



Object Oriented Programming

Instructor Name:

Lecture-7

Today's Lecture

- **Object Interaction**
- **Modularization & Abstraction**

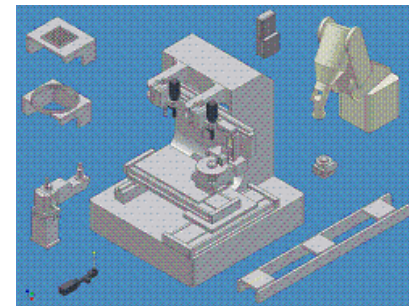
Object Interaction & Collaboration

Message Passing

- An Application is set of Cooperating Objects that communicate with each other
- Objects communicate by sending messages
- When an object sends a message to another object, an operation is invoked in the receiving object
- The aim of modelling object interaction is to determine the most appropriate scheme of message passing between objects to support a particular user requirement
- In Java , “Message Passing” is done by calling methods.

Today's Lecture

- *Modularization* divides a problem into simpler sub-parts, which can be built separately, and which interact in simple ways.
- *Abstraction* is the ability to ignore low level details of a problem to focus on the higher levels.



Today's Lecture

Use in Programming

- Use *modularization* to split a programming problem into sub-parts (modules).
 - implement the modules
- The implementation of the complete program will be easier, since *abstraction* can be used to write software in terms of the modules.

Object Interaction & Collaboration

Abstraction & Modularization Example

- 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

Object Interaction & Collaboration

Abstraction & Modularization Example

- 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

Object Interaction & Collaboration

Abstraction & Modularization Example

- 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...?

Object Interaction & Collaboration

Abstraction & Modularization Example

- 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

Object Interaction & Collaboration

Abstraction & Modularization Example

- Engineers in car company designing a car.
- Many Thinking areas
 - ✓ Shape of body
 - ✓ Size and location of engine
 - ✓ Number and size of seats in the passenger area.
 - ✓ Exact spacing of wheel
- An other engineer, whose job is to design engine
- Many thinking areas
 - ✓ Clyinder
 - ✓ Injection Mechansim
 - ✓ Carburetor
 - ✓ Elelctronics
 - ✓ Spark plug (one Engineer will desgin it)

Object Interaction & Collaboration

Further Narrow Down the Problem Basic Class Structure

- Engineer may think of the spark plug as a complex artifact of many parts. He might have done complex studies. To determine exactly what kind of metal to use for the contacts or what kind of material and production process to use for the insulation.

Abstraction

Understanding Abstraction

- A designer at the highest level will regard a wheel as a single part. Another engineer much further down the chain may spend her days thinking about the chemical composition necessary to produce the right materials to make the tires. For the tire engineer, the tire is a complex thing. The car company will just buy the tire from the tire company and then view it as a single entity. This is abstraction.
- =====
- The engineer in the car company *abstracts from the details of the tire manufacture to be able to* concentrate on the details of the construction of, say, the wheel. The designer designing the body shape of the car abstracts from the technical details of the wheels and the engine to concentrate on the design of the body (he will just be interested in the size of the engine and the wheels).

Today's Lecture

Use in OOP

- Use modularization to split a programming problem into *objects*.
- Implement the *classes* for the objects.
- The implementation of the class for the complete program will be easier, since abstraction can be used to write the class in terms of other classes (yours and predefined ones).

Today's Lecture

e.g. Airport Control System



Classes for:
Plane, Gate, Luggage,
Passenger, etc.

Use them to create objects
such as plane1, plane2,
gate2, myLuggage

Abstraction simplifies the
communication between
the objects; only use their
visible interface.

Abstraction

Abstraction in Software

- In object-oriented programming, these components and subcomponents are objects. If we were trying to construct a car in software, using an object-oriented language, we would try to do what the car engineers do.
- Instead of implementing the car as a single, monolithic object,
- we would first construct separate objects for an engine, gearbox, wheel, seat, and so on, and then assemble the car object from those smaller objects.

Abstraction & Modularization

Abstraction

- Abstraction is the ability to ignore details of parts to focus attention on a higher level of a problem.

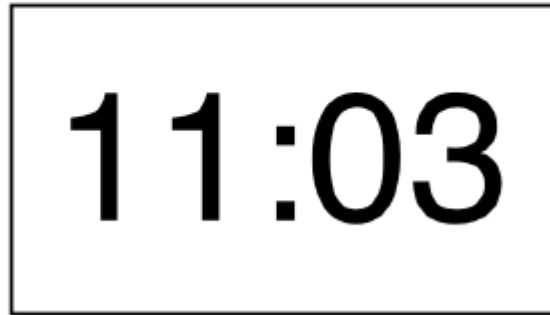
➤ Modularization

- 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

Object Interaction & Collaboration

A Digital Clock Example

Let's implement a digital clock:



- Can you guess the implementation of clock?

Object Interaction & Collaboration

A Digital Clock Example

- **First Idea:** to implement a whole clock display in single class.
- If the problem is bigger than a digital clock then??
- **Another approach (Abstraction and Modularization):**
- **Divide the problem** in sub-components and then again sub-sub-components and so on, until then individual problem are small enough to be easy to deal with (**Modularization**).
- Once sub-problem solved, don't think about in detail any more (**Abstraction**)

Modularization

Modularization in the Digital Clock Example

- One way to look at it is to consider it as consisting of a single display with four digits (two digits for the hours, two for the minutes).
- we can see that it could also be viewed as two separate two-digit displays (one pair for the hours and one pair for the minutes).
- One pair starts at 0, increases by 1 each hour, and rolls back to 0 after reaching its limit of 23. The other rolls back to 0 after reaching its limit of 59.
- The similarity in behavior of these two displays might then lead us to abstract away even further from viewing the hours display and minutes display distinctly. Instead, we might think of them as being objects that can display values from zero up to a given limit. The value can be incremented, but, if the value reaches the limit, it rolls over to zero.
- Now we seem to have reached an appropriate level of abstraction that we can represent as a class: a two-digit display class.

Modularization

Modularization in the Digital Clock Example

11:03

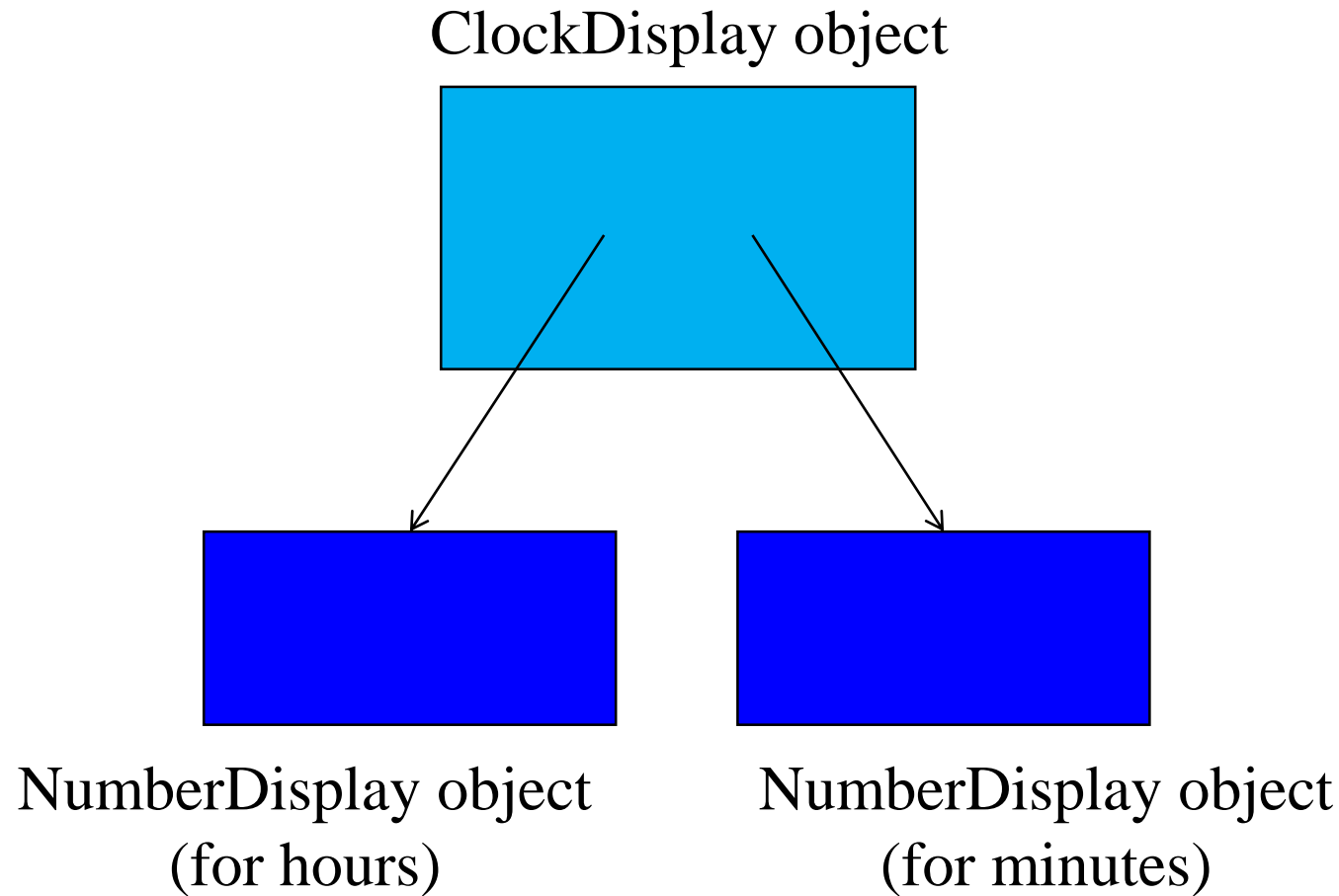
One four-digit display?

Or two two-digit displays?

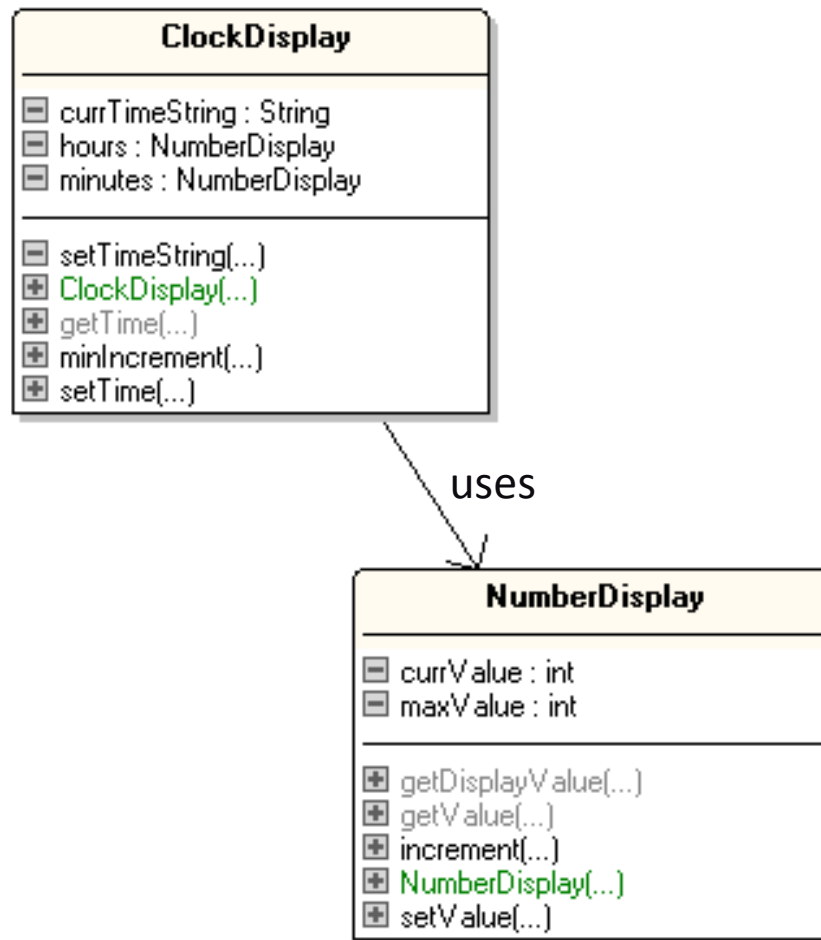
11 03

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

Objects Diagram



Classes Diagram



Modularization

Implementing Digital 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

Abstraction - NumberDisplay

Implementing Digital Clock

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

    Constructor and methods omitted.
}
```


Abstraction - NumberDisplay

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

Abstraction - ClockDisplay

Implementing Digital Clock

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

    Constructor and methods omitted.
}
```

NumberDisplay class is use in ClockDisplay class. Here we have has-a relationship

Abstraction - NumberDisplay

NumberDisplay Source Code

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

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

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

What modulu (%) operotor do here?

Abstraction - NumberDisplay

NumberDisplay Source Code

```
public int getValue()  
{  
    return value;  
}
```

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

Abstraction - NumberDisplay

NumberDisplay Source Code

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

Objects Creating Objects

ClockDisplay Source Code

```
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

ClockDisplay Source Code

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

    public ClockDisplay() {
        hours = new NumberDisplay(24);
        minutes = new NumberDisplay(60);
    }

    public void timeTick() {
        minutes.increment();
        if(minutes.getValue() == 0)
        {
            hours.increment();
        }
        updateDisplay();
    }
    .....
}
```

Objects Creating Objects

ClockDisplay Source Code

```
public void setTime(int hour, int minute)
{
    hours.setValue(hour);
    minutes.setValue(minute);
    updateDisplay();
}

public String getTime()
{
    return displayString;
}

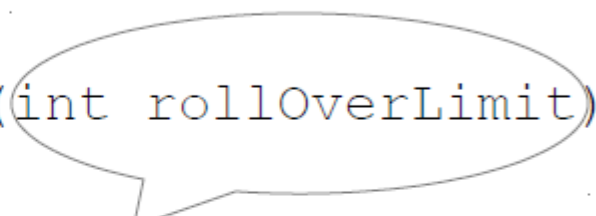
private void updateDisplay()
{
    displayString = hours.getDisplayValue() + ":" +
                    minutes.getDisplayValue();
}
}
```


Objects Creating Objects

Objects creating objects

in class NumberDisplay:

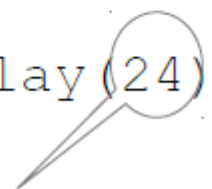
```
public NumberDisplay(int rolloverLimit);
```



formal parameter

in class ClockDisplay:

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



actual parameter

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

Internal & External Call

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)

Internal & External Call

Method calls in ClockDisplay

- internal method call

```
updateDisplay();
```

```
...
```

```
private void updateDisplay()
```

- external method call

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

R@CAP

