

# Object interaction

Creating cooperating objects

# A digital clock

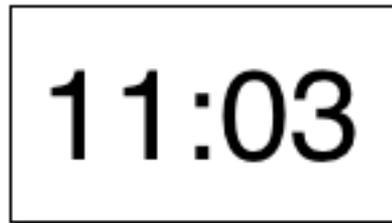
A digital clock display showing the time 11:03. The time is displayed in a large, black, sans-serif font within a white rectangular box with a thin black border. The box is centered on the slide.

11:03

# 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.

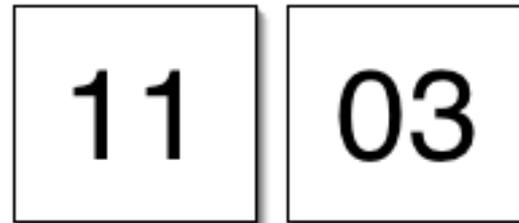
# Modularizing the clock display



11:03

One four-digit display?

Or two two-digit displays?



11 03

# Implementation - NumberDisplay

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

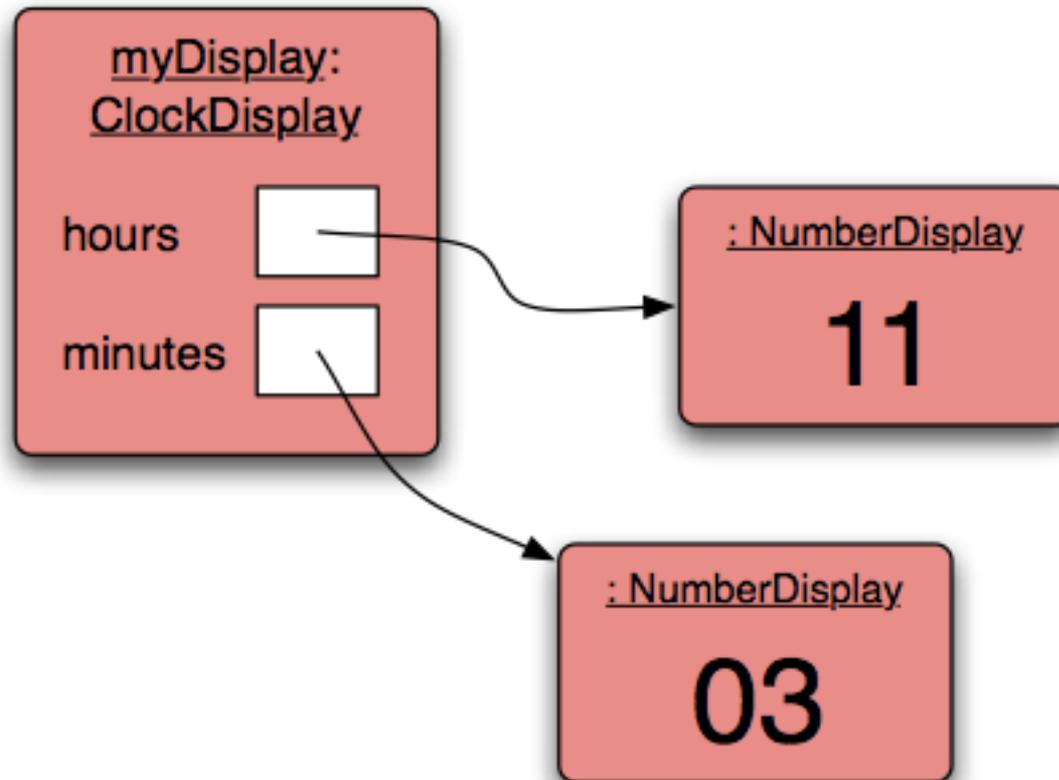
    Constructor and
    methods omitted.
}
```

# Implementation - ClockDisplay

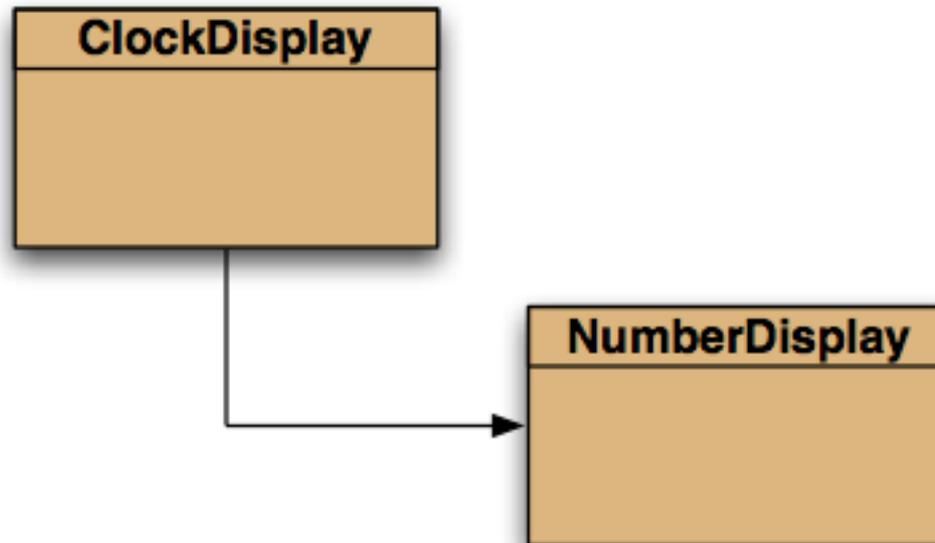
```
public class ClockDisplay
{
    private NumberDisplay hours;
    private NumberDisplay minutes;

    Constructor and
    methods omitted.
}
```

# Object diagram



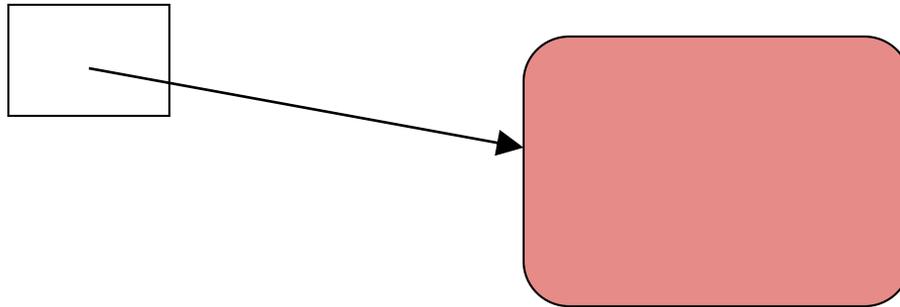
# Class diagram



# Primitive types vs. object types

`SomeObject obj;`

object type



`int i;`

primitive type



# Quiz: What is the output?

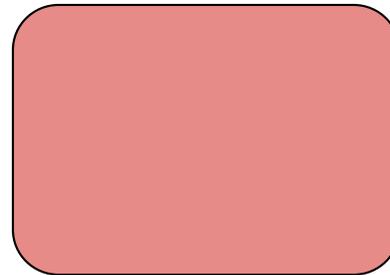
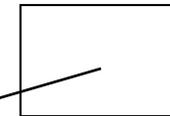
- ```
int a;  
int b;  
a = 32;  
b = a;  
a = a + 1;  
System.out.println(b);
```
- ```
Person a;  
Person b;  
a = new Person("Everett");  
b = a;  
a.changeName("Delmar");  
System.out.println(b.getName());
```

# Primitive types vs. object types

`ObjectType a;`



`ObjectType b;`



---

`b = a;`

`int a;`



32

`int b;`



32

# Source code: NumberDisplay

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

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

# 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:  
     $17 / 5 == 3$   
     $17 \% 5 == 2$

# Quiz

- What is the result of the expression  
 $8 \% 3$
- For integer  $n \geq 0$ , what are all possible results of:  
 $n \% 5$
- Can  $n$  be negative?

# Source code: NumberDisplay

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

# Concepts

- abstraction
- modularization
- classes define types
- class diagram
  
- object diagram
- object references
- object types
- primitive types

# 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:

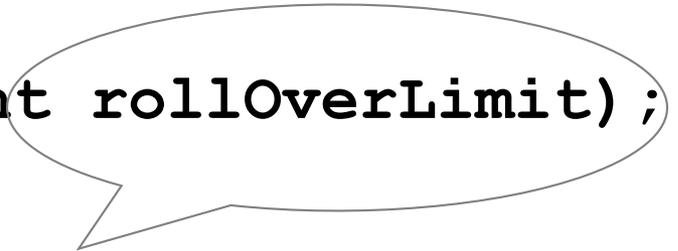
```
hours = new NumberDisplay(24);
```



*actual parameter*

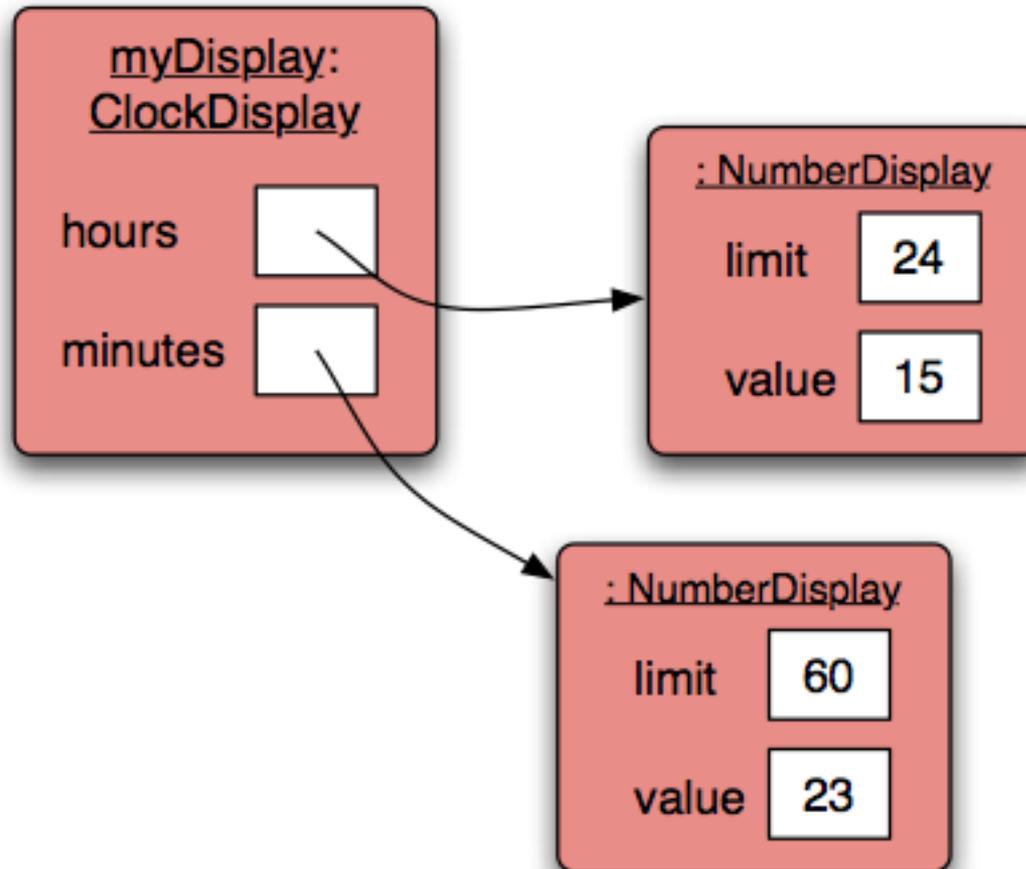
in class NumberDisplay:

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



*formal parameter*

# ClockDisplay object diagram



# Method calling

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

# External method call

- external method calls

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

```
object . methodName ( parameter-list )
```

# Internal method call

- internal method calls

```
updateDisplay() ;
```

- No variable name is required.
- **this**
  - could be used as a reference to the invoking object, but not used for method calls.

# 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 be an external call.
- ‘Internal’ means ‘this object’.
- ‘External’ means ‘any other object’, regardless of its type.

# null

- `null` is a special value in Java
- Object fields are initialized to `null` by default.
- You can test for and assign `null`:

```
private NumberDisplay hours;
```

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

```
hours = null;
```

# 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

The screenshot displays a Java IDE window titled "MailClient" with a menu bar containing "Compile", "Undo", "Cut", "Copy", "Paste", "Find...", and "Close". A "Source Code" dropdown menu is visible. The code editor shows the following code:

```
/**
 * 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);
}
```

A breakpoint is set on the line `MailItem item = server.getNextMailItem(user);`. The "BlueJ: Debugger" window is open, showing the "Threads" panel with "main (at breakpoint)". The "Call Sequence" panel shows "MailClient.printNextMailItem". The "Instance variables" panel shows "MailServer server = <object reference>" and "String user = 'feena'". The "Local variables" panel is empty. The debugger controls include "Halt", "Step", "Step Into", "Continue", and "Terminate". A status bar at the bottom of the debugger window says "Thread 'main' stopped at breakpoint." and "saved".

At the bottom of the IDE, there are three red buttons representing objects: "mailServ1: MailServer", "sophie: MailClient", and "feena: MailClient". The "feena: MailClient" button is highlighted, indicating the current object being debugged.

# Concept summary

- object creation
- overloading
- internal/external method calls
- debugger