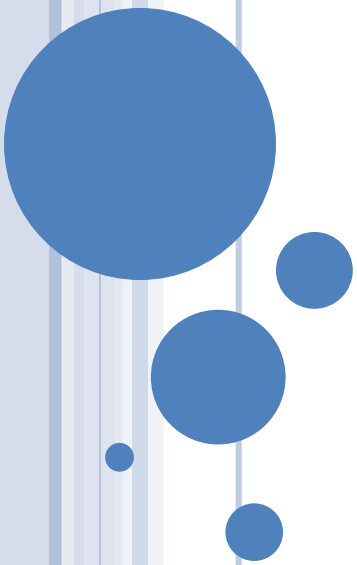


Comp 110/401 Recursion

Instructor: Prasun Dewan

Prerequisites

- Conditionals

Recursion

- English definition
  - Return (Oxford/Webster)
  - procedure repeating itself indefinitely or until condition met, such as grammar rule (Webster)
- English examples
  - **adequate**: satisfactory
  - **satisfactory**: adequate
- My definition: **recursion**: recursion
- Mathematics
  - expression giving successive terms of a series (Oxford)
- Programming
  - Method calling itself.
  - On a smaller problem.
  - Alternative to loops.

FACTORIAL(N)

```
<terminated> Factorial [Ja
3
factorial = 6
2
factorial = 2
-1
```

1*2*3*…*n

```
public static int factorial(int n) {
        int product = 1;
        while (n > 0) {
                product *= n;
                n -= 1;
        }
        return product;
}
```

```
public static void main (String[] args) {
        while (true) {   // loop condition never false
                int n = Console.readInt();
                if (n < 0)
                        break;
                System.out.println("factorial = " + factorial(n));
        }
}
```

Defining Factorial(n)

Product of the first n numbers

1*2*3*…*n

factorial(0)	= 1	
factorial(1)	= 1	= 1*factorial(0)
factorial(2)	= 2*1	= 2*factorial(1)
factorial(3)	= 3*2*1	= 3*factorial(2)
factorial(4)	= 4*3*2*1	= 4*factorial(3)
factorial(n)	= n*n-1*…*1	= n*factorial(n-1)

Defining Factorial(n)

factorial(n) = 1 if n == 0

factorial(n) = n*factorial(n-1) if n > 0

Implementing Factorial(n) (Edit)

factorial(n) = 1 if n == 0

factorial(n) = n*factorial(n-1) if n > 0

```
public static int factorial(int n) {
        …
}
```

Implementing Factorial(n)

factorial(n) = 1 if n == 0

factorial(n) = n*factorial(n-1) if n > 0

n < 0 ?

Function must return something for all cases

```
public static int factorial (int n) {
        if (n == 0)
                return 1;
        if (n > 0)
                return n*factorial(n-1);
}
```

Multiple return

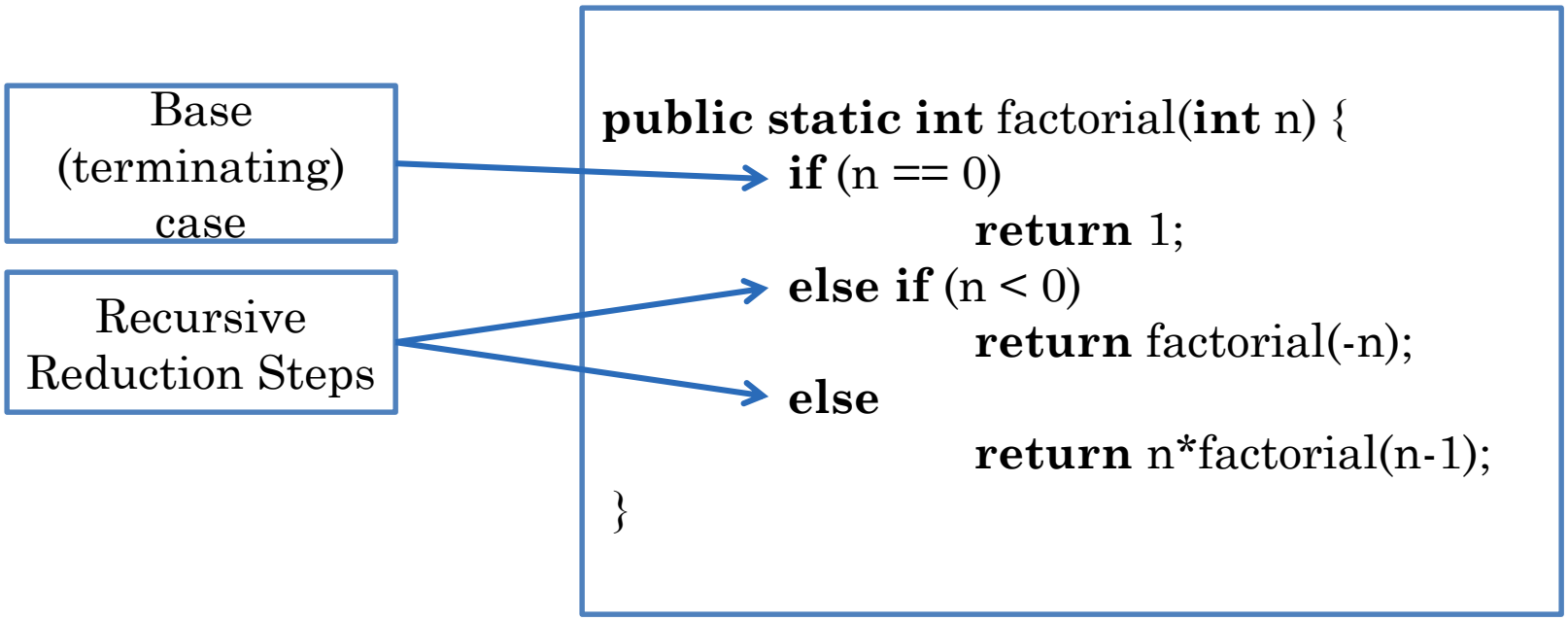
Early return

Implementing Factorial(n)

factorial(n) = 1 if n == 0

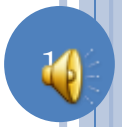
factorial(n) = n*factorial(n-1) if n > 0

factorial(n) = factorial(-n) if n < 0



Recursive Methods

- Should have base case(s)
- Recurse on smaller problem(s)
  - recursive calls should converge to base case(s)



General Form of a Recursive Method

```
if (base case 1 )
      return solution for base case 1
else if (base case 2)
      return solution for base case 2
….
else if (base case n)
      return solution for  base case n
else if (recursive case 1)
       do some preprocessing
       recurse on reduced problem
       do some postprocessing
…
else if (recursive case m)
       do some preprocessing
       recurse on reduced problem
       do some postprocessing
```

Recursion vs. Loops (Iteration)

```
public static int factorial(int n) {
        int product = 1;
        while (n > 0) {
                product *= n;
                n -= 1;
        }
        return product;
}
```

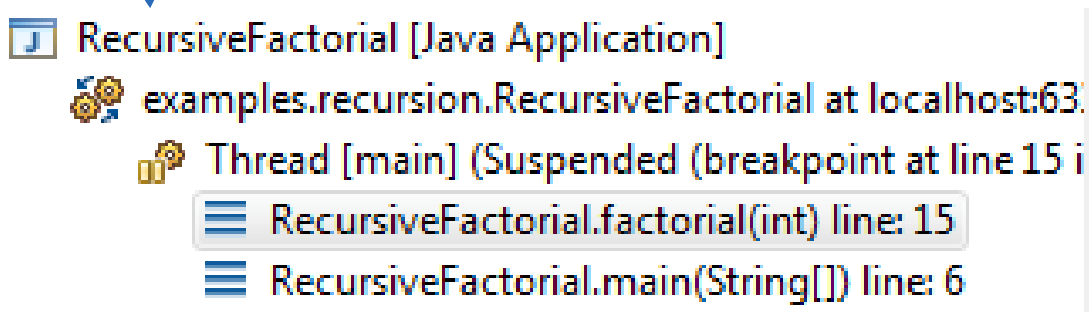
Implementation follows from definition

```
public static int factorial(int n) {
        if (n == 0)
                return 1;
        else if (n < 0)
                return factorial(-n);
        else
                return n*factorial(n-1);
}
```

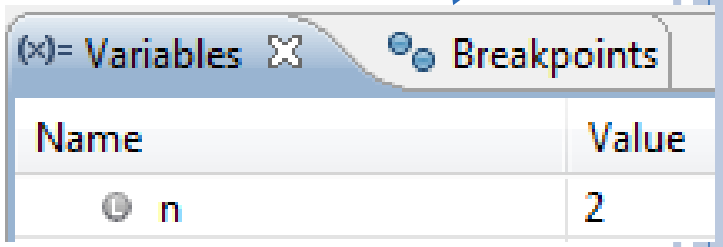
Tracing Recursive Calls

```
public class RecursiveFactorial {

    public static void main(String[] args) {
        System.out.println(factorial(2));
    }

    public static int factorial(int n) {
        if (n == 0)
            return 1;
        else if (n < 0)
            return factorial(-n);
        else
            return n*factorial(n-1);
    }

}
```

Call Stack



Locals



Tracing Recursive Calls

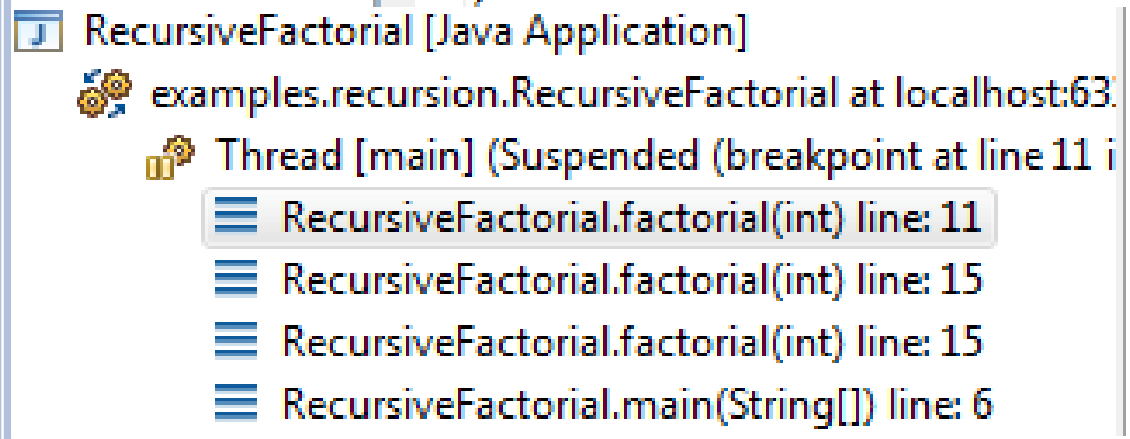
```
public class RecursiveFactorial {

    public static void main(String[] args) {
        System.out.println(factorial(2));
    }

    public static int factorial(int n) {
        if (n == 0)
            return 1;
        else if (n < 0)
            return factorial(-n);
        else
            return n*factorial(n-1);
    }

}
```

Invocation	n	return value
factorial(0)	0	?
factorial(1)	1	?
factorial(2)	2	?

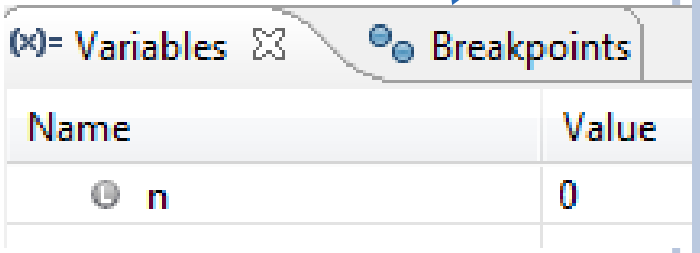
Tracing Recursive Calls

```
public class RecursiveFactorial {

    public static void main(String[] args) {
        System.out.println(factorial(2));
    }

    public static int factorial(int n) {
        if (n == 0)
            return 1;
        else if (n < 0)
            return factorial(-n);
        else
            return n*factorial(n-1);
    }

}
```

Call Stack



Locals



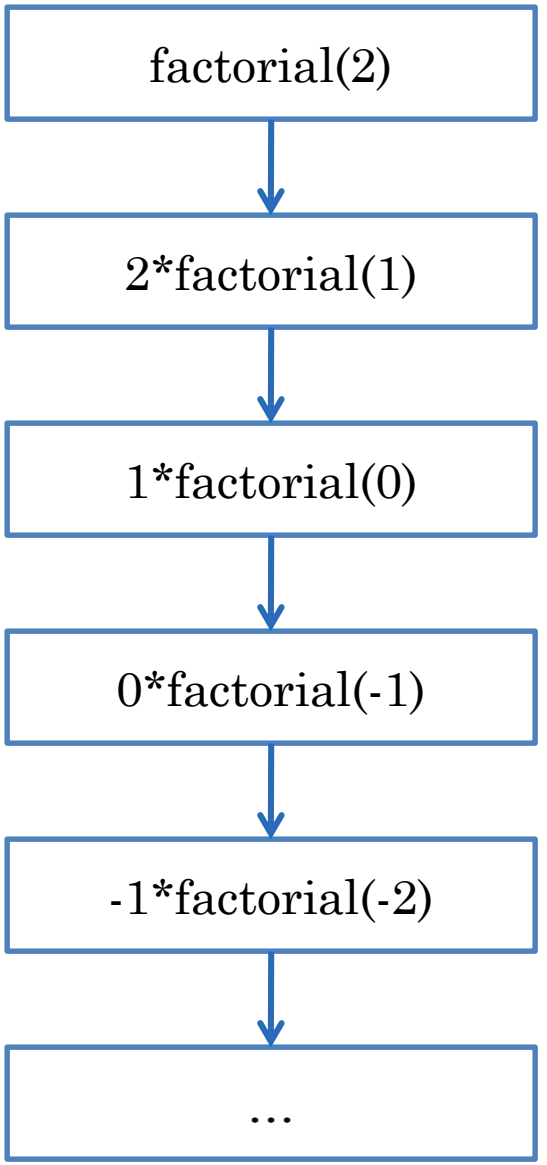
Tracing Recursive Calls

```
public class RecursiveFactorial {

    public static void main(String[] args) {
        System.out.println(factorial(2));
    }

    public static int factorial(int n) {
        if (n == 0)
            return 1;
        else if (n < 0)
            return factorial(-n);
        else
            return n*factorial(n-1);
    }

}
```

Invocation	n	return value
factorial(0)	0	1
factorial(1)	1	1
factorial(2)	2	2

Recursion Pitfalls

```
public static int factorial (int n) {
        return n*factorial(n-1);
}
```

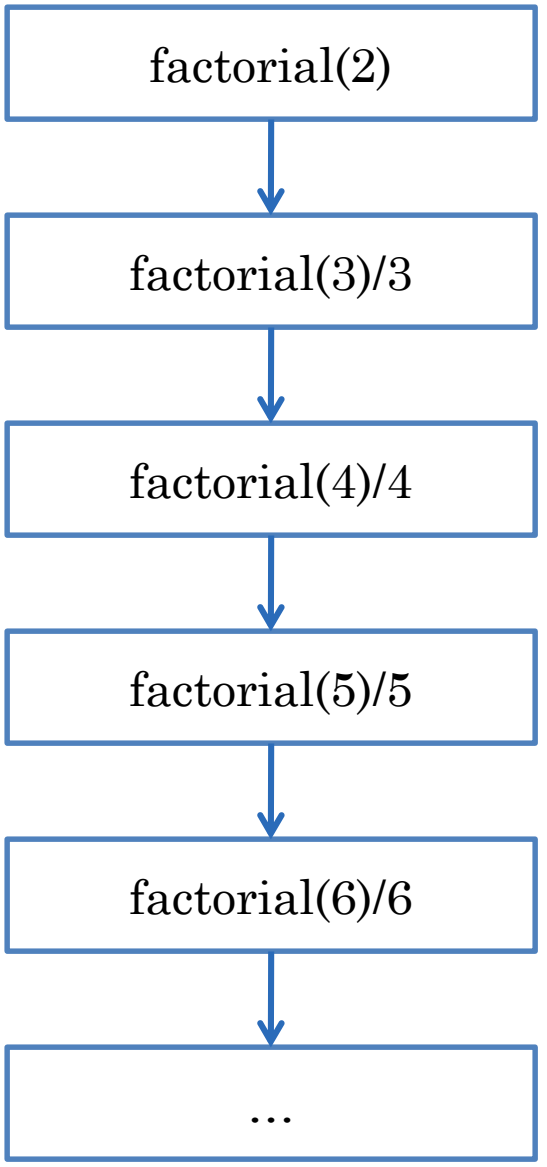


Infinite recursion! (stack overflow)

No base case

Recursion Pitfalls

```
public static int factorial (int n) {
        if (n == 0)
                return 1;
        else if (n < 0)
                return factorial(-n);
        else
                return factorial(n+1)/n;
}
```



Infinite recursion!

Recurses on bigger problem

Recursive Functions with Multiple Parameters

power(base, exponent)

$base^{exponent}$

base*base*base*…*base

Exponent # of times

power(0, exponent) = 0

power(1, exponent) = 1

power(2, exponent) = 2*2*…*2 (exponent times)

power(3, exponent) = 3*3*…*3 (exponent times)

No pattern!

Recursive Functions with Multiple Parameters (Edit)

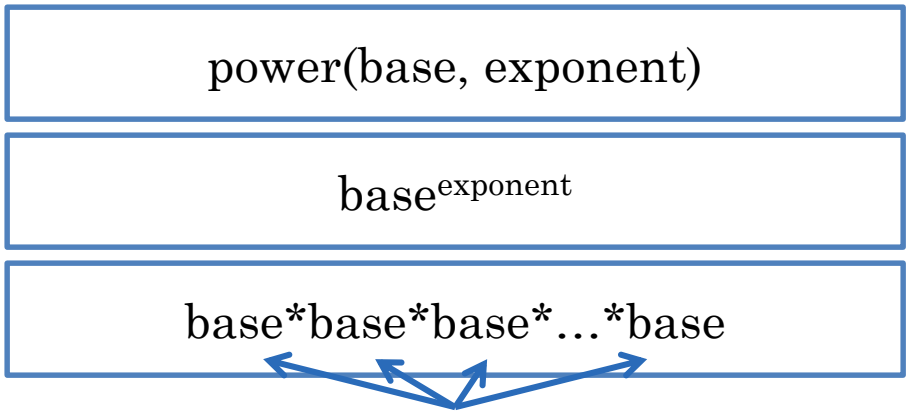
power(base, exponent)

$base^{exponent}$

base*base*base*…*base

Exponent # of times

Recursive Functions with Multiple Parameters (Edit)



power(base, 0)	= 1	
power(base, 1)	= base*1	= base*power(base, 0)
power(base, 2)	= base*base*1	= base*power(base, 1)
power(base, exponent)		= base*power(base, exponent-1)

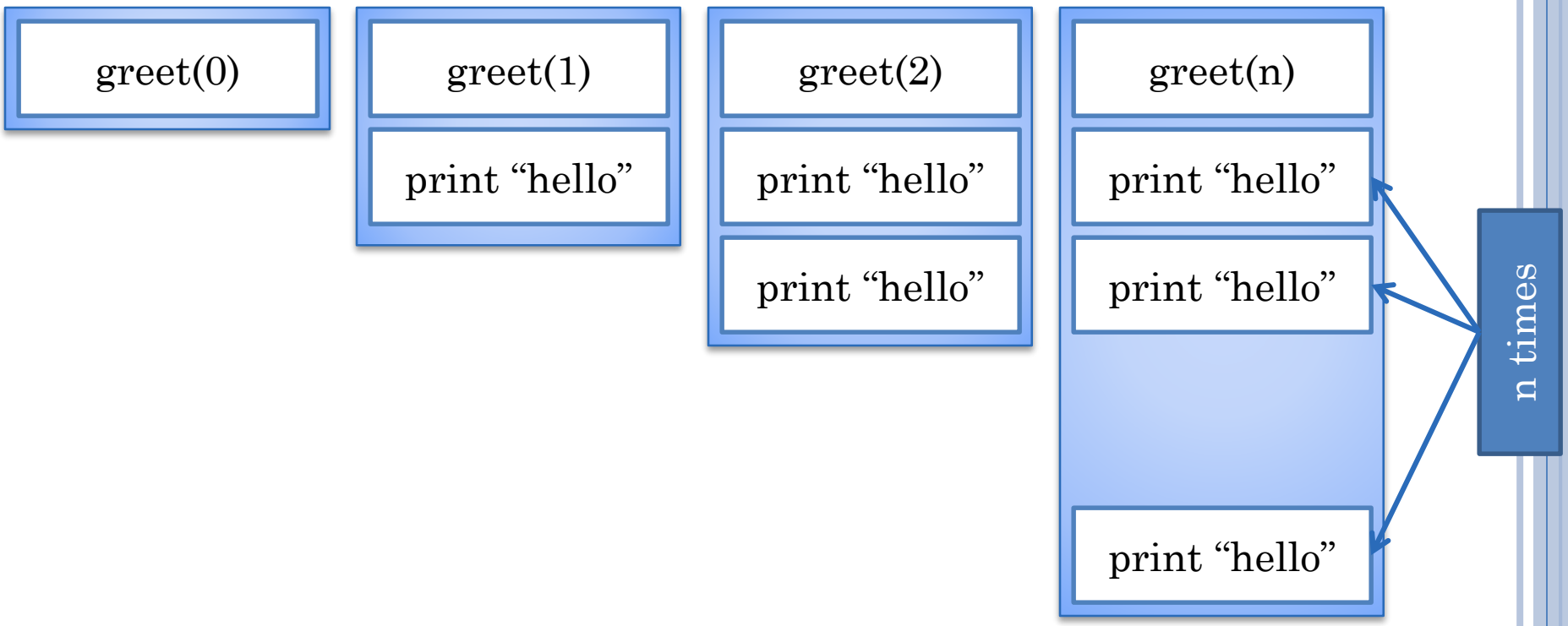
Defining power(base, exponent)

power(base, exponent) = 1 if exponent <= 0

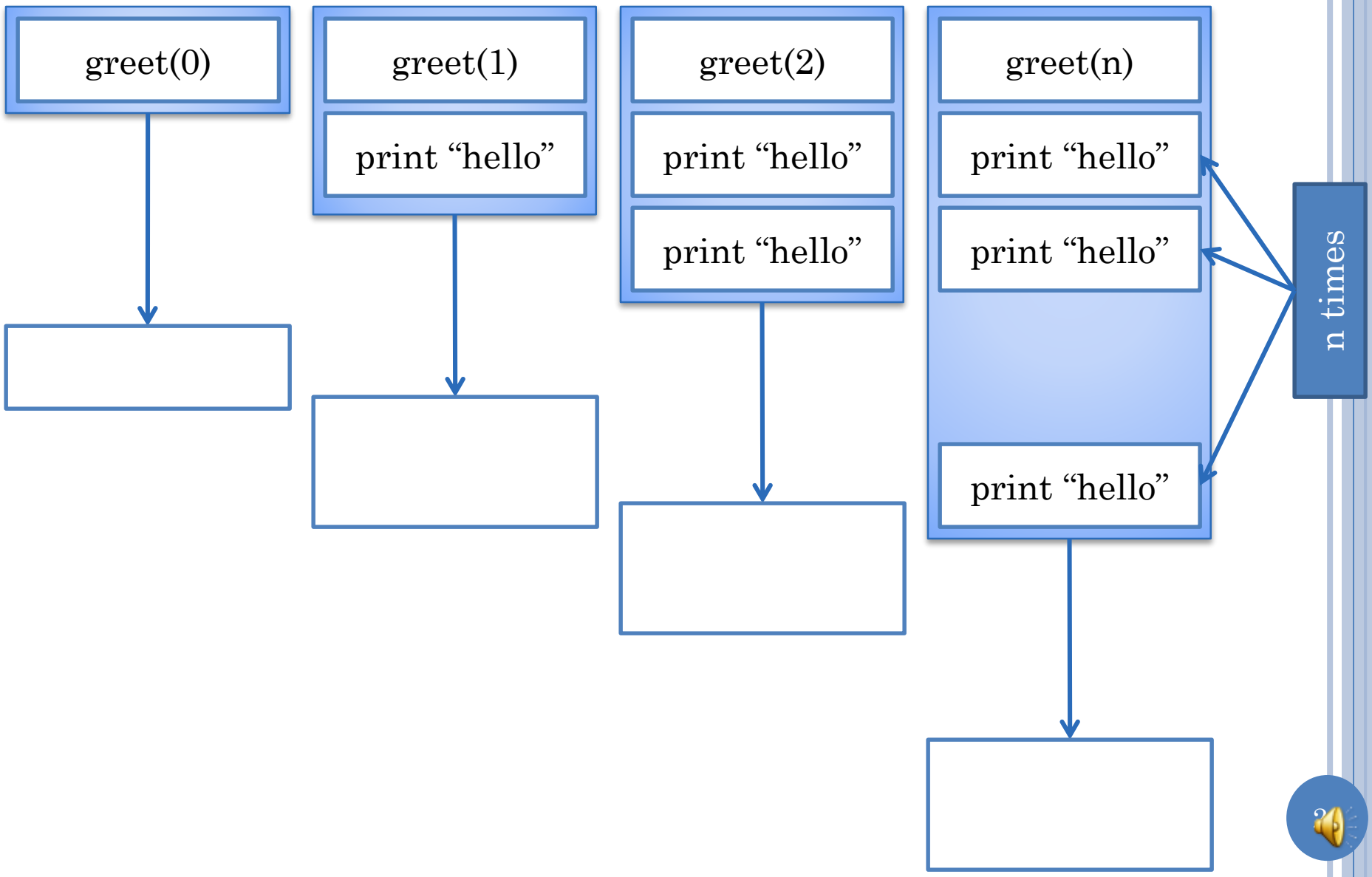
power(base, exponent) = base*power(base, exponent-1) if exponent > 0

```
public static int power(int base, int exponent) {
        if (n <= 0)
                return 1;
        else
                return base*power(base, exponent-1);
}
```

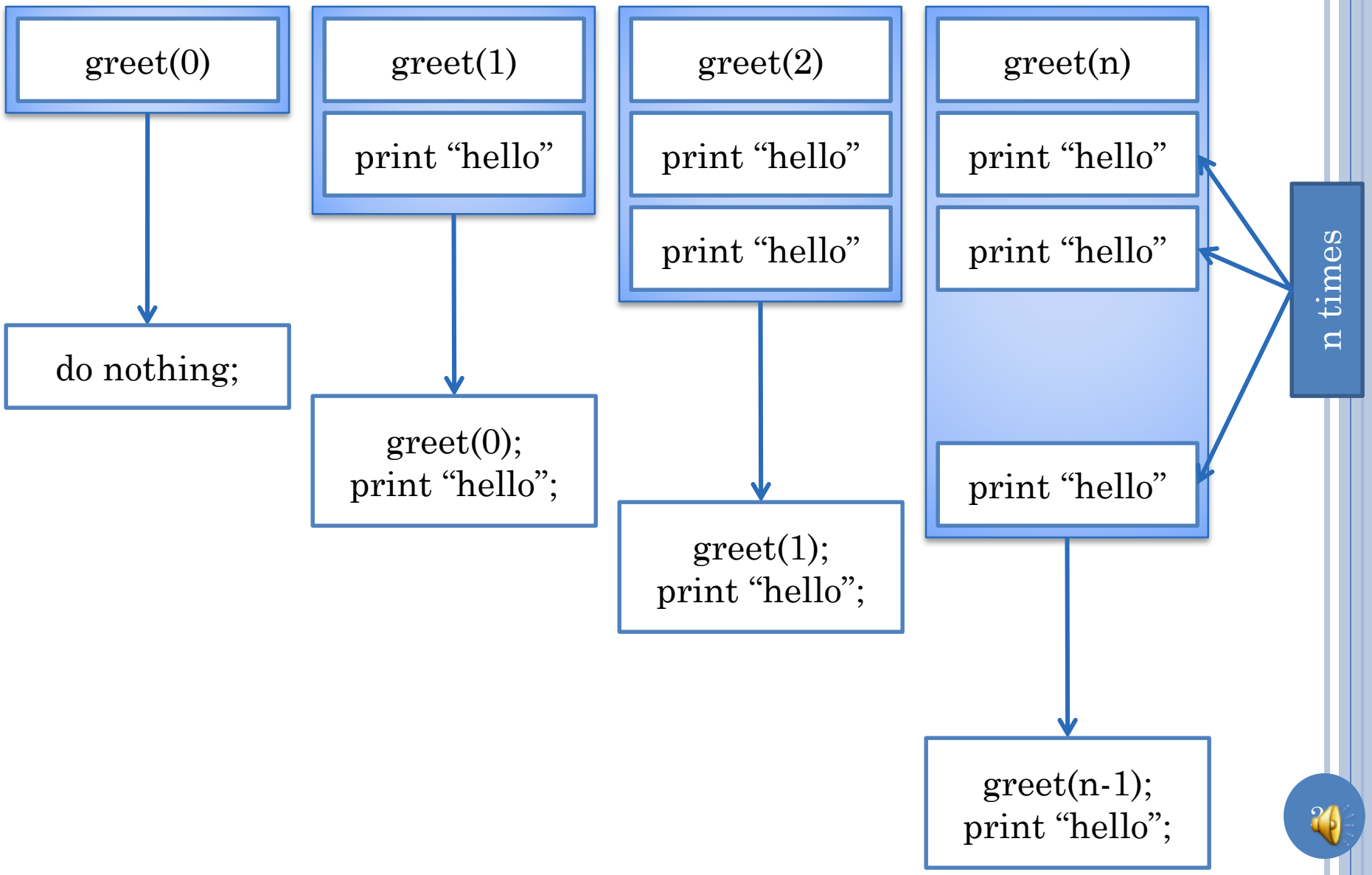
Recursive Procedures: greet(n)



Defining greet(n) (Edit)



Defining greet(n)



Defining greet(n)

greet(n)	→	do nothing;	if n <= 0
greet(n)	→	greet(n-1); print “hello”;	if n > 0

Implementing greet(n) (Edit)

greet(n) →	do nothing;	if n <= 0
greet(n) →	greet(n-1); print “hello”;	if n > 0

Implementing greet(n)

greet(n) → do nothing; if n <= 0

greet(n) → greet(n-1);
print “hello”; if n > 0

```
public static void greet (int n) {
        if (n > 0) {
                greet(n-1);
                System.out.println(“hello”);
        }
}
```

List-based Recursion: multiplyList()

multiplyList()	= 1	if remaining input is: -1
multiplyList()	= 2	if remaining input is: 2, -1
multiplyList()	= 12	if remaining input is: 2, 6, -1

multiplyList() = readNextVal()	if nextVal < 0
multiplyList() = readNextVal() * multiplyList()	if nextVal > 0

List-based Recursion: multiplyList()

multiplyList() = 1	if nextVal < 0
multiplyList() = readNextVal() * multiplyList()	if nextVal > 0

```
public static int multiplyList () {
        int nextVal = Console.readInt();
        if (nextVal  <  0)
                return 1;
        else
                return nextVal*multiplyList();
}
```

Tracing multiplyList()

```
public static int multiplyList () {
        int nextVal = Console.readInt();
        if (nextVal  <  0)
                return 1;
        else
                return nextVal*multiplyList();
}
```

Invocation	Remaining input	Return value
multiplyList()	-1	1
multiplyList()	30, -1	30
multiplyList()	2, 30, -1	60

Summary

- Recursive Functions
- Recursive Procedures
- Number-based Recursion
- List-based Recursion
- See sections on trees, graphs, DAGs and visitor for other kinds of recursion