOf10}(2)$

result

n

750

powerOf10 method

2

$10 * \text{powerOf10}(1)$

result

n

800

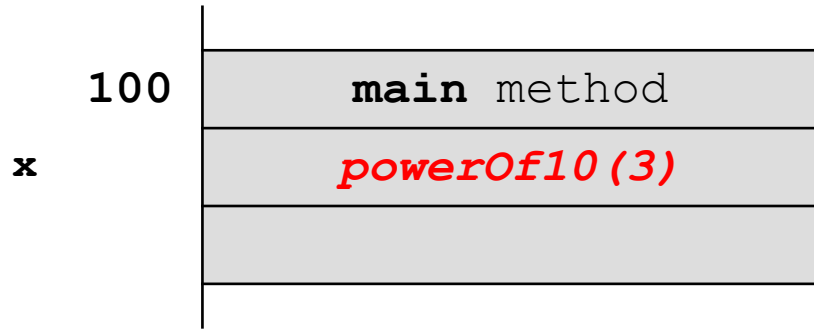
powerOf10 method

1

$10 * \text{powerOf10}(0)$



```
double x = Recursion.powerOf10(3);
```



result

n

600

powerOf10 method

3

$10 * \text{powerOf10}(2)$

result

n

750

powerOf10 method

2

$10 * \text{powerOf10}(1)$

result

n

800

powerOf10 method

1

$10 * \text{powerOf10}(0)$

result

n

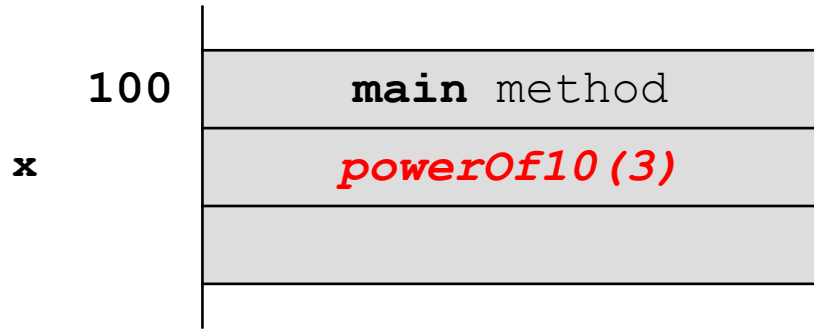
950

powerOf10 method

0



```
double x = Recursion.powerOf10(3);
```



result

600

powerOf10 method

n

3

$10 * \text{powerOf10}(2)$

result

750

powerOf10 method

n

2

$10 * \text{powerOf10}(1)$

result

800

powerOf10 method

n

1

$10 * \text{powerOf10}(0)$

result

950

powerOf10 method

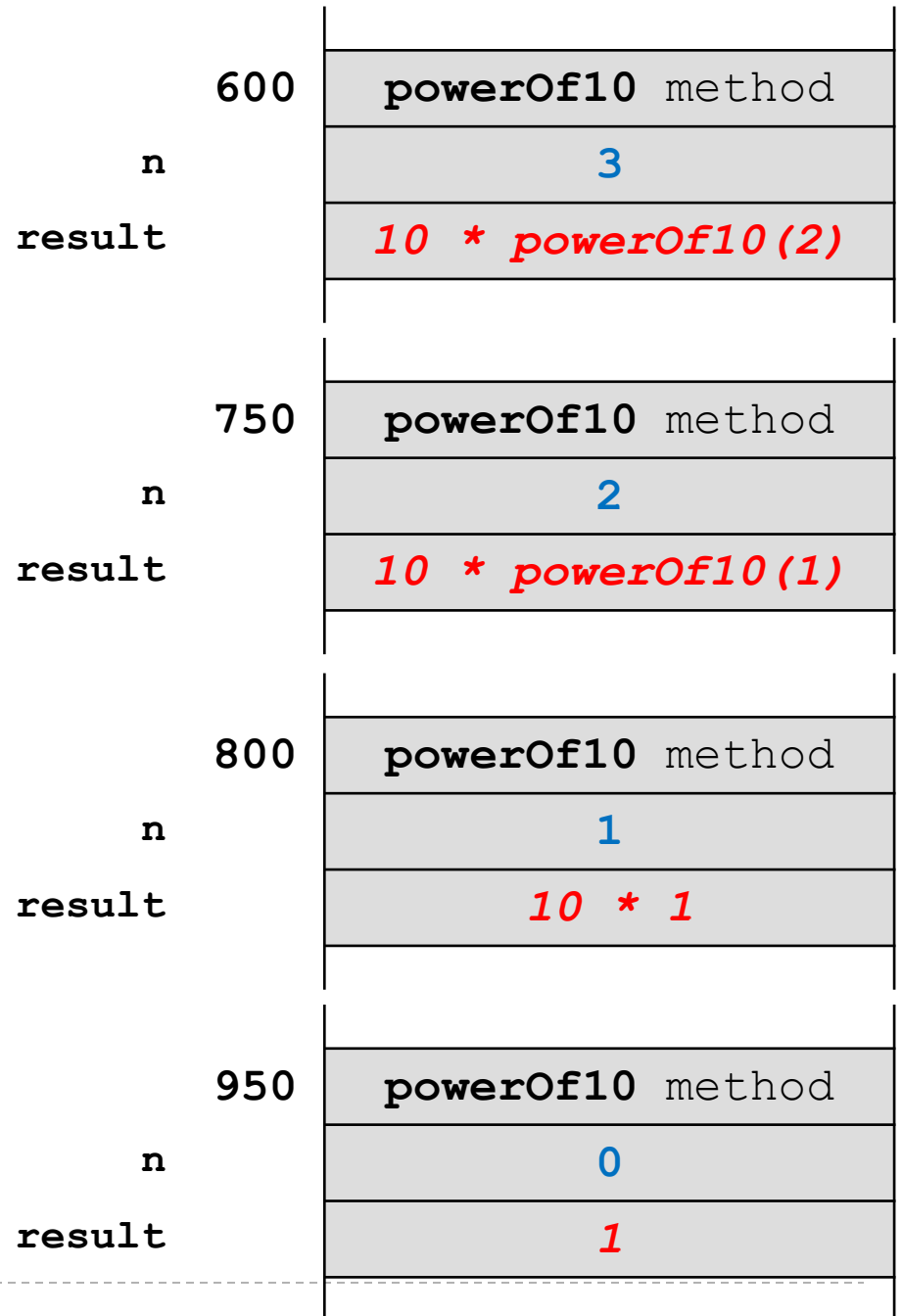
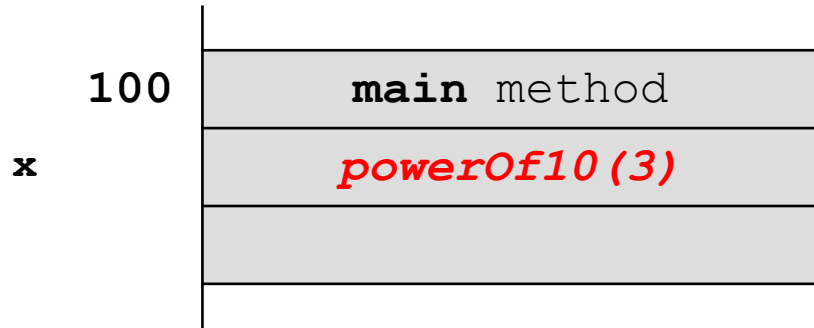
n

0

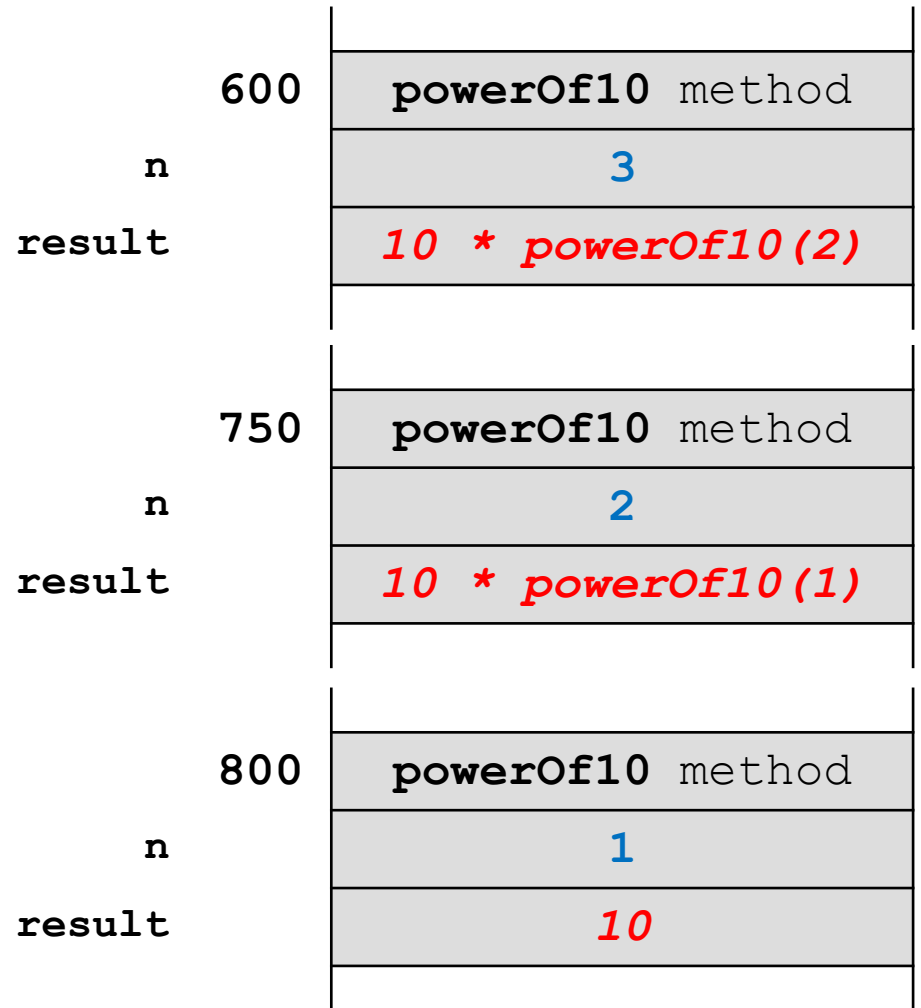
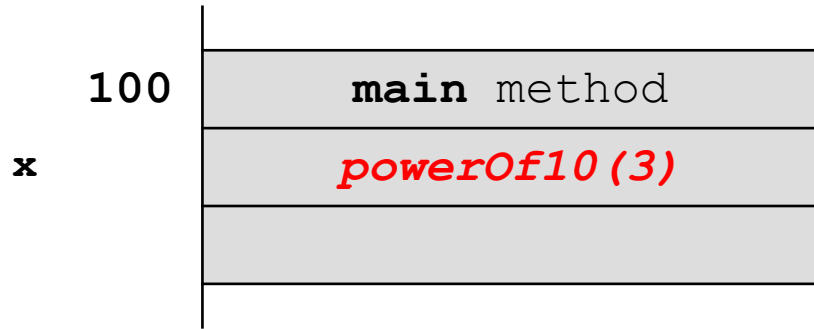
1



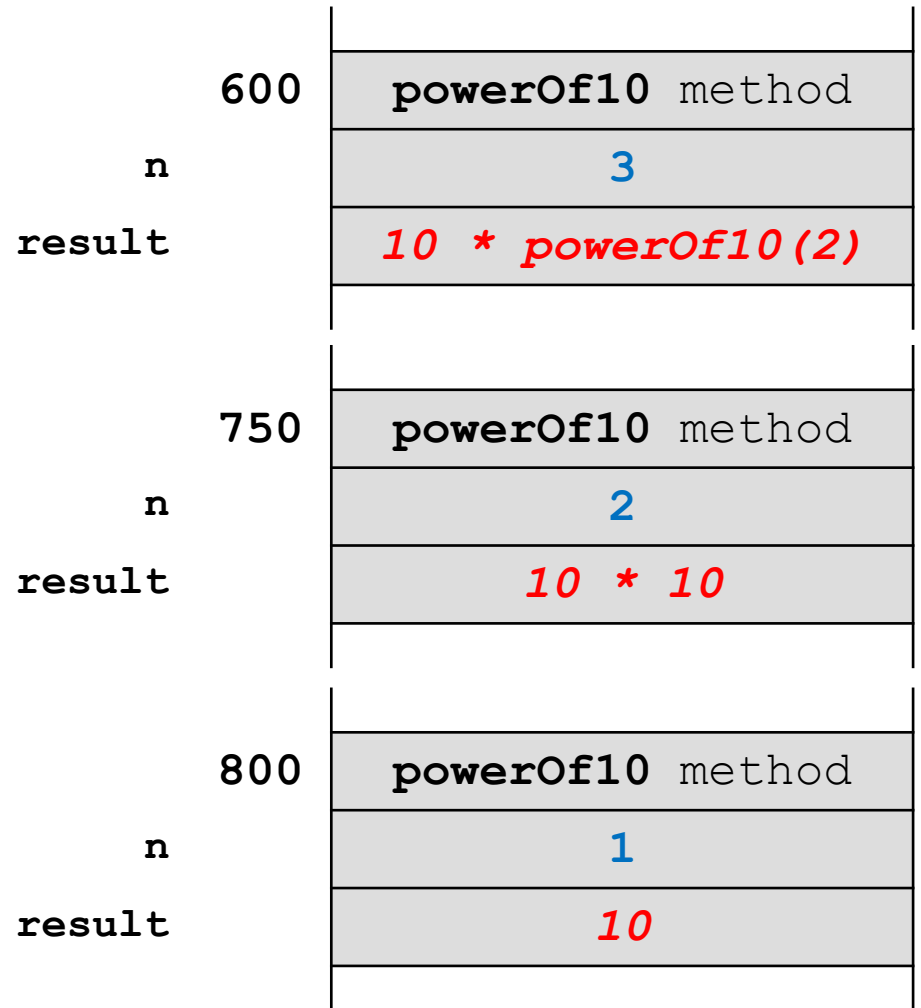
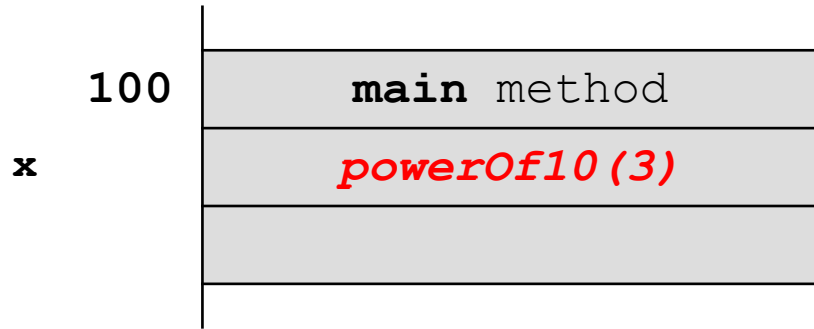
```
double x = Recursion.powerOf10(3);
```



```
double x = Recursion.powerOf10(3);
```

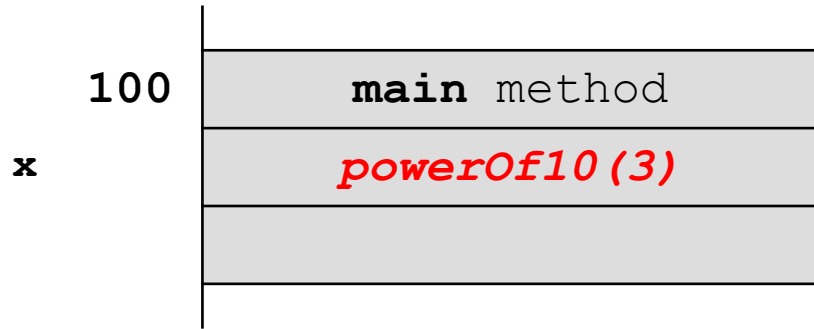


```
double x = Recursion.powerOf10(3);
```





```
double x = Recursion.powerOf10(3);
```



result

n

600

powerOf10 method

3

*10 \* powerOf10(2)*

750

powerOf10 method

2

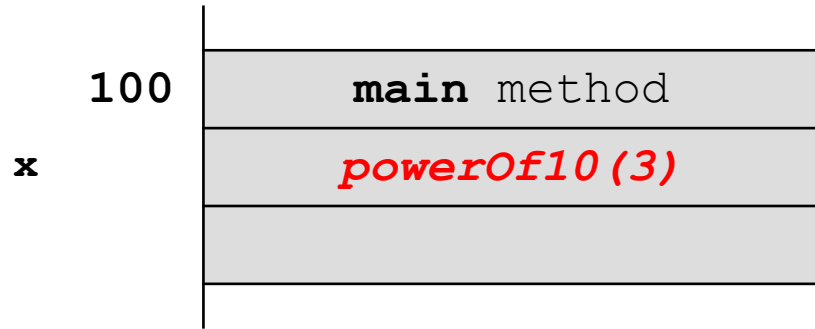
*100*

result

n



```
double x = Recursion.powerOf10(3);
```



result

n

600

powerOf10 method

3

$10 * 100$

750

powerOf10 method

2

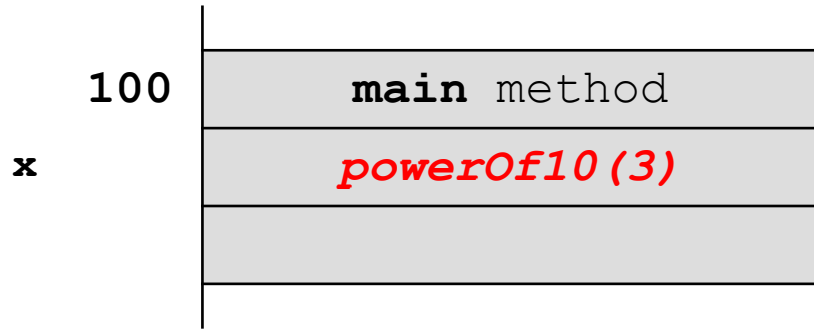
100

result

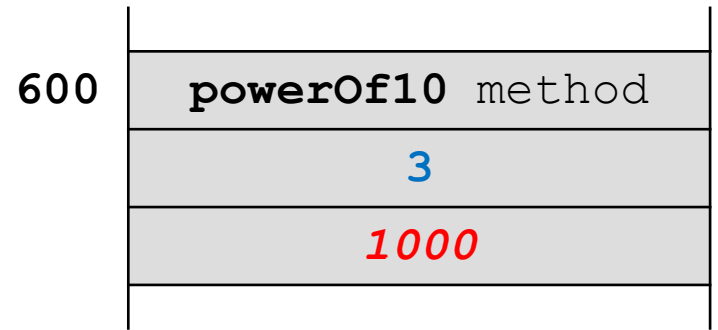
n



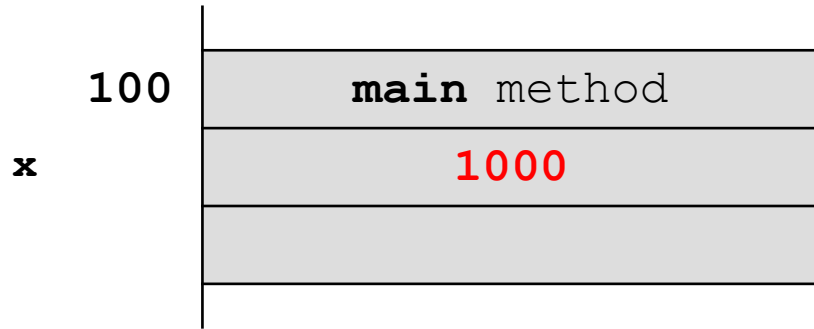
```
double x = Recursion.powerOf10(3);
```



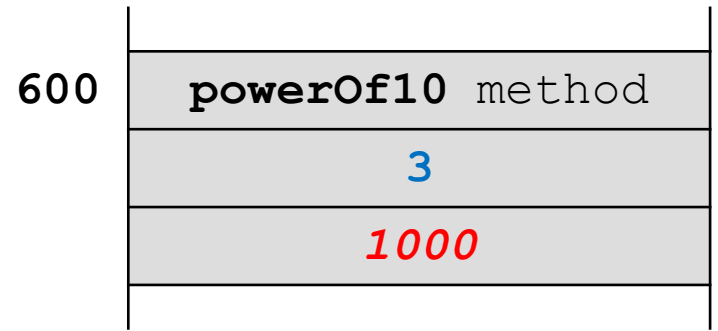
n  
result



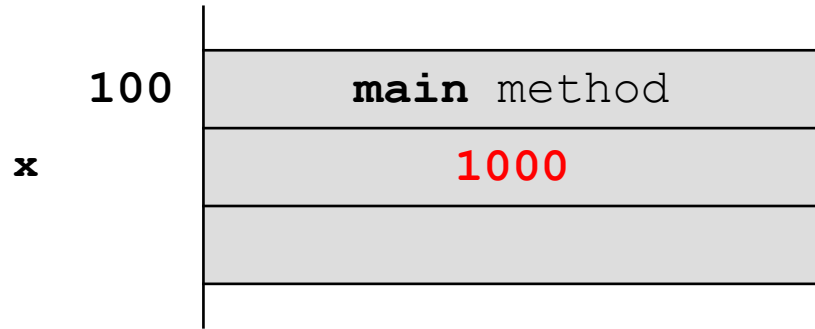
```
double x = Recursion.powerOf10(3);
```



n  
result



```
double x = Recursion.powerOf10(3);
```



# Using a stack

---

- ▶ we to simulate what the JVM is doing during the recursive method
  - ▶ each recursive call that is not a base case pushes a 10 onto the stack
  - ▶ a recursive call that reaches a base case pushes a 1 onto the stack
  - ▶ pop the stack until it is empty, multiplying the values as you go

```
public static int powerOf10(int n) {  
  
    Stack<Integer> t = new Stack<Integer> ();  
    // recursive calls  
    while (n > 0) {  
        t.push(10);  
        n--;  
    }  
    // base case: n == 0  
    t.push(1);  
  
    // accumulate the result  
    int result = t.pop();  
    while (!t.isEmpty()) {  
        result *= t.pop();  
    }  
    return result;  
}
```

# Converting a Recursive Method

---

- ▶ using a stack in the previous example is too complicated for the problem that we were trying to solve
  - ▶ but it illustrates the basic idea of using a stack to convert a recursive method to an iterative method
- ▶ more complicated examples require greater sophistication in exactly what is pushed onto the stack
  - ▶ see the make-up lab (Part 1) when it becomes available



# Implementation with ArrayList

---

- ▶ **ArrayList** can be used to efficiently implement a stack
- ▶ the end of the list becomes the top of the stack
  - ▶ adding and removing to the end of an **ArrayList** usually can be performed in  $O(1)$  time

```
public class Stack<E> {
    private ArrayList<E> stack;

    public Stack() {
        this.stack = new ArrayList<E>();
    }

    public void push(E element) {
        this.stack.add(element);
    }

    public E pop() {
        return this.stack.remove(this.stack.size() - 1);
    }
}
```

# Implementation with Array

---

- ▶ the ArrayList version of stack hints at how to implement a stack using a plain array
  - ▶ however, an array always holds a fixed number of elements
    - ▶ you cannot add to the end of the array without creating a new array
    - ▶ you cannot remove elements from the array without creating a new array
- ▶ instead of adding and removing from the end of the array, we need to keep track of which element of the array represents the current top of the stack
  - ▶ we need a field for this index

```
import java.util.Arrays;
import java.util.EmptyStackException;

public class IntStack {
    // the initial capacity of the stack
    private static final int DEFAULT_CAPACITY = 16;

    // the array that stores the stack
    private int[] stack;

    // the index of the top of the stack (equal to -1 for an empty stack)
    private int topIndex;
```

```
/**  
 * Create an empty stack.  
 */  
public IntStack() {  
    this.stack = new int[IntStack.DEFAULT_CAPACITY];  
    this.topIndex = -1;  
}
```

# Implementation with Array

---

```
IntStack t = new IntStack();
```

**this.topIndex** == -1

|                   |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |
|-------------------|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|
| <b>this.stack</b> | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0  | 0  | 0  | 0  | 0  | 0  |
| index             | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |



# Implementation with Array

---

- ▶ pushing a value onto the stack:
  - ▶ increment **this.topIndex**
  - ▶ set the value at **this.stack[this.topIndex]**

# Implementation with Array

---

```
IntStack t = new IntStack();  
t.push(7);
```

**this.topIndex == 0**

|                   |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |
|-------------------|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|
| <b>this.stack</b> | 7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0  | 0  | 0  | 0  | 0  | 0  |
| index             | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |





# Implementation with Array

---

```
IntStack t = new IntStack();  
t.push(7);  
t.push(-5);
```

**this.topIndex == 1**

|                   |   |    |   |   |   |   |   |   |   |   |    |    |    |    |    |    |
|-------------------|---|----|---|---|---|---|---|---|---|---|----|----|----|----|----|----|
| <b>this.stack</b> | 7 | -5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0  | 0  | 0  | 0  | 0  | 0  |
| index             | 0 | 1  | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |



# Implementation with Array

---

- ▶ popping a value from the stack:
  - ▶ get the value at **this.stack[this.topIndex]**
  - ▶ decrement **this.topIndex**
  - ▶ return the value
- ▶ notice that we do not need to modify the value stored in the array

# Implementation with Array

---

```
IntStack t = new IntStack();  
t.push(7);  
t.push(-5);  
int value = t.pop(); // value == -5
```

**this.topIndex == 0**

|                   |   |    |   |   |   |   |   |   |   |   |    |    |    |    |    |    |
|-------------------|---|----|---|---|---|---|---|---|---|---|----|----|----|----|----|----|
| <b>this.stack</b> | 7 | -5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0  | 0  | 0  | 0  | 0  | 0  |
| index             | 0 | 1  | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |



```
/**
 * Pop and return the top element of the stack.
 *
 * @return the top element of the stack
 * @throws EmptyStackException if the stack is empty
 */
public int pop() {
    // is the stack empty?
    if (this.topIndex == -1) {
        throw new EmptyStackException();
    }
    // get the element at the top of the stack
    int element = this.stack[this.topIndex];

    // adjust the top of stack index
    this.topIndex--;

    // return the element that was on the top of the stack
    return element;
}
```

# Implementation with Array

---

// stack state when we can safely do one more push

**this.topIndex == 14**

|                   |   |    |   |   |   |   |   |   |   |    |    |    |    |    |    |    |
|-------------------|---|----|---|---|---|---|---|---|---|----|----|----|----|----|----|----|
| <b>this.stack</b> | 7 | -5 | 6 | 3 | 2 | 1 | 0 | 0 | 9 | -3 | 2  | 7  | 1  | -2 | 1  | 0  |
| index             | 0 | 1  | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9  | 10 | 11 | 12 | 13 | 14 | 15 |



```
/**  
 * Push an element onto the top of the stack.  
 *  
 * @param element the element to push onto the stack  
 */  
public void push(int element) {  
    // is there capacity for one more element?  
    if (this.topIndex < this.stack.length - 1) {  
        // increment the top of stack index and insert the element  
        this.topIndex++;  
        this.stack[this.topIndex] = element;  
    }  
    else {
```

# Adding Capacity

---

- ▶ if we run out of capacity in the current array we need to add capacity by doing the following:
  - ▶ make a new array with greater capacity
    - ▶ how much more capacity?
  - ▶ copy the old array into the new array
  - ▶ set **this.stack** to refer to the new array
  - ▶ push the element onto the stack

```
else {  
    // make a new array with double previous capacity  
    int[] newStack = new int[this.stack.length * 2];  
  
    // copy the old array into the new array  
    for (int i = 0; i < this.stack.length; i++) {  
        newStack[i] = this.stack[i];  
    }  
  
    // refer to the new array and push the element onto the stack  
    this.stack = newStack;  
    this.push(element);  
}  
}
```



# Adding Capacity

---

- ▶ when working with arrays, it is a common operation to have to create a new larger array when you run out of capacity in the existing array
- ▶ you should use ***Arrays.copyOfOf*** to create and copy an existing array into a new array

```
else {
```

```
int[] newStack = Arrays.copyOf(this.stack, this.stack.length * 2);
```

```
// refer to the new array and push the element onto the stack
```

```
this.stack = newStack;
```

```
this.push(element);
```

```
}
```

```
}
```