

Objects First With Java
A Practical Introduction Using BlueJ

Object interaction

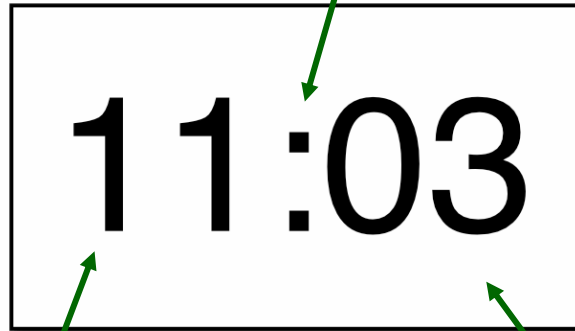
Creating cooperating objects

The next step

- **so far:**
 - what objects are
 - how they are implemented
- **now:**
 - combine classes and objects
 - let objects cooperate
 - how methods call other methods

Project: a digital clock

always the same



hours (European style: 0..23)

minutes

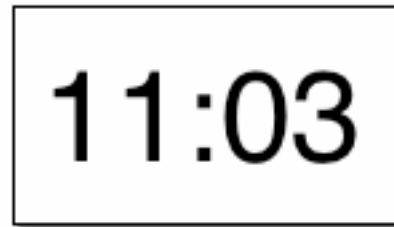
Abstraction

- Why not define the digital clock as one single class?
 - complexity → substructuring (divide and conquer)!
 - effect: “abstract” from a component’s details (not everyone needs to know everything ...)
- **Abstraction** is the ability to ignore details of parts to focus attention on a higher level of a problem.

Modularization

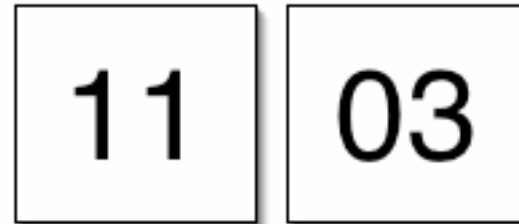
- **Modularization**: process of dividing a whole into well-defined parts, which can be built and examined separately, and which interact in well-defined ways.
- Modularization and abstraction complement each other.
- Both concepts are crucial for object-oriented software (components and subcomponents being objects here).

Modularizing the clock display



One four-digit display?

Or two two-digit displays?



One or two classes?
Differences?

Implementation - NumberDisplay

```
public class NumberDisplay
{
    fields:    private int limit;    when to roll back to 0?
               private int value;  current value?

    public NumberDisplay(int rollOverLimit)
    {
        limit = rollOverLimit;
        value = 0;
    }

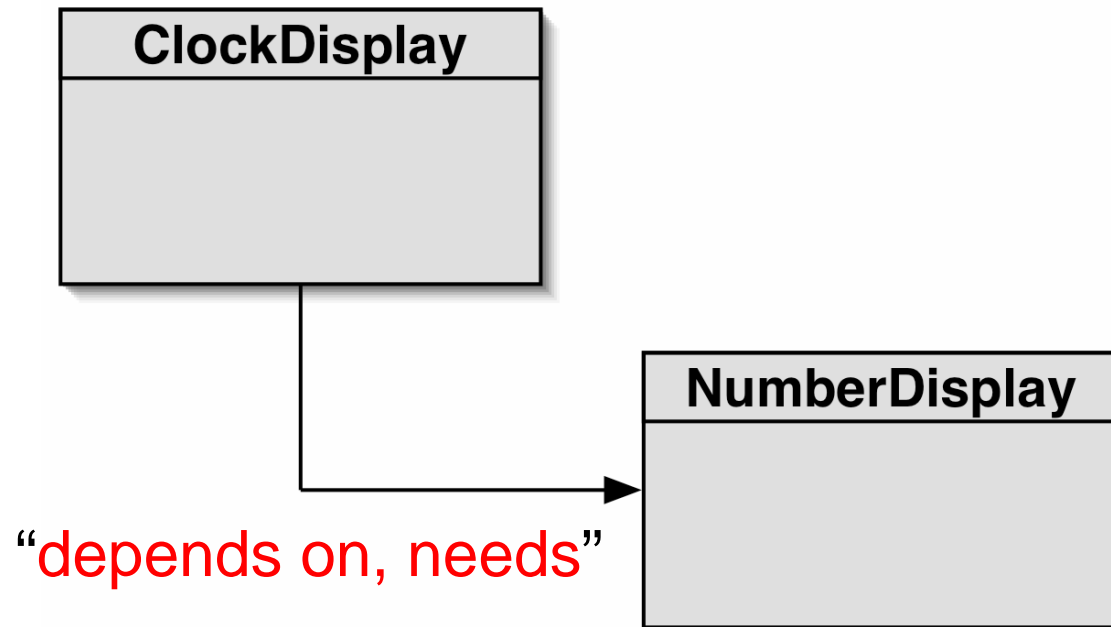
    methods omitted.
}
```

Implementation - ClockDisplay

```
public class ClockDisplay
{
    private NumberDisplay hours;
    private NumberDisplay minutes;
    Class names can be used as  
types!
    Constructor and  
methods omitted.
}
```

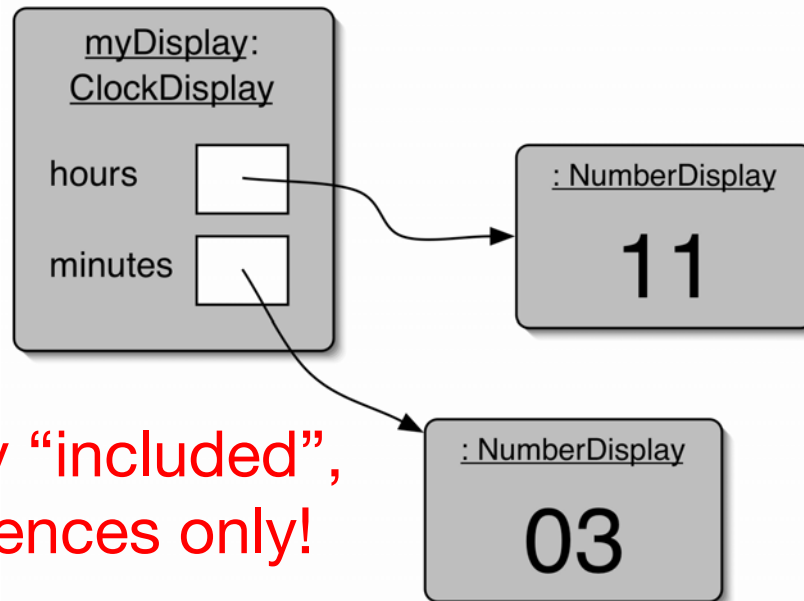

Class diagrams

- **class diagrams** show classes and their relationships – hence, a *static* view of a program



Object diagrams

- **object diagrams** show objects and their relationships at one moment in time – hence, a *dynamic* view of a program



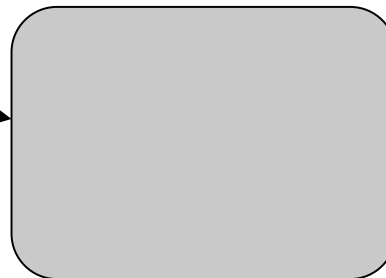
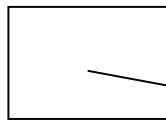
not really “included”,
but references only!

Starting the program

- Start program by creating an object of type/class `ClockDisplay`.
(`bluej`)
- Then: automatic creation of two `NumberDisplay` objects for own purposes

Primitive types vs. object types

```
SomeObject obj;
```



object type

(by classes, given by Java system or by programmer)

```
int i;
```



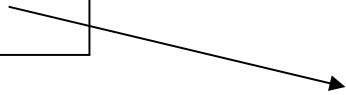
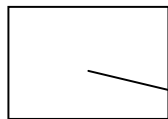
**Note different way of storing!
(direct value vs. reference)**

primitive type

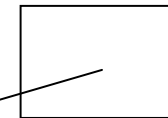
(predefined in Java)

Primitive types vs. object types

`SomeObject a;`



`SomeObject b;`



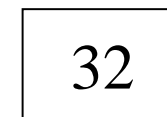
two references of
the same type!

`b = a;`

`int a;`



`int b;`



Source code: NumberDisplay

```
public NumberDisplay(int rollOverLimit)
{
    limit = rollOverLimit;           constructor!
    value = 0;
}
```

4 methods: `getValue`, `setValue`, `getDisplayValue`, `increment`

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

Source code: NumberDisplay

```
public String getDisplayValue()  
{  
    if(value < 10)  
        return "0" + value;  
    else  
        return "" + value;  
}  
    why not directly return value?
```

```
public void setValue(int replacementValue)  
{  
    if((replacementValue >= 0) &&  
        (replacementValue < limit))  
        value = replacementValue;  
}  
    what happens if replacement value is illegal? OK?
```

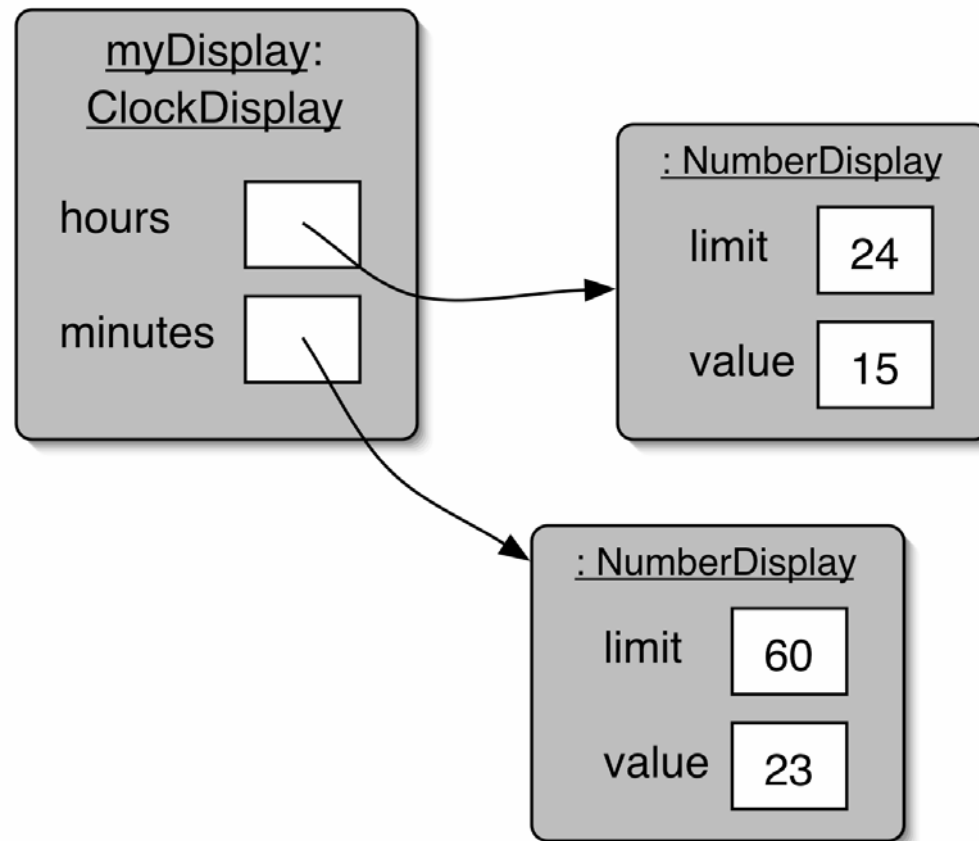
Logical operators

- operate on boolean values
- produce a new boolean value as result
- available in Java:
 - && (and)
 - || (or)
 - ! (not)

Further operators

- **String concatenation**
 - with “+” operator
 - spaces/blanks must be added explicitly
 - If one operand is a string, both the others and the result are converted to string.
 - the “+=” operator acts as for int or double data types
- **modulo: %**

ClockDisplay object diagram



where do the three objects come from?

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);
        updateDisplay();
    }
}
```

Alternative

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

    public ClockDisplay(int hour, int minute)
    {
        hours = new NumberDisplay(24);
        minutes = new NumberDisplay(60);
        setTime(hour, minute);
    }
}
```

multiple constructors in one class possible!

Overloading

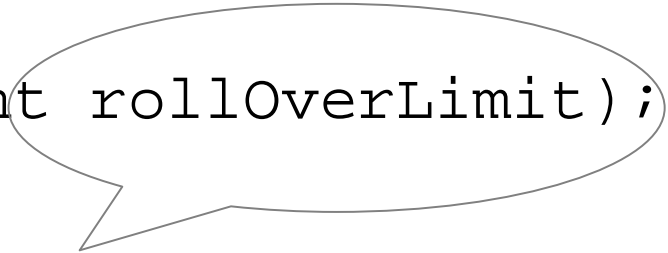
Automatic creation

- how and where?
 - in `ClockDisplay`'s constructor
 - automatically executed whenever a `ClockDisplay` object is created
 - explicitly done with `new` command
 - `new` creates an object of the specified class and executes the respective constructor

Objects creating objects

in class NumberDisplay:

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



formal parameter

in class ClockDisplay:

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



actual parameter

Method calling

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

Internal vs. external

- internal method calls
 - internal means: called method belongs to calling class (here: `ClockDisplay`)

```
updateDisplay();
```

```
private void updateDisplay()
```

- external method calls
 - external means: called method belongs to other class (here: `NumberDisplay`)

```
minutes.increment();
```


Calling external methods

object . methodName (parameter-list)

Review: concepts

- abstraction
- modularization
- classes define types
- class diagram
- object diagram
- object references
- primitive types
- object types
- object creation
- overloading
- internal/external method call