Agenda

- Recap: Self-Review
- ClockDisplay: abstraction, modularization
- ClockDisplay: Object interaction
-
-

Self-review Solutions

a) List the name of this method and the name and type of its parameter:

```
public void setCredits(int creditValue)
{
    credits = creditValue;
}
```

b) Write out the outer wrapping (header) of a class called Person.

```
public class Person {
}
```

c) Write out definitions for the following instance variables (fields) of the class Person:

- A field called name of type String: `private String name;`
- A field of type int called age: `private int age;`
- A field of type String called code: `private String code;`
- A field called credits of type int: `private int credits;`

d) Write out a class and a constructor for a class called Module. The constructor should take a single parameter of type String called moduleCode and assign it to the field code.

Self-review Solutions

Write out a class and a constructor for a class called Module. The constructor should take a single parameter of type String called moduleCode and assign it to the field code.

```
public class Module {

        private String code;

        public Module (String moduleCode){
              code= moduleCode;

        }

}
```

Self-review

e) Write out a constructor for a class called Person. The constructor should take two parameters: a String called myName and an int called myAge. The parameters should be used to set the values of the fields name and age.

```
public Person(String myName, int myAge ){
      name = myName;
      age= myAge;

}
```

Self-review

Write an accessor method called getName that returns the value of a field called name, whose type is String.

```
public String getName(){
      return name;
}
```

Correct the error in this method:

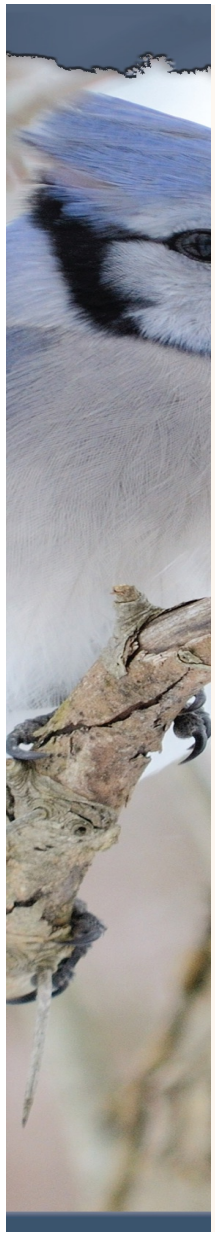
```
public ~~void~~ int getAge(){
    return age;
}
```

Variable vs parameter

Variablen :
Speicherplatz für Daten mit
z.B. Instanz variable (fields) mit Gültigkeitsbereich innerhalb des Objektes
z.B. lokale Variablen, mit Gültigkeitsbereich innerhalb einer Methode

Parameter:
Übergeben Werte an Methoden / Konstruktoren

```
public myMethod (int myParameter){
}
```



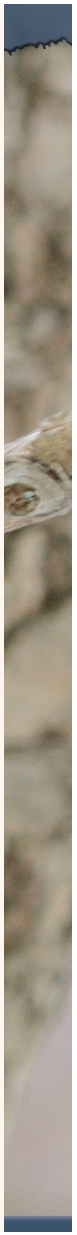
Object interaction

Creating cooperating objects

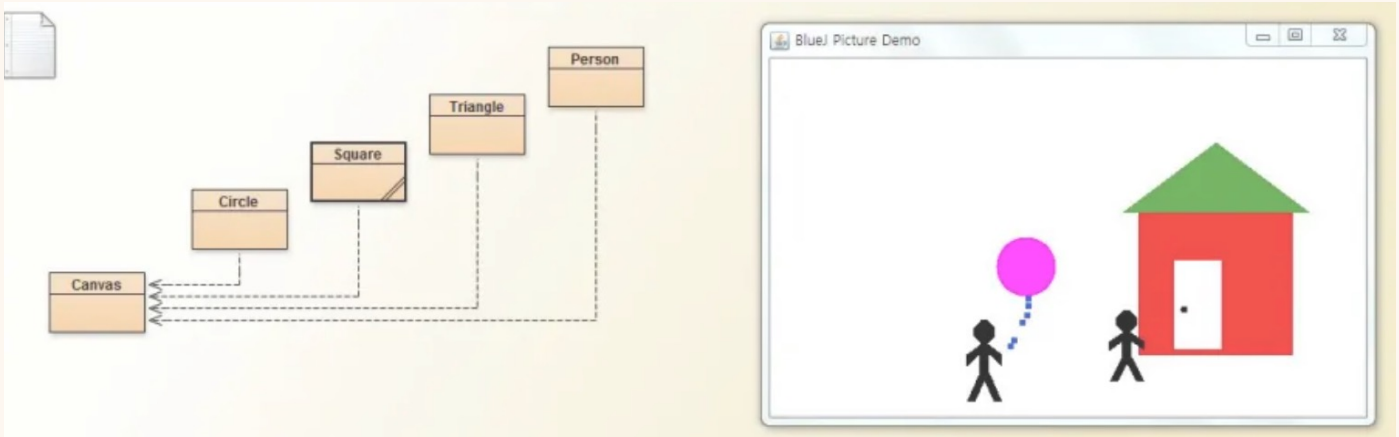
https://www.youtube.com/watch?v=H4z88Z2P1z0&list=PL9HfA4ZKbzin-tNeJi09vJxll_RiOn9eB&index=3

Debugger:

https://www.youtube.com/watch?v=AbEVfqG-sZc



Class House



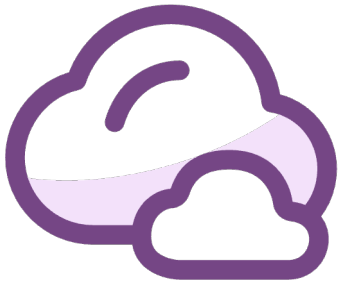
Abstraction and modularization

Abstraction is the ability to ignore details of parts to focus attention on a higher level of a problem.

Modularization is the process of dividing a whole into well-defined parts, which can be built and examined separately, and which interact in well-defined ways.

slido

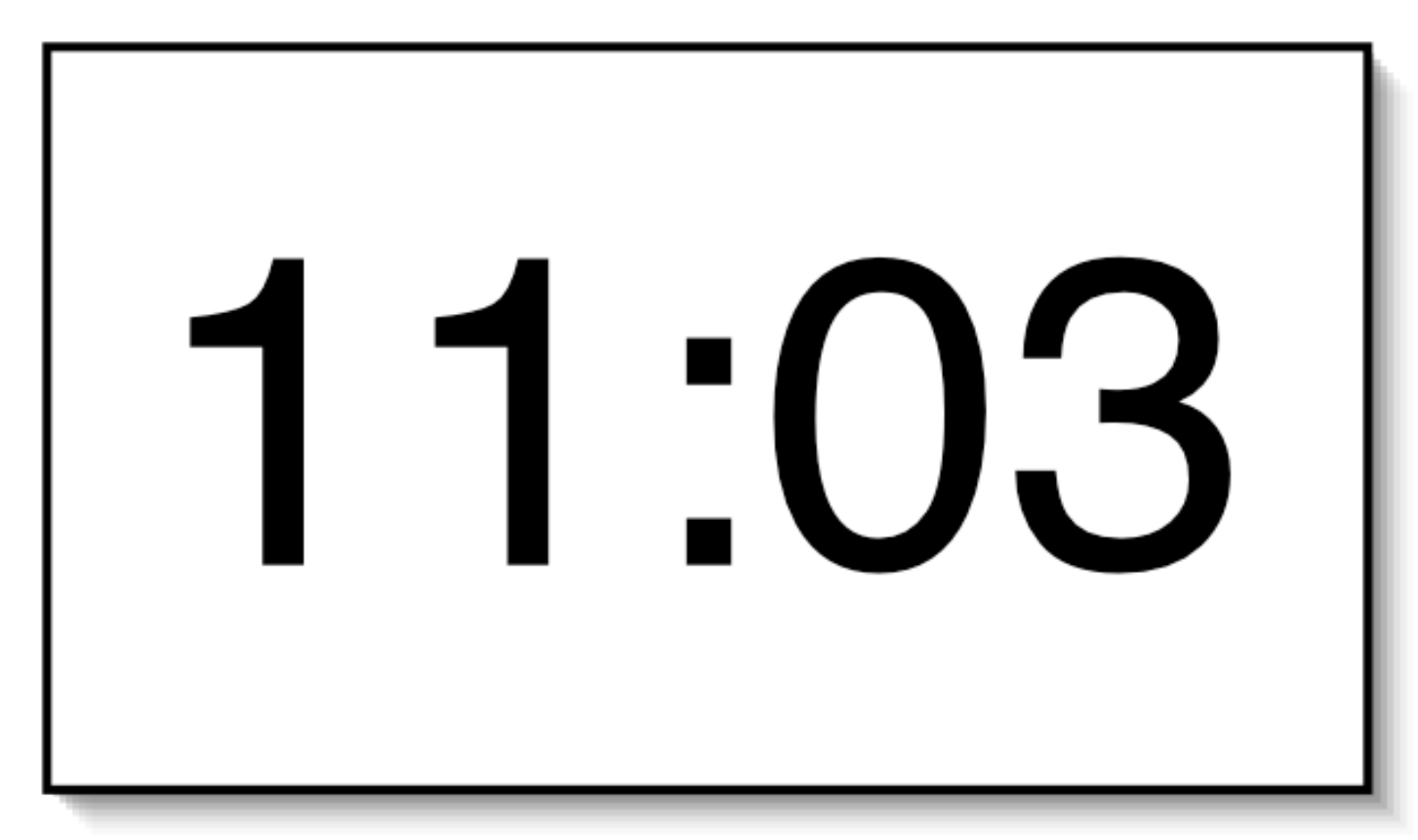
Please download and install the Slido app on all computers you use



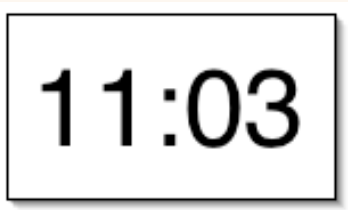
Beispiele für Modularisierung?

ⓘ Start presenting to display the poll results on this slide.

A digital clock

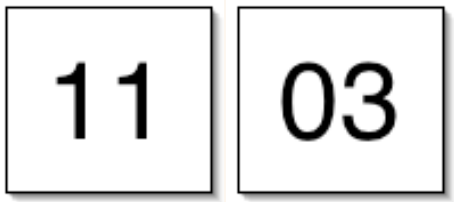


Modularizing the clock display



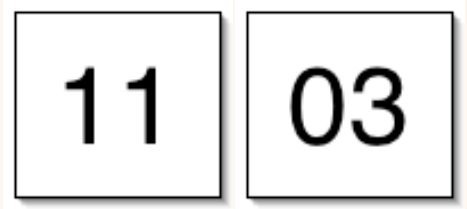
One four-digit display?

Or two two-digit displays?

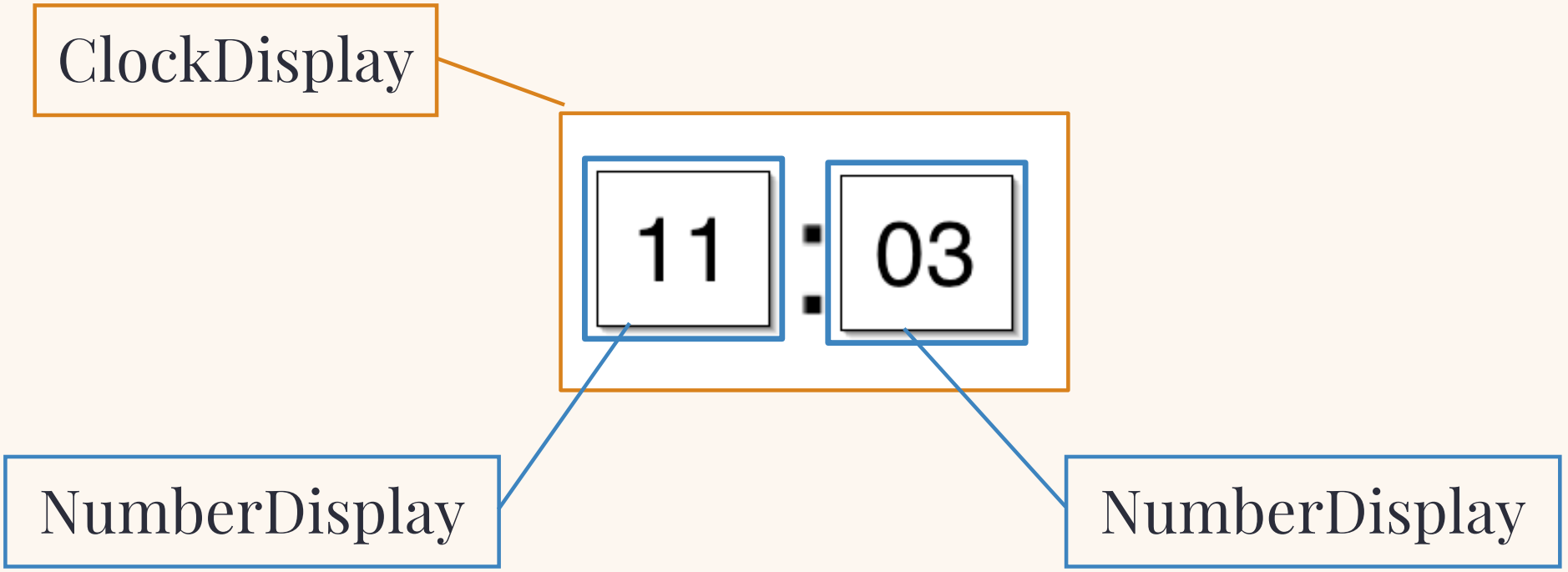


What do they have in common?

Or two two-digit displays?



Modularizing the clock display



Modeling a two-digit display

We call the class **`NumberDisplay`**.

Two integer fields:

- The current value.
- The limit for the value.

The **current value** is incremented (hochgezählt) until it reaches its **limit**.

It ‘rolls over’ to zero at this point.

Implementation - ClockDisplay

```
public class ClockDisplay
{
    private NumberDisplay hours;
    private NumberDisplay minutes;

    Constructor and
    methods omitted.
}
```

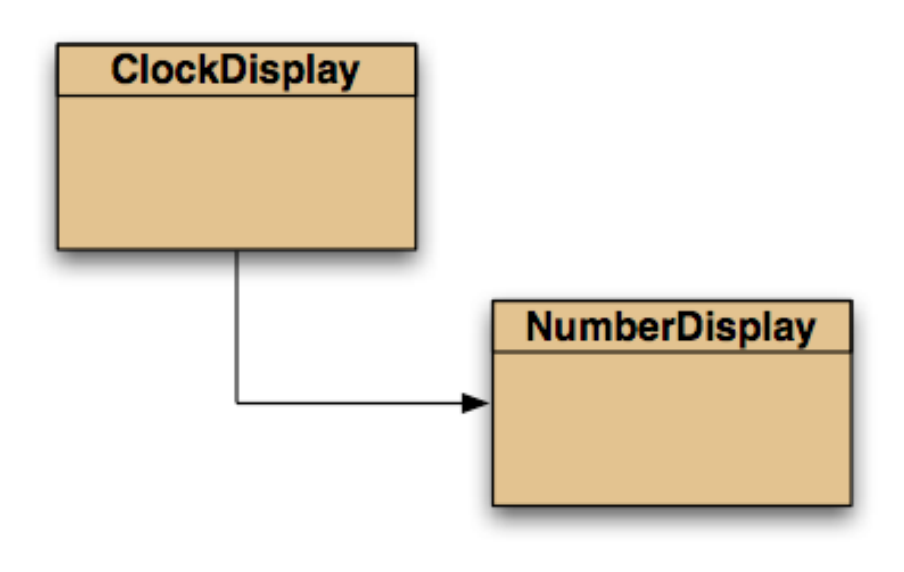
Classes as types

Data can be classified under many different types; e.g. integer, boolean, floating-point.

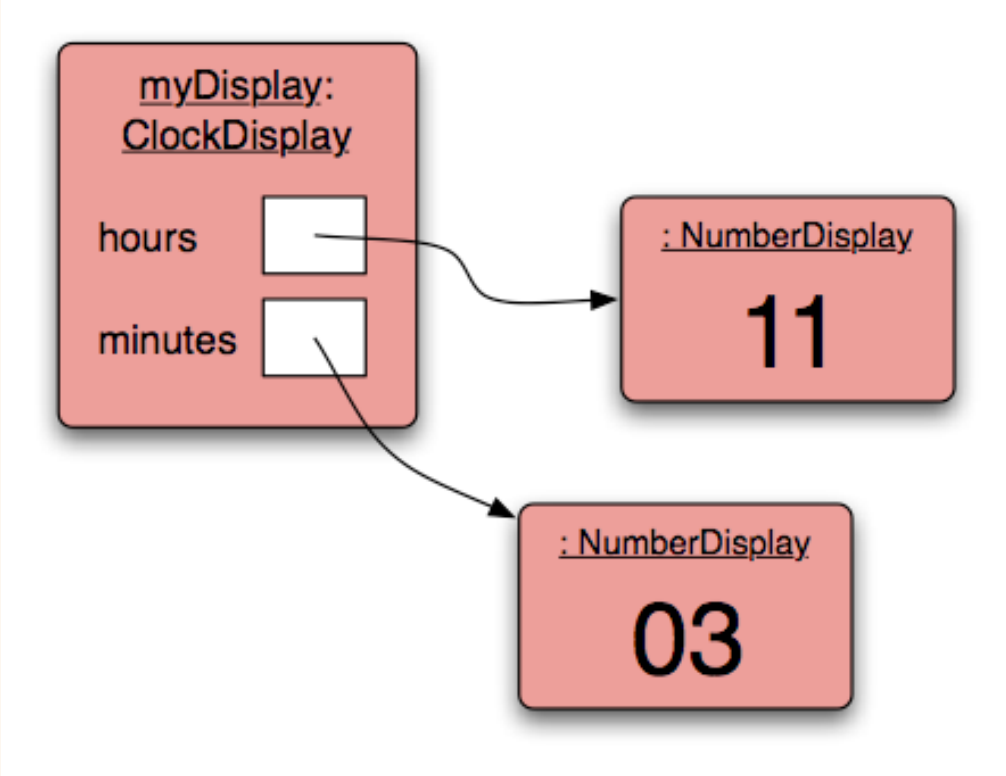
In addition, every class is a unique data type; e.g. **`String`**, **`TicketMachine`**, **`Circle`**.

Data types, therefore, can be composites and not simply values.

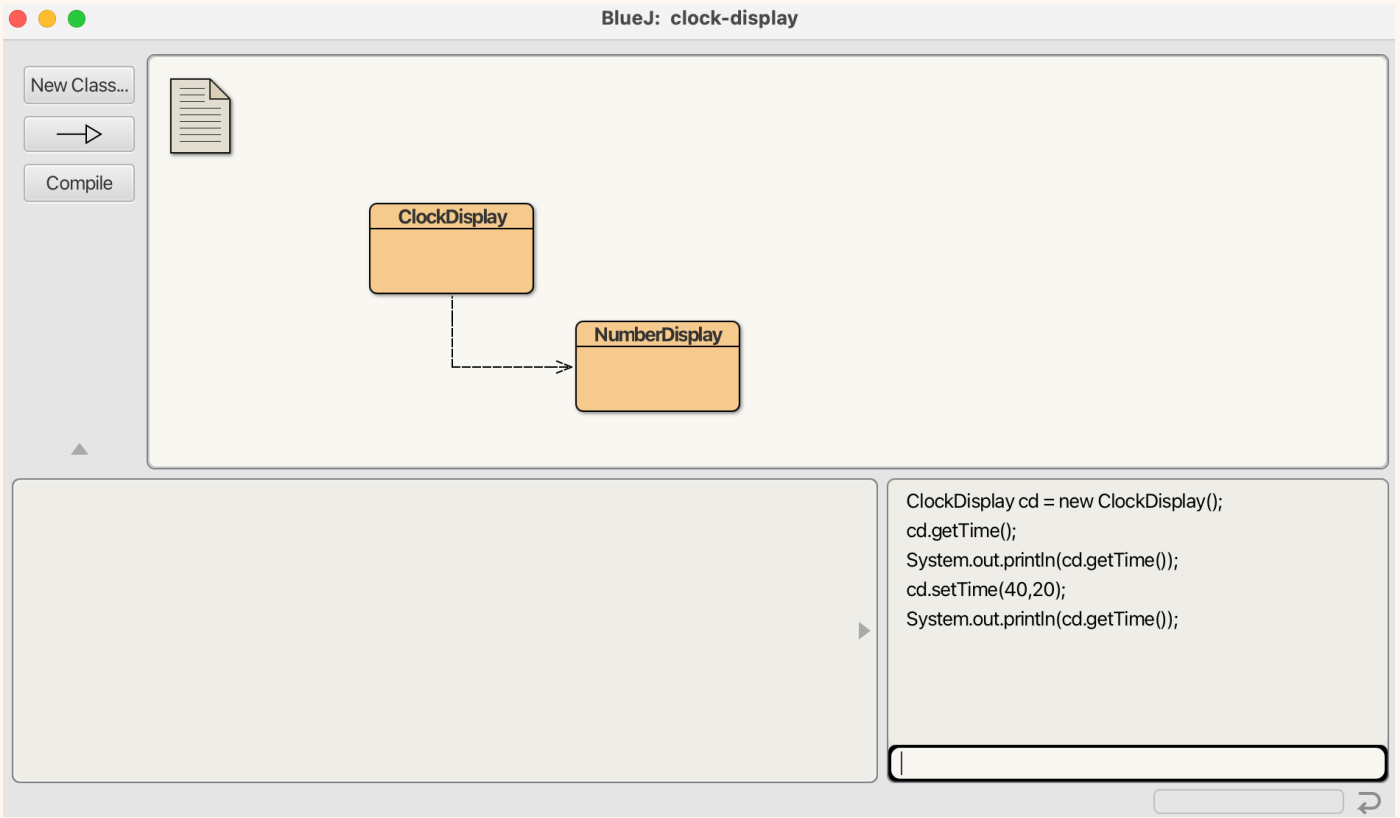
Class diagram



Object diagram



Playing with the codepad



Exploring NumberDisplay

```
public class NumberDisplay
{
    private int limit;
    private int value;

    /**
     * Constructor for objects of class NumberDisplay.
     * Set the limit at which the display rolls over.
     */
    public NumberDisplay(int rollOverLimit)
    {
        limit = rollOverLimit;
        value = 0;
    }

    /**
     * Return the current value.
     */
    public int getValue()
    {
        return value;
    }

    /**
     * Return the display value (that is, the current value as a two-digit
     * String. If the value is less than ten, it will be padded with a leading
     * zero).
     */
    public String getDisplayValue()
    {
        if(value < 10) {
            return "0" + value;
        }
        else {
            return "" + value;
        }
    }
```

Implementation - NumberDisplay

```
public class NumberDisplay
{
    private int limit;
    private int value;

    public NumberDisplay(int limit)
    {
        this.limit = limit;
        value = 0;
    }
    ...
}
```

Objects creating objects

in class ClockDisplay:

```
hours = new NumberDisplay(24);
```

actual parameter

in class NumberDisplay:

```
public NumberDisplay(int rollOverLimit);
```

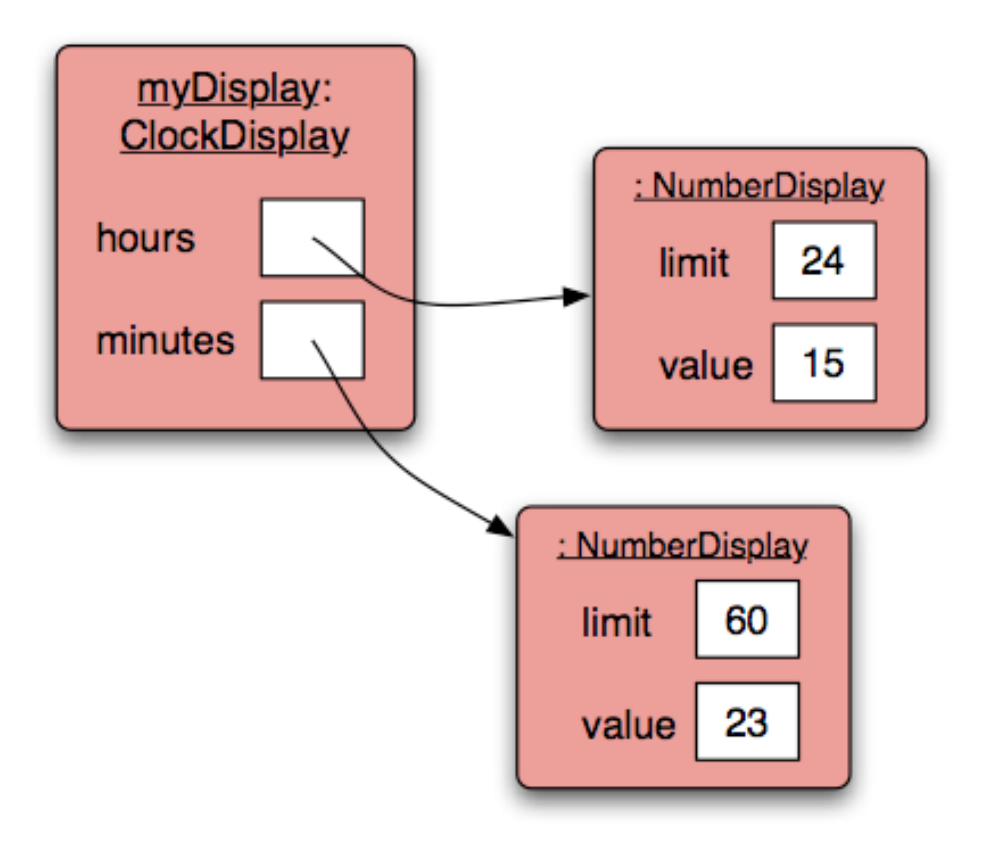
formal parameter

Objects creating objects

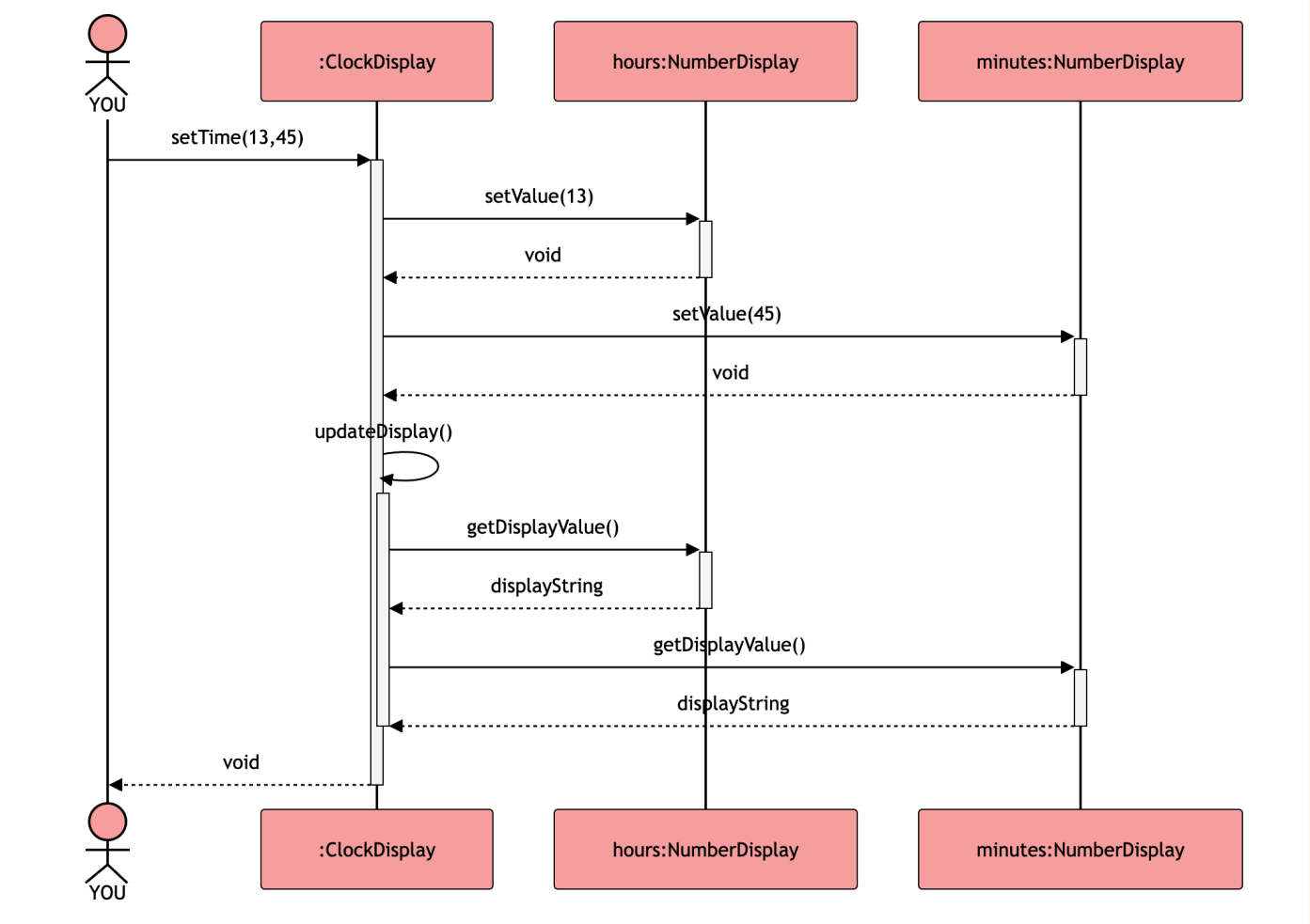
```
public class ClockDisplay
{
    private NumberDisplay hours;
    private NumberDisplay minutes;
    private String displayString;

    public ClockDisplay()
    {
        hours = new NumberDisplay(24);
        minutes = new NumberDisplay(60);
        …
    }
}
```

ClockDisplay object diagram



Object interaction



https://home.htw-berlin.de/~kleinen/material/info1/clock_display_refactoring/original_clock_display/

Source code: NumberDisplay

```
public String getDisplayValue()
{
    if(value < 10) {
        return "0" + value;
    }
    else {
        return "" + value;
    }
}
```

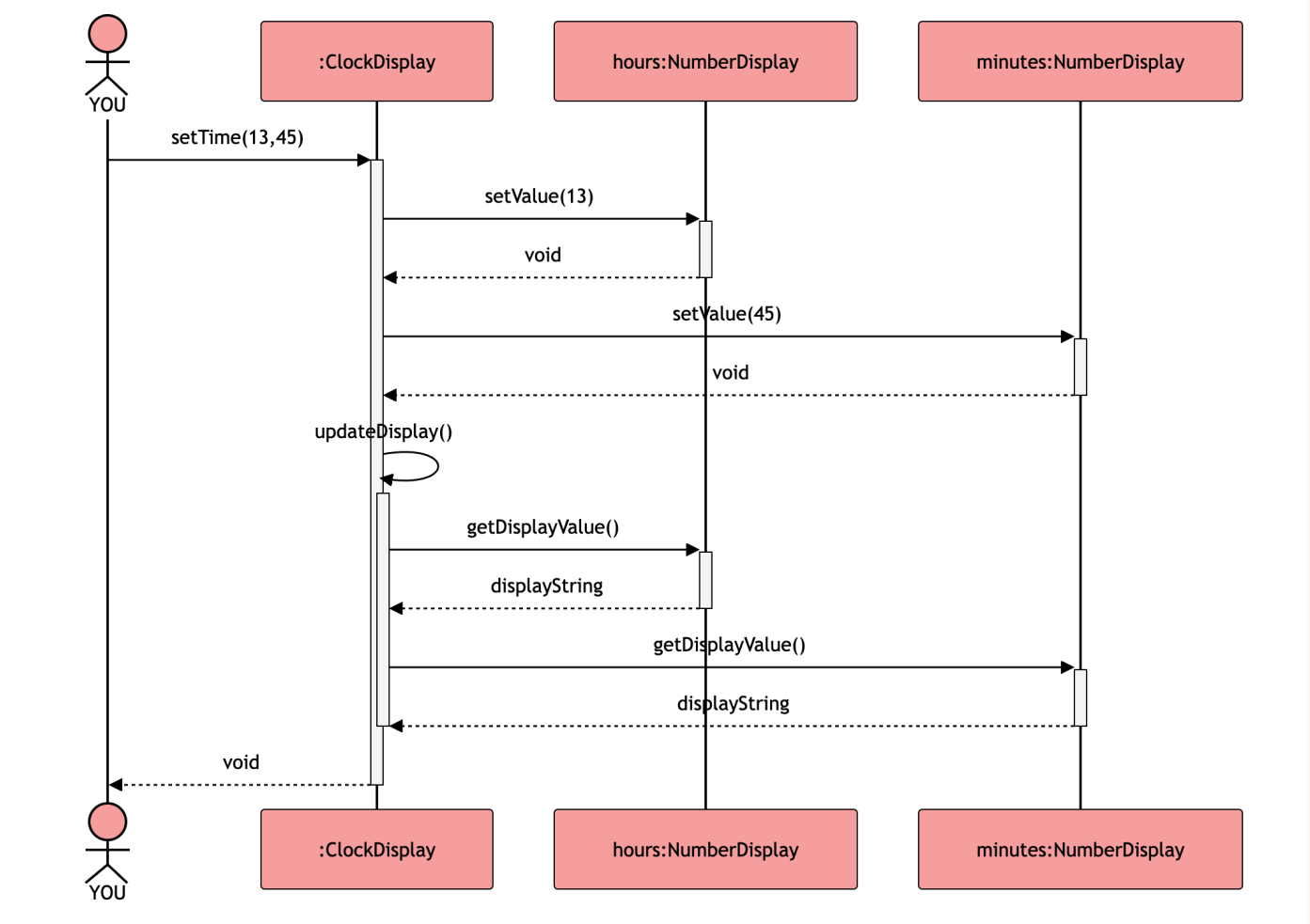
increment method

```
public void increment()
{
    value = value + 1;
    if(value == limit) {
        // Keep the value within the limit.
        value = 0;
    }
}
```

Agenda

- ClockDisplay: abstraction, modularization
- Debugger
- Variable scope
- Primitive vs object type

Object interaction



https://home.htw-berlin.de/~kleinen/material/info1/clock_display_refactoring/original_clock_display/

increment method

```
public void increment()
{
    value = value + 1;
    if(value == limit) {
        // Keep the value within the limit.
        value = 0;
    }
}
```

Alternative increment method

```
public void increment()
{
    value = (value + 1) % limit;
}
```

Check that you understand how the rollover works in this version.

Object interaction

Two objects interact when one object calls a method on another.

The interaction is usually all in one direction (cf, ‘client’, ‘server’).

The client object can ask the server object to do something.

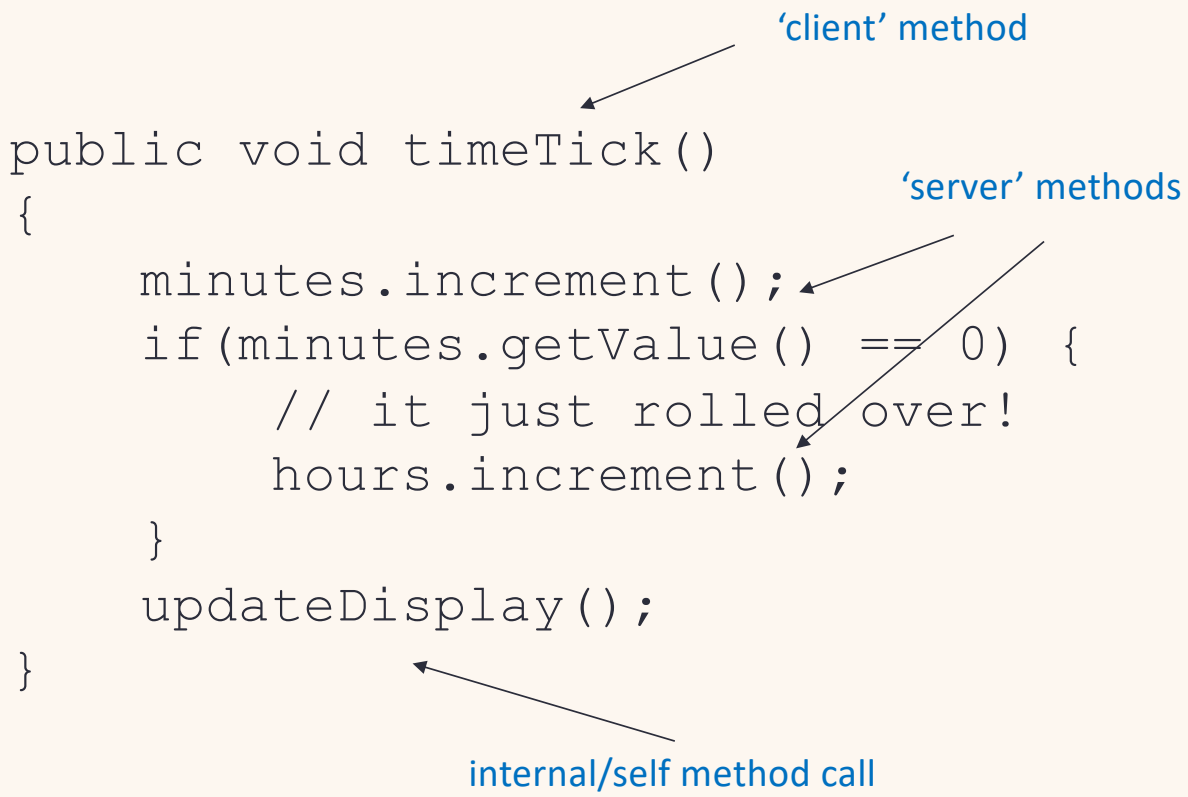
The client object can ask for data from the server object.

Object interaction

Two NumberDisplay objects store data on behalf of a ClockDisplay object.

- The ClockDisplay is the ‘client’ object.
- The NumberDisplay objects are the ‘server’ objects.
- The client calls methods in the server objects.

Method calling



External method calls

General form:

object . *methodName* (*params*);

Examples:

```
hours.increment();

minutes.getValue();
```

Internal method calls

No variable name is required:

```
updateDisplay();
```

Internal methods often have **`private`** visibility.

- Prevents them from being called from outside their defining class.

Internal method

```
/**
 * Update the internal string that
 * represents the display.
 */
private void updateDisplay()
{
    displayString =
        hours.getDisplayValue() + ":" +
        minutes.getDisplayValue();
}
```

Method calls

NB: A method call on *another object of the same type* would also be an external call.

‘Internal’ means ‘this object’.

‘External’ means ‘any other object’, regardless of its type.

The **this** keyword

Used to distinguish parameters and fields of the same name. E.g.:

```
public ClockDisplay(int limit)
{
    this.limit = limit;
    value = 0;
}
```

Variable Scope in Java

Instance Variables (Attributes)

- Usually marked as private.
- Only accessible inside the class's methods.

Example:

```
private int age; // instance variable
```

Variable Scope in Java

Local Variables

- Declared inside a method or block.
- Valid from where they’re declared until the block ends.
- Example:Usually marked as private.

Only accessible inside the class’s methods.

Example:

```
public void example() {
    int x = 5; // local variable
    // x is only valid inside this method
}
```

Variable Scope in Java

Method Parameters

- Treated like local variables.
- Valid only inside the method body.

Example:

```
public void example(String name ) {
    lastname = name
// 'name' is a parameter with local scope
}
```

Variable Shadowing

If a local variable or parameter has the same name as an instance variable, it hides (shadows) the instance variable.

- Use keyword `this` to refer to the instance variable:

```
public class Person {
    private String name;
    void setName(String name) {
        this.name = name;
     // 'this.name' = instance var, 'name' = parameter
    }
}
```

String concatenation

```
4 + 5
 9
"wind" + "ow"
 "window"
"Result: " + 6
 "Result: 6"
"# " + price + " cents"
 "# 500 cents"
```

→ overloading

- Different ways to overload the method
- By changing the no. of arguments
- By changing the datatype

Quiz

```
System.out.println(5 + 6 + "hello");
```

```
11hello
```

```
System.out.println("hello" + 5 + 6);
```

```
hello56
```

The debugger

- Useful for gaining insights into program behavior …
- … whether or not there is a program error.
- Set breakpoints.
- Examine variables.
- Step through code.

The debugger