Software design

Lecture 14 of TDA 540
Object-Oriented Programming

Jesper Cockx

Fall 2018

Chalmers University of Technology — Gothenburg University

Last week: recap

- Pre- and post-conditions (`@requires` and `@ensures`)
- Invariants (`@invariant`)
- The `Exception` hierarchy
- Software verification
  - Code inspection
  - Testing (unit testing and integration testing)
  - Formal verification

This week

Recap of classes and inheritance

How to design a software project?

+ some other topics:

- Enumerated types (enums)
- Recursion

Classes and inheritance recap

Classes

A class describes a collection of objects with the same public interface and representation of their internal state:

```
class Glass {  // user-defined class
  private double volume; // state
  void addWater(double amount) { // operation
    volume = volume + amount;
  }
}
```

What is in a class?

A class defines:

- How objects of that class are represented in computer memory (the attributes)
- What methods are available on objects of the class (the methods)
- How to create new objects of that class (the constructors)

Each class also defines a new type.

Objects vs. classes

A class is a static entity:
it refers to a piece of code.

An object is a dynamic entity:
it is only created when the program executes.

An object is an instance of a certain class.

Attributes

An attribute (also called an instance variable or a field) represents part of an object's state.

- Each object has its own copy of the attributes
- Attributes can be of *primitive* or *reference* type
- `final` attributes cannot change once the object has been created

Attributes are declared in the class body:

```
class Glass {
  double volume;            // current contents in ml
  final double maxVolume; // maximum volume
  // ...
}
```

Methods

A method (also called an instance method or a member function) represents an operation that can be executed on objects of the class.

A method can modify the object state and/or return information about the object state.

Methods are declared in the class body:

```
class Glass {
  // ...
  public void addWater(double x) { volume += x; }
  // ...
}
```

Constructors

A constructor is a special method that creates a new object of the class.

- Constructors have the *same name* as the class
- A constructor has no return type (not even `void`!)
- It should give an initial value to all attributes (uninitialized attributes get default value)

Constructors are declared in the class body:

```
public Glass(double size) {
  volume = 0;
  maxVolume = size;
}
```

Using a constructor

To use a constructor, we use the `new` keyword:

```
Glass glass = new Glass(100);
```

The result of `new Glass(100)` is a reference to the new object.

Visibility of members

The visibility of a class member (attribute or method) determines where in a program we can refer to that member:

- `private`: `x` is only visible in the enclosing class
- `protected`: `x` is visible within the same package
- `public`: `x` is visible everywhere in the program

Encapsulation / information hiding

An important role of classes is to hide information from the rest of the program.

The client only has to know the public methods and constructors = the API (*Application Programming Interface*).

The (private) state can change while the rest of the program stays the same.

$\Rightarrow$ Abstraction!

Information hiding: example

Public interface:

```
class Glass {

  /* attributes invisible */

  Glass(double size) {
    /* body invisible */
  }

  double getVolume() {
    /* body invisible */
  }

  void addWater(double amount) {
    /* body invisible */
  }
}
```

Client code:

```
Glass glass;
glass = new Glass(500);

// we don't have to worry how addWater
// and getVolume are implemented

glass.addWater(300);
if (glass.getVolume() > 100) {
  System.out.println(
    "Can drink water!");
}
```

Static members

A static member belongs to the whole class, not an individual object.

- A `static` attribute is shared among all object of the class
- A `static` method can only use static attributes and other static methods
- A constructor can never be `static`

Static members are accessed using the class name:

```
class CoinPurse {
  static int[] COIN_SIZES =
    { 1 , 2 , 5 , 10 };
  // ...
}
```

```
int[] coins =
  CoinPurse.COIN_SIZES;
for (i : coins) {
  // ...
}
```

Static or instance?

Rule of thumb:

Does it make sense to call (method) or access (attribute) `m` independent of specific objects of its class?

1. Yes: you probably need a static member
2. No: you should go with an instance member

In most cases, the answer should be no!

Inheritance

Inheritance = relation between a general class (the superclass) and a more specific one (the subclass)

Example: a car is a vehicle
$\Rightarrow$ `Car` is a subclass of `Vehicle`

In Java:

```
class Vehicle { ... }
class Car extends Vehicle { ... }
```

All members (attributes and methods) of `Vehicle` are automatically also members of `Car`.

Inheritance and types

Every class `C` corresponds to a `type`.

If `C` is a subclass of another class `B`, then `C` is a subtype of `B`: an object of type `C` can be used as an object of type `B`.

```
class Vehicle { ... }
class Car extends Vehicle { ... }
```

A car is a vehicle!

Liskov's substitution principle

A program that expects an object of a superclass should also work when given an object of a subclass instead.

e.g. a program that works with a `Vehicle` should also work for a `Car`.

- Subclass can only add new attributes and methods, never remove them
- Return types of methods can only become more specific
- Argument types can only become more general

Overriding

Overriding = redefine a method from the superclass

```
class Account {
  int balance;

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

```
class NoOverdrawnAccount
  extends Account {
   // redefinition of withdraw
   @Override
   void withdraw(int amount) {
     if (amount <= balance)
       balance -= amount;
   }
}
```

super: referencing the superclass

The keyword super denotes a reference to the current object as an instance of the superclass.

```java
class Account {
  int balance;
  void withdraw(int amount) {
    balance -= amount;
  }
}
```

```java
class NoOverdrawnAccount
  extends Account {
   @Override
   void withdraw(int amount) {
     if (amount <= balance)
       // call withdraw
       // from Account
       super.withdraw(amount);
     }
}
```

When to create a new subclass?

Not every kind of object needs its own subclass:

- If objects vary in their behaviour
  $\Rightarrow$ different subclasses
- If objects only vary in some values
  $\Rightarrow$ one class is enough

Interfaces

An interface is a list of abstract operations describing the *public interface* (API) of a class.

```
public interface IGlass {
  double getCurrentVolume();
  void addWater(double amount);
  void removeWater(double amount);
}
```

All methods are automatically `public` and `abstract`.

No attributes[1] or constructors.

[1]Except for `static final` attributes

Interfaces and classes

A class can implement one or more interfaces:

- it must override all methods of the interfaces
- it can also introduce other members (private or public) without restrictions

```
class Glass implements IGlass {
  private double volume;
  int getCurrentVolume() {
    return volume;
  }
  // ... other attributes and methods ...
}
```

Interfaces and inheritance

An interface also can inherit from one or more interfaces (but not from classes), by providing additional public methods (or constants).

```
interface IAccount {
  void deposit(long amount);
  // ...
}

interface ISavingAccount
  extends IAccount {
    static final double INTEREST = 0.001;
    void addInterest();
}
```

Polymorphism

Polymorphism: we can switch between different concrete implementations of an interface without changing anything else in the program!

```
interface List<E> {
  E get(int index);
  void add(int index, E e);
  int size();
}
```

```
List<String> l;
l = // choose any List implementation
l.add(0, "hej");
l.add(1, " då");
if (l.size() >= 2)
  String s = l.get(0) + l.get(1);
  System.out.println(s);
```

Polymorphism

Advantages of using polymorphism:

Decoupling You can think about (and use!) an interface without worrying about the implementation.

Cohesion If you know how to use one implementation of `List`, you know how to use all of them.

Component-based design You can switch out one part of the code for another without changing the overall behaviour.

Enumerated types

Enumerated types (enums)

An enumerated types (enum) is a type with a finite number of values.

- yes / no / don't know
- days of the week: Monday / Tuesday / .../ Sunday
- age ranges: infant / adolescent / adult / senior

Enum example

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

Type `Answer` has 3 values:
`Answer.YES`, `Answer.NO`, and `Answer.DONT_KNOW`.

Software design

Software design

Knowing how to program is only the first step towards writing good programs.

good

$\simeq$

correct, readable, modifiable, efficient, ...

There are many design principles and techniques that help to write better programs.

Software design

Software design step by step:

- Gather requirements
- Determine classes and their responsibilities
- Define a public interface for each class
- Determine the (private) instance variables and constructors
- Implement methods and constructors
- Test the program

Design principles

- Don’t repeat yourself
- Keep it simple
- Hide implementation details
- Design for change

Design principle 1: Don’t repeat yourself

- use constants with expressive names
- write new methods that abstract common functionality
- use libraries whenever possible instead of implementing it yourself

Design principle 2: Keep it simple

- when a method or class becomes too big, split it up in multiple parts
- use inheritance to hide details and keep high-level code understandable
- use expressive constructs (e.g. exceptions) only when they simplify the program

Design principle 3: Hide implementation details

- hide details with visibility modifiers (`private` and `protected`) whenever possible
- define interfaces that define the outside-facing behaviour of a class
- use interfaces rather than concrete classes for the types of arguments and variables (e.g. use `List<String> x` instead of `ArrayList<String> x`)

Design principle 4: Design for change

- write generic code: don't commit to a specific class if you don't have to (e.g. instead of `class CoinPurse where ...` define `class Purse<A> where ...`)
- abstract beyond the specific example
- but don't overdo it!

Top-down design vs bottom-up design

Top-down design: Start with abstract high-level interfaces and refine the components iteratively by adding details until everything is concrete.

Bottom-up design: Design/reuse individual concrete components and combine them to build more complex components until the overall functionality is implemented.

Top-down design example

Step 1: define high-level abstract interface

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

Step 2: refine some aspects of the interface by adding details ...

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

Step 3: ...until everything is concrete

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

Bottom-up design example

Step 1: design/reuse individual concrete components

```
class Account { /* ... */ }
class Person { /* ... */ }
class ArrayList<E> { /* ... */ } // taken from the Collections framework
```

Step 2: combine them to build more complex components ...

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

Step 3: ...until the overall functionality is implemented

```
class Bank {
  final float interest = 0.02;
  ArrayList<PersonalAccount> accounts;
  void depositInterest() {
    for (a : accounts) { a.deposit(a.balance * interest); }
  }
  // ...
}
```

Top-down vs bottom-up design

Java supports both top-down and bottom-up design:

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

Which one is best depends on the specific problem.

Designing individual classes

Discovering classes and methods

If you're unsure what classes to create, look at the problem description:

- Nouns correspond to classes
- Verbs correspond to methods

Don't overdo it: not every noun needs to be its own class.

Cohesion

A class is cohesive if all its public methods are closely related to the concept represented by the class.

If a class contains many unrelated methods (= low cohesion), it's often possible to split it into multiple classes.

Cohesion example

```java
public class CoinPurse { // ...
  public void add1KrCoins(int amount) { ... }
  public void add2KrCoins(int amount) { ... }
  public void add5KrCoins(int amount) { ... }
  public void add10KrCoins(int amount) { ... }
}
```

$$\Downarrow$$

```java
public class Coin { // ...
  public int getValue() { ... }
}

public class CoinPurse { // ...
  public void addCoins(Coin coin, int amount) { ... }
}
```

Relationships between classes

- Dependency: `A` *knows about* `B`
  `A` **uses** objects of type `B` as method argument, return value, or local variable
  e.g. a `Car` can transport a `Person`
- Aggregation: `A` *has a* `B`
  `A` **has** one or more objects of type `B` as instance variable(s)
  e.g. a `Car` has four `Tire`s
- Inheritance: `A` *is a* `B`
  `A` **is a** subclass of `B`
  e.g. a `Car` is a `Vehicle`

Coupling

Coupling = how much classes depend on each other

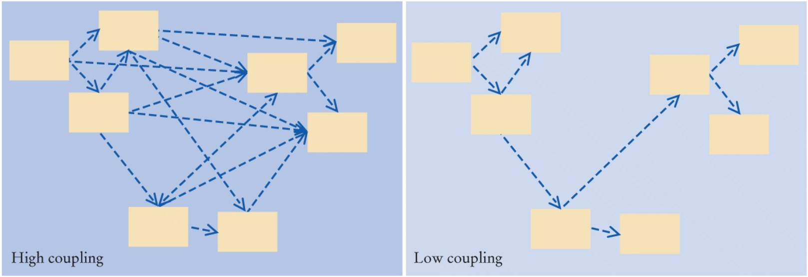


Figure 12.4

A small program with high coupling is much harder to change than a large program with low coupling.

$\Rightarrow$ Avoid unneccessary coupling!

Single- vs bi-directional relations

Single-directional relation: `A` knows about `B`

Bi-directional relation:
`A` knows about `B` and `B` knows about `A`

- An `Account` knows which `Person` it belongs to:

```
class Account {
  Person owner; /* ... */
}
```

- A `Person` has a list of all its `Accounts`:

```
class Person {
  List<Account> accounts; /* ... */
}
```

Keeping bidirectional relations consistent

It's important to keep bidirectional relations consistent:

- If `account.getOwner() == john` then `account` must be in `john.getAccounts()`
- If `account` is in `john.getAccounts()` then `account.getOwner()` must be `john`

Tip: give *one* class the responsibility for managing the relation.

```java
class Person {
  private List<Account> accounts;
  public Person() { accounts = new ArrayList<Account>(); }
  protected addAccount(Account a) {
    assert a.getOwner() == this; // check that this person
    accounts.add(a);             //   is indeed the owner
  }
}

class Account {
  private Person owner;
  public Person getOwner() { return owner; }
  public Account(Person owner) {
    this.owner = owner;
    owner.addAccount(this); // make sure owner knows
  }                         //   about this account
}
```

Refactoring

Refactoring

Software development always involves trial and error: you hardly ever get the program right at the first try!

Refactoring = changing the design or implementation without changing the overall functionality:

- introduce constants
- move some code to a separate method
- make a private method public or vice versa
- replace a concrete type by a more abstract type
- add a new superclass or interface to a class
- ...

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; }
```

Refactoring and testing

The goal of refactoring is to change the implementation without changing the overall functionality.

Use tests to ensure you don't change the functionality by accident.

To refactor effectively, you need good tests!

Recursion

Recursion in programming

A recursive method is a method that calls itself on different arguments:

```
// compute 2^x, for x ≥ 0
int pow2(int x) {
  if (x == 0)
    return 1;
  else
    return 2 * pow2(x - 1);
}
```

`pow2(x - 1)` is a recursive call of the method `pow2` to itself.

Recursion in programming

To be terminating, a recursive method must:

- Call itself only on smaller arguments

  e.g. `pow2(x)` calls itself on `pow(x-1)`

- Include a base case where it doesn't call itself at all

  e.g. `pow(0)` does not call itself

Example: factorial

Factorial of a nonnegative integer n:

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

Example: factorial

Factorial of a nonnegative integer n:

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

Example: factorial

Factorial of a nonnegative integer n:

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

$$n! \triangleq \begin{cases} 1 & \text{if } 0 \leq n \leq 1 \\ n \cdot (n-1)! & \text{if } n > 1 \end{cases}$$

← base case

← recursive/inductive case

Example: factorial

Factorial of a nonnegative integer n:

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

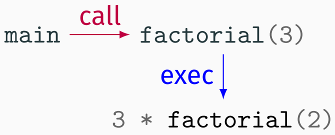
```
int factorial(int n) {
  if (n <= 1)
    return 1;                        // base case
  else
    return n * factorial(n - 1); // recursive case
}
```

↑ recursive call

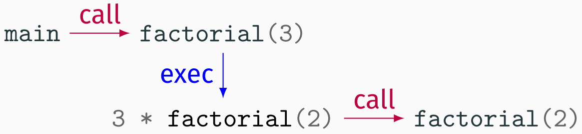
How does recursion work?

`main` —call→ `factorial(3)`

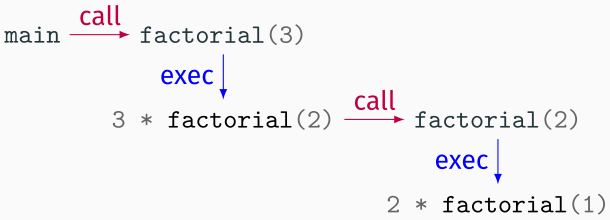
How does recursion work?



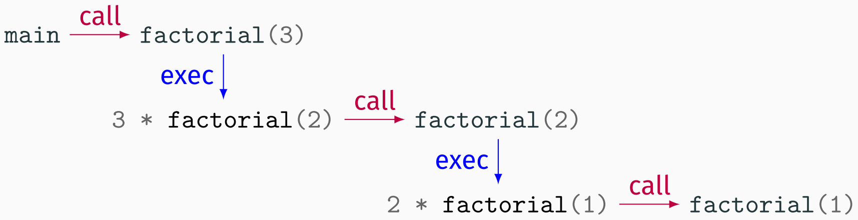
How does recursion work?



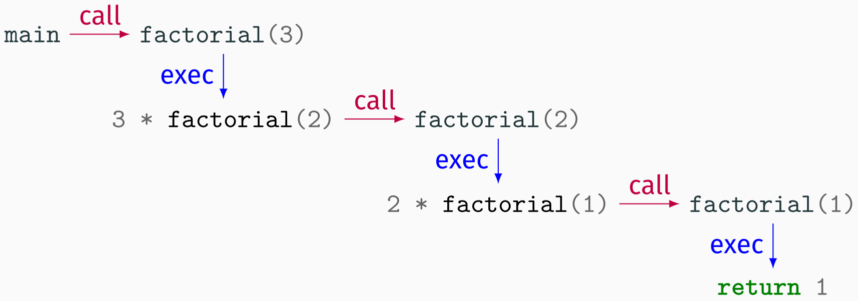
How does recursion work?



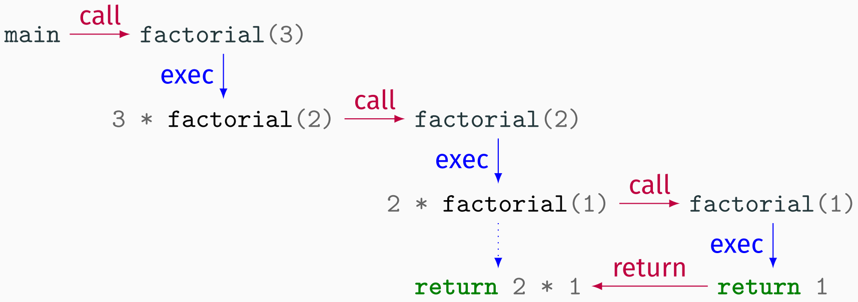
How does recursion work?



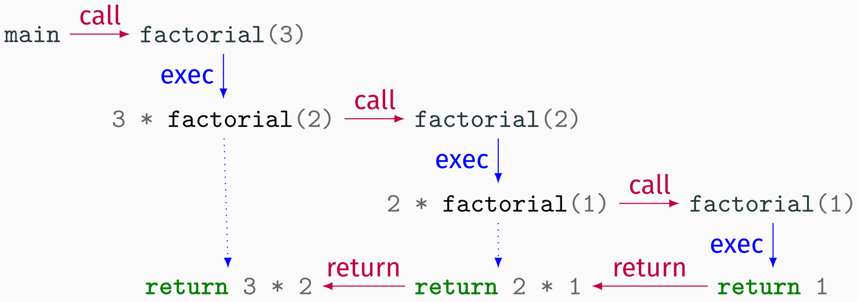
How does recursion work?



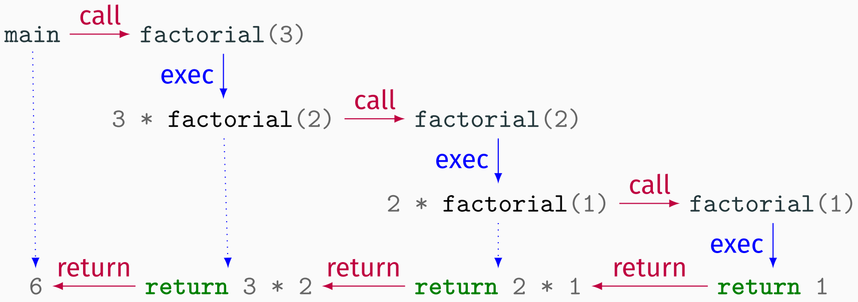
How does recursion work?



How does recursion work?



How does recursion work?



Thinking recursively

Recursion is a way to apply the divide and conquer approach.

When solving a problem, ask yourself:

1. What is the simplest possible form of this problem?
2. How can I reduce the problem to a simpler problem of the same kind?

The Tower of Hanoi

Goal: move all disks from the left peg to the middle peg

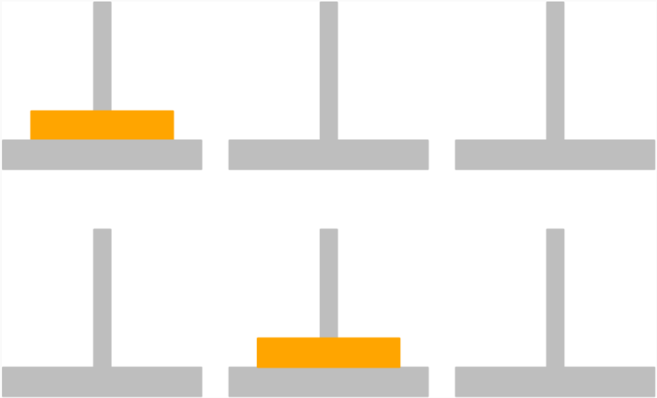
Rules:

1. One move = take the disk on top of one peg and place it on top of another peg
2. You can move only one disk at a time
3. A larger disk can never be placed on top of a smaller disk

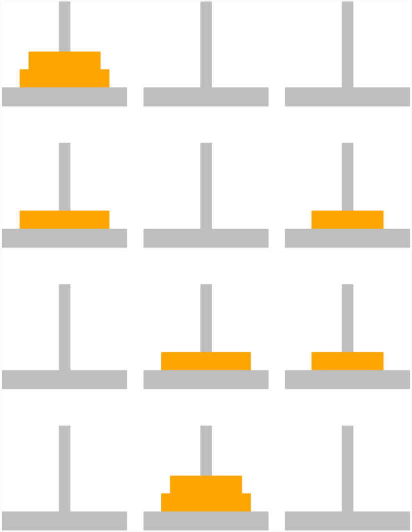
The original Tower of Hanoi

> *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.

The Tower of Hanoi: one disk

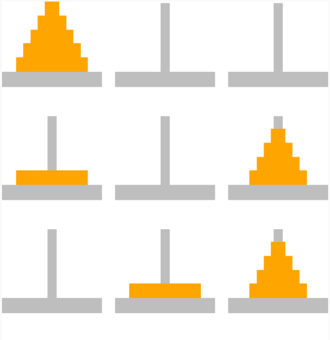


The Tower of Hanoi: two disks



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 moveDisks(int n,
  Peg source, Peg destination, Peg spare) {
  if (n == 1)
    // base case
    moveOneDisk(source, destination);
  else {
    // recursively move n - 1 to spare
    moveDisks(n - 1, source, spare, destination);
    // move largest disk to destination
    moveOneDisk(source, destination);
    // recursively move n - 1 to destination
    moveDisks(n - 1, spare, destination, source);
  }
}
```

Got time for 64 disks?

- For n disks, solving the puzzle takes $2^n - 1$ moves
- 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.

What's next?

This was the final lecture!
Thank you for your attention.

To do:

- Finish the final two labs
- Start preparing for the exam
  - Take a look at the study guide on the website
  - You can ask any questions to the lab assistants, on the discussion group, or by sending me an email.

Good luck!