

Object-oriented programming

Lecture 4: methods and top-down design

Krasimir Angelov

September 2018

Chalmers University of Technology

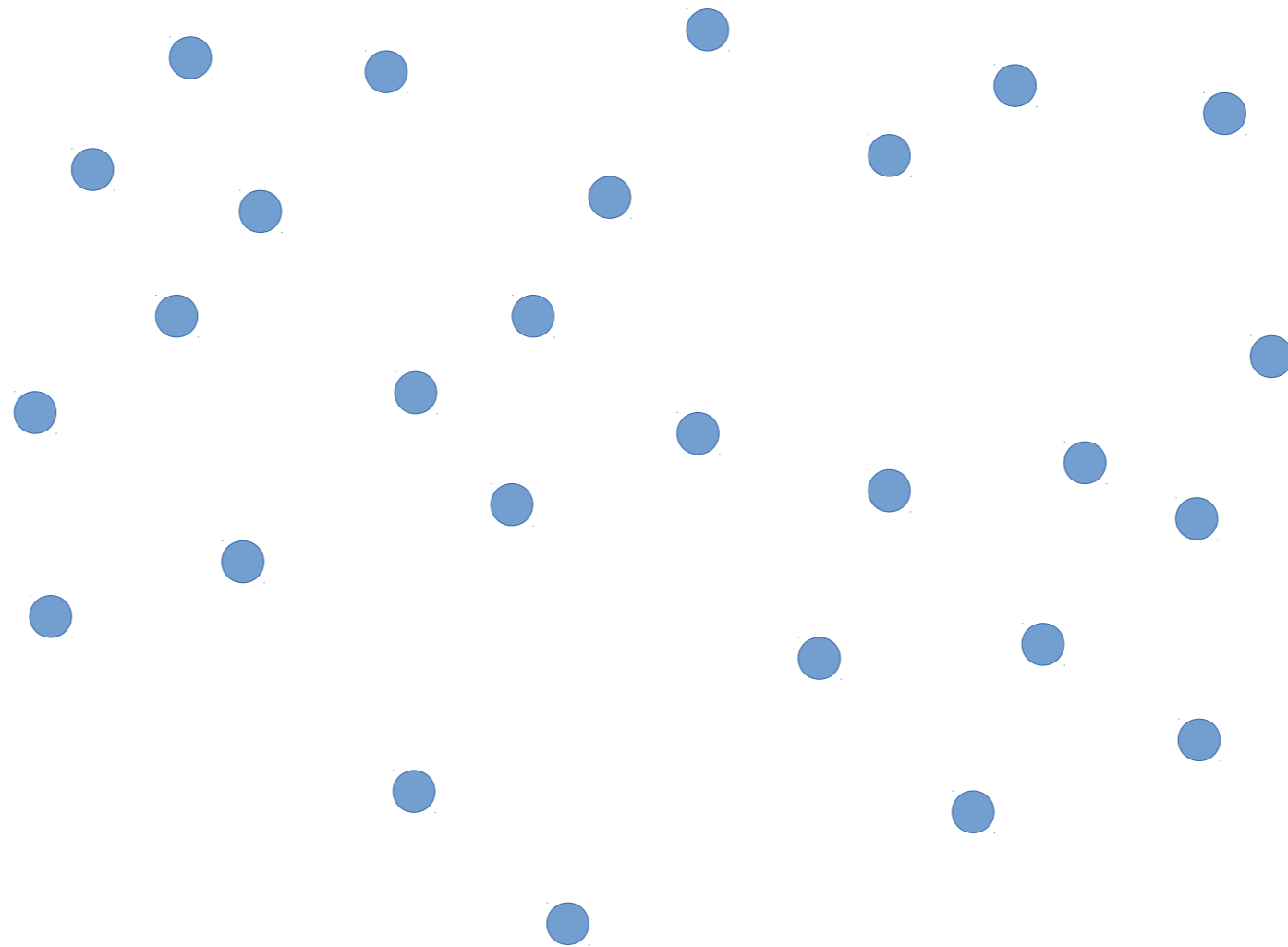
Reminder

- Just a few days left to the deadline for lab 1

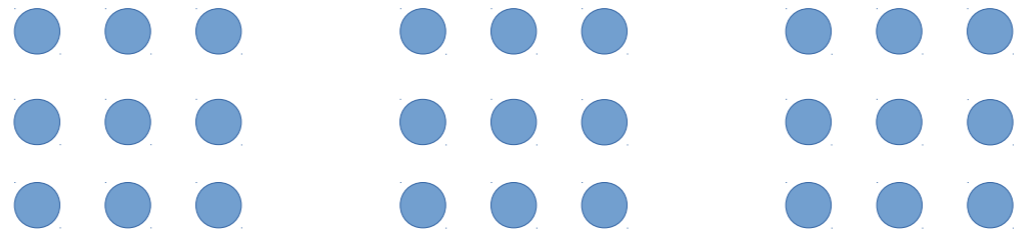
- `double`: a real number limited in size and precision
- Iteration: `while`-, `do`- and `for`-statements
- Variable scope
- Repeated Program Execution
- `printf`

Abstraction

Psychological Experiment



Psychological Experiment



Programming = modelling

- A computer program is a *model* of a real or imaginary world; usually there is a complex system which must be modelled
- In object oriented programming this world consists of a number of *objects* which together solve a given task.
 - The different objects have specific *responsibilities*
 - The objects cooperate by communication with each other via messages
 - A message to an object is a call from another object to make something done



To do a good model of reality, and therefore to allow a good program design, is a challenge.

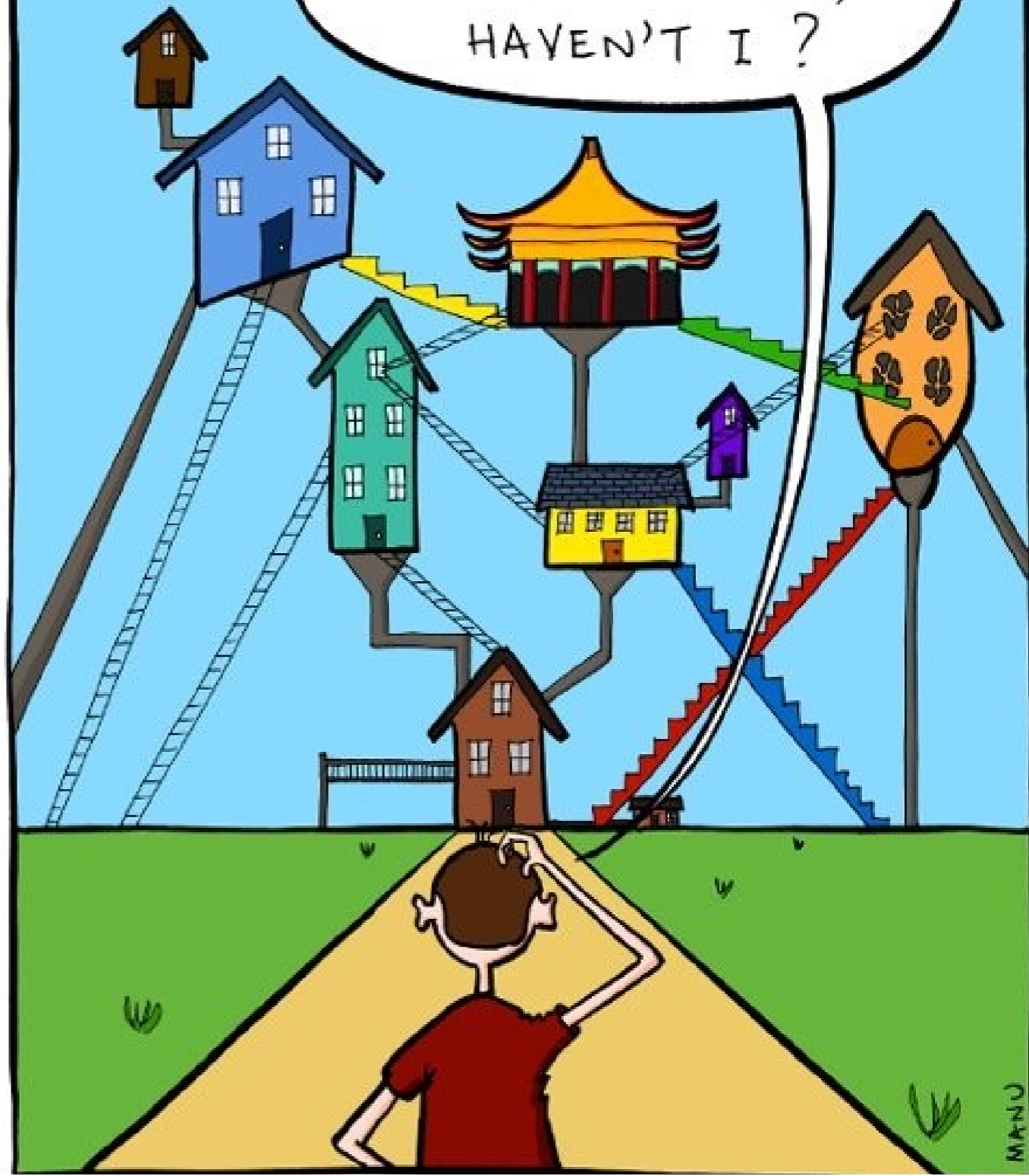
THE LIFE OF A SOFTWARE
ENGINEER.

CLEAN SLATE. SOLID
FOUNDATIONS. THIS TIME
I WILL BUILD THINGS THE
RIGHT WAY.

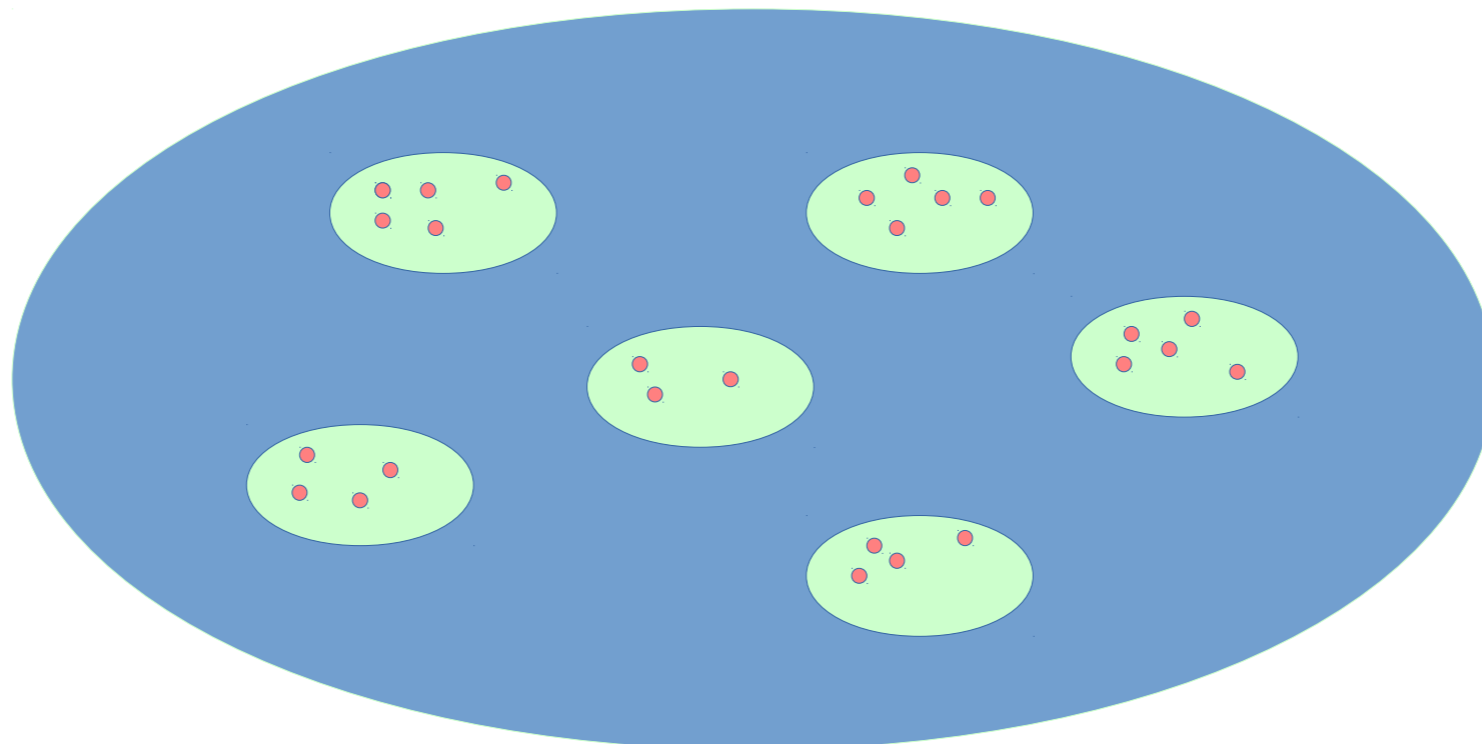


MUCH LATER...

OH MY. I'VE
DONE IT AGAIN,
HAVEN'T I ?



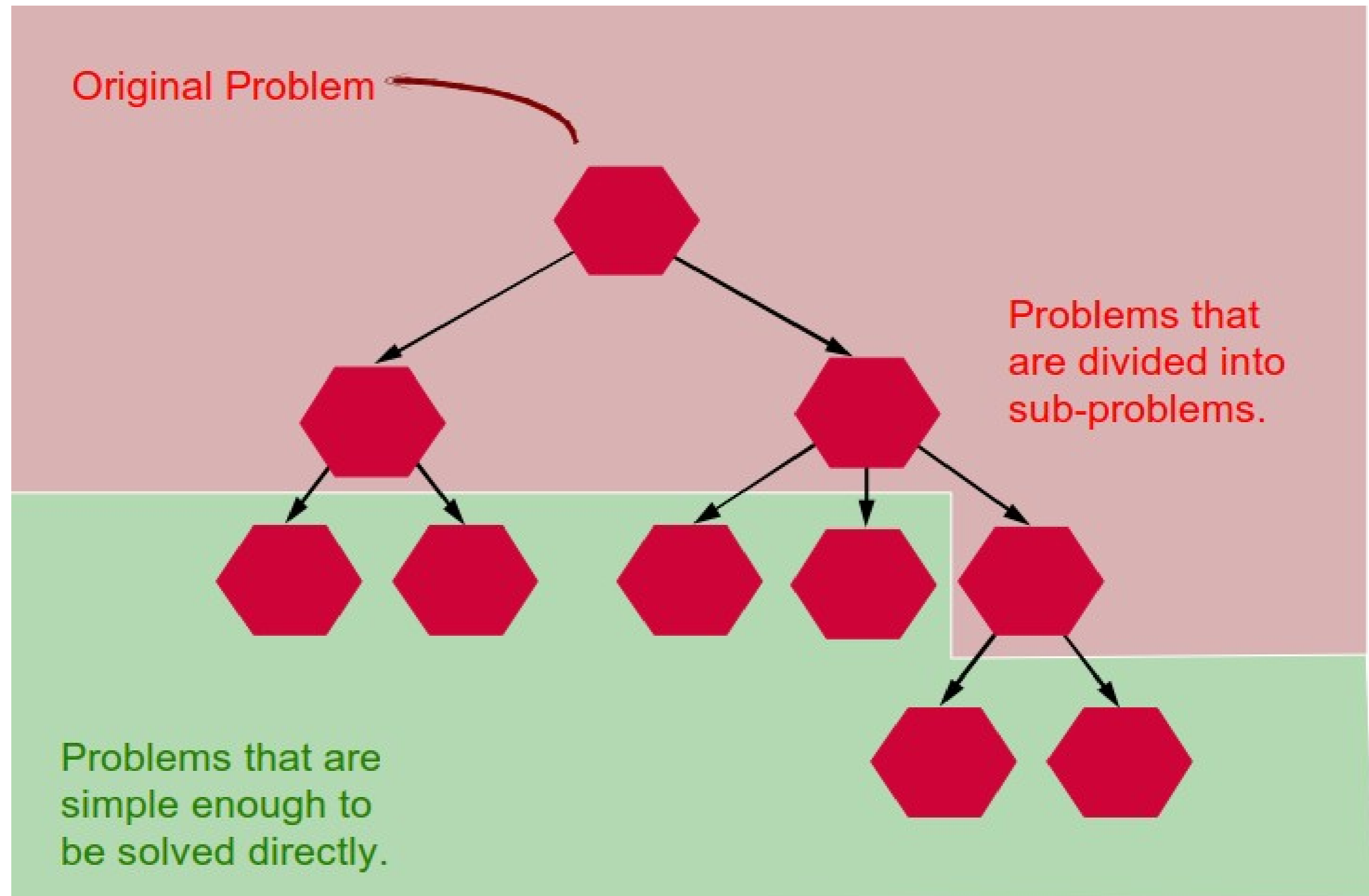
- An abstraction helps to hide some details in what we do, in order better understand other aspects.
- Abstraction is the most important tool we have to handle complexity and to find general solutions.
- Because of all the details, we cannot see the woods for the trees. Two problems can look different while on a higher level of abstraction they are identical



Top-down design

- One problem solving method which builds on using abstractions is **top-down** design
- Top-down design means that we first see the problem on a higher abstraction level and divide it into smaller and simpler problems.
- Every sub-problem is then considered on its own. This reveals more of the aspects of the original problem. On the other hand every sub-problem only affects some of the aspects and not others so we are still not overwhelmed with details.
- If necessary the sub-problems are again divided into even smaller problems, until they are so simple that can be solved immediately.
- Top-down-design is based on the principle:
divide-and-conquer

Top-down design



- Related to top-down design is **bottom-up** design
- Bottom-up design means to start with developing small and generally **reusable** programming components and then combine them to bigger and more powerful tools
- An important aspect of object oriented programming, which aligns with bottom-up design, is re-usability
- Re-usability is the strive to design classes which are so **general** that they can be used in many programs.
- In Java there is a standard library which contains a large number of such general classes; the standard library can therefore be seen as a "toolkit" from which we can pick up components for the system that we want to build.
- For the development of Java programs we usually combine top-down and bottom-up design.

- In a Java program:
abstraction mechanisms = classes + methods
- Top-down design in Java =
division it into classes and methods, which on the other hand are divided into new classes and methods.
- We should strive for a ***modular design*** where every sub-problem (= class or method) does a well defined *task* and is as independent from the others as possible.

Modular design

- A well done modular design means that the system is divided in clearly **identifiable abstractions**. The assumptions with such a system are:
 - It is easy to modify
 - The components can be reused
 - There is a clear division of responsibilities between the components
 - The complexity is reduced
 - The components can be replaced
 - Simplifies testing
 - Allows parallel development



Design systems around *stable abstractions* and *exchangeable* components, in order to allow small and stepwise changes.

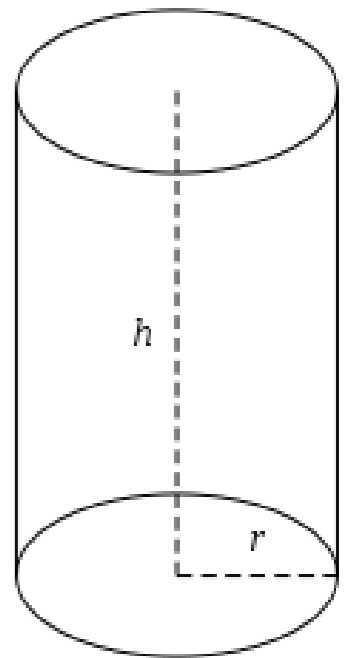
Example

Problem: write a program which reads the radius and the height of a cylinder, then computes and prints the area and the volume of the cylinder. The area A and the volume V of a cylinder can be computed with the following formulas:

$$A = 2\pi rh + 2\pi r^2 \qquad V = \pi r^2 h$$

where r is the radius and h is the height of the cylinder

- Algorithm:
 1. Read the radius of the cylinder r
 2. Read the height of the cylinder h
 3. Compute the area of the cylinder
 4. Compute the volume of the cylinder
 5. Print the area A and the volume V





- ```
import javax.swing.*;
import java.util.*;

public class CalcCylinder {
 public static void main(String[] args) {
 String input = JOptionPane.showInputDialog("Give radius and hight:");
 if (input != null) {
 Scanner sc = new Scanner(input);
 double radius = sc.nextDouble();
 double height = sc.nextDouble();
 double area = 2 * Math.PI * radius * height +
 2 * Math.PI * Math.pow(radius, 2);
 double volume = Math.PI * Math.pow(radius, 2) * height;
 JOptionPane.showMessageDialog(null, "The area of the cylinder is "
 + area +
 "\nThe volume of the cylinder is "
 + volume);
 }
 }
}
```



- [illegible]

# Solution 2, continuation

```
private static double computeArea(double radius, double height) {
 return 2 * Math.PI * radius * height +
 2 * Math.PI * Math.pow(radius, 2);
}

private static double computeVolume(double radius, double height) {
 return Math.PI * Math.pow(radius, 2) * height;
}
} // Slut av klassen CalcCylinderV2
```

- Comments: we have declared the methods `computeArea` and `computeVolume` as **private** because they are helper methods to let the main program to do its task.

- We can divide the problem further by isolating sub-problems

```
private static double computeArea(double radius, double height) {
 return computeSideArea(radius, height) + 2 * computeCircleArea(radius);
}

private static double computeVolume(double radius, double height) {
 return computeCircleArea(radius) * height;
}

private static double computeSideArea(double radius, double height) {
 return computeRectangleArea(computeCircleLength(radius), height);
}

private static double computeCircleArea(double radius) {
 return Math.PI * Math.pow(radius, 2);
}

private static double computeCircleLength(double radius) {
 return 2 * Math.PI * radius;
}

private static double computeRectangleArea(double width, double height) {
 return width*height;
}
```

# Solution 4

- In the previous solutions we had a class which contains both the main program and the private class methods `computeArea`, `computeVolume`, `computeSideArea`, etc.
- It is possible (and useful) to put these methods in another class and then then the external methods must be made *public*.

```
public class Cylinder {
 public static double computeArea(double radius, double height) {
 return computeSideArea(radius, height) + 2 * computeCircleArea(radius);
 }

 public static double computeVolume(double radius, double height) {
 return computeCircleArea(radius) * height;
 }

 private static double computeSideArea(double radius, double height) {
 return computeRectangleArea(computeCircleLength(radius), height);
 }

 private static double computeCircleArea(double radius) {
 return Math.PI * Math.pow(radius, 2);
 }
}
```

What are the advantages of this design?

# Solution 4, continuation

```
import javax.swing.*;
import java.util.*;

public class CalcCylinderV4 {
 public static void main(String[] args) {
 String input = JOptionPane.showInputDialog("Give radius and hight:");
 if (input != null) {
 Scanner sc = new Scanner(input);
 double radius = sc.nextDouble();
 double height = sc.nextDouble();
 double area = Cylinder.computeArea(radius, height);
 double volume = Cylinder.computeVolume(radius, height);
 JOptionPane.showMessageDialog(null, "The area of the cylinder is "
 + area +
 "\nThe volume of the cylinder is"
 + volume);
 }
 }
}
```

**Note:** for the program to work the class `Cylinder` must be in the same folder as the class `CalcCylinderV4`

Pause (15 min)

---

# Methods

---

# The parts of a method

```
// Methods which return values
modifier type name(parameter list) {
 variables and statements
 return result;
}
```

```
// Methods which don't return value
modifier void name(parameter list) {
 variables and statements
}
```

- Methods can be ***class methods*** or ***instance methods***
- Methods can return a value or do not return a value
- Methods can also be `private` or `public`

# The parts of a method

- The statement

```
return x;
```

terminates the method and the value `x` will be the result which is returned from the method.

- A method which does not return value (a `void`-method) have no `return`-statement or have `return`-statement which doesn't have a result.

type of return value

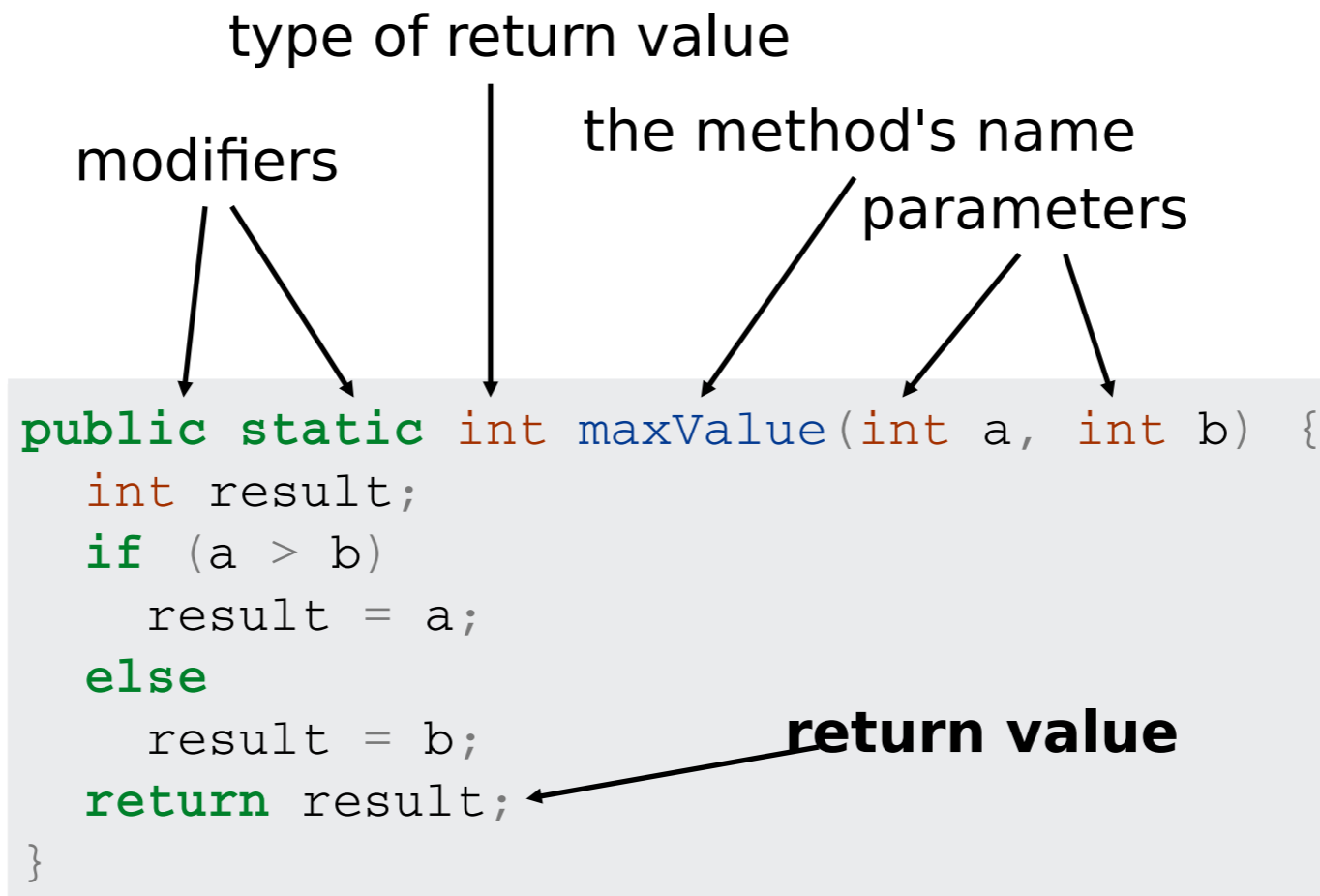
modifiers

the method's name

parameters

```
public static int maxValue(int a, int b) {
 int result;
 if (a > b)
 result = a;
 else
 result = b;
 return result;
}
```

return value

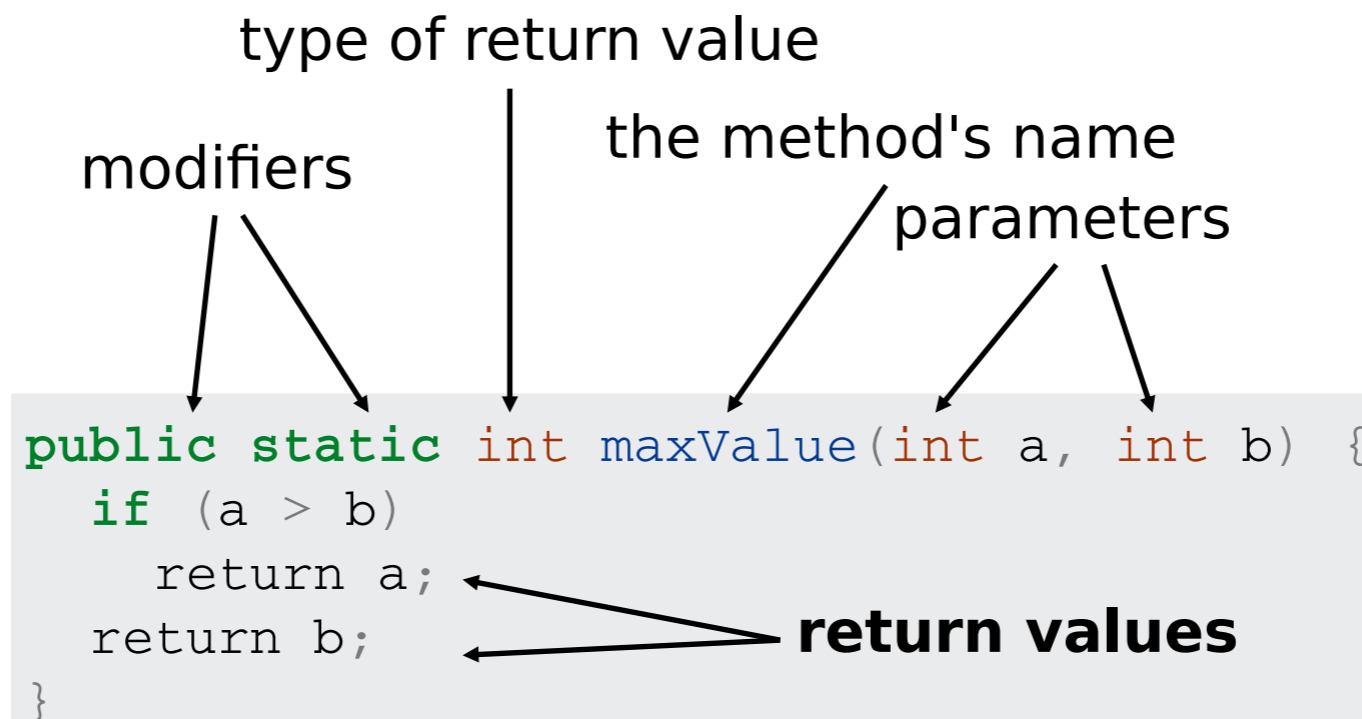


# The parts of a method

- It is important to remember that:

`return x;`

terminates the method, because the following example also works:



The diagram illustrates the components of a Java method signature. It shows a code snippet with arrows pointing to specific parts and labels:

- modifiers**: Points to `public` and `static`.
- type of return value**: Points to `int`.
- the method's name**: Points to `maxValue`.
- parameters**: Points to `int a` and `int b`.
- return values**: Points to the `return a;` and `return b;` statements inside the method body.

```
public static int maxValue(int a, int b) {
 if (a > b)
 return a;
 return b;
}
```

```
public static int maxValue(int a, int b);
```

- A methods **signature** consists of:
  - the method's *name*
  - the method's *return type*
  - the method's *parameter list* with their types and order
  - whether it is a class or instance method
- A method call can be seen as a message that the sender sends to the receiver
- The parameter list describes what type of data the sender can send in the message. Conversely, the result type describes what type of answer the sender gets in response from the receiver.



# Formal and actual parameters

```
import javax.swing.*;
import java.util.*;

public class Example {
 public static void main(String[] args) {
 String input = JOptionPane.showInputDialog("Give three integers");
 Scanner sc = new Scanner(input);
 int value1 = sc.nextInt();
 int value2 = sc.nextInt();
 int value3 = sc.nextInt();
 int big = maxValue(value1, value2);
 big = maxValue(big, value3);
 JOptionPane.showMessageDialog(null,
 "The biggest of the integers " + value1 + ", " + value2 +
 " and " + value3 + " is " + big);
 }

 public static int maxValue(int a, int b) {
 int result;
 if (a > b)
 result = a;
 else
 result = b;
 return result;
 }
}
```

actual parameters

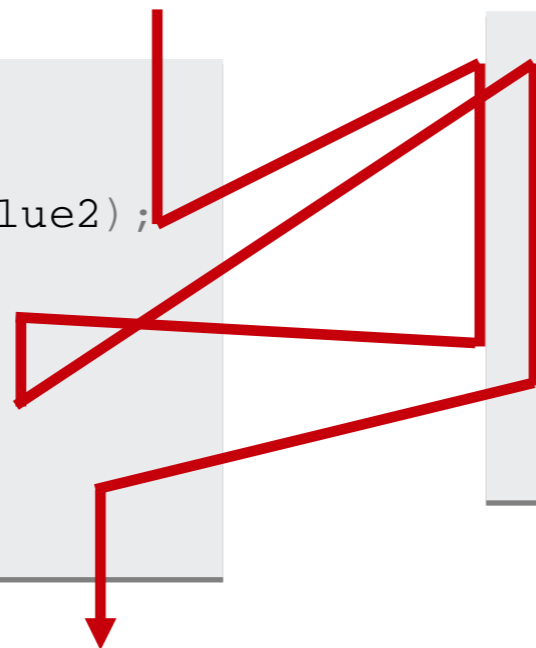
formal parameters

- With a call to a method the following happens:
  - the values of the actual parameters are copied to the corresponding formal parameters
  - the execution continues with the first statement in the called method.
- when the execution of the called method is completed, the execution of the caller is resumed.

## Execution order

```
...
int big = maxValue(value1, value2);
...
big = maxValue(big, value3);
...
```

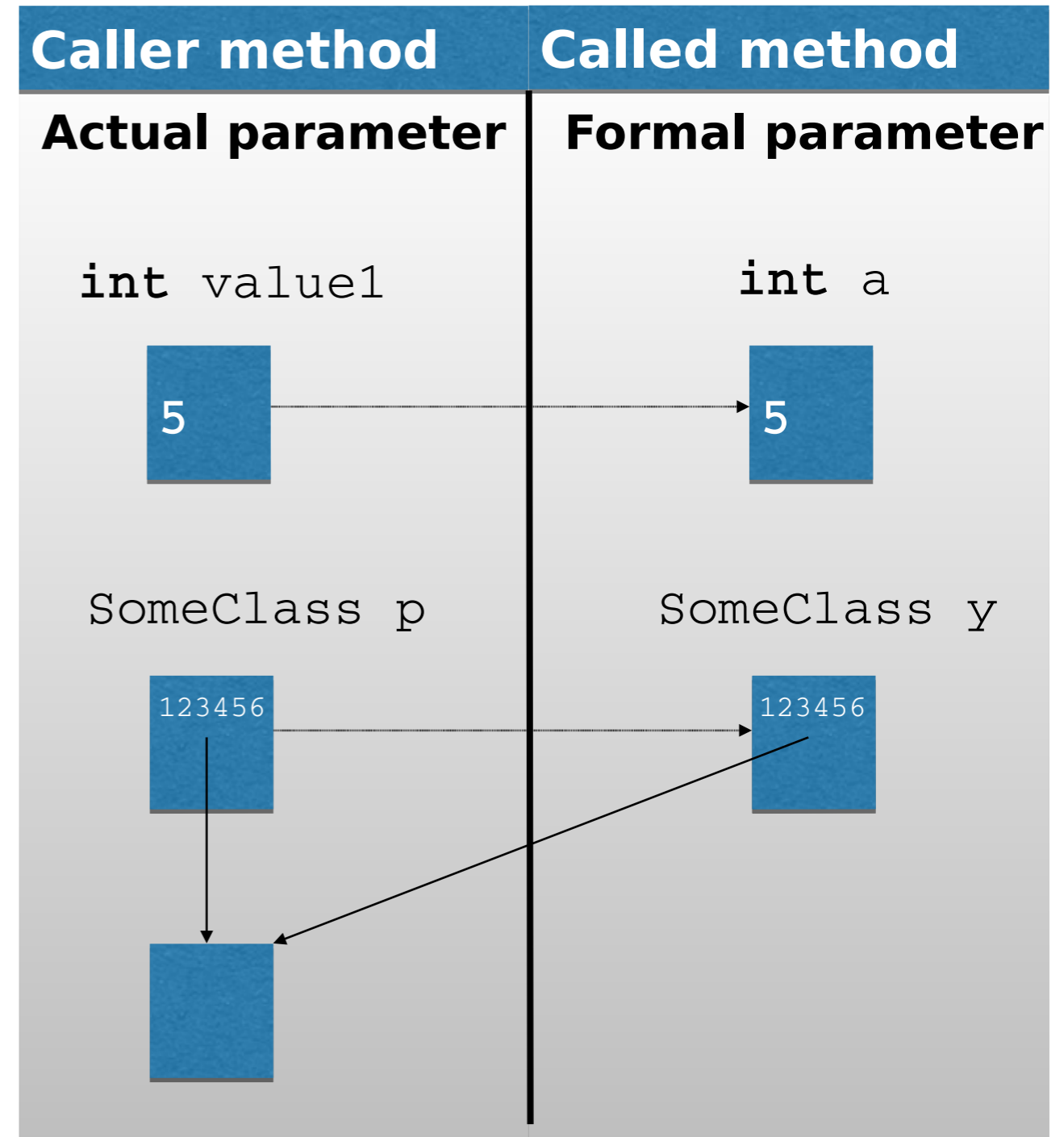
```
public static int maxValue(int a, int b) {
 int result;
 if (a > b)
 result = a;
 else
 result = b;
 return result;
}
```



- All primitive data types and all classes can be used in the parameter list and/or as a result type
- In Java the parameter passing is always by value, but what constitutes a value is different for classes and primitive types:
  - When the actual parameter is a primitive type the value is for example an integer like 1 or 2, or a real number like 3.15, or a boolean constant like `true` or `false`.
  - When the parameter is an object (an instance of a class) then the value in the parameter is just a reference to the object. This means that the actual and the formal parameter have access to the same physical *object*.

# Parameter passing (example)

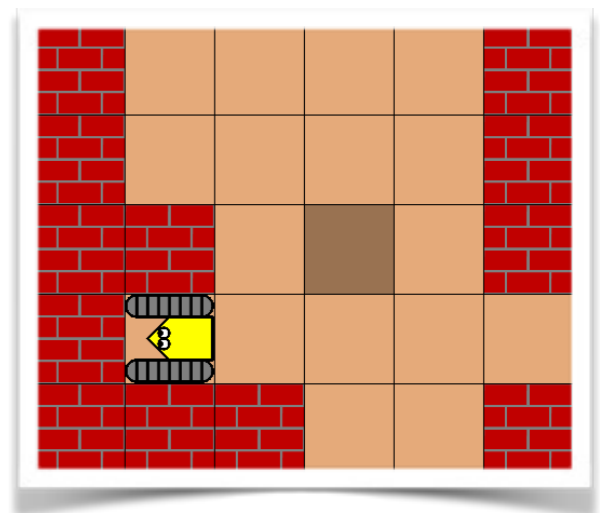
- The value of the actual parameter `value1` is copied to the formal parameter `x`
- `value1` and `x` are different physical locations
- A change in the value of `a` doesn't affect the value in the variable `value1`
- The value of the actual parameter `p` is copied to the formal parameter `y`
- `p` and `y` are going to refer to the *same* physical object
- A change in the object which is referenced by the variable `y` affects the object which is referenced by the variable `p`, since they are the same objects



# Lab 2

---

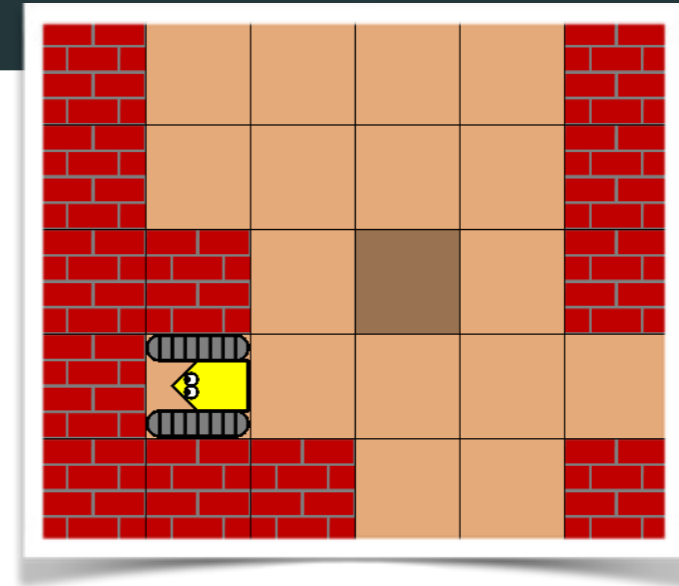
- In Lab 2 we will program a robot which is modelled with the given class `Robot`
- The goal with the lab is to break the task that the robot must do into sub-problems and to build abstractions which can be implemented as simple and reusable methods – with the help of the operations that the robot provides
- The robot lives in a simple world and can only do these simple operations:



| Method                              | Use                                                                                                                                               |
|-------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>void move()</code>            | <b>Move one step forward, if the robot ends up outside of the world, or in a wall this causes an execution error</b>                              |
| <code>boolean frontIsClear()</code> | <b>returns <i>true</i> if it is possible for the robot to <code>move()</code> without causing execution error, otherwise returns <i>false</i></b> |
| <code>void turnLeft()</code>        | <b>rotate 90° to the left</b>                                                                                                                     |
| <code>void makeLight()</code>       | <b>Lights up the the current position. If the position is already lighted up, an execution error occurs.</b>                                      |
| <code>boolean onDark()</code>       | <b>returns <i>true</i> if the robot stays on a dark position, otherwise returns <i>false</i></b>                                                  |
| <code>int getDirection()</code>     | <b>returns the robot's direction</b>                                                                                                              |

# Lab 2 (example)

- Suppose that the situation is like on the picture. The goal is to move the robot to the dark position. In that case an operation for turning around would be a useful abstraction.

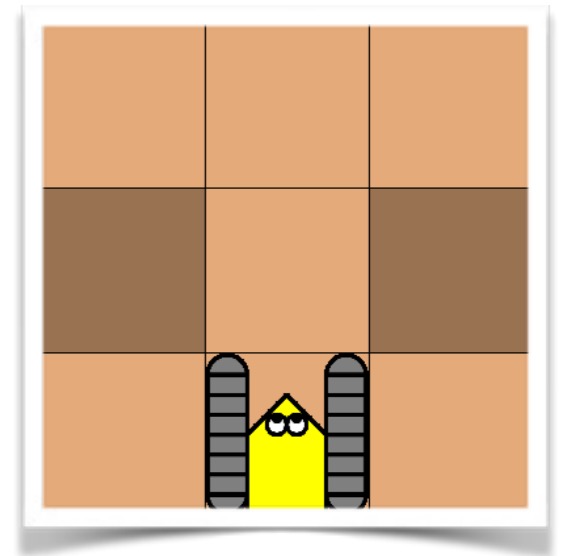


- The robot doesn't have an operation `turnAround`, instead it must be implemented with two calls to the operation `turnLeft` or two calls to `turnRight`.
- In the lab the robot is an instance variable called `robot`. This means that the abstraction `turnAround` must be implemented as an instance method:

```
// before: none
// after: the robot is facing the opposite direction
public void turnAround() {
 robot.turnLeft();
 robot.turnLeft();
}
```

# Preconditions and postconditions

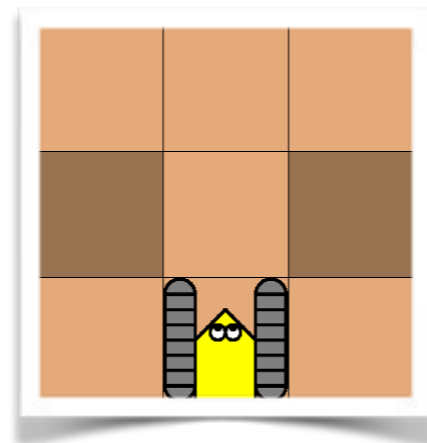
- Suppose that we have a situation like on the picture. Our task is to move the robot to the light position in front of the robot and make it dark. This can be done with the abstraction `moveAndDarken`.
- The implementation has the following look:



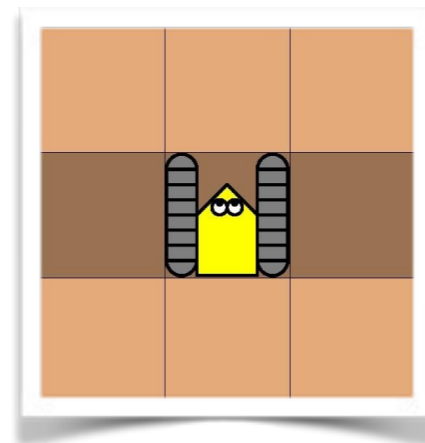
```
public void moveAndDarken() {
 robot.move();
 if (!robot.onDark())
 robot.makeDark();
}
```

# Preconditions and postconditions

- The method (the abstraction) works perfectly for the scenario from which we started:

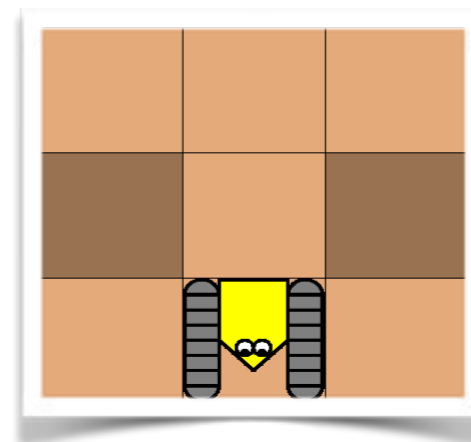
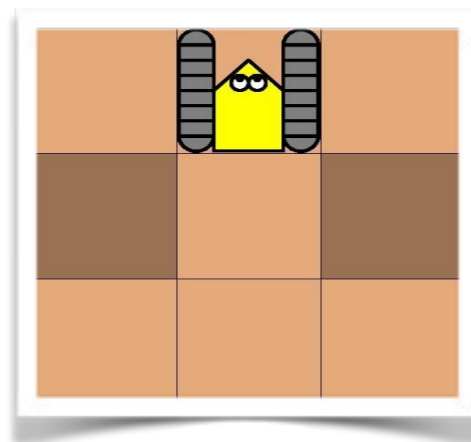
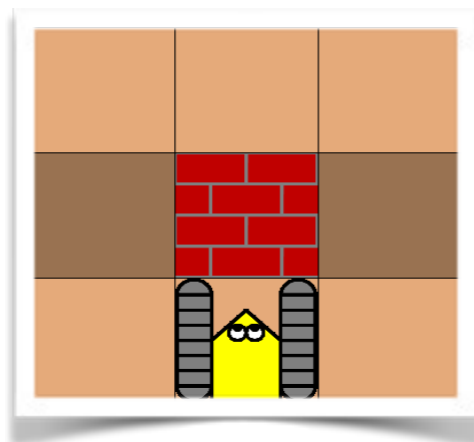


*Before*



*After*

- But there are several scenarios where the method doesn't work, for example:



# Preconditions and postconditions

- When we use a method we must know what assumptions are made in the implementation of the method, i.e. conditions that must be valid for the method to work correctly. These assumptions are called **preconditions** and must be specified.
- It is also important to know if the method causes any *side effects* when it is executed. The goal with the method `moveAndDarken` is to move the robot to the next position forward and to make it dark. This means that the robot will be on a new position after the execution of the method? This is a side effect which must be specified in a **postcondition**.
- There must be a way for the caller to check the precondition. In our example this can be done with the method `frontIsClear()`.

```
// before: the robot is not facing a wall or at the
// edge of the world
// after: the cell in front is dark, and the robot moved to that
// cell and has not changed direction
public void makeNorthDark() {
 robot.move();
 if (!robot.onDark())
 robot.makeDark();
}
```

# Pre- and Post-condition

- A cylinder cannot have a negative radius or negative height
- A circle cannot have a negative radius

```
public class Cylinder {
 // before: radius >= 0 && height >= 0
 // after: the area of a cylinder with assigned radius and height
 public static double computeArea(double radius, double height) {
 return computeSideArea(radius, height) + 2 * computeCircleArea(radius);
 }
 // before: radius >= 0 && height >= 0
 // after: the volume of a cylinder with assigned radius and height
 public static double computeVolume(double radius, double height) {
 return computeCircleArea(radius) * height;
 }
 // before: radius >= 0 && height >= 0
 // after: the side area of a cylinder with assigned radius and height
 private static double computeSideArea(double radius, double height) {
 return computeRectangleArea(computeCircleLength(radius), height);
 }
 // before: radius >= 0
 // after: the area of a circle with assigned radius
 private static double computeCircleArea(double radius) {
 return Math.PI * Math.pow(radius, 2);
 }
}
```

- The programmer must know the following in order to correctly use a method:
  - The method's name
  - The method's parameter list
  - The method's return type
  - What the method does:
    - which preconditions must apply
    - what postconditions (or side effects) method has
  - Complexity?
- In order to use the method, the programmer doesn't need to know how the method is implemented. It is interesting to know what the method does, not how it does it!

- @return is a predefined annotation
- @before is a self-defined annotation

- The command:

```
javadoc -tag before:a:"Before:" Cylinder.java
```

creates a documentation for the class Cylinder in the form of html-files.

- Examples of other pre-defined annotations:

- @author
- @before
- @version
- @exception
- @param

```
public class Cylinder {
 /** @before radius >= 0 && height >= 0
 * @return the area of a cylinder with assigned radius and height
 */
 public static double computeArea(double radius, double height) {
 return computeSideArea(radius, height) + 2 * computeCircleArea(radius);
 }
 /** @before radius >= 0 && height >= 0
 * @return the volume of a cylinder with assigned radius and height
 */
 public static double computeVolume(double radius, double height) {
 return computeCircleArea(radius) * height;
 }
 /** @before radius >= 0 && height >= 0
 * @return the side area of a cylinder with assigned radius and height
 */
 private static double computeSideArea(double radius, double height) {
 return computeRectangleArea(computeCircleLength(radius), height);
 }
 /** @before radius >= 0
 * @return the area of a circle with assigned radius
 */
 private static double computeCircleArea(double radius) {
 return Math.PI * Math.pow(radius, 2);
 }
}
```

## computeArea

```
public static double computeArea(double radius,
 double height)
```

### Returns:

the area of a cylinder with assigned radius and height

### Before:

radius >= 0 && height >= 0

## computeVolume

```
public static double computeVolume(double radius,
 double height)
```

### Returns:

the volume of a cylinder with assigned radius and height

### Before:

radius >= 0 && height >= 0

# Live coding: Lab 2

---

***abstraction***

***reusability***

***modular design***

***top-down***

***bottom-up***

***divide-and-conquer***

---

***class methods***

***instance methods***

**public and private**

**method signature**

**precondition**

**postcondition**

**actual and formal parameters**