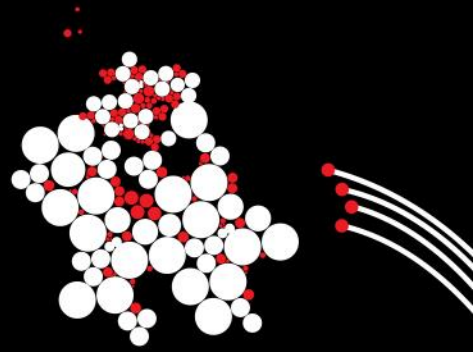
UNIVERSITY OF TWENTE.

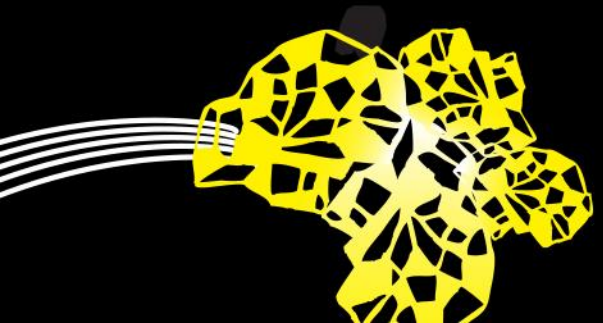


P2.1: CLASSES AND OBJECTS

TOM VAN DIJK

201700117-1B MODULE 2: SOFTWARE SYSTEMS

18 NOVEMBER 2019



PROGRAMMING LINE OVERVIEW

Week 1 Values and variables Control flow	**Week 2** Classes and objects Testing	**Week 3** Interfaces and Inheritance Subtyping Security 1
Week 4 Arrays and Lists List implementations Collections	**Week 5** Stream I/O and MVC Exceptions Security 2	**Week 6** Concurrency Project kick-off IDE Tips & Tricks
Week 7 Basic Networking Networking and Multithreading GUIs	**Week 8/9** Advanced Java facilities Test	**Week 10** Project Test resit

CONTENTS

- Object-oriented programming
  - The structure and abstractions to help you think
  - Concepts: classes, encapsulation, public vs private
- How to design your world of objects
- Documenting your code

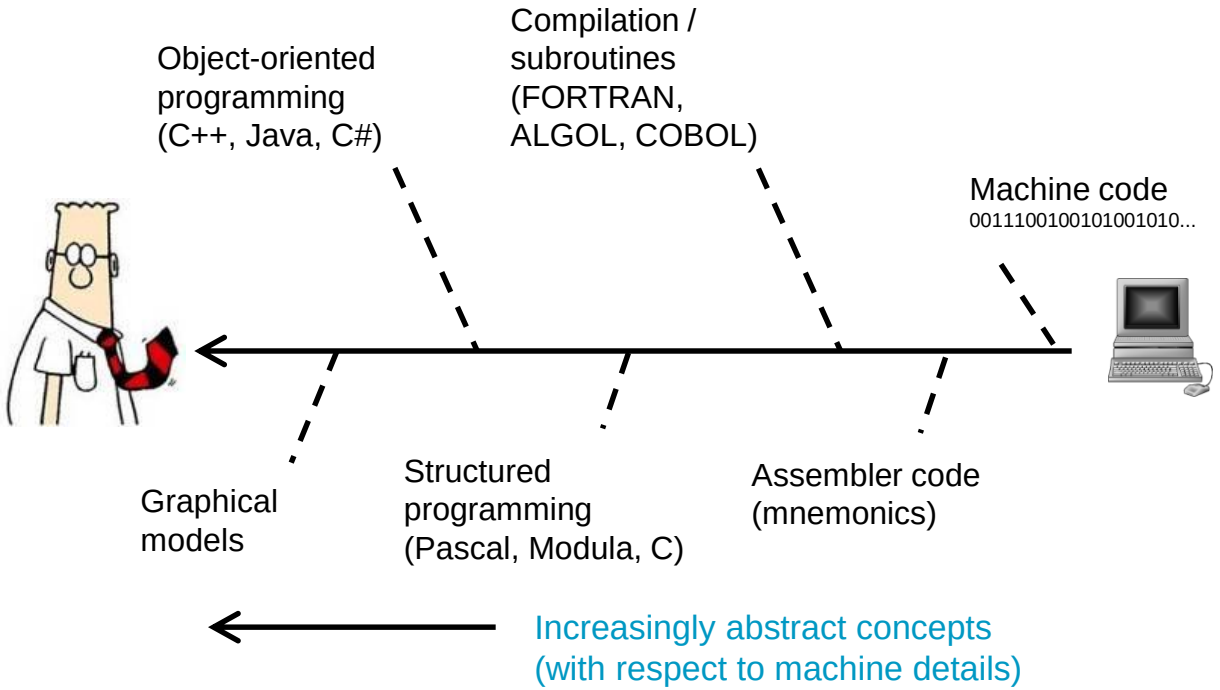
ESSENCE OF PROGRAMMING

- Give instructions to the processor to make it do what we “expect”
- Processors understand binary data (rows of 0s and 1s)
- People understand concepts (ideas)

ESSENCE OF PROGRAMMING

- There is a gap between concepts and binary data!
- Programming languages bridge this gap
- Structure and abstractions allow us to **reason** about our software
  - Is the software correct? Is it efficient?
  - What is/was the intent/design of the programmer?
  - Separation of concerns (within the team, within the software)

PROGRAMMING TIMELINE



OBJECT-ORIENTED PROGRAMMING

Goals

- Add structure to software
- Separation of concerns

  a design principle for separating a computer program into distinct sections such that each section addresses a separate concern (Wikipedia)
- Don’t repeat yourself (DRY)

OBJECT-ORIENTED PROGRAMMING

Object-orientation

- A computer program as a collection of objects that interact
- Each object is an instance of a class
  - … every specific chair is an instance of the idea/class “chair” …
- Objects contain data (fields) and code (methods)
  - Every instance of the class has these fields but with values
  - The current values of the fields = the state of an object
  - Example: the length of a chair, the number of guests of a room

OBJECT-ORIENTED PROGRAMMING

Classes have fields

- For example: a Box has fields label, length, height, width
- Fields have a type
  - Primitive (int, float, boolean, char)
  - Reference (to an object)

```
class Box
{
    String label;
    int length;
    int height;
    int width;
}
```

OBJECT-ORIENTED PROGRAMMING

Classes have fields

- For example: a Box has fields label, length, height, width
- Fields have a type (primitive or reference)
- Perspective: classes “group” variables together that belong together

OBJECT-ORIENTED PROGRAMMING

Classes have methods

- Getters and setters
  - Manipulate the state of the object (values in the fields)
  - Example: `g.getName()` to get the name of the Guest `g`
- Complex operations
  - Example: `h.getCurrentGuestCount()`
  - Example: `h.checkin(Guest g, Room r)`

OBJECT-ORIENTED PROGRAMMING

Classes have methods

- Getters and setters
- Complex operations
- Constructors (with parameters, but no return value)
  - Creates a new object: a **new** instance of the class

    Example: `House h = new House("my little house");`
  - Java creates a default constructor!
- Destructors (not needed in Java: Java has garbage collection)

OBJECT-ORIENTED PROGRAMMING

Some terminology

- Classes are sometimes called composite types
- Fields are also called attributes or properties
- Methods are declared and invoked
- The parameters (declaration) are also called arguments (invocation)
- Methods have a return type, or “return type” void
  - Class methods that return a value are also called functions or queries
  - Void methods are also called commands

OBJECT-ORIENTED PROGRAMMING

Other things classes have

- Classes live in a package
  - Convenient for the programmer (more structure!)
  - Example: `java.util.List`
  - Example: `nl.utwente.tcs.myfirstprogram.HelloWorld`
- Classes can have subclasses and nested classes
  - Topic of next week!
  - Subclasses inherit the members of their superclass

OBJECT-ORIENTED PROGRAMMING

Fields and methods have modifiers

- Fields and methods have an access modifier

Modifier	Same class?	Subclass?	Package?	Rest of the world?
public	Yes	Yes	Yes	Yes
private	Yes			
protected	Yes	Yes		
(none)	Yes	Yes	Yes	

OBJECT-ORIENTED PROGRAMMING

Fields and methods have modifiers

- Fields can be static
  - a **single** field that lives in the class
  - shared by all instances
- Methods can be static
  - may only access static fields

OBJECT-ORIENTED PROGRAMMING

Fields and methods have modifiers

- Fields can be final
  - may not be modified (after object construction)
- Methods can be final too
  - subclasses may not override (replace) the method

OBJECT-ORIENTED PROGRAMMING

Classes in memory

- Classes are reference types
- A variable is stored in memory
  - A **int** or a **float** has 32 bytes
  - A **long** or a **double** has 64 bytes
- Primitive types: store the **value** of the variable
- Reference types: store a pointer to the instance (or null)
- Two different variables may point to the same object (aliasing)

EXAMPLE

```
class Room {
    String building;
    int number;
}
```

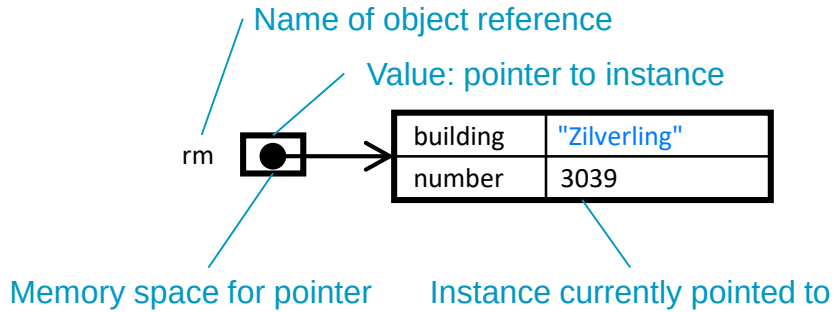
File “Room.java”

```
package your.package.name;
public class Room {
    private String building;
    private int number;

    // methods and constructors
}
```

In Java

```
...
Room rm;
```

In memory



PROGRAM DESIGN

- A program must be designed before code is implemented
  - Start with system specification
  - Nouns indicate the relevant concepts
  - Concepts are turned into classes

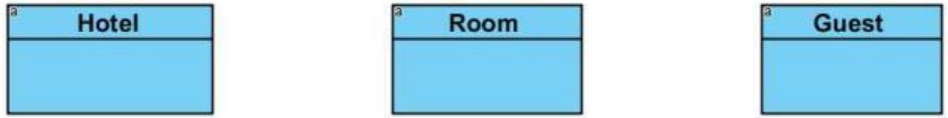
The sooner you start to code, the longer the program will take.
- Roy Carlson

PROGRAM DESIGN

- A program must be designed before code is implemented
- Example: Hotel Information System
- Initial requirements:
  - System to record guests of a hotel, including their name and in which room they stay
- What concepts do we need?

PROGRAM DESIGN: CONCEPTS ARE CLASSES

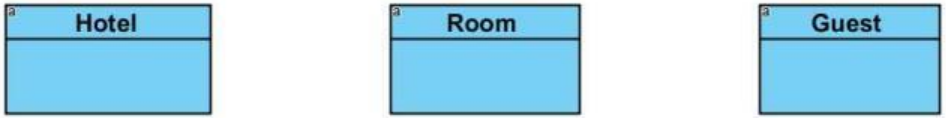
- First design step: class diagram



- Ultimately, program manipulates objects
  - Objects are values of the classes above
  - Objects represent specific hotels, rooms and guests
  - Objects are constructed by instantiating classes

PROGRAM DESIGN: CONCEPTS ARE CLASSES

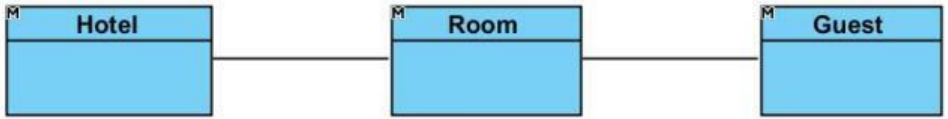
- First design step: class diagram



- Examples
  - ‘Hotel Fawlty Towers’
  - ‘Room 101’, ‘Room 102’, etc.
  - ‘Major Gowen’, ‘Miss Tibbs’, etc.

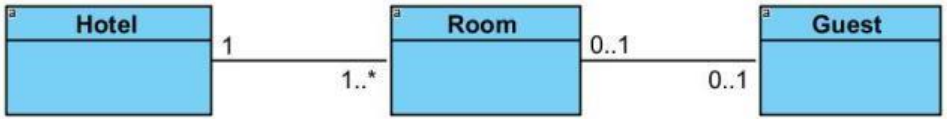
PROGRAM DESIGN: RELATIONS BETWEEN CONCEPTS

- What relations can be defined between these concepts?
  - `Hotel` *has* `Room`s, `Room` *belongs to* a `Hotel`
  - `Guest` *occupies a* `Room`, `Room` *has* `Guest`
- Second design step: extend class diagram with associations



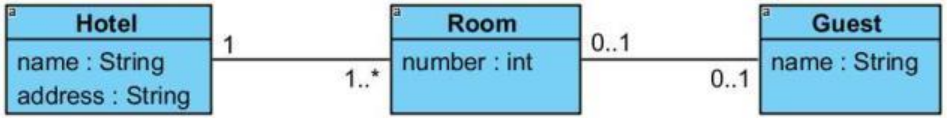
PROGRAM DESIGN: RELATIONS BETWEEN CONCEPTS

- Multiplicities: how many of these are there?
  - `Hotel` → `Room`: many; `Room` → `Hotel`: exactly one
  - `Guest` → `Room`: zero or one, `Room` → `Guest`: zero or one
- Third design step: extend class diagram with multiplicities



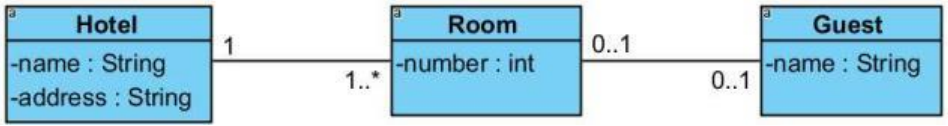
PROGRAM DESIGN: PROPERTIES/ATTRIBUTES

- What properties do our concepts have?
  - `Hotel`: name (a `String`), address (a `String`)
  - `Room`: number (an **`int`**)
  - `Guest`: name (a `String`)
- Third design step: extend class diagram with attributes



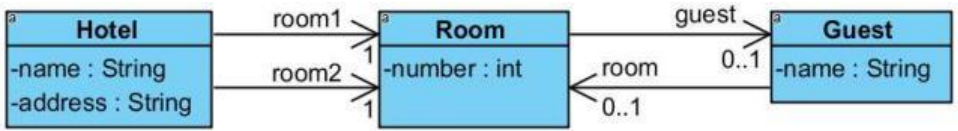
PROGRAM DESIGN: PROPERTIES/ATTRIBUTES

- Design principle: encapsulation
  - Attributes are “owned” by objects and not publicly accessible
- Fourth design step: extend attributes with visibility indicators



PROGRAM DESIGN: FROM ASSOCIATIONS TO FIELDS

- Make a choice which associations to code up
  - Does a hotel “know” its rooms?
  - Does a room “know” its hotel?
  - Does a room “know” its (optional) guest?
  - Does a guest “know” his room?



- Fifth design step: named & directed associations

For simplification, our `Hotel` now has exactly 2 `Rooms`

These are *our* answers here, but not the only or (necessarily) best ones

In fact, there is hardly ever a single or absolutely best choice

PROGRAM DESIGN: OPERATIONS

- Every class is responsible for part of the action
  - For this purpose, classes have operations
  - Queries: reveal some of the internal state
  - Commands: change the internal state

Examples

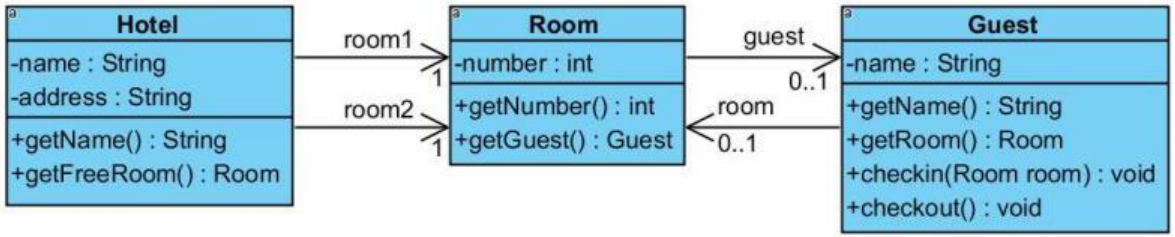
- For `Hotel`: what's it's name? What is a free `Room`? (queries)
- For `Room`: what's its number, etc. (queries)
- For `Guest`: check into a `Room` (command)

PROGRAM DESIGN: OPERATIONS

Examples

- For `Hotel`: what's it's name? What is a free `Room`? (queries)
- For `Room`: what's its number, etc. (queries)
- For `Guest`: check into a `Room` (command)

- Sixth design step: show (public) operations



NOW WE CAN START CODING THIS UP

- Each class in the diagram is implemented as a separate Java class (separate file)

```
public class Hotel {
    private String name;    // attribute
    private String address; // attribute
    private Room room1;     // association
    private Room room2;     // association

    public String getName() {   // query

        return name;        // method implementation
                            // not in design!
    }

    // more stuff
}
```

REMARKS

- Program design is not a complete system
  - Parts like the user interface are missing
  - Concepts, attributes and operations are incomplete
  - Gradual further development
- Program design is not an executable program
  - Gives a specification and structure
  - Details are missing, e.g., method bodies
  - Next step: detailed implementation
- There may be more than one (good) design!

CONSTRUCTORS FOR ROOM AND HOTEL

```
public class Room {
    private int number;
    private Guest guest;

    /*  Constructor
        Does not initialise guest attribute
    */
    public Room(int number) {
        this.number = number;
    }

    // to be continued
}
```

reference to field

formal parameter

reference to parameter

Simplified: address omitted

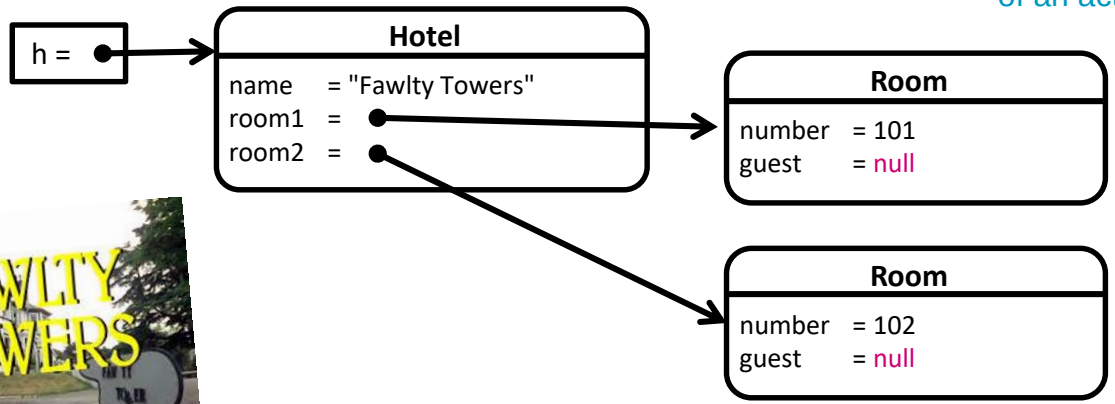
```
public class Hotel {
    private String name;
    private Room room1;
    private Room room2;

    public Hotel(String name) {
        this.name = name;
        room1 = new Room(101); // constructor call
        room2 = new Room(102); // constructor call
    }

    // more stuff
}
```

actual parameter (argument)

CLASS INSTANTIATION: CREATING AN OBJECT

- The piece of code `Hotel h = new Hotel("Fawlty Towers");` results in the creation of the following structure (in memory)

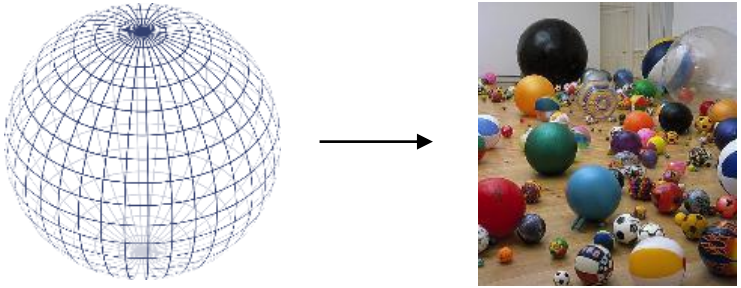
`null` stands for the absence of an actual value



AGAIN: CLASSES VERSUS OBJECTS (INSTANCES)

Class

Generic description of attributes and behavior



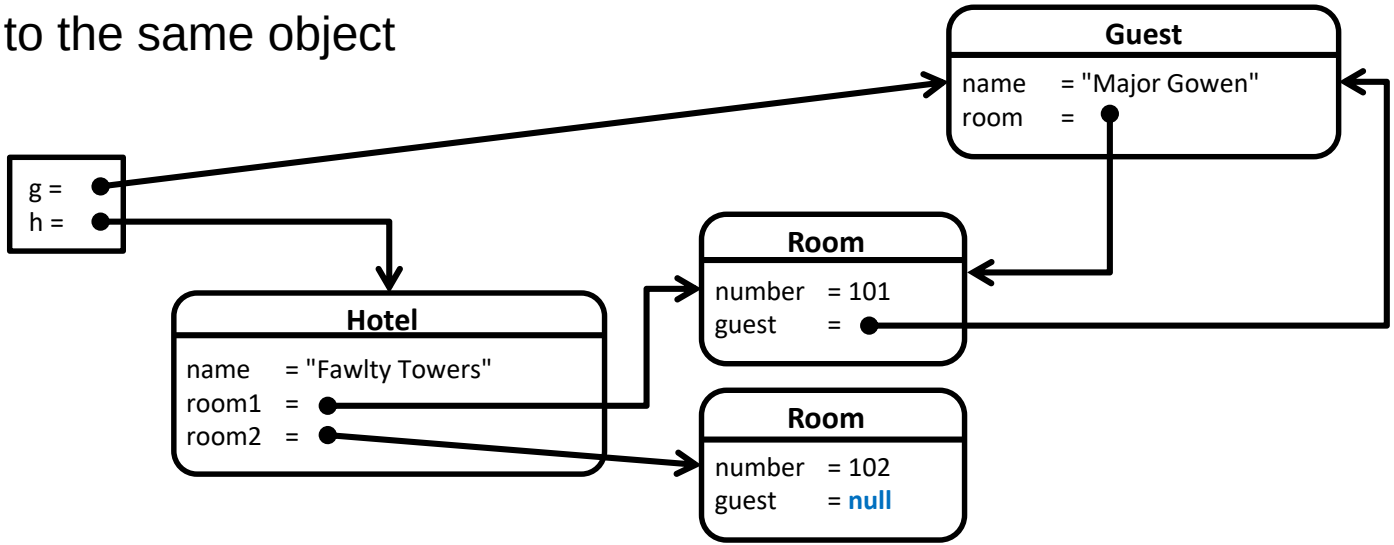
Object

Instance of a class with specific attributes and behavior

A GUEST IN OUR HOTEL

- Objects can point back and forth
- There can be multiple pointers to the same object

```
Hotel h = new Hotel("Fawlty Towers");
Guest g = new Guest("Major Gowen");
g.checkin(h.getFreeRoom());
```



CLASSES AND INSTANCES: STRINGS

- We have actually omitted something 😉
  - Strings are also classes, and hence reference types
  - `String` values are also pointers to (`String`) objects
- You hardly ever have to think about this
  - Except when comparing String values, which should never be done using `==`

KINDS OF VARIABLES

```
public class Hotel {
  public static final double VAT = 0.21;
  private String name;

  public double getBill(Guest guest) {
    Room room = this.getRoom(guest);
    return room.getPrice() * (1 + Hotel.VAT);
  }
}
```

- `VAT`: class variable (**`static`**), one per class
- `name`: instance variable, one per object
- `guest`: formal parameter, one per method invocation
- `room`: local variable, one per method invocation
- **`this`**: “automatic” pseudo-variable, refers to the object itself

VARIABLES: THIS

- What will happen in this constructor?

```
public Hotel(String name) {
    name = name;
    room1 = new Room(101);
    room2 = new Room(102);
}
```

- Will just (re-)assign to formal parameter `name`, not the attribute!
  - Always use **`this`** to refer to attributes

```
public Hotel(String name) {
    this.name = name;
    room1 = new Room(101);
    room2 = new Room(102);
}
```

STYLE

- The use of `{`
  ```
  // braces like this
  }
  ```
- Instead of
  - `static public`

  use
  - `public static`
- All these are pretty arbitrary
  - Just use them!
  - Checked by `checkstyle` plugin

It's okay to figure out murder mysteries, but you shouldn't need to figure out code. You should be able to read it.
-Steve McConnell

Programs must be written for people to read, and only incidentally for machines to execute.
- H. Abelson and G. Sussman

NAMING CONVENTIONS: LIKE IT OR LUMP IT!

- Names of different kinds of concepts are shaped differently
  - This helps recognition and thus readability
  - ... once you get used to it (so get used to it!)
- Choose meaningful names
  - Whole words or traceable abbreviations
  - Single letters only if the scope is very very small (few lines at most)
  - Reserved keywords not usable as names (`class`, `if`, `true`)
- Upper- and lowercase:
  - Class names always start Uppercase
  - Package, variable and method names always start lowercase
  - Constant names are all caps (only UPPERCASE)
- *All* names `areCamelCase and_not_underscore`
  - except `CONSTANT_NAMES` (where `CAMELCASE` does not work)

COMMENTS

- Text for documenting the code
  - One-line: starts with `//` reaches till the end of the line
  - Multi-line: Between `/*` and `*/`
- Useful for programmer
  - Increases comprehensibility
  - Especially when working in a team
  - Improves maintainability
- Is required in this course!

COMMENTS: JAVADOC

- Special kind of comments for documenting how to use a class
  - Multi-line between `/**` and `*/`
  - Start with textual description.
- Use tags to document specific information
- Important tags (some concepts are introduced in the next weeks)
  - @author (who is the author of this class or method)
  - @param (what does each parameter of the method mean)
  - @return (what does the method return)
  - @throws (what exceptions can the method throw)
  - @requires (preconditions of the method)
  - @ensures (postconditions of the method)

TAKE HOME MESSAGES

- Why object-oriented?
  - Structure and abstractions allow us to reason about our software
  - Separation of concerns
  - Encapsulation (public vs private)
- Objects are instances of classes
- Classes have fields, methods, constructors
- Classes are references types
- Your code must be readable!

  Good code is written for people first, and for machines second!