



# Enums, recursion, software design

Lecture 14 of TDA 540 (Objektorienterad Programmering)

---

Carlo A. Furia   Alex Gerdes

Chalmers University of Technology – Gothenburg University

Fall 2016

# Pop quiz!

1. Go to [kahoot.it](https://kahoot.it)
2. Enter **PIN** shown on **projector screen**
3. Pick a **nickname** and go!

# Enumerated types (enums)

**Enumerated types** are a convenient solution to create type with a only **finite number of values**.

- yes, no, I don't know
- Swedish counties: Stockholm, Uppsala, Kalmar, Västra Götaland, Dalarna, ...
- age ranges: infant, adolescent, adult, senior

```
enum Answer { YES, NO, DONT_KNOW };
```

Type Answer is a **reference type** that implicitly inherits from class Enum and includes 3 distinct constant values Answer.**YES**, Answer.**NO**, and Answer.**DONT\_KNOW**.

Even if they are reference types, using == compares instances of **enum** types by **value**.

# Enumerated types (enums)

Enumerated types are just a **special syntax for classes** with some restrictions. In particular, you can redefine how values are displayed and compared.

```
enum Answer {
    YES("Y"),
    NO("N"),
    DONT_KNOW("?");

    Answer(String repr)
    { this.repr = repr; }

    private String repr;

    @Override
    public String toString()
    { return this.repr; }
}
```

```
Answer a = Answer.YES;
System.out.println(a); // print "Y"
```

Understanding the behavior of the various **programming language constructs** is only the first step towards writing **good** programs.

- **good**  $\simeq$  correct, readable, modifiable, efficient, . . .

We give an overview of some **design principles** and **techniques** that can help write better programs.

- Design **techniques**:
  - top-down design
  - bottom-up design
  - refactoring
  - test-driven development
- Design **principles**:
  - do not repeat yourself
  - keep it small
  - information hiding
  - design for change

# Top-down design

1. start designing the **high-level** (abstract) components (abstract classes and method signatures)
2. **refine** (reduce abstraction) the components by adding details
3. until everything is **concrete** and executable

Step 1:

```
interface AccountI {  
    void deposit(int amount);  
}
```

Step 2:

```
abstract class Account  
    implements AccountI  
{ int balance;  
  
    // set balance to 0  
    Account() { }  
  
    // add 'amount' to 'balance'  
    abstract  
    void deposit(int amount);  
}
```

Step 3:

```
class Account  
    implements AccountI  
{ int balance;  
  
    // set balance to 0  
    Account() { balance = 0; }  
  
    // add 'amount' to 'balance'  
    void deposit(int amount)  
    { balance += amount; }  
}
```

# Top-down design

Top-down design is also applicable at the level of **individual method implementations** as stepwise refinement.

Step 1:

```
// input: non-null array a
// output: sum of values in a
int sum(int[] a)
{ }
```

Step 3:

```
// input: non-null array a
// output: sum of values in a
int sum(int[] a)
{ int sum = 0;
  // for each position k in a:
  // add a[k] to sum:
  sum += a[k];
  return sum; }
```

Step 2:

```
// input: non-null array a
// output: sum of values in a
int sum(int[] a)
{ int sum = 0;
  // for each position k in a:
  // add a[k] to sum
  return sum; }
```

Step 4:

```
// input: non-null array a
// output: sum of values in a
int sum(int[] a) {
  int sum = 0;
  // for each position k in a:
  for (int k=0; k < a.length; k++)
  { // add a[k] to sum:
    sum += a[k]; }
  return sum; }
```

# Bottom-up design

1. start designing **individual components** independently, or reuse those provided by libraries
2. **combine** the components to build **more complex** components
3. until the **overall functionality** is implemented

Step 1:

```
class Account
{ /* ... */ }
```

```
class Person
{ /* ... */ }
```

*// Collections Framework*

```
interface List<E>
{ /* ... */ }

class ArrayList<E>
{ /* ... */ }
```

Step 2:

```
class PersonalAccount
    extends Account
{
    Person owner;
    // ...
}
```

Step 3:

```
class Bank
{
    final float interest = 0.02;

    ArrayList<PersonalAccount> accounts;

    void depositInterest()
    {
        for (a : accounts)
            a.deposit(a.balance * interest);
    }
    // ...
}
```



# Top-down and bottom-up

Object-oriented programming languages support both top-down and bottom-up development:

- **top-down**: inheritance, abstract classes, interfaces
- **bottom-up**: encapsulation, polymorphism, assertions & exceptions

In practice programs are often developed with a **combination** of top-down and bottom-up, with a mixture that depends on the specific problem being targeted.

# Refactoring

Any realistic software development process goes through **trials and errors**: you hardly ever get the program right at the first attempt!

**Refactoring** is the activity of **changing** parts of the design or implementation of a program under development to **improve** and **adapt** it.

Examples of refactoring:

- introduce constants
- extract common implementations in two or more methods
- change the public interface of a class
- change the inheritance hierarchy
- ...

# Refactoring: method extraction example

Before refactoring:

```
void deposit(int amount)
{ if (amount > 0)
  balance += amount; }
```

```
void withdraw(int amount)
{ if (amount > 0)
  balance -= amount; }
```

After refactoring:

```
void deposit(int amount)
{ if (isPositive(amount))
  addAmount(amount); }
```

```
void withdraw(int amount)
{ if (isPositive(amount))
  addAmount(-amount); }
```

```
private boolean isPositive(int amount)
{ return amount > 0; }
```

```
private void addAmount(int amount)
{ balance += amount; }
```

# Test-driven development

In a previous class, we already emphasized the importance of writing **tests**. **Test-driven development** revolves around this principle:

- test **extensively**
- test **early**
- test **often**
- **refactor** implementation and tests accordingly

Test-drive development is often advocated with aggressive refactoring.

# Design principles

## Do not repeat yourself.

- use constants
- refactor methods
- use libraries

## Keep it small.

- refactor methods and classes
- use inheritance to gradually extend public interfaces
- use expressive constructs (e.g. exceptions) when appropriate

## Information hiding.

- use visibility modifiers appropriately
- public interfaces vs. private implementation details
- use abstract classes and interfaces

## Design for change.

- use genericity
- generalize (abstract) beyond the specific example
- but do not overdo it!

# Recursion in programming

**Recursion** is a **style of programming** where methods are defined in terms of themselves.

The **definition** of a **method**  $m$  is **recursive** if the implementation of  $m$  includes a call to  $m$  (directly or indirectly).

```
// compute  $x^y$ , for  $y \geq 0$   
int pow(int x, int y) {  
    if (y == 0)  
        return 1;  
    else  
        return x * pow(x, y - 1);  
}
```

↑  
recursive call

# Recursion in mathematics

**Recursion** is a **style of definition** where concepts are defined in terms of themselves.

The **definition** of a **concept** is **recursive** if it defines the concept in terms of an instance of the concept itself.

Definition of **natural numbers**:

- **0** is a natural number
- if  **$n$**  is a natural number then  **$n + 1$**  is a natural number.



**recursive/inductive definition**

# Recursion: from math to programming

Recursion in programming provides a natural way of implementing recursive definitions in mathematics.

**Factorial** of a nonnegative integer  $n$ :

$$n! \triangleq \underbrace{n \cdot (n-1) \cdot \dots \cdot 1}_{n \text{ terms}}$$



# Recursion: from math to programming

Recursion in programming provides a natural way of implementing recursive definitions in mathematics.

**Factorial** of a nonnegative integer  $n$ :

$$n! \triangleq n \cdot \underbrace{(n-1) \cdot \dots \cdot 1}_{n \text{ terms}} = n \cdot \underbrace{(n-1) \cdot \dots \cdot 1}_{n-1 \text{ terms}}$$

# Recursion: from math to programming

Recursion in programming provides a natural way of implementing recursive definitions in mathematics.

**Factorial** of a nonnegative integer  $n$ :

$$n! \triangleq \underbrace{n \cdot (n-1) \cdot \dots \cdot 1}_{n \text{ terms}} = n \cdot \underbrace{(n-1) \cdot \dots \cdot 1}_{n-1 \text{ terms}}$$

$$n! \triangleq \begin{cases} 1 & \text{if } 0 \leq n \leq 1 \leftarrow \text{base case} \\ n \cdot (n-1)! & \text{if } n > 1 \leftarrow \text{recursive/inductive case} \end{cases}$$

# Recursion: from math to programming

Recursion in programming provides a natural way of implementing recursive definitions in mathematics.

**Factorial** of a nonnegative integer  $n$ :

$$n! \triangleq \begin{cases} 1 & \text{if } 0 \leq n \leq 1 \leftarrow \text{base case} \\ n \cdot (n-1)! & \text{if } n > 1 \leftarrow \text{recursive/inductive case} \end{cases}$$

```
int factorial(int n) {  
    if (n <= 1)  
        return 1; // base case  
    else  
        return n * factorial(n - 1); // recursive case  
}
```

↑  
recursive call

# How does recursion work?

- Each recursive call runs an **independent instance** of the recursive method. (Independent means that it has its own private copy of actual arguments and local variables.)
- When a recursive instance terminates, execution resumes in the calling instance **after the recursive call**.

main  factorial(3)

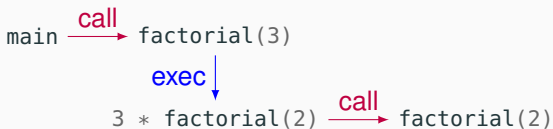
# How does recursion work?

- Each recursive call runs an **independent instance** of the recursive method. (Independent means that it has its own private copy of actual arguments and local variables.)
- When a recursive instance terminates, execution resumes in the calling instance **after the recursive call**.

main  $\xrightarrow{\text{call}}$  factorial(3)  
                   $\downarrow$  exec  
                  3 \* factorial(2)

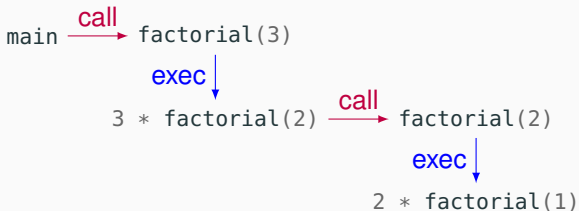
# How does recursion work?

- Each recursive call runs an **independent instance** of the recursive method. (Independent means that it has its own private copy of actual arguments and local variables.)
- When a recursive instance terminates, execution resumes in the calling instance **after the recursive call**.



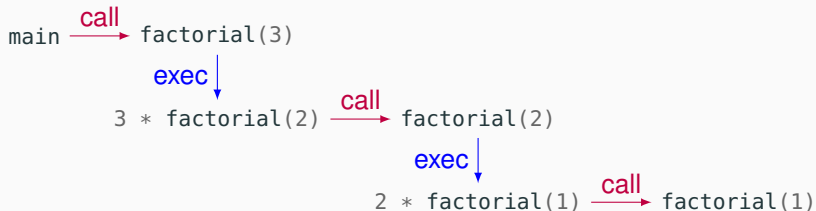
# How does recursion work?

- Each recursive call runs an **independent instance** of the recursive method. (Independent means that it has its own private copy of actual arguments and local variables.)
- When a recursive instance terminates, execution resumes in the calling instance **after the recursive call**.



# How does recursion work?

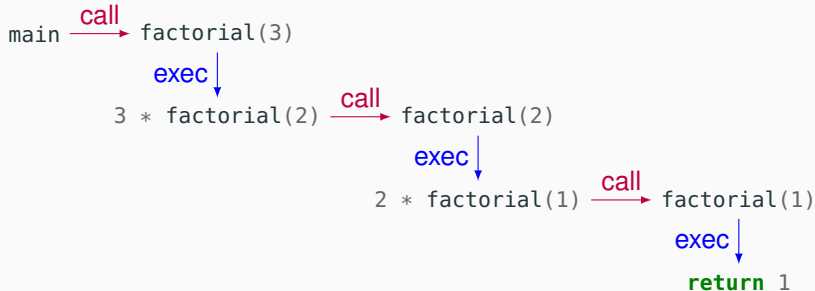
- Each recursive call runs an **independent instance** of the recursive method. (Independent means that it has its own private copy of actual arguments and local variables.)
- When a recursive instance terminates, execution resumes in the calling instance **after the recursive call**.





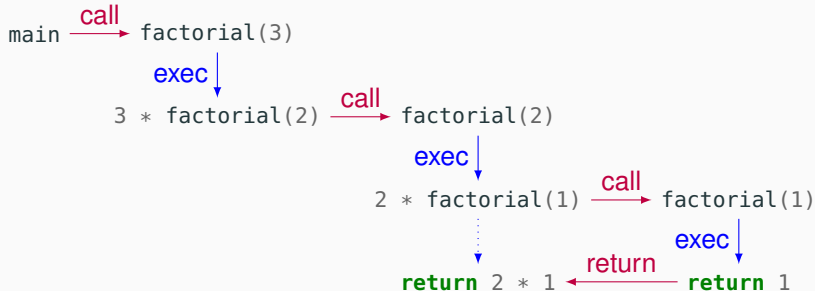
# How does recursion work?

- Each recursive call runs an **independent instance** of the recursive method. (Independent means that it has its own private copy of actual arguments and local variables.)
- When a recursive instance terminates, execution resumes in the calling instance **after the recursive call**.



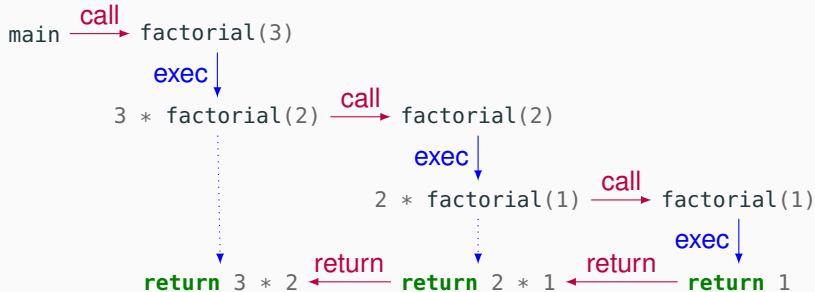
# How does recursion work?

- Each recursive call runs an **independent instance** of the recursive method. (Independent means that it has its own private copy of actual arguments and local variables.)
- When a recursive instance terminates, execution resumes in the calling instance **after the recursive call**.



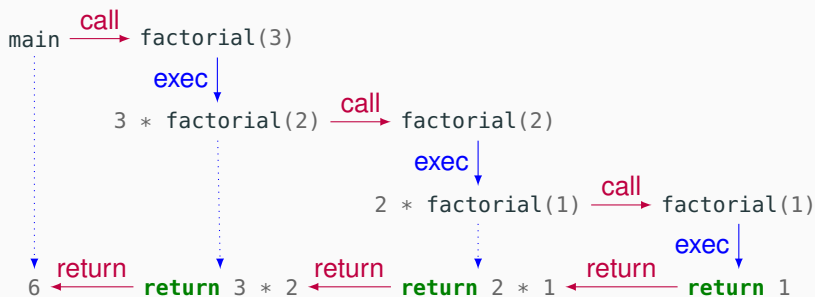
# How does recursion work?

- Each recursive call runs an **independent instance** of the recursive method. (Independent means that it has its own private copy of actual arguments and local variables.)
- When a recursive instance terminates, execution resumes in the calling instance **after the recursive call**.



# How does recursion work?

- Each recursive call runs an **independent instance** of the recursive method. (Independent means that it has its own private copy of actual arguments and local variables.)
- When a recursive instance terminates, execution resumes in the calling instance **after the recursive call**.



# Recursion as a design technique

**Recursion** as a programming technique is useful to design programs using the **divide and conquer** approach:

To solve a **problem instance**  $P$ , **split**  $P$  into problem instances  $P_1, \dots, P_n$  chosen such that:

1. Solving  $P_1, \dots, P_n$  is **simpler** than solving  $P$  directly
2. The solution to  $P$  is a **simple combination** of the solutions to  $P_1, \dots, P_n$

# The Tower of Hanoi

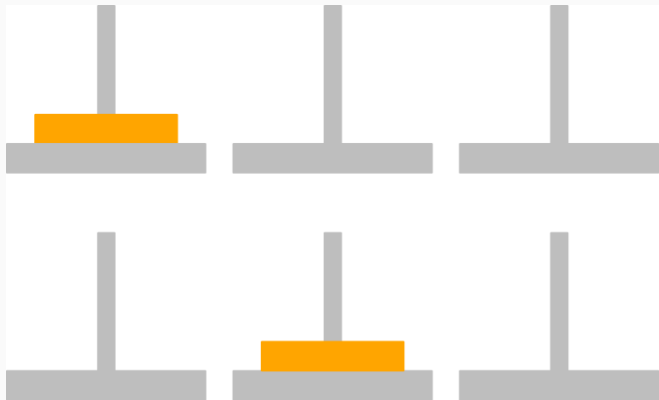
**Goal:** move sorted stack of disks to from the left to the middle peg



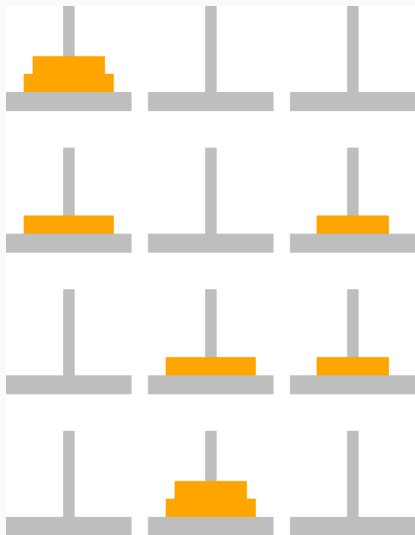
**Rules:**

1. Move one disk at a time
2. A move consists of taking the disk on top of one peg and placing it on top of another peg
3. Throughout the game, a larger disk can never be placed on top of a smaller disk

# The Tower of Hanoi: one disk



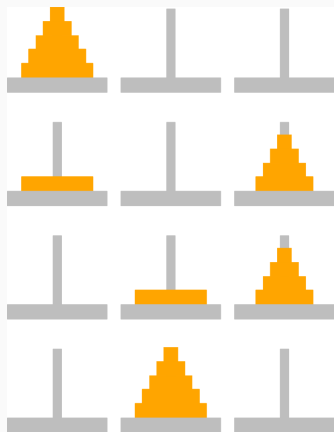
# The Tower of Hanoi: two disks





# The Tower of Hanoi: $n$ disks

1. **Recursively** move  $n - 1$  disks on a spare peg
2. Move remaining largest disk to destination peg
3. **Recursively** move  $n - 1$  disks from spare peg to destination peg



# The Tower of Hanoi: $n$ disks

```
// move 'n' top disks
// from 'source' peg to 'destination' peg via 'spare' peg
public void move(int n,
    PegPosition source, PegPosition destination, PegPosition spare) {
    if (n == 1)
        // base case
        move(source, destination);
    else {
        // recursively move n - 1 to spare
        move(n - 1, source, spare, destination);
        // move largest disk to destination
        move(1, source, destination, spare);
        // recursively move n - 1 to destination
        move(n - 1, spare, destination, source);
    }
}
```

# The original Tower of Hanoi

*In the great temple of Benares, under the dome that marks the center of the world, three diamond needles, a foot and a half high, stand on a copper base. God on creation strung 64 plates of pure gold on one of the needles, the largest plate at the bottom and the others ever smaller on top of each other. That is the tower of Brahma. The monks must continuously move the plates until they will be set in the same configuration on another needle. The rule of Brahma is simple: only one plate at a time, and never a larger plate on a smaller one. When they reach that goal, the world will crumble into dust and disappear.*

Édouard Lucas, *Récréations mathématiques*, 1883.

## Got time for 64 disks?

- For  $n$  disks, the recursive solution generated by our program enumerates  $2^n - 1$  moves
- We could show that this is the minimum number of moves to solve the problem following the rules
- If one move takes 1 millisecond,  $2^{64} - 1$  milliseconds is about 580 million years
- For comparison: dinosaurs got extinct about 65 million years ago, humans are about 2.5 million years old

**Bottom line:** recursion is a powerful abstraction tool, which can be very effective at expressing the solutions to complex problems in a simple way.

# Recursion vs. Iteration

In principle, anything that can be done using recursion can be done using iteration (loops) as well, and vice versa.

**Recursive** factorial:

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

**Iterative** factorial:

```
int factorial(int n) {  
    int factorial = 1;  
    for (int k = n; k > 1; k--)  
        factorial *= k;  
    return factorial;  
}
```

However, when the **divide and conquer approach** is naturally applicable, recursion often leads to more readable and clearer programs.