Object-Oriented Programming In Mechatronic Systems

Summer School

Module 6
Aachen, Germany, August 10th, 2018

Cybernetics Lab IMA & IfU
Faculty of Mechanical Engineering
RWTH Aachen University

Recap

Recap

Module 4 was about the more advanced concepts of OOP

- `this` and `super`
- Exceptions
- Packages
- Java API and
- Data structures (like ArrayList, HashMap, …)

… and concepts like recursion

Recap: `this` – When an Object Refers to Itself

`this` represents a reference (a pointer) to the current object

```
public class Rectangle {
   private Point2D lowerLeft;

   private Point2D upperRight;
   …
   public void setLowerLeft(Point2D lowerLeft) {
      this.lowerLeft = lowerLeft;
   }

   public Point2D getLowerLeft() {
      return lowerLeft;
   }
   …
}
```

Recap: `super` – When an Object Refers to its Super-Class Parts

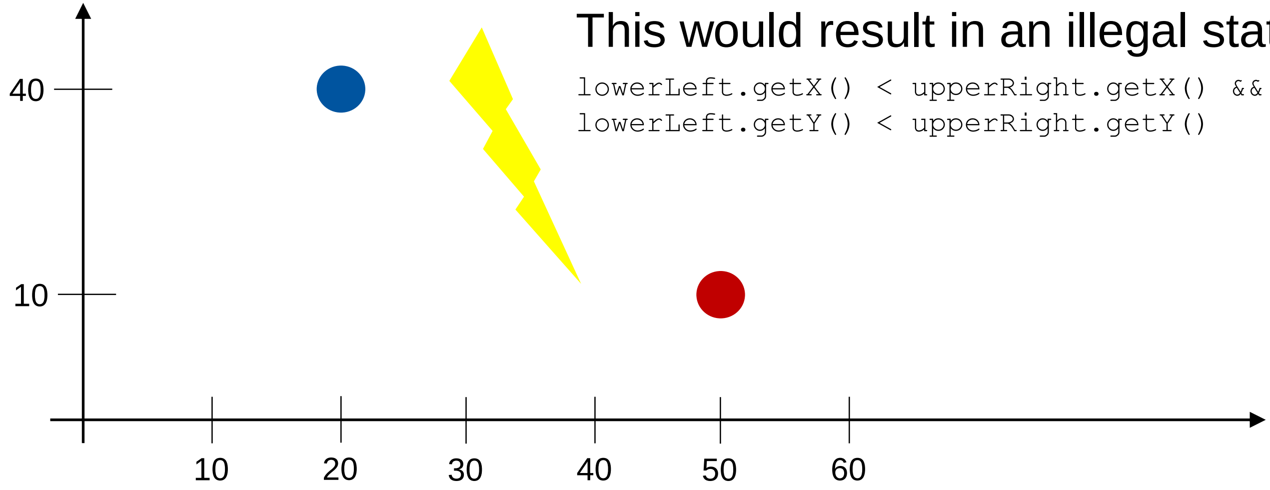
```
public class Square extends Rectangle {
      …
   public Square(Point2D lowerLeft, Point2D upperRight){
      super(lowerLeft, upperRight);
      if(this.upperRight.getX() - this.lowerLeft.getX()
         != this.upperRight.getY() - this.lowerLeft.getY())
      {
         throw new IllegalStateException();
      }
   }
}
```

A Square is a special case of a rectangle, where the sides have equal length

- Square extends and specializes Rectangle
- To save the two points needed to describe a rectangle, we need to call the constructor of Rectangle
- Use **super** to call the overridden method or constructor of a **superclass**

Recap: Exception Handling

```
public class Application {

   public static void main(String[] args) {
      Rectangle r = new Rectangle(new Point2D(50,10), new
         Point2D(20,40));
      System.out.println("The area of r is: " +
         r.calculateArea();
   }
}
```



This would result in an illegal state!

```
lowerLeft.getX() < upperRight.getX() &&
lowerLeft.getY() < upperRight.getY()
```

Recap: Exception Handling

```
public class Application {

   public static void main(String[] args) {
      try {
         Rectangle r = new Rectangle(new Point2D(50,10), new
            Point2D(20,40));
         System.out.println("The area of r is: " +
            r.calculateArea();
      } catch (IllegalStateException e) {
         System.err.println("The initialization of rectangle
            failed. Reason: " + e.getMessage();
      }
   }
}
```

If the initialization fails (due to a created illegal state), an `IllegalStateException` is thrown: Now, we can react accordingly, by catching the `Exception`.

The Catch or Specify Requirement

- Valid code must honor the *Catch or Specify* Requirement
- If code might throw certain exceptions, code **must** be enclosed by…
  - … a `try` statement that catches the exception **or**
  - … a method that is marked via the `throws` clause (telling the caller that the method can throw such exceptions)
- Code that does not honor the requirement doesn't compile!

Recap: Exception Handling

Catching and Handling Exceptions

Three exception handler components: `try`, `catch`, and `finally`

```
try
{
    statements that can throw exceptions
}
catch (exception-type identifier)
{
    statements executed when exception is thrown
}
finally // not mandatory!
{
    statements that are always executed
}
```

Recap: Packages

Definition

- A *package* is a grouping of related types (e.g. classes or interfaces)
- Make stuff easier to find and use
  - … to avoid **naming conflicts**
  - … to **control access**.

Usage

- Examples: `java.util` (for utilities) or `javax.swing` (for creating GUIs)
- We have to **either import it** by using the `import` keyword…

```
import java.util.ArrayList
```

- … or type in the full name of the class everywhere in our code!
- You can bundle your own code in packages: use the `package` statement

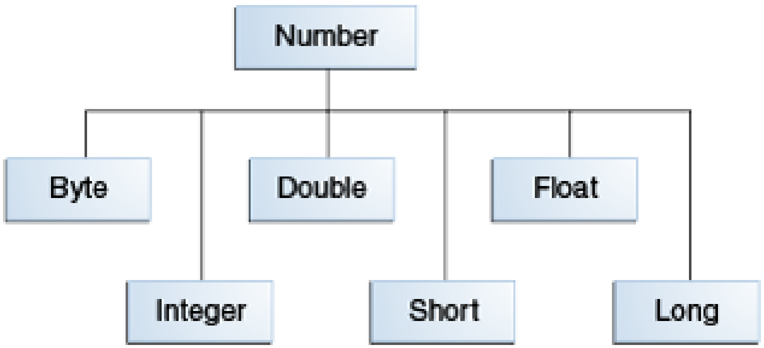
Recap: Wrapper Classes

Problem

- `int`, `double`, `float`, … are primitive data types and therefore **not defined by classes ➔ you cannot create an object of type int**
- Often you have **data structures** that can hold objects of a specific type, **but only objects**.

Solution: Wrapper Classes

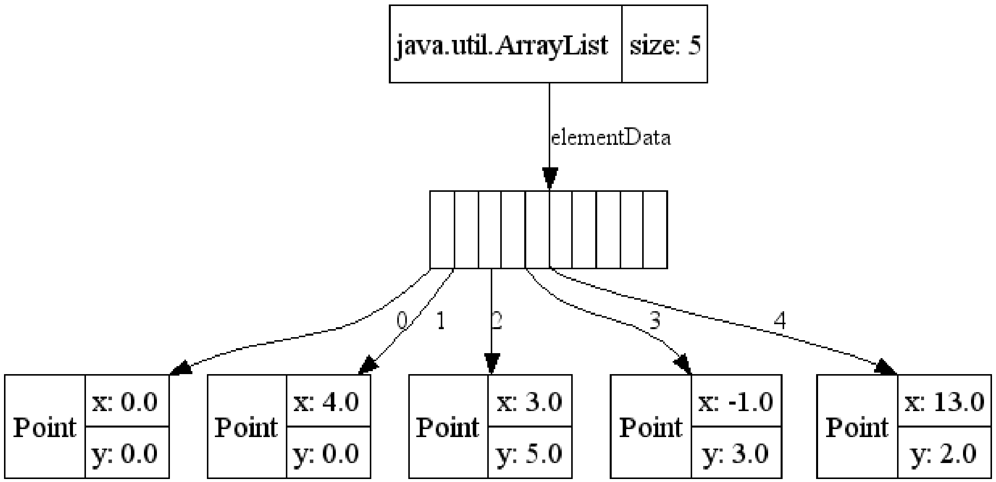
- Java provides **wrapper classes for each of the primitive data types**.
- Wrapping can be done by compiler (compiler boxes primitive in its wrapper class) and unboxes them if needed.



Recap: Java API and `ArrayLists`

Class `java.util.ArrayList`

- `ArrayList` extends `AbstractList` and implements the `List` interface
- **Data structure to hold objects** (e.g. class `Integer` or class `Point`)!
- **Automatically manages its size.**
- **Provides convenient methods to remove, find and add objects to the list.**



Recap: Java API and `ArrayLists`

`ArrayList` methods (Excerpt)

- **`void add(int index, Object element)`**
  Inserts the specified element at the specified position index in this list.
- **`void add(Object element)`**
  Inserts the specified element at the end of this list.
- **`void clear()`**
  Removes all of the elements from the ArrayList.
- **`Object remove(int index)`**
  Removes the element at the specified position in this list.
- **`int size()`**
  Returns the number of elements in this list.

Recap: Java API and `Map`

Aka an *associative array*

- **It's a collection of (key, value) pairs**
- Each possible key appears just once in the collection
- Important operations are *add*, *remove* or *lookup*

KEYS	VALUES
Jan	327.2
Feb	368.2
Mar	197.6
Apr	178.4
May	100.0
Jun	69.9
Jul	32.3
Aug	37.3
Sep	19.0
Oct	37.0
Nov	73.2
Dec	110.9
Annual	1551.0

Aug → 37.3

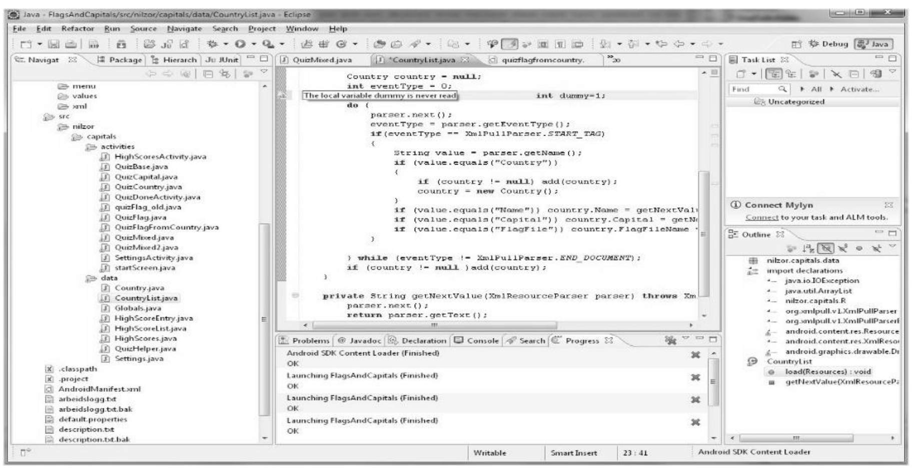
Unified Modeling Language (UML)

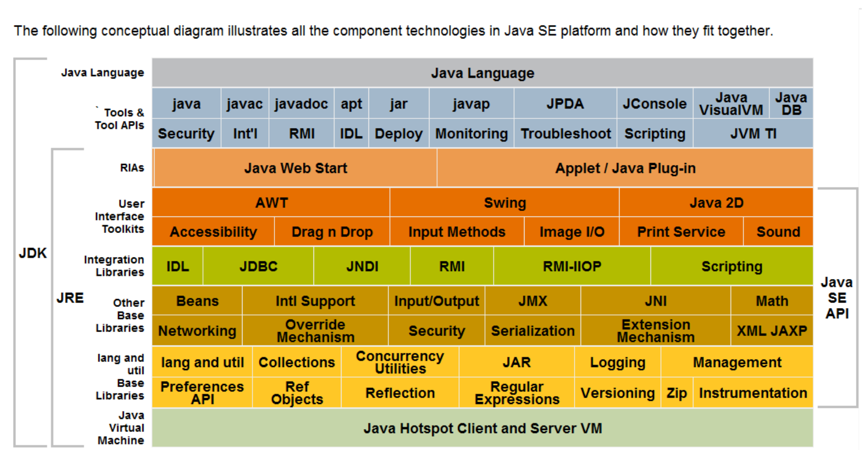
Modeling software before programming

Motivation

- Your company is given the task of developing a software system…
- Let's say a software for handling customer complaints!
- How would you proceed?

Would you just start programming?





Motivation

Planning upfront might be a good idea, but how do you communicate your ideas, plans and needs?



You need a common ground for communication, some kind of a language: Unfortunately, a natural language cannot succeed in this task, because it is ambiguous and complex!

Standardization and Visual Modeling

We need a standardized modeling notation, that…

- … must have well-defined semantics,
- … must be well suited for representing aspects of a system and
- … must be well-understood among project participants (which can be dozens of peoples)

It would be great to have some visual modeling, because…

- … **models are visual**: They are potentially a more efficient and effictive form of communication than prose,
- … **models are more precise**: It is hard to recognize missing elements in written forms of requirements, but with a visual model it is more noticeable,
- … **models can represent ideas from different angles / perspectives**

Unified Modeling Language (UML) is a visual modeling language

Basic idea of UML

To provide the stakeholders of an object-oriented software development process with a common and standardized development and analysis tool.

UNIFIED MODELING LANGUAGE ™

- **Industry Standard** for specifying, visualizing, constructing, and documenting the artifacts of software systems
- UML uses **mostly graphical notations** to express the OO analysis and design of software projects
- UML **simplifies** the complex process of software design

Unified Modeling Language

Why we use UML?

- **Use graphical notations (remember visual modeling):** more clearly than natural language (imprecise) and code (too detailed)
- Help acquire an overall view (different perspective) of a system
- **UML is independent on any programming language or technology**
- **UML moves us from fragmentation to standardization**

Unified Modeling Language

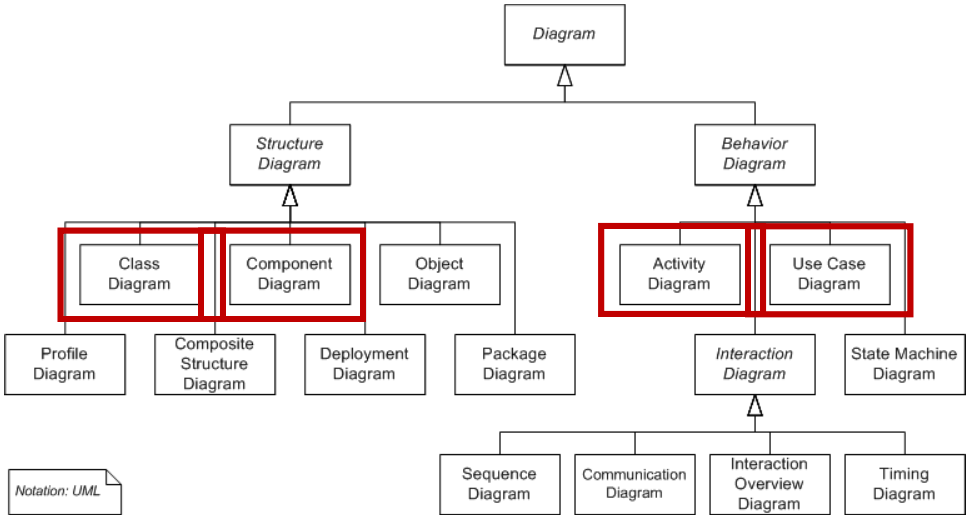
UML is standardized

- Current Version: **2.5** (May 2015)
- ISO/IEC DIS 19505-1
- ISO/IEC DIS 19505-2
- **Constantly further developed**: http://www.omg.org/spec/UML/Current
- **UML defines different types of diagrams for modeling**

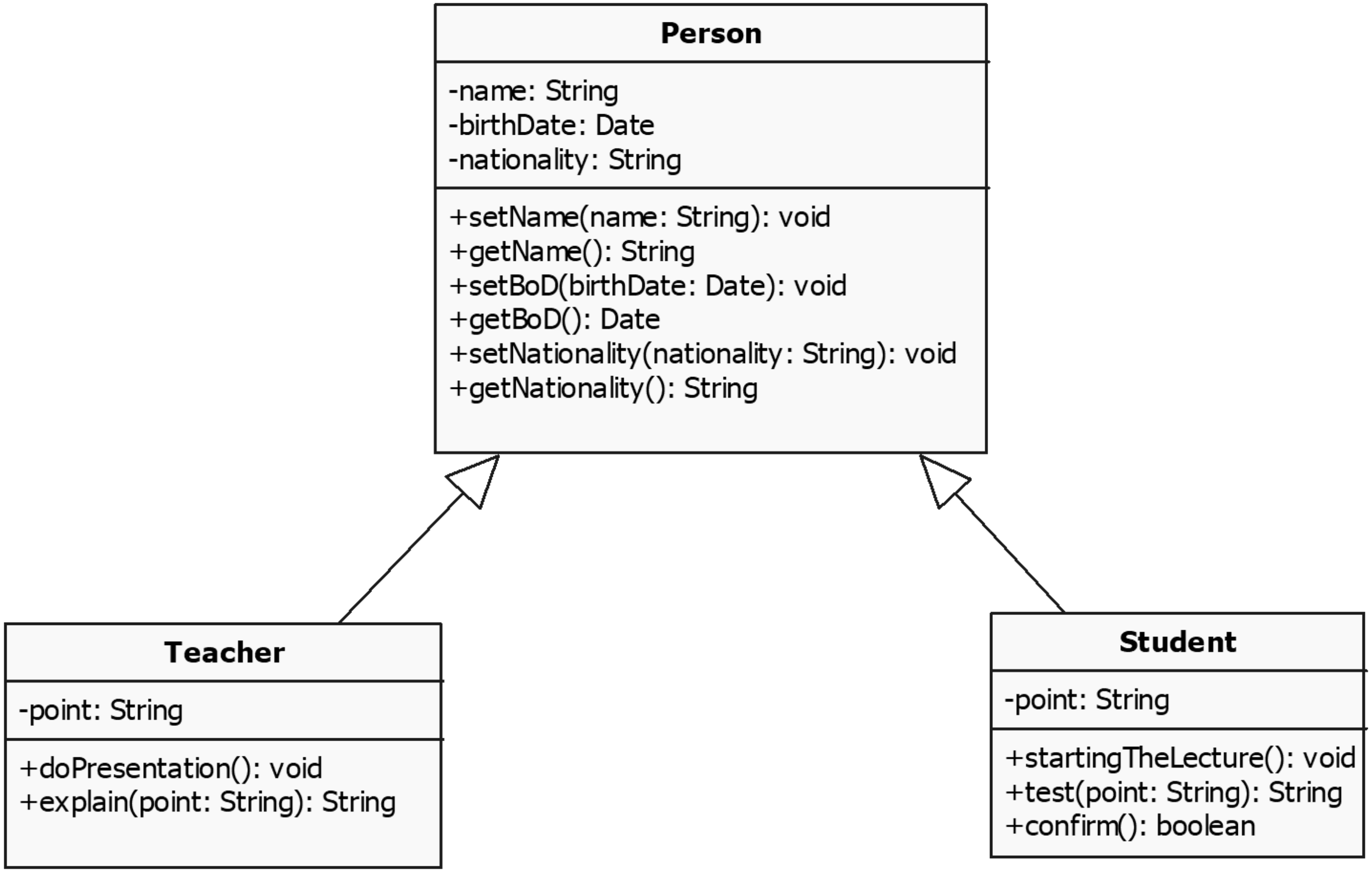
Two main diagram types

- **Structure** diagrams
- **Behaviour** diagrams

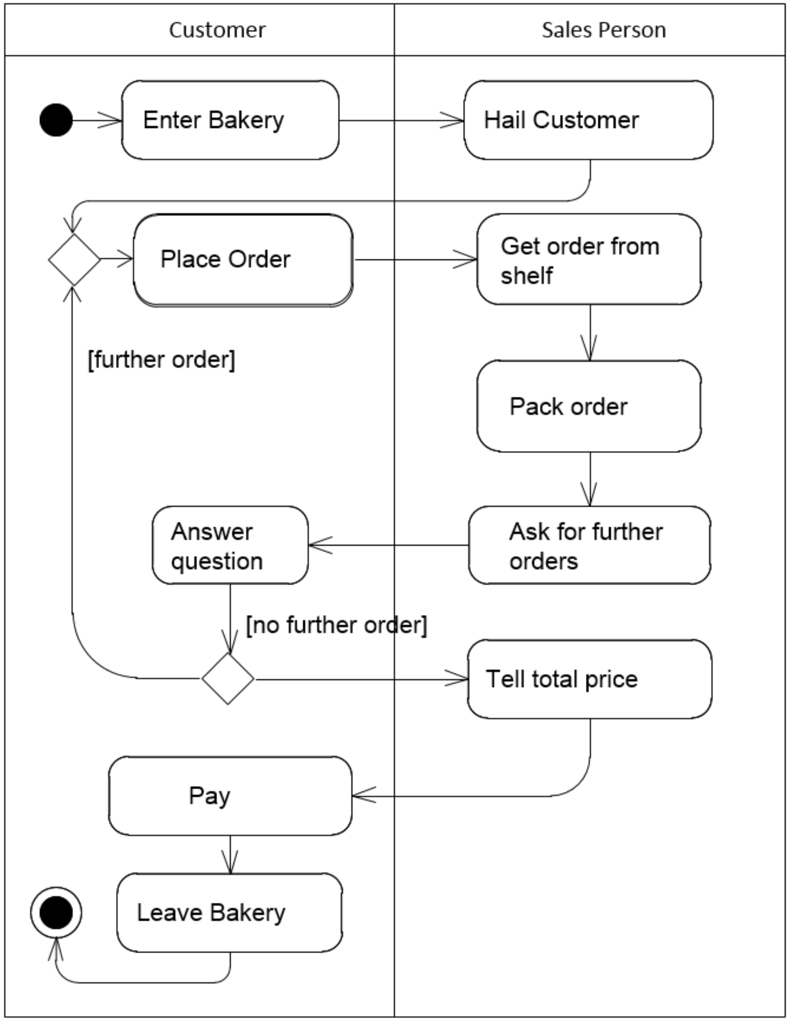
We are focusing on four diagram types:



UML Class Diagram



UML Activity Diagram

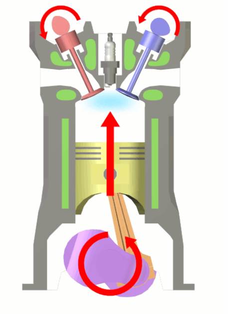


UML Activity Diagram

Activity Diagram

Bringing a system to life, i.e. describing its dynamic behavior

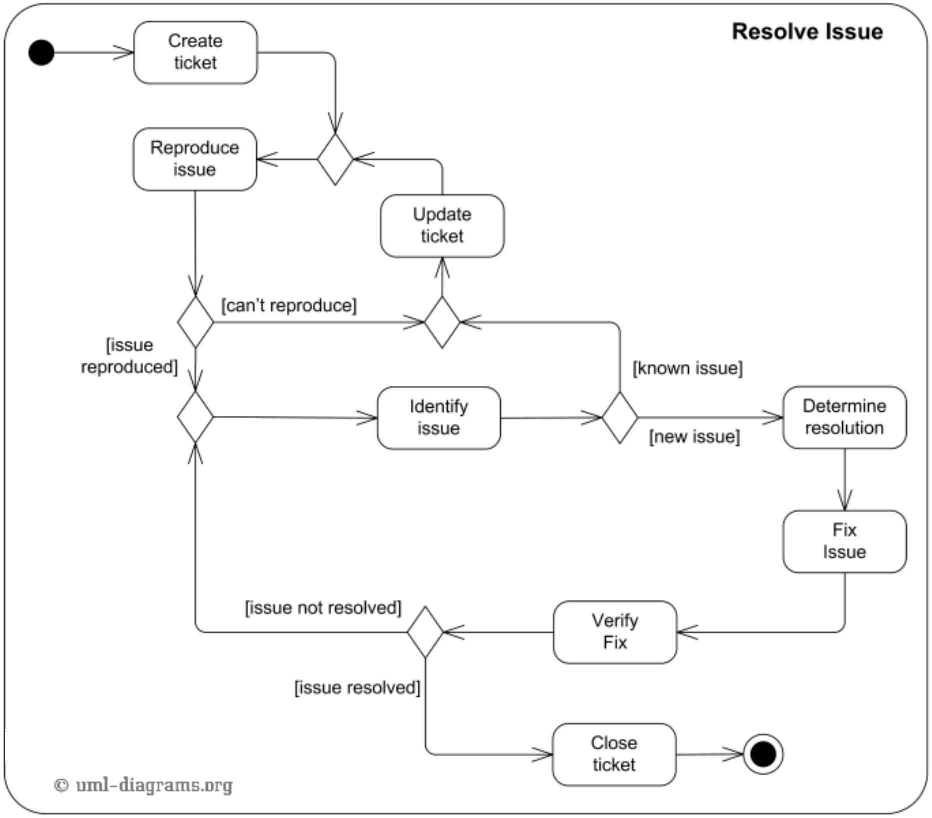
- Previously, we have met component diagrams
- They describe the (static) structure of a system
- … and not the flow of events!

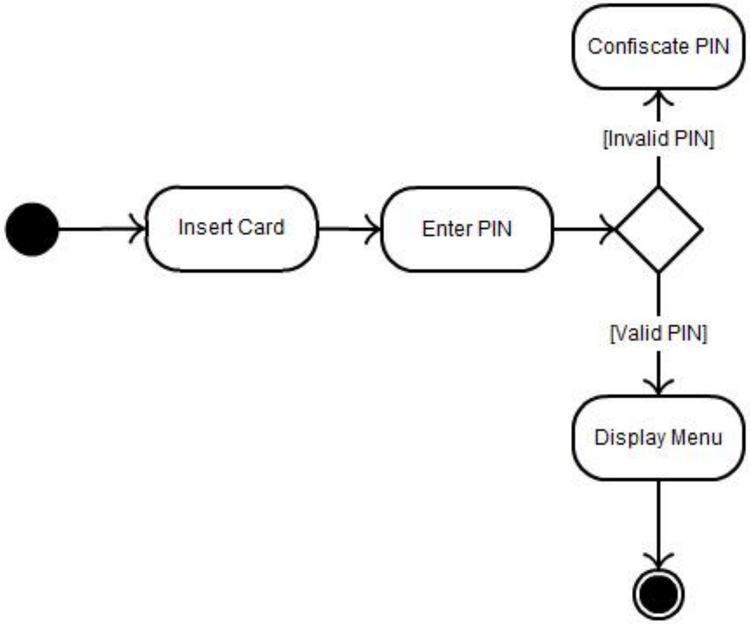


Different dynamic aspects of a system can be UML-modeled

- Interaction between components: **sequence and communication diagram**
- The process of state changes: **state diagram**
- Processes and algorithms: **activity diagram**
- Activity diagrams are similar to flowcharts

Activity Diagram: Examples

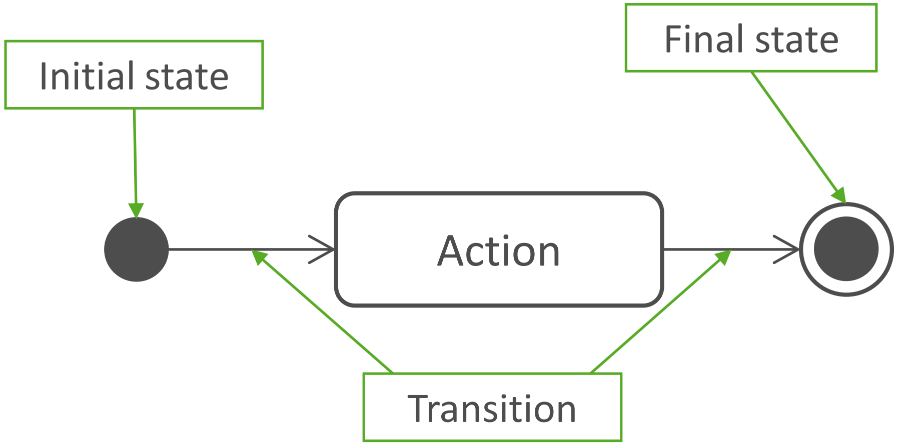




Activity Diagram: Building Blocks

Minimal requirements

- An **initial state** (black circle)
- A **final state** (encircled black circle)
- At least one **action** (rounded rectangle)



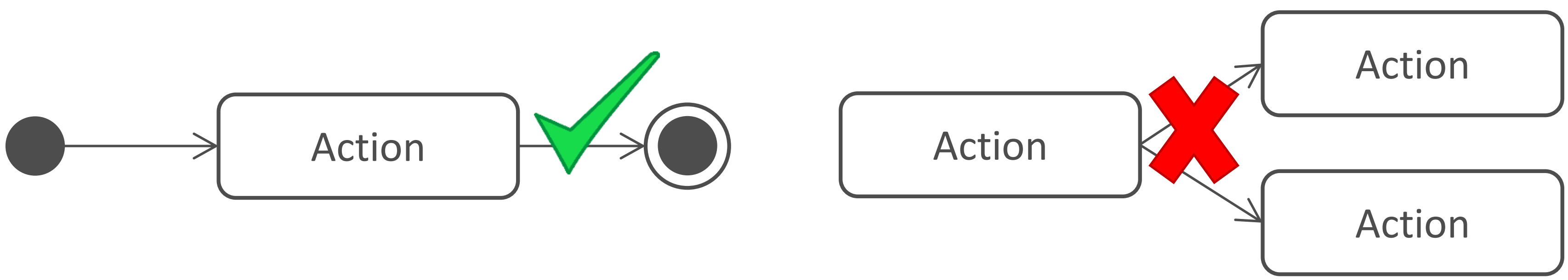
Activity Diagram: Building Blocks

Actions

- An **executable unit**
- In the context of the model not decomposable
- In a programming language such as a method call or a computation

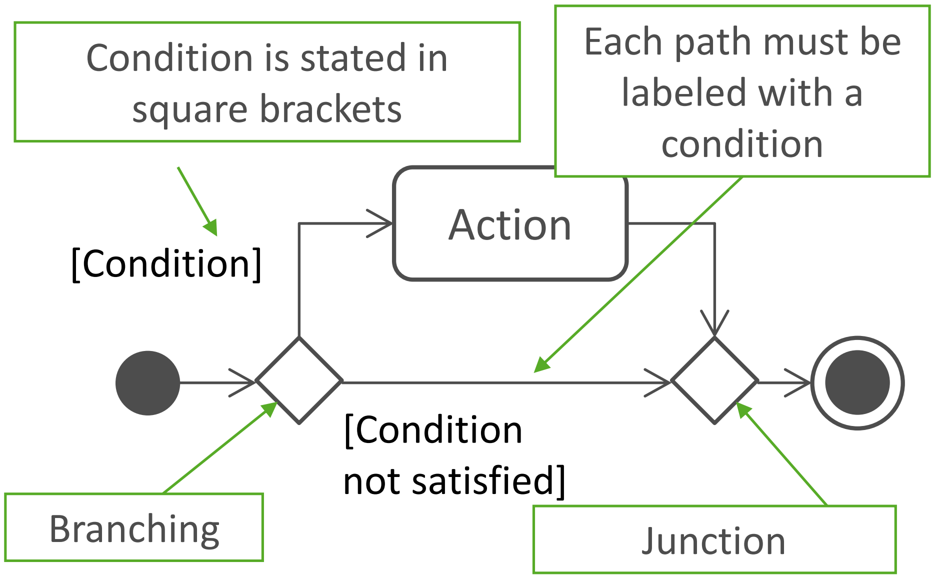
Transactions

- Initial state and action each have only **one outgoing transition**
- Final state and action each have only **one incoming transition**



Branching / Decisions based on conditions (1/3)

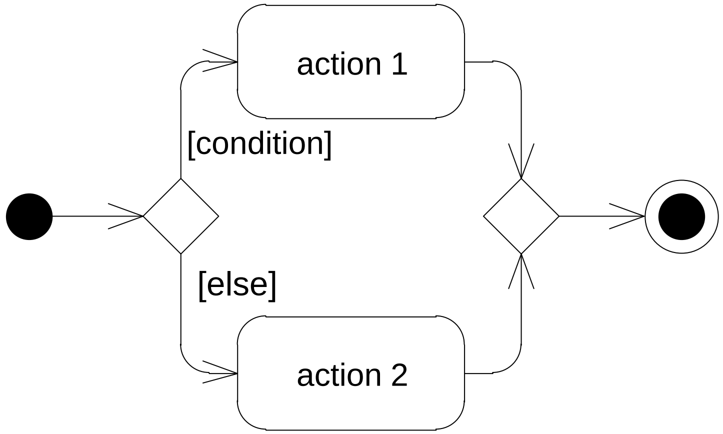
- Activity diagrams can also model branching or decisions within the activity flow: Diamonds for representation
- Reminder (Java): `if (Condition){action();}`

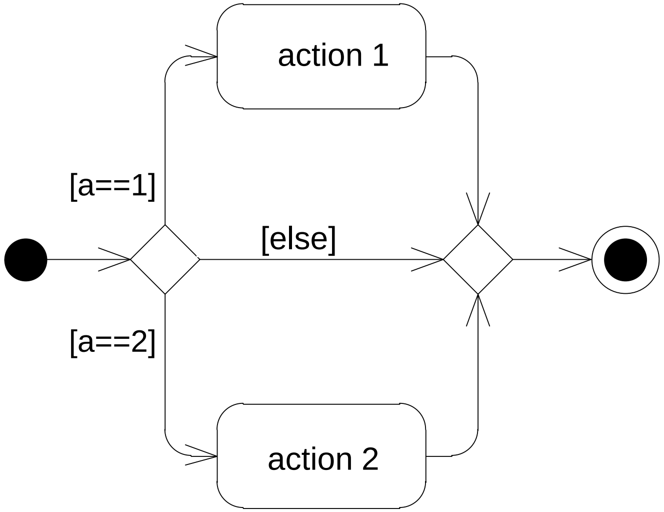


Branching / Decisions based on conditions (2/3)

```
if (condition){
    action1();
} else {
    action2();
}
```

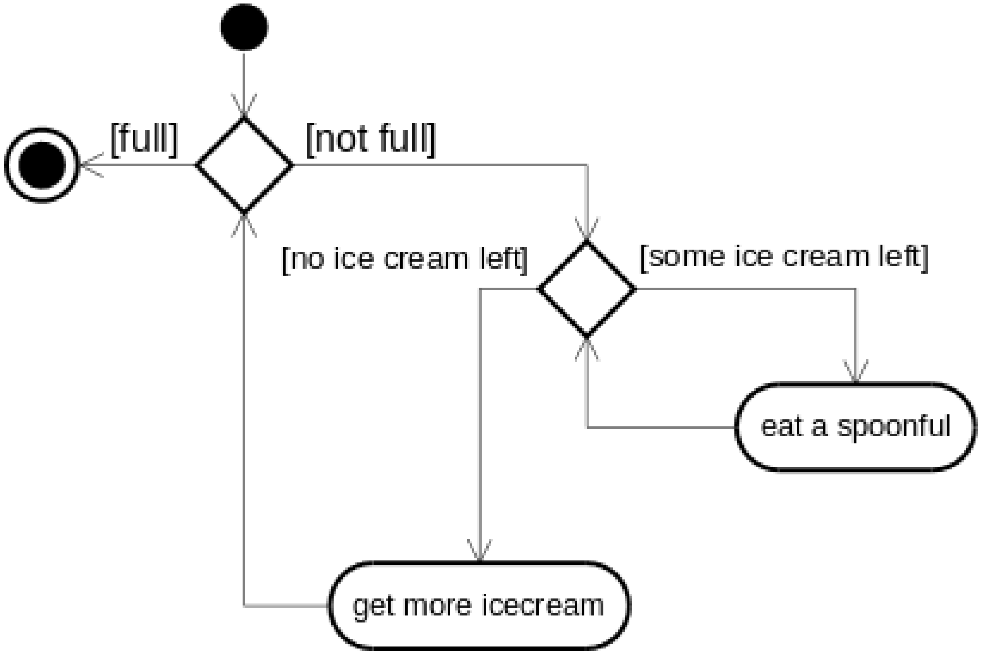
```
switch (a){
    case 1: aktion1(); break;
    case 2: aktion2(); break;
    default:
}
```





Branching / Decisions based on conditions (3/3)

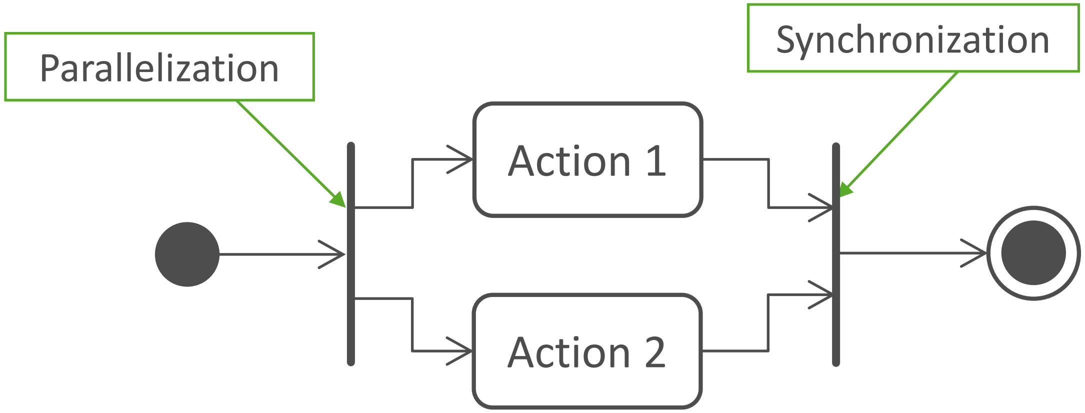
Looping can be represented as well!



Activity Diagram: Building Blocks

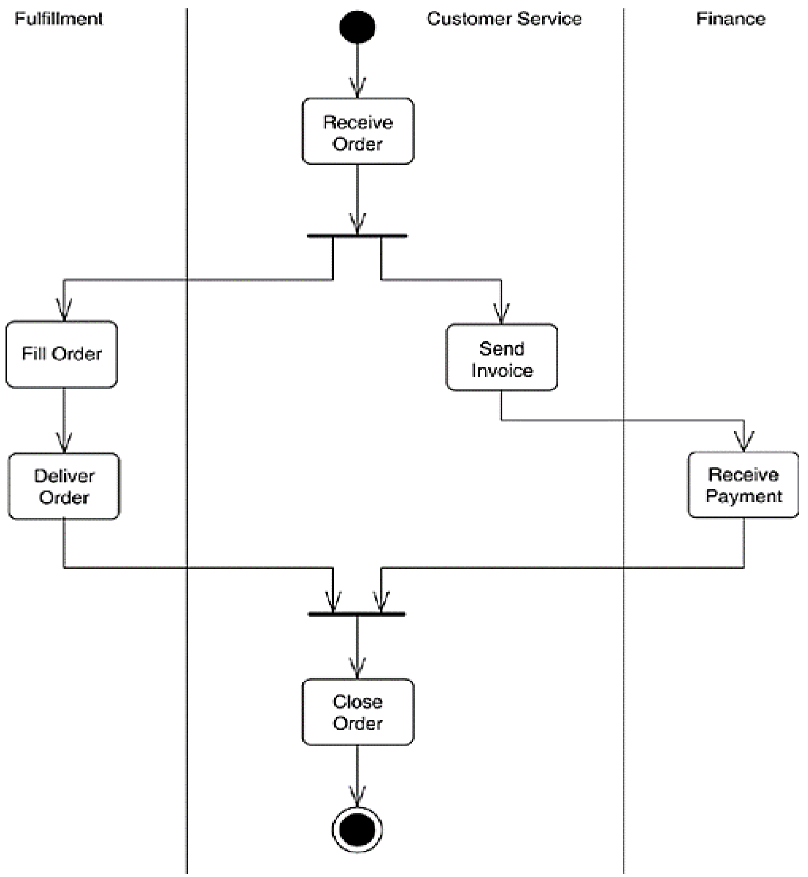
Parallelization

- Parallelization is used to spilt one control flow in several ones
- Two actions are executed in parallel …
- After execution they are merged together (aka synchronization)
- In Java that could be done via **Threads** (not covered during lecture)



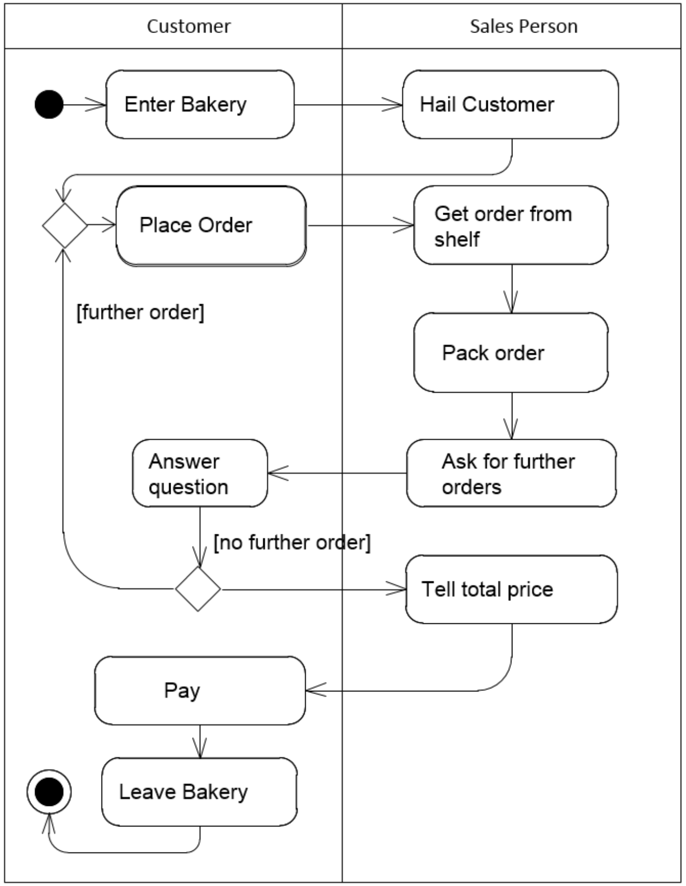
Activity Diagram: Building Blocks

Swim lanes to assign actions to components / classes



Activity Diagram: Example

Shopping trip to the bakery ☺



Activity Diagram Generation

Get activity diagram from textual description

Root finding with the bisection method

Task: Find the square root of a given x.

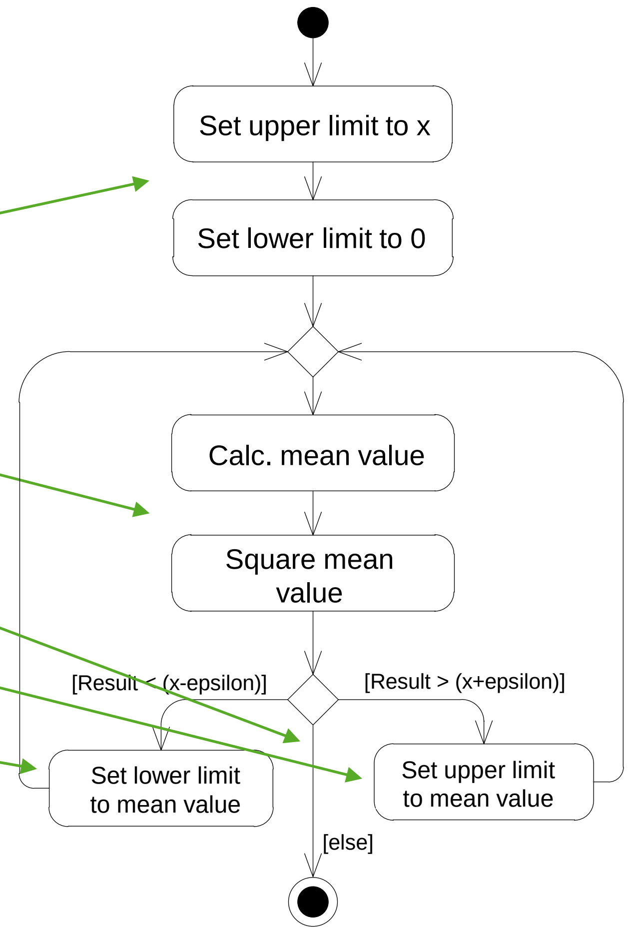
Choose x as an upper limit and 0 as a lower limit.

Calculate the mean value of upper and lower limit and square this value

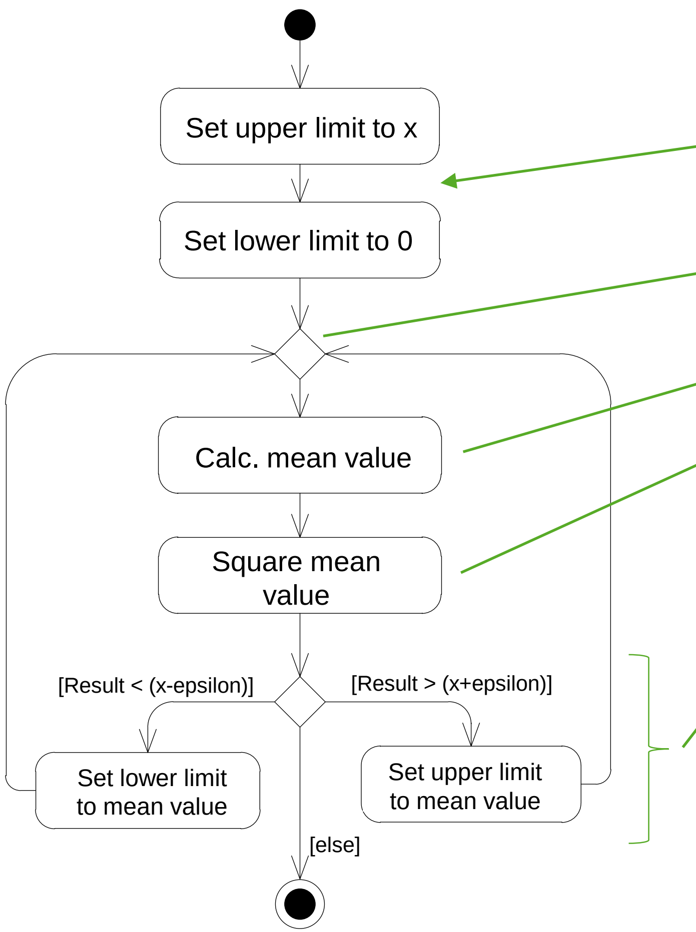
If the result is within a given epsilon environment (which can be set) to x, end the calculation

If the result is bigger than x, set the mean value as the new upper limit an repeat the calculation.

If the result is smaller than x, set the mean value as the new lower limit an repeat the calculation.



Get source code from activity diagram




```
double squareRoot(double x, double eps) {
    double upper = x;
    double lower = 0;
    double mV = 0;
    boolean traced = false;
    do {
        mV = (upper + lower) / 2;
        double square = mV * mV;
        if (square > x + eps) {
            upper = mV;
        } else if (square < x - eps) {
            lower = mV;
        } else {
            traced = true;
        }

    } while (!traced);
    return mV;
}
```

UML Class Diagram

Class Diagram: Introduction

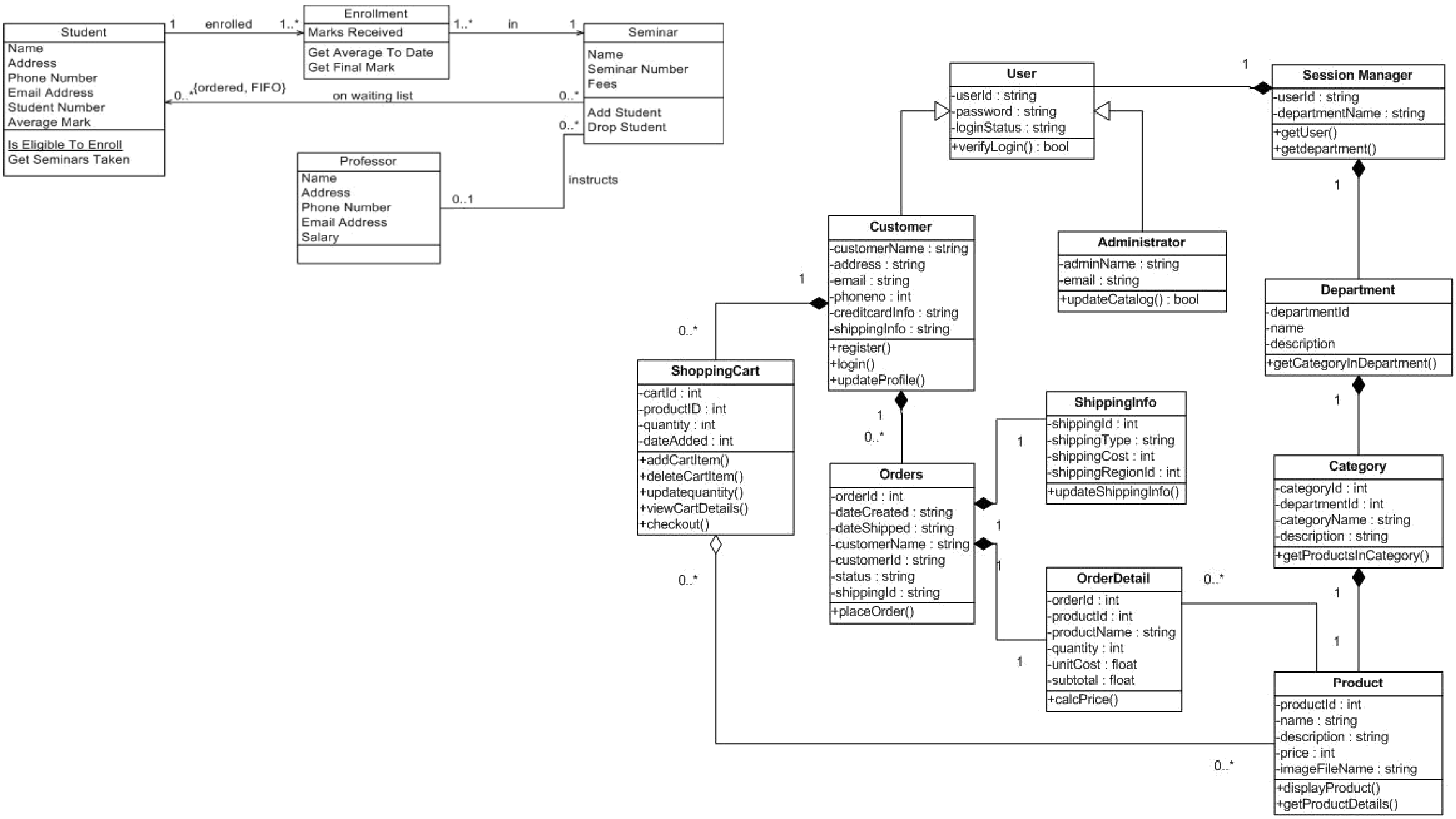
Structure of an OO-software project

- Class diagrams visualize the static structure of object-oriented SW
- The structure of a source code can be represented by a class diagram
- Class diagram can be based on source code or
- Source code can be based on class diagram

Class diagrams reveal

- Classes and their
  - Attributes (+visibilities)
  - Methods (+visibilities)
- Relationship between classes, e.g. inheritance and dependencies

Class Diagram: Examples



Class Diagram: Building Blocks

Element	Notation	Example/ Explanation
Class	Name	Car
Class with attributes and methods	Name attributes methods	Car speed: int getSpeed() : int setSpeed (speed : int): void
Visibility of attributes and methods	private: - public: + protected: #	Car - speed: int + getSpeed () : int + setSpeed (an speed:int):void

Class Diagram: Building Blocks

access modifier
+ = public
- = private
= protected

Class Name
-attribute:DataType
+Method(parameter:DataType):Return DataType

General class syntax

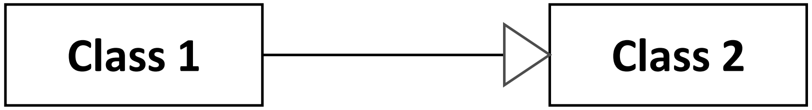
Part
-id:String -name:String
+Part(id:String, name:String) +getId():String +getName():String -setId(id:String):void -setName(name:String):void

Concrete (remember lecture 4)

Building blocks (relationships)

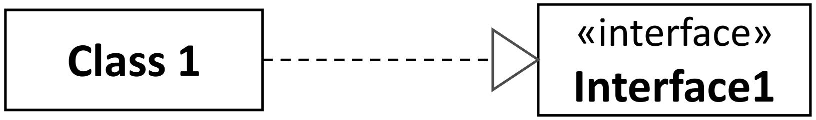
- The **generalization** is the “normal” inheritance from OOP. The triangle is attached to the superclass!

Generalization

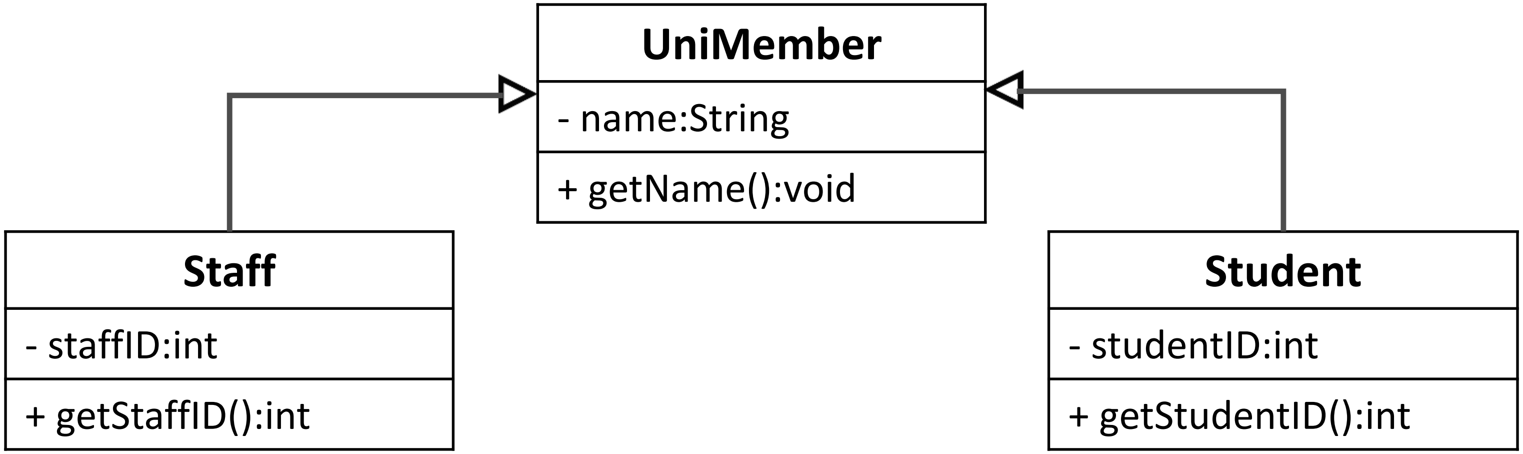


- The **realization** is the equivalent to the implementation of an interface.

Realization



Building blocks (generalization)



Class Diagram: Building Blocks

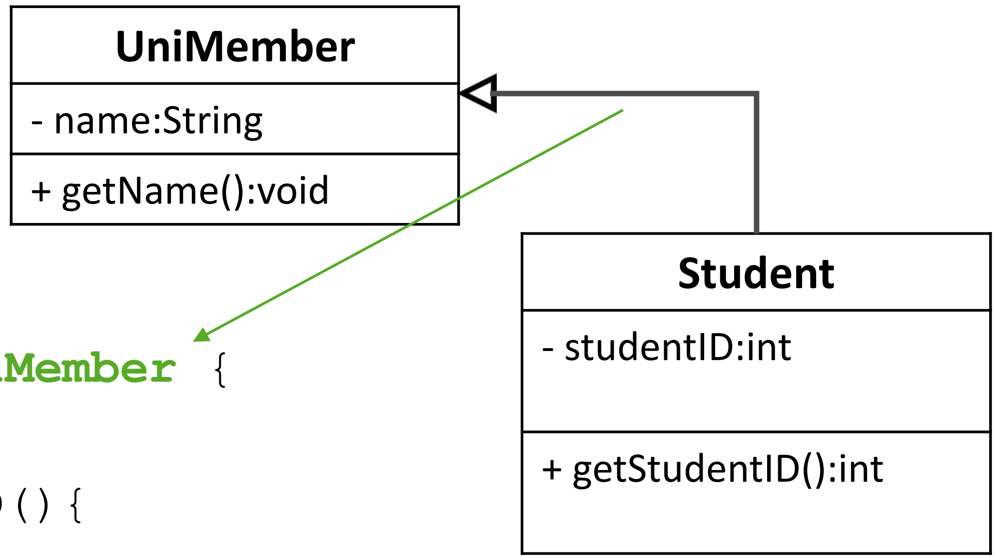
Building blocks (generalization in code)

UniMember
- name:String
+ getName():void

```
class UniMember{
  private String name;

  public void getName(){
    System.out.println("I‘m " + name);
  }
}
```

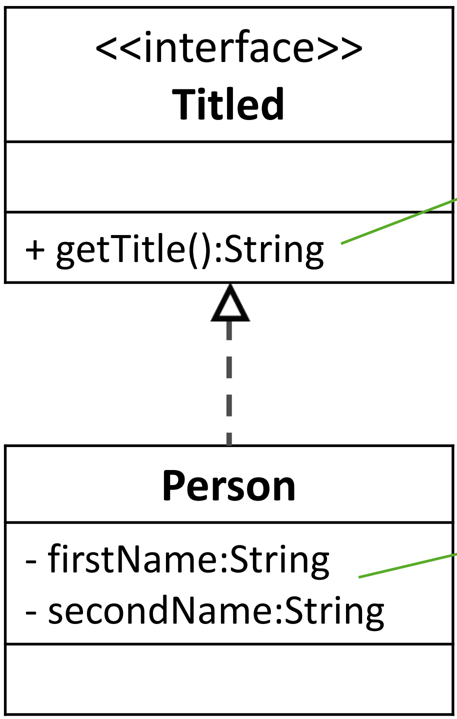
Concrete implementation = behavior
Not part of class diagram!

Building blocks (generalization in code)




```
class Student extends UniMember {
  private int studentID;

  public int getStudentID(){
    return studentID;
  }
}
```

Building blocks (realization)




```
public interface Titled {

  public String getTitle();

}
```

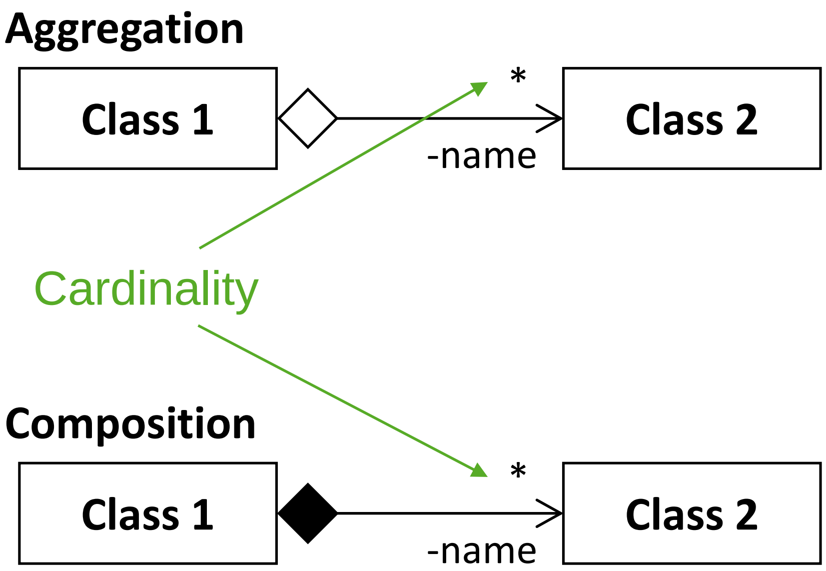
```
public class Person implements Titled {

  private String firstName;
  private String lastName;

  public String getTitle() {
    return firstName + “ “ + lastName;
  }
}
```

Class Diagram: Building Blocks

Building blocks (relationships)

- The **aggregation** is a weak dependency where instances of class 1 use instances of class 2. Class 1 and class 2 can exist independently of each other.

- The **composition** is a strong dependency where instances of class 1 consist of instances of class 2. The life cycles of each class are coupled together. If class 1 gets deleted class 2 gets deleted as well.



Building blocks (cardinality)

Cardinality specifies the range of possible objects existing in a relationship between two classes.

Cardinalities	Meaning
0..1	zero or one instances (n..m indicates n to m instances)
0..* or *	no limited number of instances
1	number of instances (here 1)
1..*	at least one instance

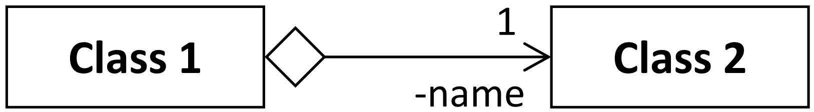
! **Cardinalities larger than 1 are realized using Arrays, Lists etc.**

Class Diagram: Building Blocks

Aggregation in code

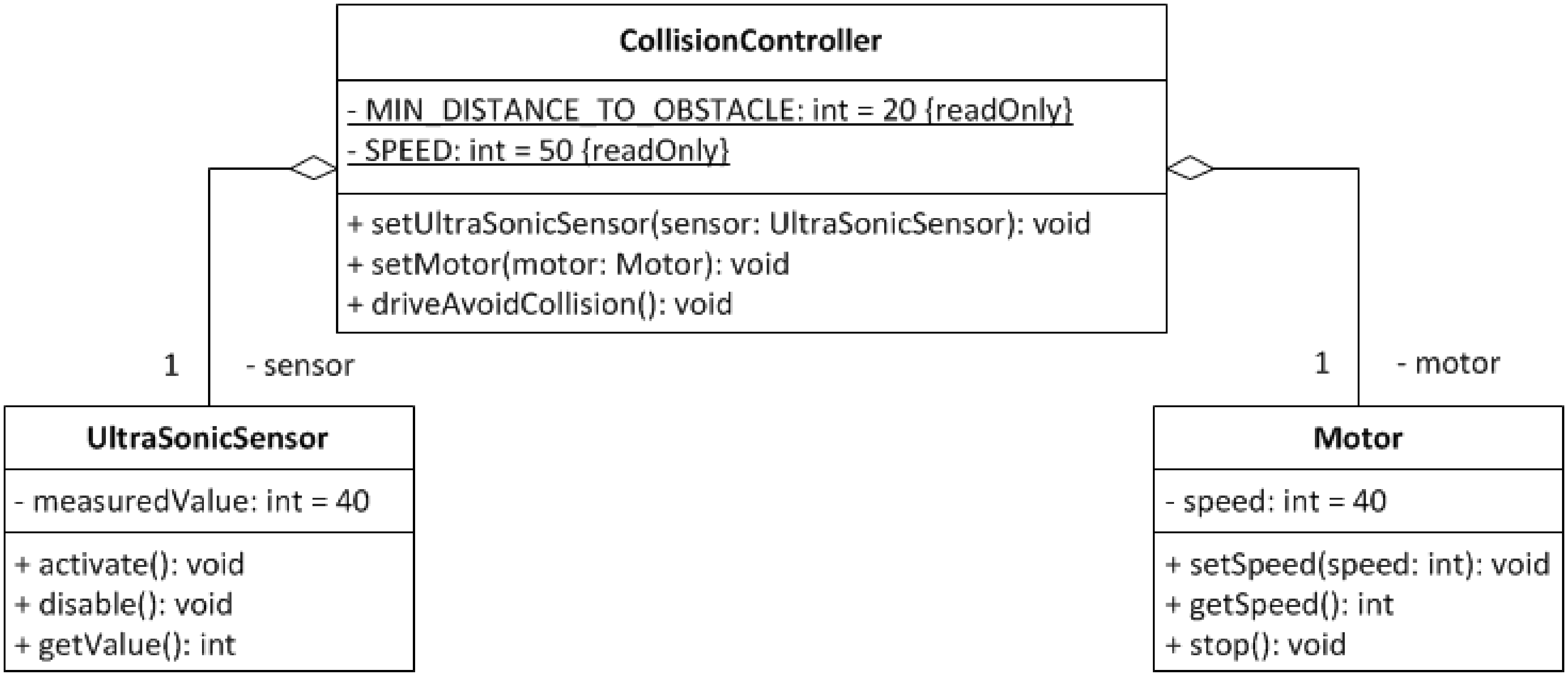
```java
public class Class1 {
  private Class2 name;

  public void setName(Class2 name) {
    this.name = name;
  }
  public Class2 getName() {
    return name;
  }

  //…
}
```

Aggregation



! The deletion of the whole does not carry on to the parts.

Class Diagram: Example



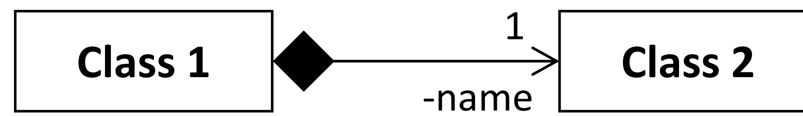
Class Diagram: Building Blocks

Composition in code

```java
public class Class1 {
  private Class2 name;

  public Class1() {
    name = new Class2();
  }
}
```

```java
public class Class1 {

  private Class2 name = new Class2();

}
```

Composition



! The deletion of the whole carries on to the parts.

Thank you very much!