Data Abstraction
Recursion (5.1)

Janyl Jumadinova

October 28, 2020
Recursion:
when a method calls itself.

Example: the factorial function
\[ n! = 1 \times 2 \times 3 \times \ldots \times (n-1) \times n \]
The Recursion Pattern

Recursion:
when a method calls itself.

Example: the factorial function
\[ n! = 1 \times 2 \times 3 \times \ldots \times (n-1) \times n \]
Example: Factorial Function

Recursive definition:

\[ f(n) = 1 \text{ if } n = 0 \]
\[ f(n) = n \times f(n - 1) \text{ otherwise} \]
**Example: Factorial Function**

Recursive definition:

\[
f(n) = 1 \text{ if } n = 0 \\
f(n) = n \times f(n - 1) \text{ otherwise}
\]

```java
public static int factorial(int n) throws IllegalArgumentException {
    if (n < 0) 
        throw new IllegalArgumentException(); // argument must be nonnegative
    else if (n == 0) 
        return 1; // base case
    else 
        return n * factorial(n - 1); // recursive case
}
```
Content of a Recursive Method

Base case(s):
Values of the input variables for which we perform no recursive calls are called base cases (there should be at least one base case).
Every possible chain of recursive calls must eventually reach a base case.

Recursive calls:
Calls to the current method.
Each recursive call should be defined so that it makes progress towards a base case.
Content of a Recursive Method

**Base case(s):**
- Values of the input variables for which we perform no recursive calls are called **base cases** (there should be at least one base case).
- Every possible chain of recursive calls must eventually reach a base case.
Content of a Recursive Method

Base case(s):
- Values of the input variables for which we perform no recursive calls are called base cases (there should be at least one base case).
- Every possible chain of recursive calls must eventually reach a base case.

Recursive calls:
- Calls to the current method.
- Each recursive call should be defined so that it makes progress towards a base case.
Recursion trace:
- A box for each recursive call
- An arrow from each caller to callee
- An arrow from each callee to caller showing return value
Factorial Function Example:

- There are \( n + 1 \) activations to compute \( \text{factorial}(n) \).
- Parameter decreases from \( n \) to \( n-1 \) from the first to second call, and so on until reaching parameter 0.
Factorial Function Example:

- There are $n + 1$ activations to compute $\text{factorial}(n)$.
  - Parameter decreases from $n$ to $n-1$ from the first to second call, and so one until reaching parameter 0.
- Each individual activation executes a constant number of operations: $O(1)$ each.
Factorial Function Example:

- There are \( n + 1 \) activations to compute \( \text{factorial}(n) \).
  - Parameter decreases from \( n \) to \( n-1 \) from the first to second call, and so on until reaching parameter 0.
- Each individual activation executes a constant number of operations: \( O(1) \) each.
- Overall time complexity: \( O(n) \).
Linear Recursion

**Test for base cases:**

- Begin by testing for a set of base cases (there should be at least one).
- Every possible chain of recursive calls must eventually reach a base case, and the handling of each base case should not use recursion.
Linear Recursion

Test for base cases:

- Begin by testing for a set of base cases (there should be at least one).
- Every possible chain of recursive calls must eventually reach a base case, and the handling of each base case should not use recursion.

Recur once:

- Perform a single recursive call
- This step may have a test that decides which of several possible recursive calls to make, but it should ultimately make just one of these calls
- Define each possible recursive call so that it makes progress towards a base case.
Example: Linear Sum

```
linearSum(data, 5)

linearSum(data, 4)

linearSum(data, 3)

linearSum(data, 2)

linearSum(data, 1)

linearSum(data, 0)
```

```
return 15 + data[4] = 15 + 8 = 23

return 13 + data[3] = 13 + 2 = 15

return 7 + data[2] = 7 + 6 = 13

return 4 + data[1] = 4 + 3 = 7

return 0 + data[0] = 0 + 4 = 4

return 0
```