

Programming Practice and Applications

Object interaction

Michael Kölling

Abstraction and modularisation

- **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.

A digital clock

11:03

Modularising the clock display

11:03

One four-digit display?

Or two two-digit displays?

11

03

Implementation - NumberDisplay

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

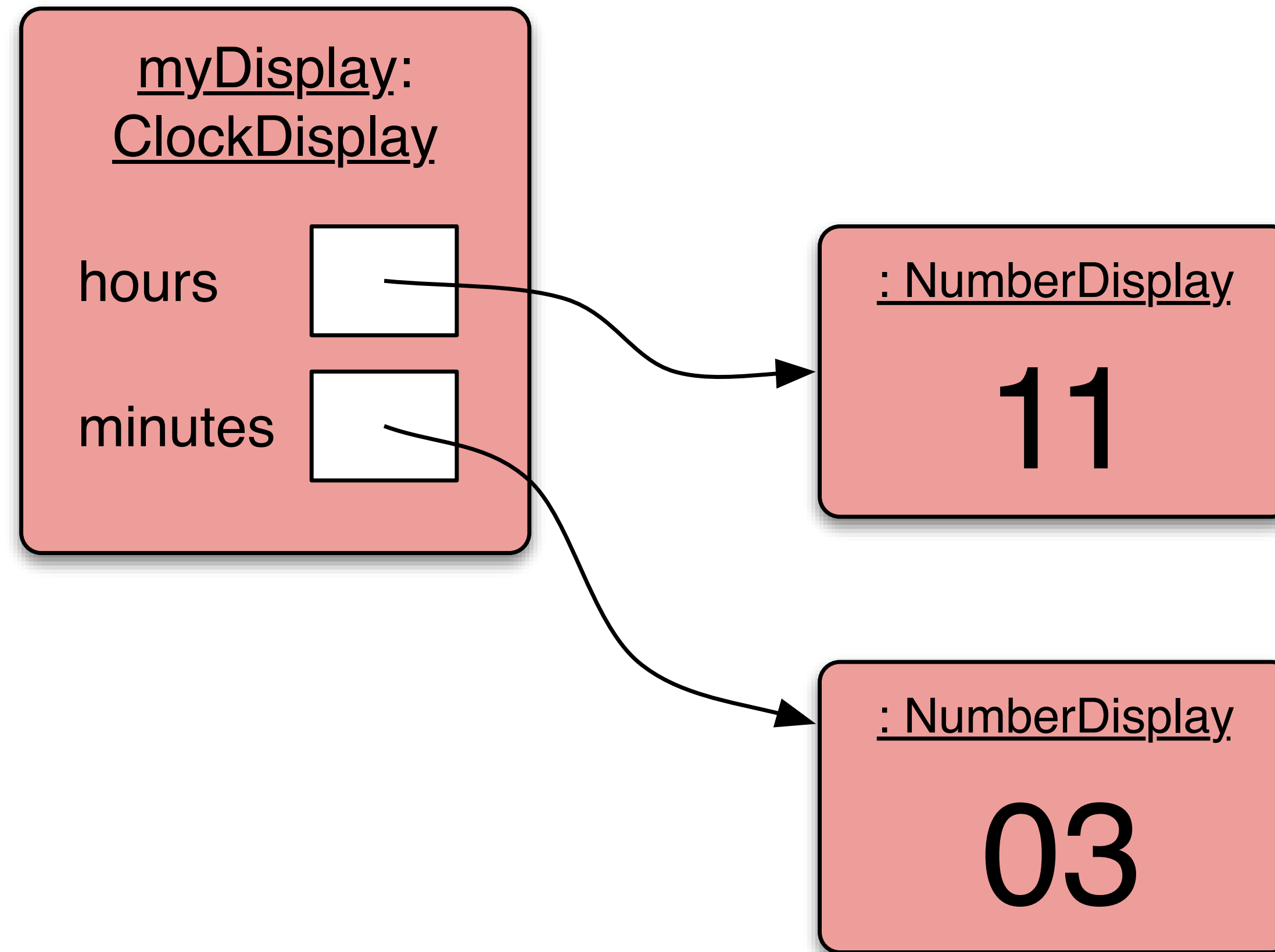
    Constructor and
    methods omitted.
}
```

Implementation - ClockDisplay

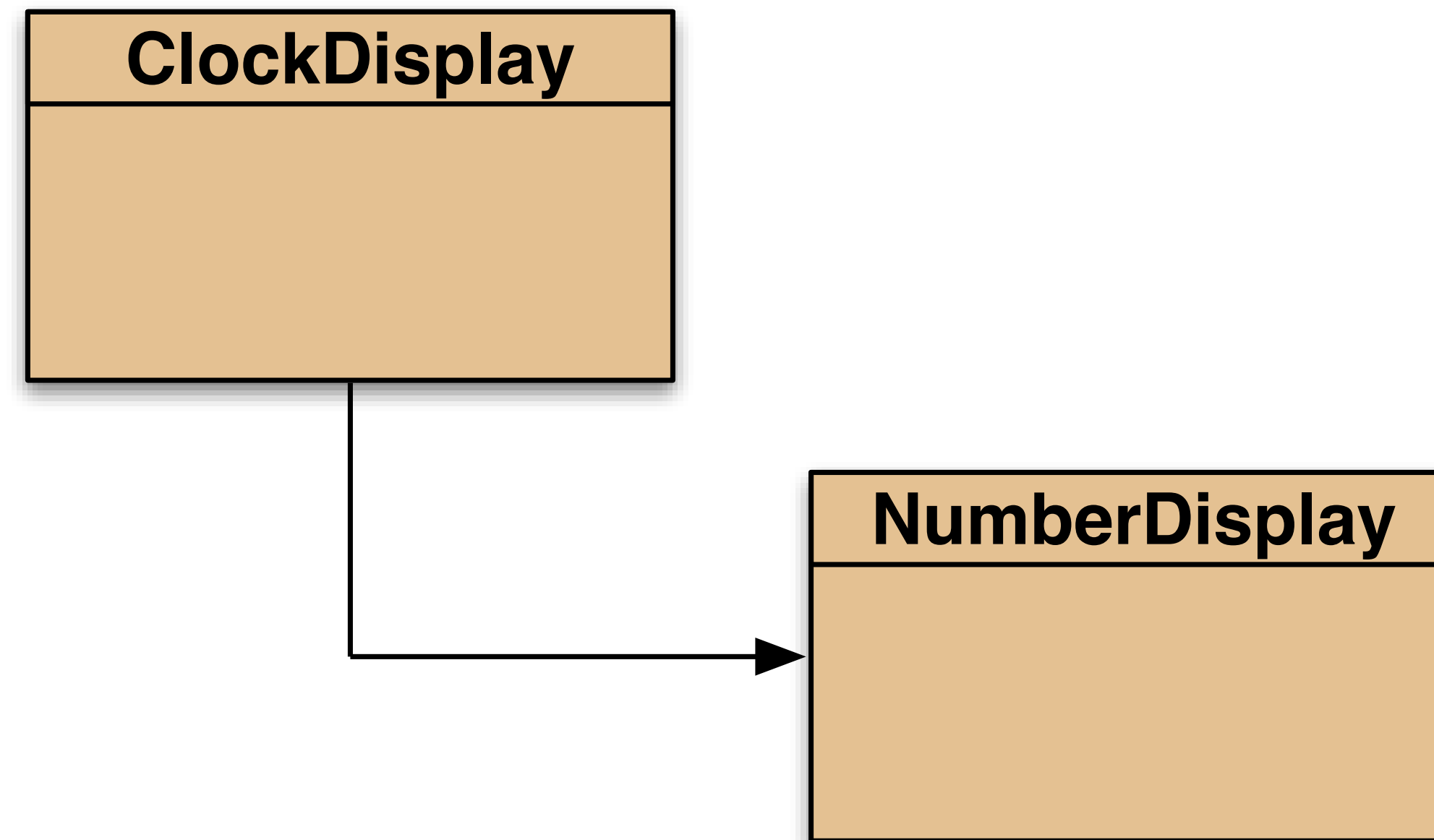
```
public class ClockDisplay
{
    private NumberDisplay hours;
    private NumberDisplay minutes;

    Constructor and
    methods omitted.
}
```

Object diagram



Class diagram



Modelling 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 until it reaches its limit.
- It ‘rolls over’ to zero at this point.

Implementation - NumberDisplay

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

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

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;  
    }  
}
```

The modulo operator

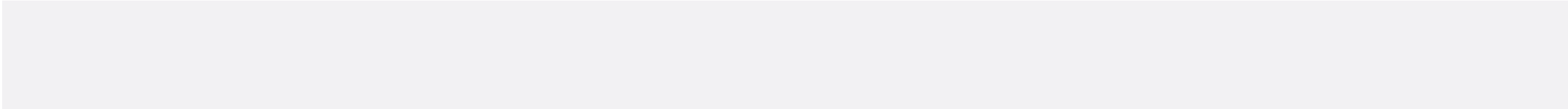
- The 'division' operator (`/`), when applied to int operands, returns the *result of an integer division*.
- The 'modulo' operator (`%`) returns the *remainder of an integer division*.
- E.g., generally:
 $17 / 5$ gives result 3, remainder 2
- In Java:
 $17 / 5 == 3$
 $17 \% 5 == 2$

increment method

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

How can this be rewritten?

Alternative increment method

```
public void increment()  
{  
      
}
```

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

Implementation - ClockDisplay

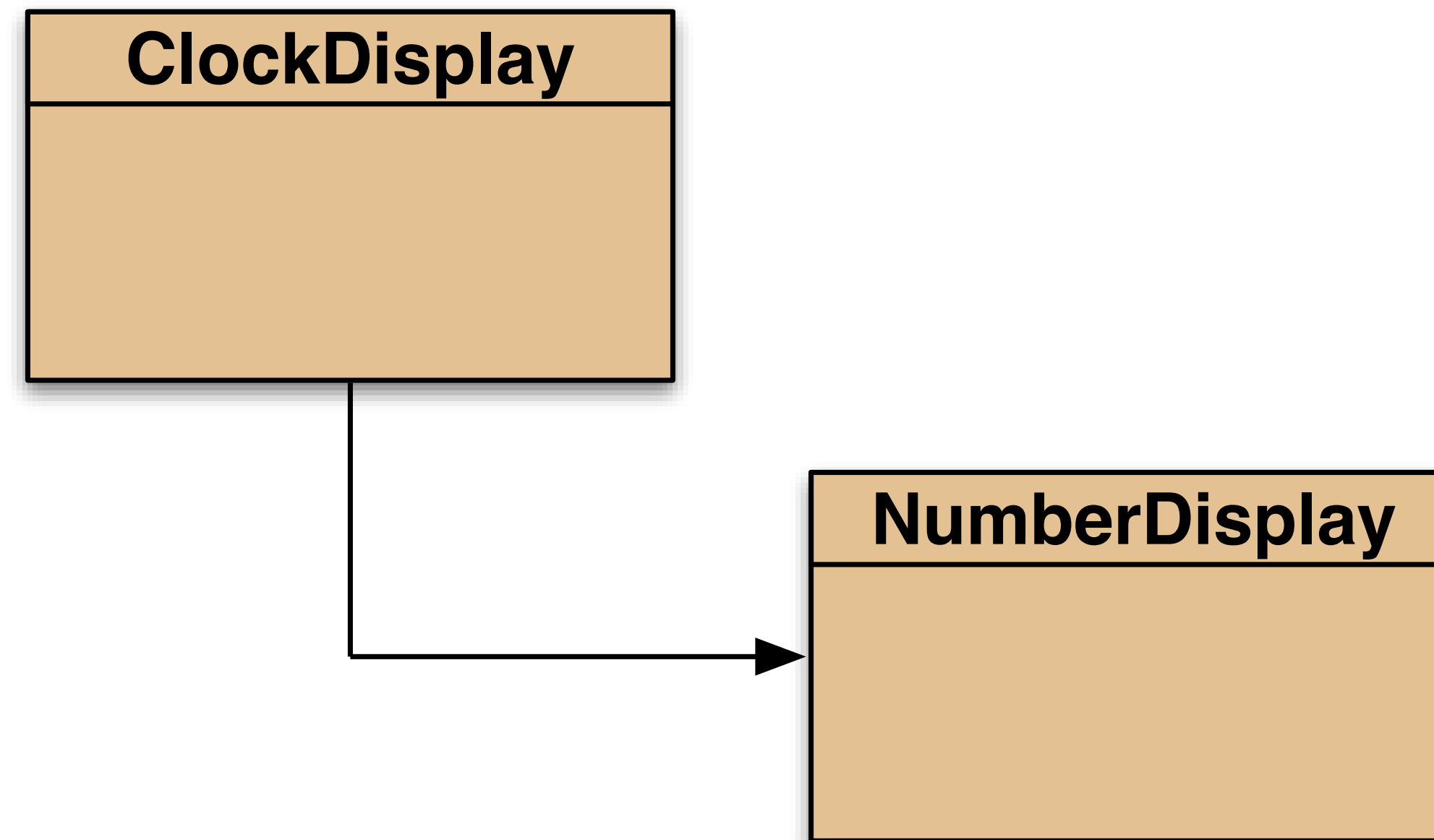
```
public class ClockDisplay
{
    private NumberDisplay hours;
    private NumberDisplay minutes;

    Constructor and
    methods omitted.
}
```

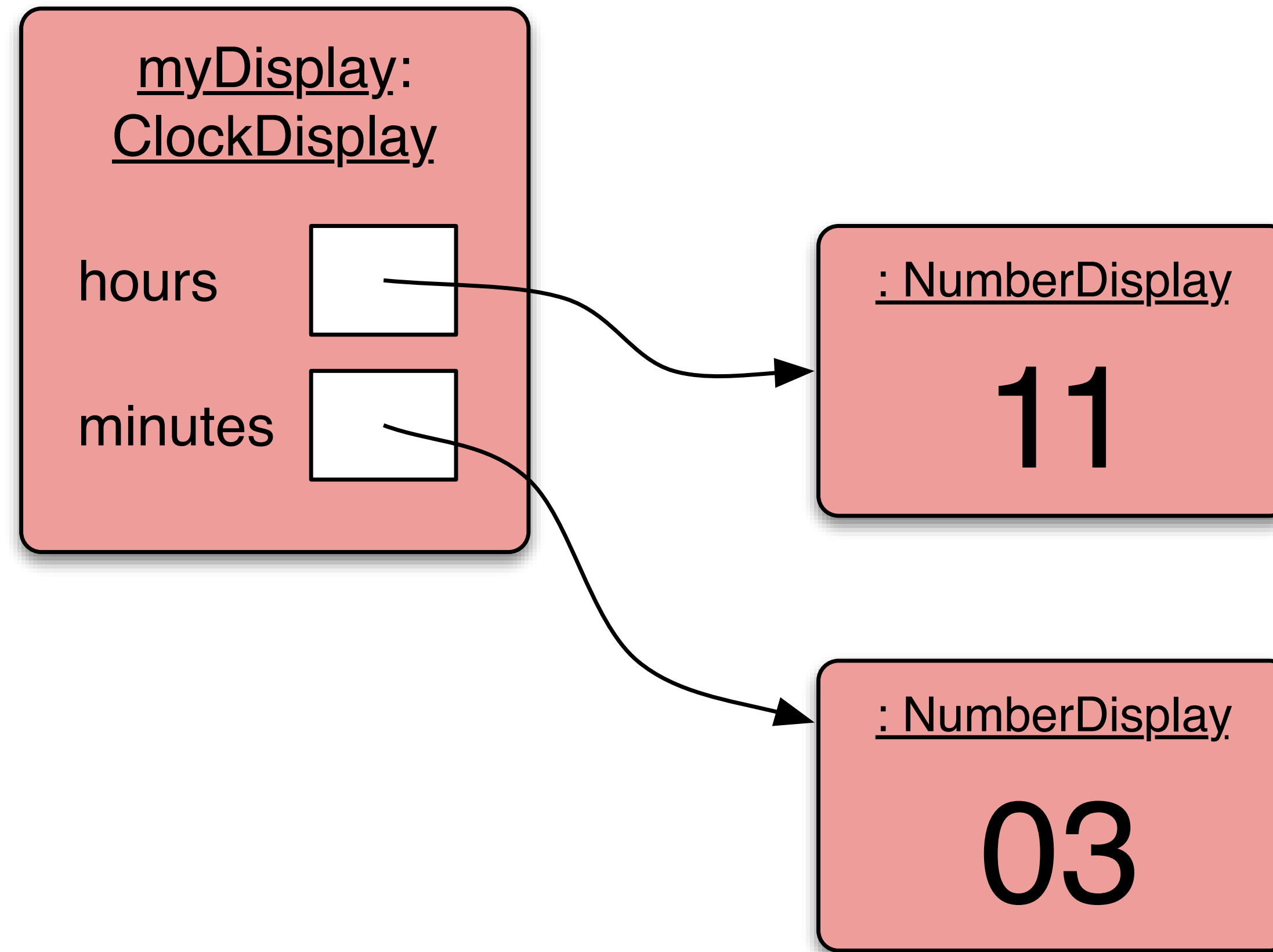

Classes as types

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

Class diagram



Object diagram



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);
        ...
    }
}
```

Objects creating objects

in class ClockDisplay:

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

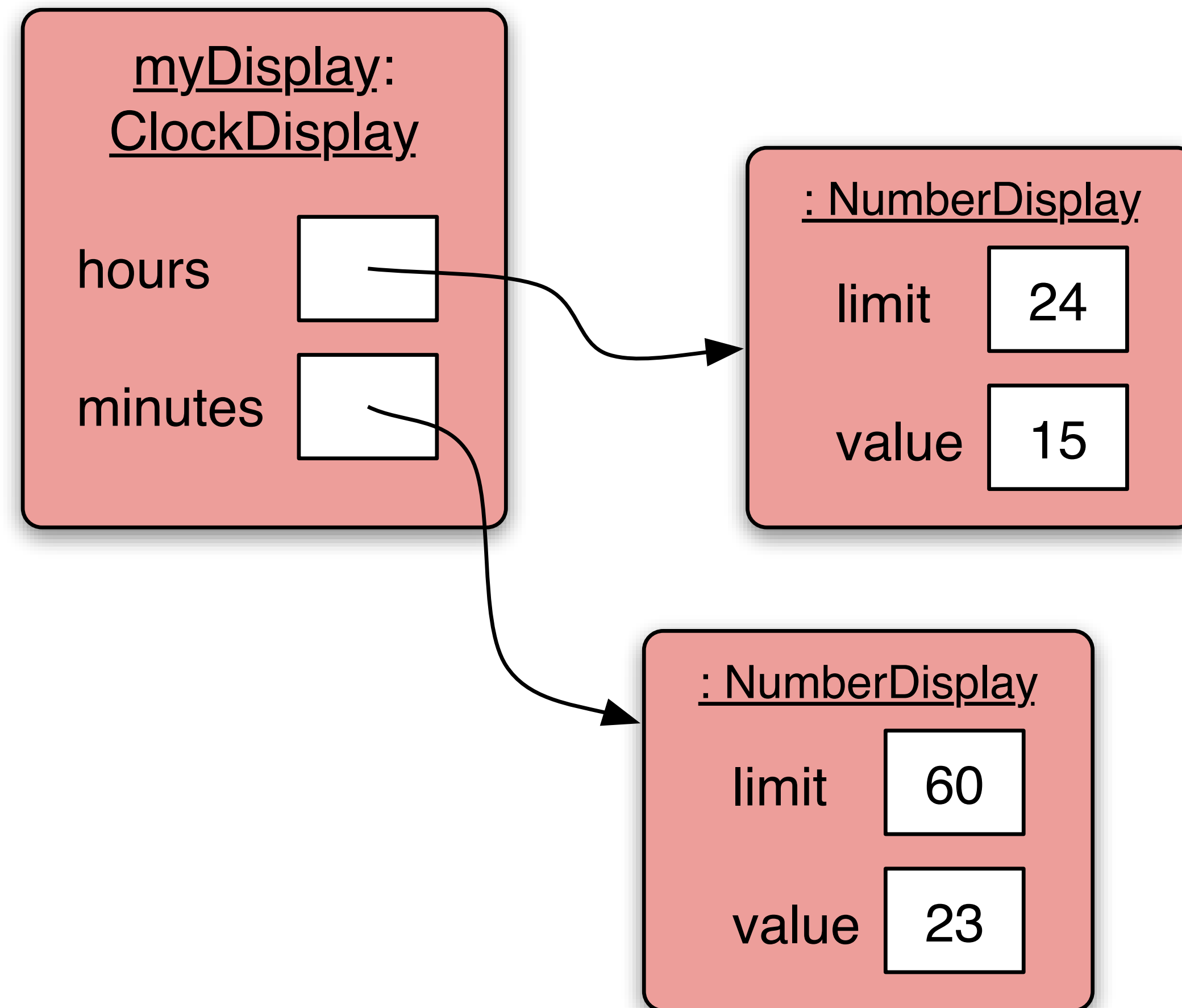
actual parameter

in class NumberDisplay:

```
public NumberDisplay(int rolloverLimit);
```

formal parameter

ClockDisplay object diagram



null

- `null` is a special value in Java
- All object variables are initialised to `null`.
- You can assign and test for `null`:

```
private NumberDisplay hours;
```

```
if(hours == null) { ... }
```

```
hours = null;
```

Object interaction

- Two objects interact when one object calls a method on another.
- The interaction is usually all in one direction.
- One object can ask another object to do something.
- One object can ask for data from the other object.

Object interaction

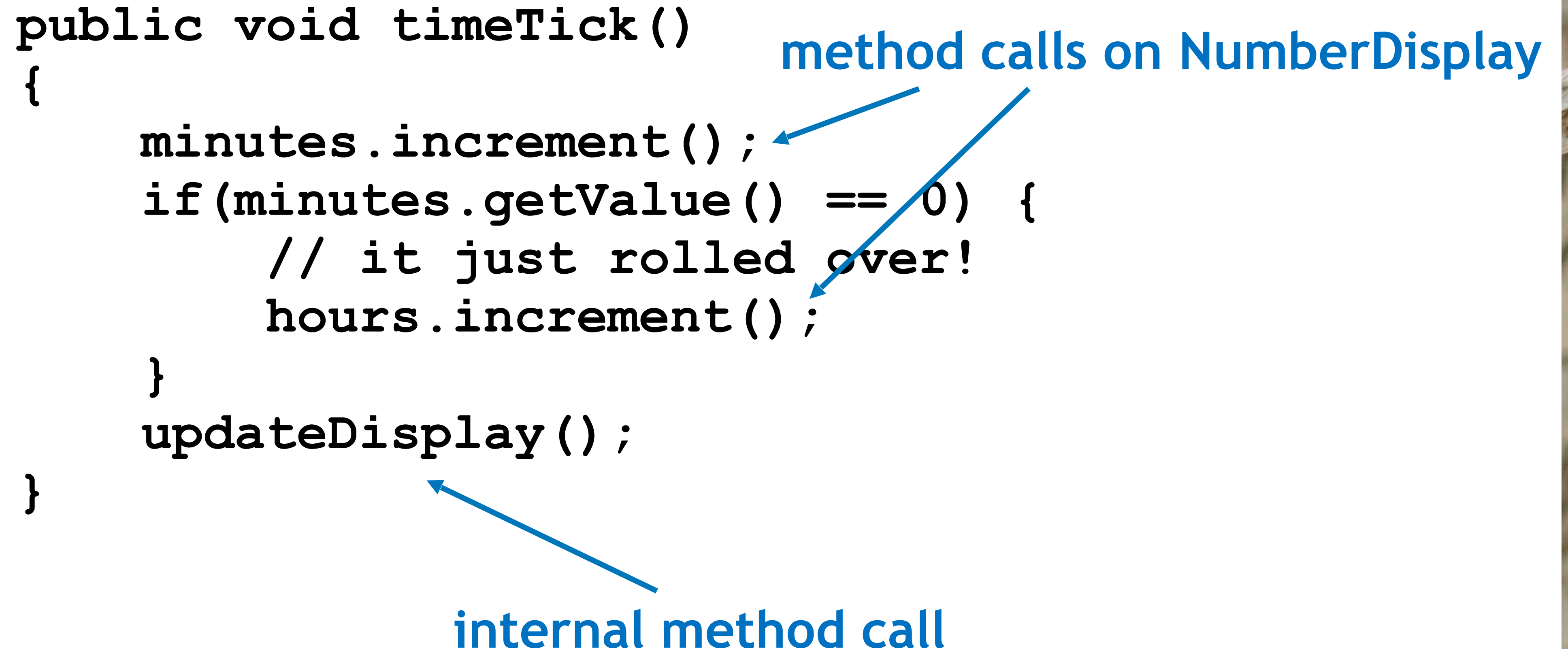
- Two NumberDisplay objects store data on behalf of a ClockDisplay object.
 - ClockDisplay calls methods in the NumberDisplay objects.

Method calling

```
public void timeTick()  
{  
    minutes.increment();  
    if (minutes.getValue() == 0) {  
        // it just rolled over!  
        hours.increment();  
    }  
    updateDisplay();  
}
```

method calls on NumberDisplay

internal method call

The diagram illustrates method calls in a Java code snippet. The code defines a method named 'timeTick' which calls 'minutes.increment()', 'hours.increment()', and 'updateDisplay()'. A blue arrow points from the text 'method calls on NumberDisplay' to the 'increment()' calls. Another blue arrow points from the text 'internal method call' to the 'updateDisplay()' call.

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 debugger

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

The debugger

The screenshot displays a Java IDE window titled "MailClient" with a menu bar (Compile, Undo, Cut, Copy, Paste, Find..., Close) and a "Source Code" dropdown. The code is as follows:

```
/**
 * Print the next mail item (if any) for this user to the text
 * terminal.
 */
public void printNextMailItem()
{
    MailItem item = server.getNextMailItem(user);
    if(item == null) {
        System.out.println("No new mail.");
    }
    else {
        item.print();
    }
}

/**
 * Send the given message to the given recipient via
 * the attached mail server.
 * @param to The intended recipient.
 * @param message The text of the message to be sent.
 */
public void sendMailItem(String to, String message)
{
    MailItem item = new MailItem(user, to, message);
    server.post(item);
}
```

The debugger window, titled "BlueJ: Debugger", is open over the code. It shows:

- Threads:** main (at breakpoint)
- Call Sequence:** MailClient.printNextMailItem
- Static variables:** (empty)
- Instance variables:** MailServer server = <object reference>, String user = "feena"
- Local variables:** (empty)

At the bottom of the debugger window, there are control buttons: Halt (Stop), Step (Green arrow), Step Into (Green arrow with L), Continue (Green arrow with R), and Terminate (Red X). A status bar at the bottom of the IDE shows "Thread 'main' stopped at breakpoint." and a "saved" indicator. Below the IDE, there are three red buttons representing objects: "mailServ1: MailServer", "sophie: MailClient", and "feena: MailClient". The current object is "feena : MailClient".

Concepts covered this week

- abstraction
- modularisation
- classes define types
- class diagram
- object diagram
- object references
- object types
- primitive types
- object creation
- internal/external method calls