# Java for Advanced Programmers

## Object-Oriented Programming Basics

Bernd Kiefer
Jörg Steffen

November 9, 2022

# The Idea behind OO Programming

## Objects: The central abstraction

It encapsulates:

- Object properties
- Object behaviours

# The Idea behind OO Programming

## Objects: The central abstraction

It encapsulates:

- Object properties: data fields
- Object behaviours: methods

# The Idea behind OO Programming

## Objects: The central abstraction

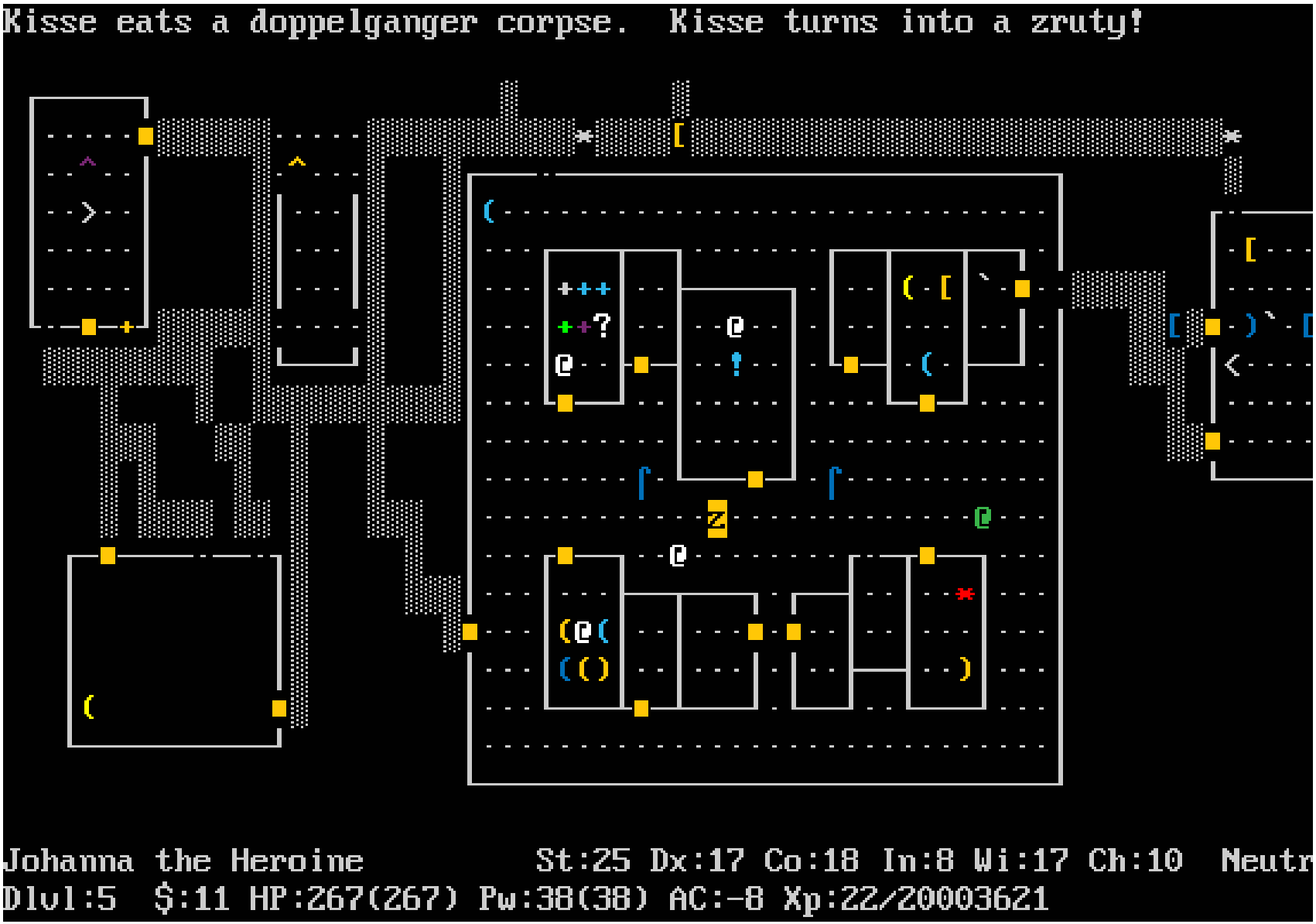
It encapsulates:

- Object properties: data fields
- Object behaviours: methods
- An object is a chunk of memory in a running system

## Classes: Blueprints

- A class defines the shape and behavior of objects
- In Java, every object is an instance of some class
- The class it belongs to is the datatype of an object

# Our running example



# Our Hero Class

```
class Hero {
  Role r;
  int x, y;

  /** Change my position by the given vector
   *  of length <= sqrt(2)
   * @param deltaX the x component of the vector
   * @param deltaY the y component of the vector
   * @return true if this movement is possible
   */
  boolean moveTo(int deltaX, int deltaY) {
    ...
  }

  ...
}
```

# Creating Objects: Constructors

- Special methods: no return type, name equal to class name
- If none is specified, there is always the empty constructor: `Hero()`
- The default constructor sets all fields to default values
- You can create your own constructors (as many as you want)

```
Hero() { x = -1; y = -1; }

Hero(Role myRole, int startX, int startY) {
  r = myRole; x = startX; y = startY;
}
```

- If you specify a non-empty constructor, the empty constructor is not created automatically

# Creating Objects: `new`

- Objects are independent blocks of memory
  - for every field
  - plus overhead for class information
- Create a new object (reserve memory) with the `new` keyword:

```
Hero r1 = new Hero();
Hero r2 = new Hero(new Rogue(), 0, 0);
```

- Object lifetime:
  - Alife as long as there is some variable or life object pointing to it
  - Otherwise, their memory is freed for reuse
  - Beware: creating and freeing many objects comes at a cost!

# Specifying Methods

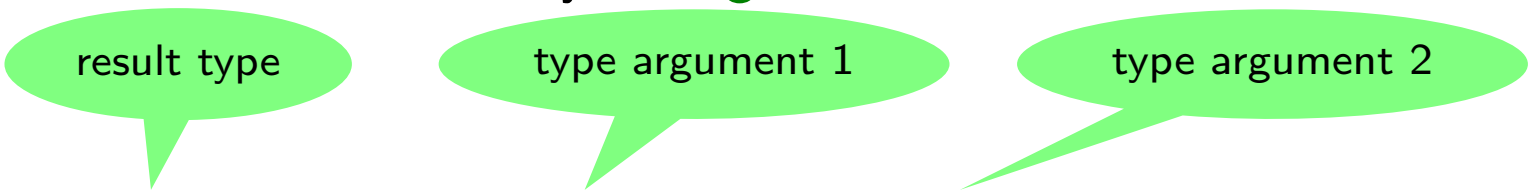
- A method is defined by its signature

```
boolean moveTo(int deltaX, int deltaY)
```

- `void` methods may be exited using simply `return;`

# Specifying Methods

- A method is defined by its signature




- Methods returning no value get `void` as return type
- Non-`void` method must have at least one `return` statement followed by an expression of the right type:

```
return Math.abs(deltaX) + Math.abs(deltaY) <= 2;
```

- `void` methods may be exited using simply `return;`

# Method Polymorphism

## Multiple methods with the same name, but different signature

```
boolean moveTo(int deltaX, int deltaY)
boolean moveTo(double deltaX, double deltaY)
```

# Method Polymorphism

## Multiple methods with the same name, but different signature

```
boolean moveTo(int deltaX, int deltaY)
boolean moveTo(double deltaX, double deltaY)
```

## Restriction: Compiler can decide which to call at compile time

- No methods differing only in return type
- No methods where, e.g., one argument type is a subclass:

```
void encounter(Hero h)   ...

void encounter(Rogue h)   ...
```

# Guidelines for Classes

## Intuitive

- Size of methods (~ max 1 page)
- Number of fields
- Little/No Duplication of code
- Hide internals: fields and implementation

# Guidelines for Classes

## Intuitive

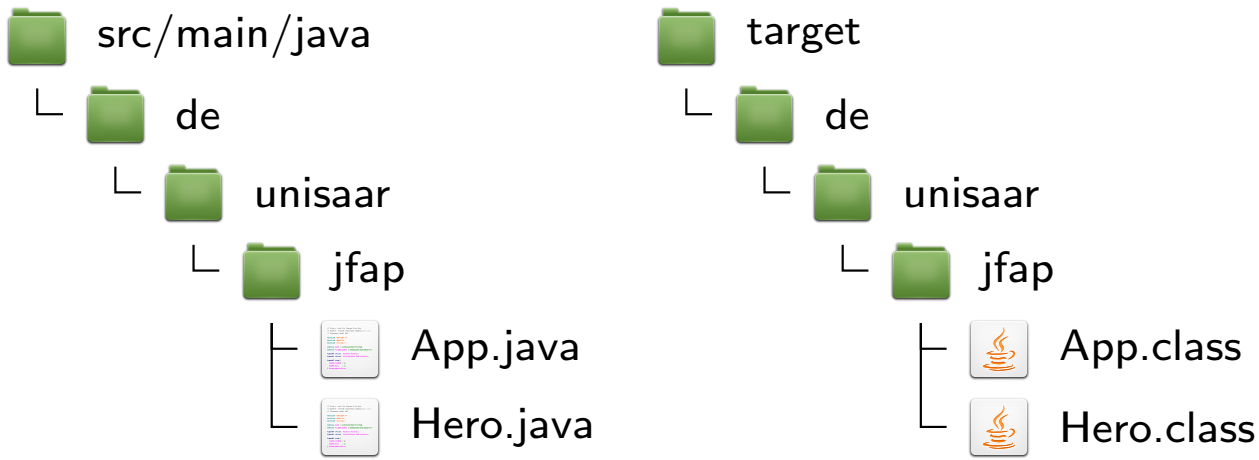
- Size of methods (~ max 1 page)
- Number of fields
- Little/No Duplication of code
- Hide internals: fields and implementation

## Textbook

- Single Responsibility Principle
- Design by Contract / Implementation by Design

# Project setup

- Packages are used to structure projects
- Analogous to folder structure on file systems
- The java compiler and class loader will search sub-folders accordingly
- Our default package structure:




- First line in front of all other code:
  `package de.unisaar.jfap;`

# Test-driven development

- First write tests that check the designed functionality
- Then: implement the class until all tests succeed
- We will do the non-fundamentalist version

# Test-driven development

- First write tests that check the designed functionality
- Then: implement the class until all tests succeed
- We will do the non-fundamentalist version

## Unit Tests

- Purpose: test isolated, atomic aspects of a class
- Tests must be independent, i.e., running test B must work without test A running first

# Test-driven development

- First write tests that check the designed functionality
- Then: implement the class until all tests succeed
- We will do the non-fundamentalist version

## Unit Tests

- Purpose: test isolated, atomic aspects of a class
- Tests must be independent, i.e., running test B must work without test A running first
- Benefit: Sleep better if you have to change innards of a class that is used in a zillion different places
- Limitations: Hard to test complex situations/classes
- Test coverage hints on how much of your code is tested

# Test Class Example

```
import static org.junit.Assert.*;

import org.junit.Test;

class TestHero {

  /** Test if the hero moves according to the specs */
  @Test
  public void testMoveTo() {
    Hero h = new Hero(0,0);
    h.moveTo(1,1);
    assertEquals(1, h.x);
    assertEquals(1, h.y);
  }

  ...
}
```