



Chapter 3: Object interaction

Creating cooperating objects

This Lecture

This lecture extends lecture 2  with:

- some techniques/concepts to help us design code: *abstraction* and *modularisation*
- *object diagrams* to help us visualise the state of a running program
- more on creating objects and calling methods

A digital clock

Let's implement a digital clock:

A rectangular box with a black border and a subtle drop shadow, containing the time 11:03 in a large, bold, black sans-serif font.

11:03

Abstraction and modularization

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

- **Abstraction:**
 - treat each module as an atomic unit or “black box”
 - I don’t know how a car engine works, but I can drive
 - To me, the car is a black box
 - solve the problem at a high level before solving each sub-problem

Modularization

Suppose you want to go to Spain on holiday

- You can divide the problem into modules:
 - Getting to the airport
 - Flying from Britain to Spain
 - Getting from Spanish airport to your hotel
- Each module can be solved independently, which simplifies things
 - Use a taxi company to get to the airport, a travel agent to get to Spain
and a shuttle bus to get to your hotel

- You can have a hierarchy of modules
 - Getting to the airport has modules:
 - Find the phone number of a taxi company
 - Book a taxi
 - Set your alarm
 - Setting your alarm has modules...

Key points:

- Dividing a problem into smaller problems simplifies things
- If modules are independent they are easier to solve

Top-down design

- Do not start by planning how to set your alarm
- Do not start by planning how to get to the airport
- Instead, abstract over these details
 - Assume you will figure out later how to set your alarm and get to the airport
- Start with the highest-level decision:
 - Where in Spain do you want to go? Madrid, Malaga, Granada...?

- Now look at the next-highest decision:
 - How should I get to Malaga? Fly from Bristol, Gatwick, Heathrow...?
 - At this point think about flights and airports, but abstract over how to get to the airport
- Now you can plan how to get to the airport
 - But still abstract over how to set your alarm
- Eventually you can deal with how to set your alarm

Key point:

- Working top-down as in this example is usually a good strategy

Abstraction

- Abstracting over a module means: we assume we will solve it later
 - Completely ignoring parts of the problem is asking for trouble
 - But abstraction does **not** mean simply ignoring parts of the problem
- Abstraction means ignoring the *implementation* of other modules while we are implementing the current one

- Although we defer the implementation of modules, we decide:
 - what modules the problem involves
 - what each module does for us (e.g. gets us to the airport)
 - what each module requires (e.g. to be at the airport before we can fly)
 - we could call this the *interface* of the module, as opposed to its implementation

We can use the (interface of the) module in our planning

- I know I'll get to the airport somehow...

Abstraction

- If we were to write code at an early stage of planning it might look like this:

```
public void goToSpain() {  
    goToAirport();  
    flyToSpain();  
    goToHotel();  
}
```

- We have written `goToSpain()` before the other methods (modules)
- That is, we abstract over the other modules

- Of course it doesn't compile until we define all the methods
- Although this is a very abstract, high-level algorithm:
 - It's nonetheless complete in the sense that everything that must be done will be done within one of the method calls
 - It's useful as we can check that it's complete, that things occurs in the correct order, that the components are not mutually exclusive etc.
- It's easier to solve the core of the problem at this abstract level first, and then fill in the details later

Encapsulation

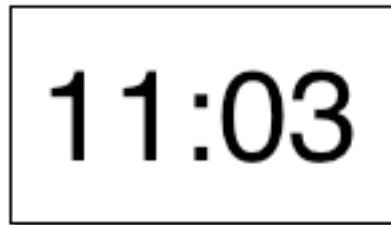
We can relate abstraction and modularization to our other OO design principles

- Note how each module *encapsulates* part of the problem
- Note how designing at an abstract level uses this encapsulation to simplify solving the core of the problem
- Whenever you see a new design principle, you should try to relate it to the others in this way
- This will deepen your understanding of what they mean

Object-Oriented Design

- Planning a trip to Spain is really an example of (part of) the problem of OO design
 - In general it's a very difficult problem
 - There are methodologies which give us step-by-step instructions for making designs
 - We will return to design and design methodologies in later lectures

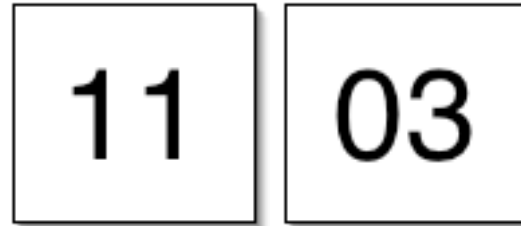
Modularizing the clock



11:03

One four-digit display?

Or two two-digit displays?



11 03

Design - deciding what classes to use and how they should interact - is one of the hardest parts of OOP

Implementing the clock

Let's use 2 classes:

- NumberDisplay implements 2-digit displays
- ClockDisplay implements the clock, using 2 NumberDisplay objects

In OO terms:

- We've *modularised* the problem into 2 parts
- We've *encapsulated* displaying digits with one class and clocks with another
- We'll *(re)use* NumberDisplay within ClockDisplay to obtain a 4-digit display
- We can now even write some *abstract code* (next few slides)
- Once we've implemented our 2 classes, we could *reuse* them in other applications
- It's good to explicitly describe your design in such OO terms to help make yourself (and others) conscious of what is going on

Implementation - NumberDisplay

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

    Constructor and  
methods omitted.
}
```

Using an abstract NumberDisplay

- Now we've decided to use a NumberDisplay class, we can *abstract* over it
- Specifically, we can refer to NumberDisplay in ClockDisplay even though we haven't finished writing NumberDisplay
 - Again, it won't compile yet, but we're making progress with our design
 - We'll implement the details of NumberDisplay later

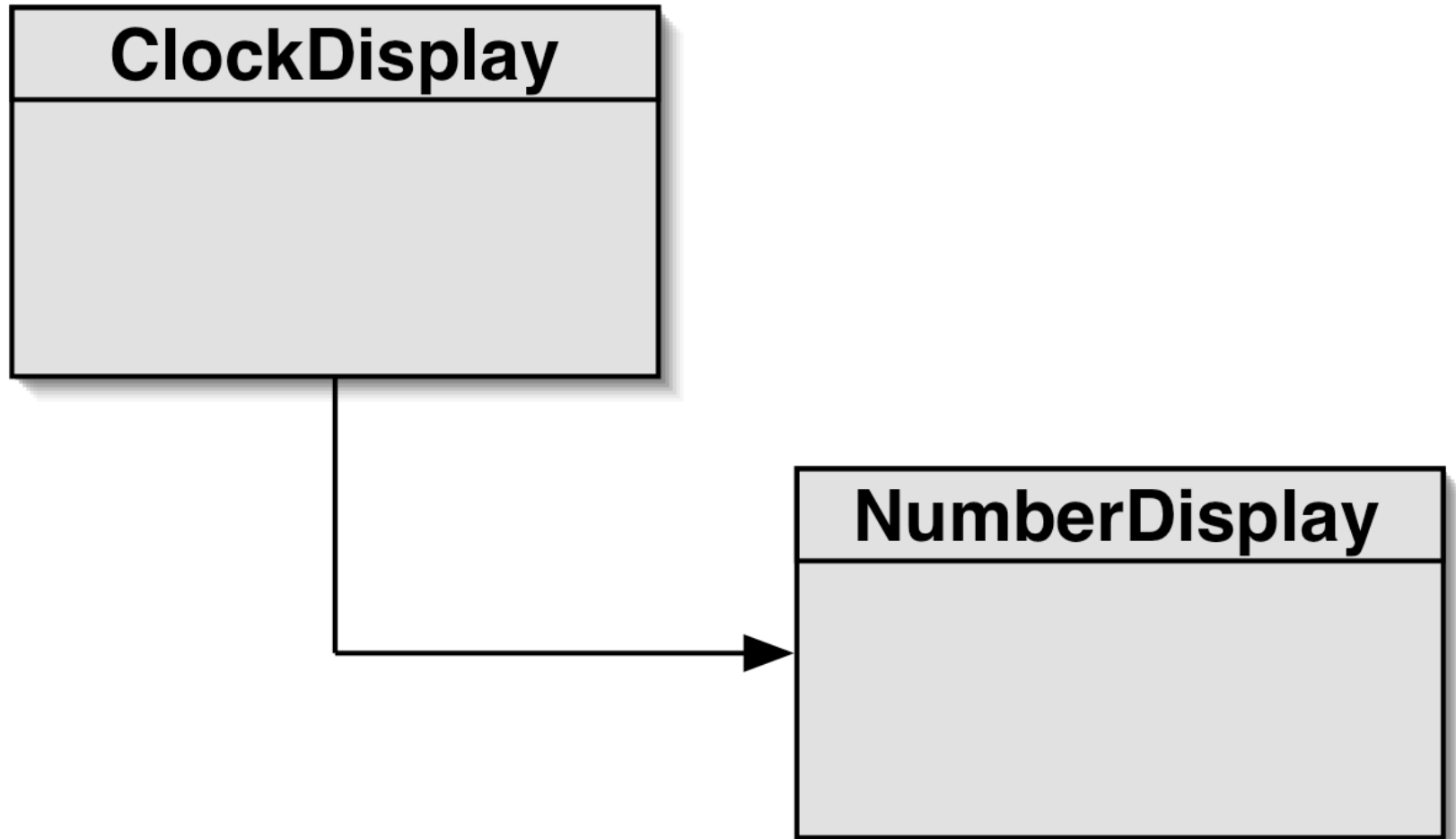
Implementation -ClockDisplay

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

    Constructor and
    methods omitted.

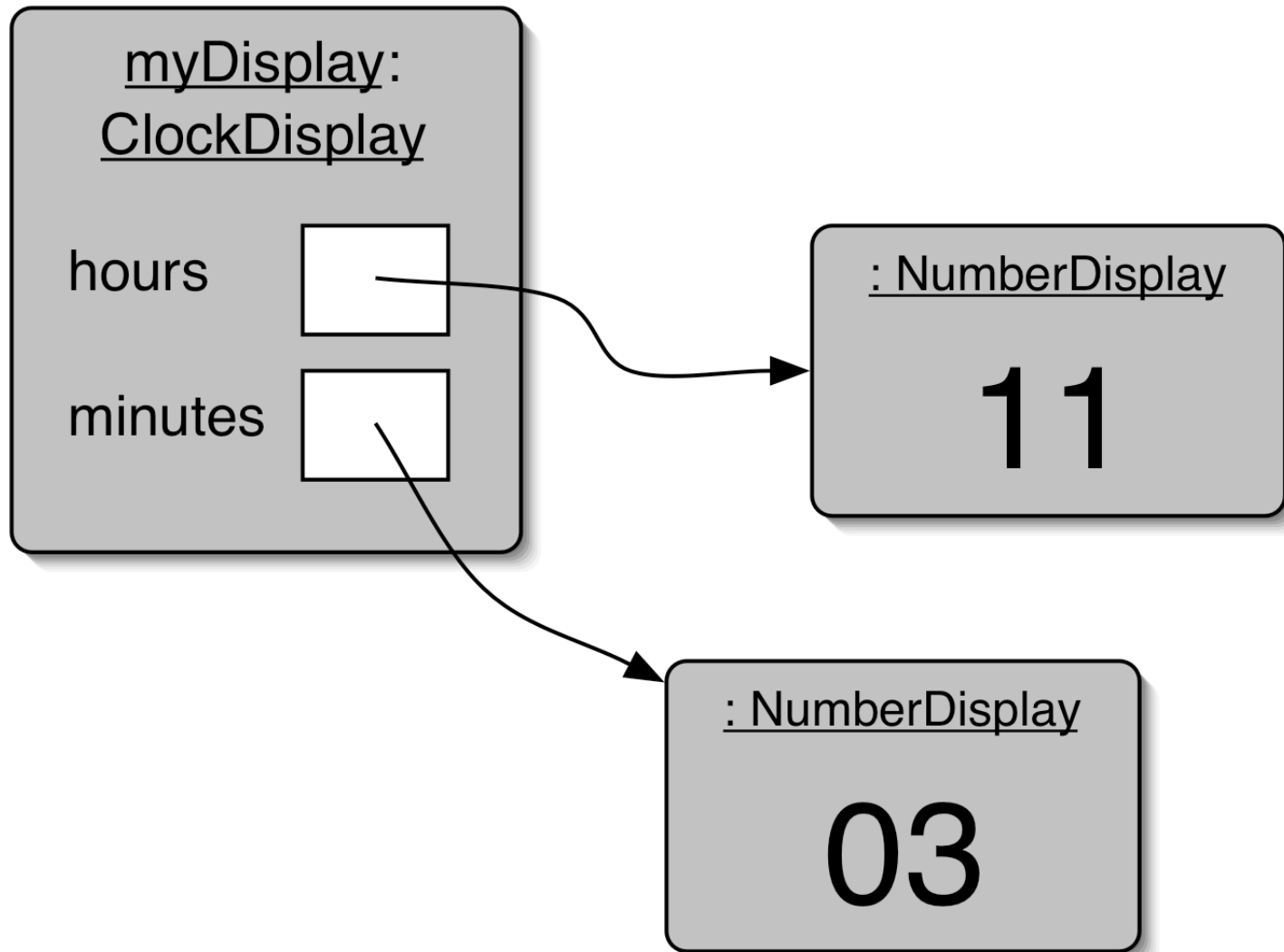
}
// Note use of class NumberDisplay
// as a type.
```

Class diagram



Arrow indicates ClockDisplay uses NumberDisplay

Object diagram



Class and object diagrams

- BlueJ shows class diagrams only
- Class diagrams are static: do not change as program runs
- Object diagrams are dynamic: change as objects created/deleted during program execution
- You must learn to draw and think in terms of object diagrams

Primitive types

- For efficiency Java stores some simple kinds of data as primitive types: int, byte, short, long, double, float, char, boolean
 - The primitive types are built-in; you cannot define new primitive types
 - They are not objects
 - You cannot e.g. associate methods with them

Object types

- Java has many built-in object types (e.g. String), and you *can* make new ones:
 - Each Class is a type
- Some differences between primitive and objects including:
 - How they are stored
 - What happens when you assign them

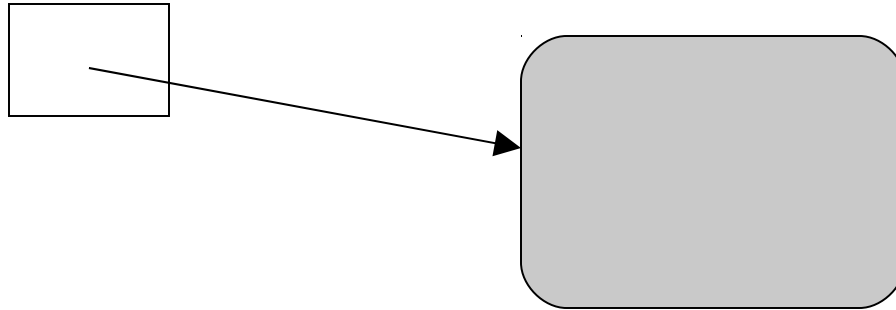
Primitive and object types

- Primitive variables store their values internally
- Object variables store a reference to the object
 - The object itself is stored elsewhere
- Comparison to C:
 - A reference is like a pointer in C
 - Because *all* objects in Java are accessed through references, things are much simpler than in C, where we sometimes access things through pointers and sometimes not
 - Pointers are a major source of nasty bugs in C

Primitive and object types

object type

`SomeObject obj;`



primitive type

`int i;`



Primitive and object types

Consequence:

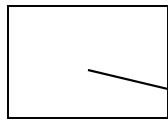
- When you assign primitive b to a , b 's value is copied into a
- When you assign an object b to a , only the reference is copied.

Result: you have 2 references to same object

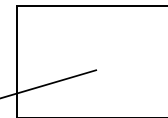
- To copy the object you need to do something else
- Of course an object can store primitive types (and some do nothing else)

Primitive and object types

`SomeObject a;`



`SomeObject b;`



`b = a;`

`int a;`



`int b;`



Source code: NumberDisplay


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

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

```
// Note: % is modulo operator: returns
// remainder of integer division
// E.g. 15 % 12 is 3
// E.g. 15:00 hours is 3pm
```

Source code: NumberDisplay

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



Note: "" is an empty String.

This is a trick to convert `value` to a String
To match the return type

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();
    }
}
```

Creating objects

On previous slide:

```
private NumberDisplay hours;
```

```
...
```

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

- We use `new` followed by call to constructor to make a new object.
- Note the special syntax for creating string objects used earlier:

```
String myString = "hello";
```

not:

```
String myString = new String("hello");
```


More of ClockDisplay

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

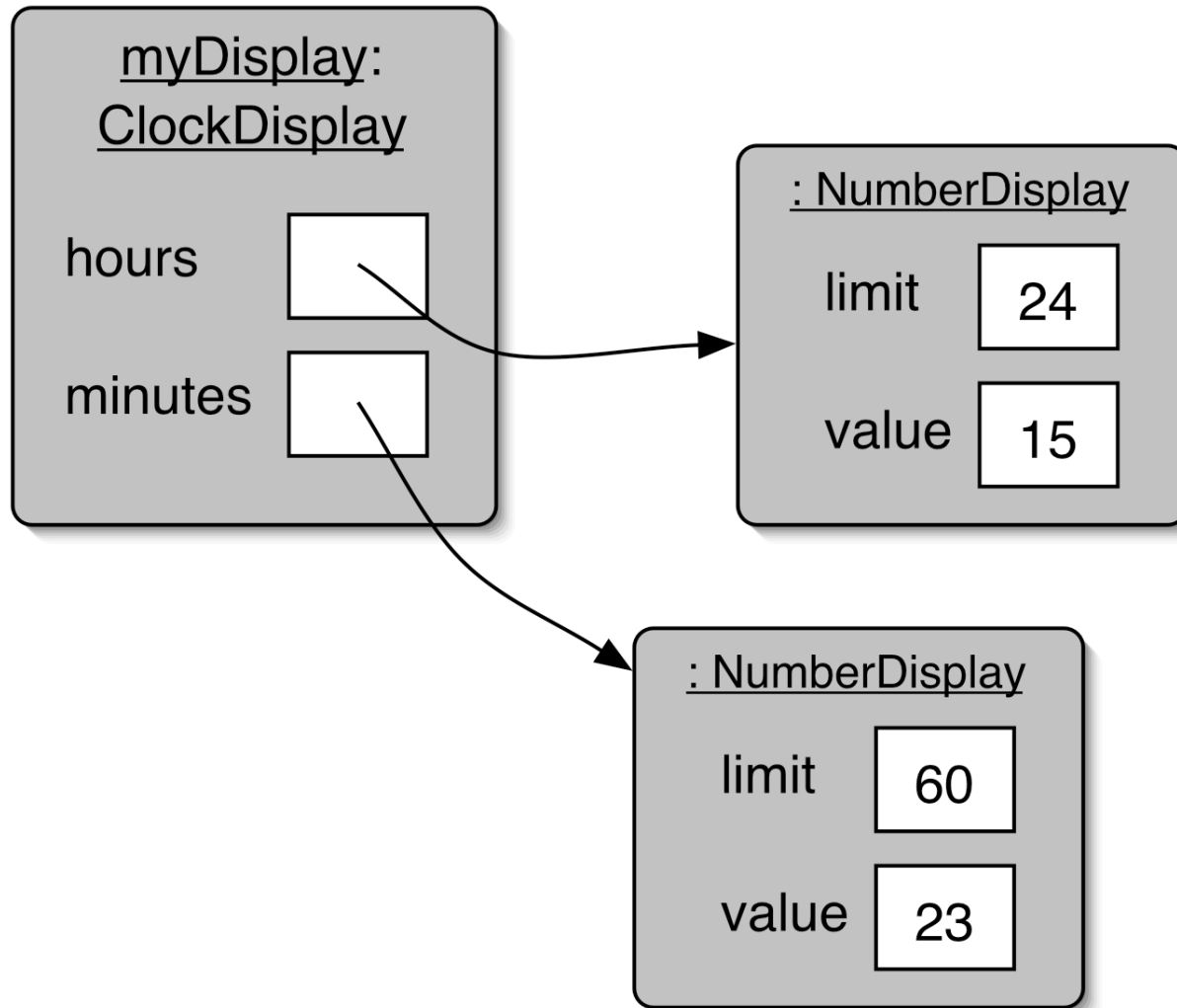
Even More of ClockDisplay

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

Object interaction

- Now we've seen the source code for both classes
- Next let's look more closely at their interaction

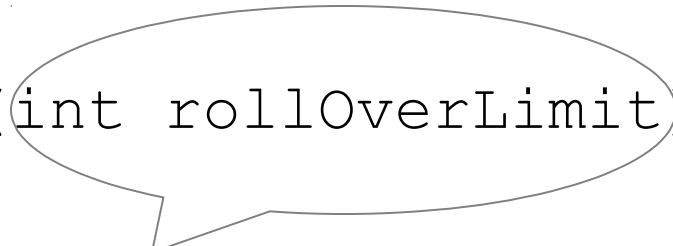
ClockDisplay object diagram



Objects creating objects

in class NumberDisplay:

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



formal parameter

in class ClockDisplay:

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



actual parameter

(Formal and actual parameters are the same as in C)

Internal and external calls

- Internal method calls are to methods in the same class

methodName (parameter-list)

- External method calls are to methods in other classes

object . methodName (parameter-list)

Method calls in ClockDisplay

- internal method call

```
updateDisplay();
```

```
...
```

```
private void updateDisplay()
```

- external method call

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

Scope and “this”

- From Chapter 2:
 - A variable can only be used within its scope
 - The scope of a parameter is its method
 - The scope of a field is the entire class
- What happens if a parameter and field have the same name?
 - Java uses the definition from the closest enclosing block, i.e. the parameter.
 - We can still access the field using ‘this’
 - You can read ‘this’ as ‘this object’

This

```
public class MailItem {  
    private String from;  
    private String to;  
  
    public MailItem(String from, String to) {  
        this.from = from;  
        this.to = to;  
    }  
}
```

...

- The value of parameter `from` is assigned to field `from`
- Note we can also return the current object with:

```
return this;
```

Concepts

- abstraction
- modularization
- top-down design
- encapsulation
- classes define types
- class diagram
- object diagram
- object references
- primitive types
- object types
- object creation
- internal/external method call
- debuggers †

†Covered in chapter 3 of text but not in handout

Extra Material

- The following is extra material for reference
- You should not need it to complete the assignments

Numbers in Java

- In Java the size of primitives is the same no matter what OS you run (unlike C)

- This helps make Java code portable

- Use int for small integers, long for big ones, and double for real values

- No warning is given for over/underflows

- Rounding errors can occur with real values

```
double f = 4.35;
```

```
System.out.println(100*f);
```

```
// prints 434.999999999999994
```

- Use BigInteger and BigDecimal classes for arbitrary precision calculations

Range of common Java numbers

Type	Range	Bytes
int	-2,147,483,648 ... +2,147,483,647	4
long	-9,223,372,036,854,775,808... +9,223,372,036,854,775,807	8
float	About $\pm 10^{38}$ and 7 significant digits	4
double	About $\pm 10^{308}$ and 15 sig. digits	8