



Object interaction

Creating cooperating objects

A digital clock

11:03

Abstraction and modularization

- **Abstraction**
definition
a high-level
description
of a system
that ignores
implementation
details
- **Modularization**
dividing a
system into
parts that can
be developed
and tested
independently



ignore
ntion on

s of
fined

ct

which
ys


Modularizing the clock display



11:03

One four-digit display?

Or two two-digit displays?



11



03

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 until it reaches its limit
- It *rolls over* to zero at this point

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.
}
```

Source code: NumberDisplay

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

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

*** value is between 0 --> (limit - 1)**

Source code: NumberDisplay

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

Source code: setValue()

```
public void setValue(int replacementValue)
{
    if((replacementValue >= 0) &&
        (replacementValue < limit))
    {
        value = replacementValue;
    }
}
```

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.



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);
```

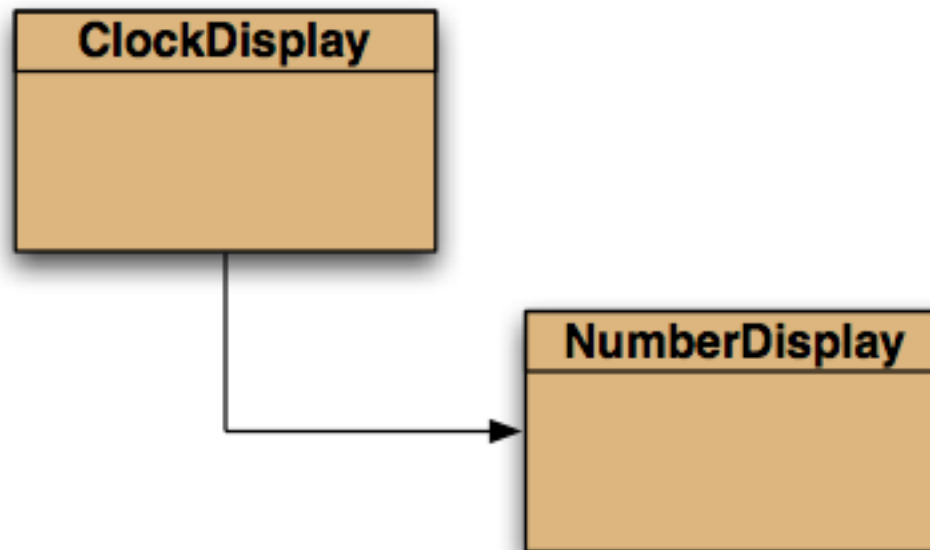
actual parameter

in class NumberDisplay:

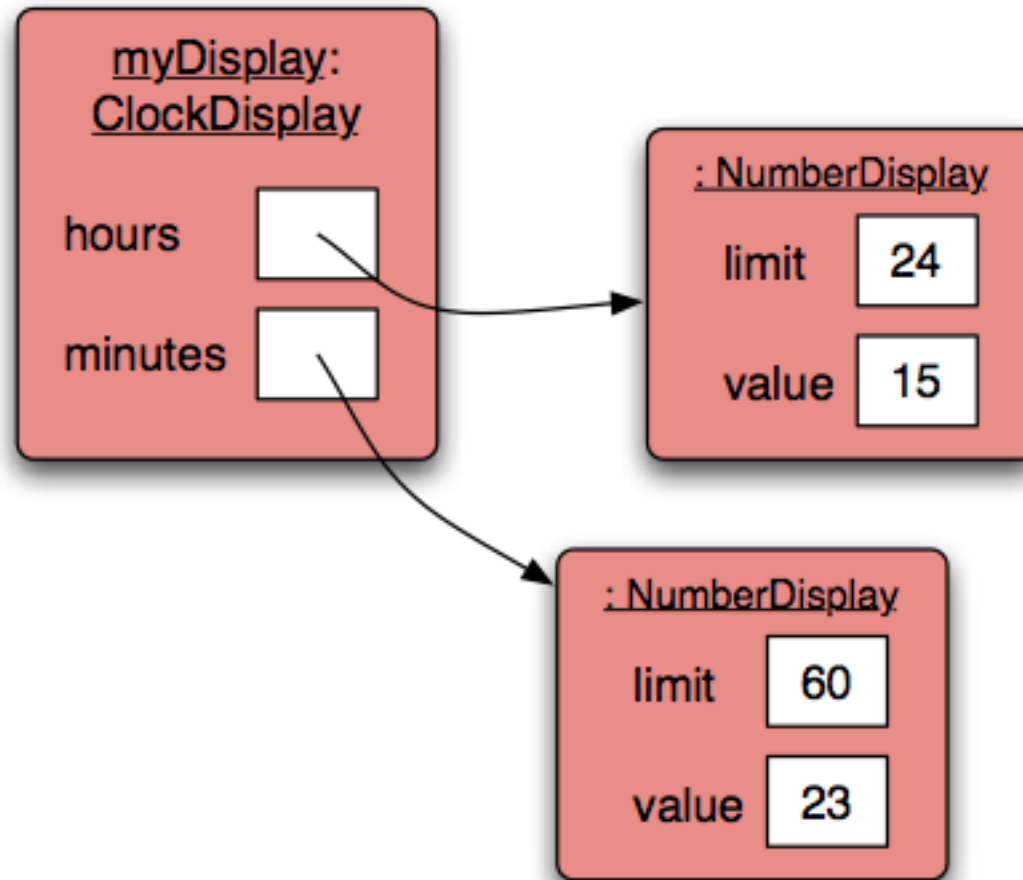
```
public NumberDisplay(int rolloverLimit)
```

formal parameter

Class diagram (static view)



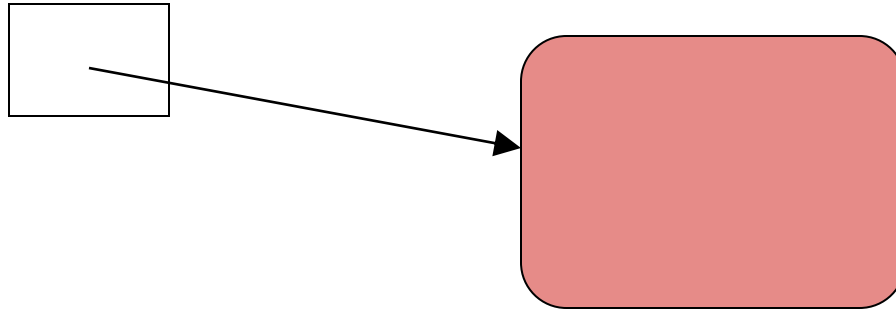
Object diagram (dynamic view)



Primitive types vs. Object types

SomeObject obj;

object type



int i;

primitive type



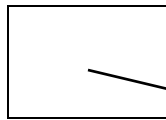
Quiz:

What is the output?

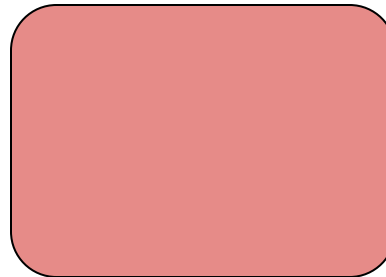
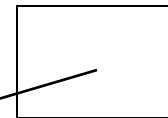
- ```
int a;
int b;
a = 32;
b = a;
a = a + 1;
System.out.println(b);
```
- ```
Person a;  
Person b;  
a = new Person("Everett");  
b = a;  
a.changeName("Delmar");  
System.out.println(b.getName());
```

Primitive types vs. object types

`ObjectType a;`



`ObjectType b;`



`b = a;`

`int a;`



`int b;`





Object interaction

- Two objects interact when one object calls a method on another
- The interaction is usually all in one direction (*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

'client' method

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

'server' external
methods

internal/self method call

External method calls

- General form of *external* method call:

object . methodName (params)

- Examples:

hours.increment()

minutes.getValue()

Internal method calls

- No variable name is required for *internal* method calls:
`updateDisplay() ;`
- Internal methods often have **private** visibility to prevent them from being called from outside their defining class
- Method is found in *this* same invoking class/object where the call is made

Internal method

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

Method calls

- *Internal* means *this object*
- *External* means *any other object*, regardless of its type
- NOTE: A method call on *another object of the same type* would also be an external call

Method / Constructor Overloading

- **Overloading:**
with a different set of parameters:

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

Quiz: is this correct ?!

```
private int value;  
  
public void setValue(int value) {  
    value = value;  
}
```

The `this` keyword

- Used to distinguish parameters and fields of the same name
- *this* could also be used as a reference to the invoking object instead of method calls

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

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;
```