CM10227/ CM50258: Lecture 5

Dr Rachid Hourizi and Dr. Michael Wright

November 14, 2016

Housekeeping

- We will post the first large Coursework on Moodle tomorrow (Friday)
- Everyone must complete this coursework (CM10227 and CM50258)
- It describes a considerably larger (Java) task than the lab sheet exercises
- Take an Iterative development approach
  - Start with a program that does very little
  - Make sure that it works
  - Add functionality incrementally
  - Compiling and testing as you go
- Dont leave it to the last minute (Hand in = 24th Nov)
- Do remember that help is available in all the usual places

Back to Java

- Java forces us to
  - **define** a combination of data and operations once (a Class)
  - and then **create** one or more examples (instances) of that class (Objects)

Review

- Class bodies contain fields, constructors and methods.
- Fields store values that determine an objects state.
- Constructors initialize objects.
- Methods implement the behaviour of objects.
    - Mutators (mutator methods) change the state of a object
    - Accessors (accessor methods) provide information about the state of an object

Review

- Objects can make decisions via conditional (if) statements.
- A true or false test allows one of two alternative courses of action to be taken.

Interacting Objects

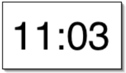
Developping Interactive Classes: A digital clock

- In the following slides, we will (start to) develop and Object Oriented (Java) program
- that provides the data structures and functionality that we will need to create a digital clock

- We will take a similar approach to the one that underpinned the TicketMachine code in the last lecture
- i.e. develop one class that contains a main method (ClockDisplay in this case)
- ... and then develop other class(es) that will be instantiated within that main method

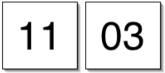
- As we do so, we will consider challenges of **abstraction** and **modularization**:
  - **Abstraction** is the ability to ignore details of parts to focus attention on a higher level of a problem.
  - **Modularization** is the process of dividing a whole into well-defined parts, which can be built and examined separately, and which interact in well-defined ways.

Modularizing the clock display



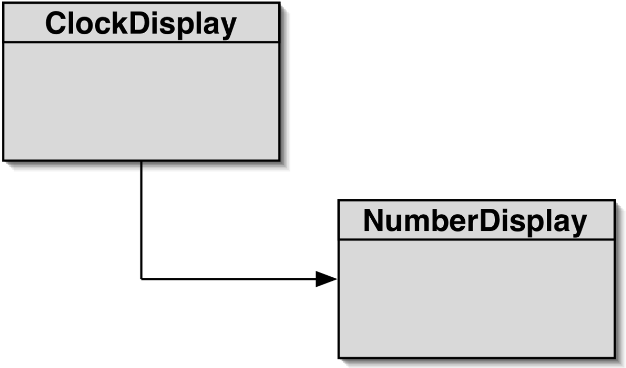
One four-digit display?

Or two two-digit displays

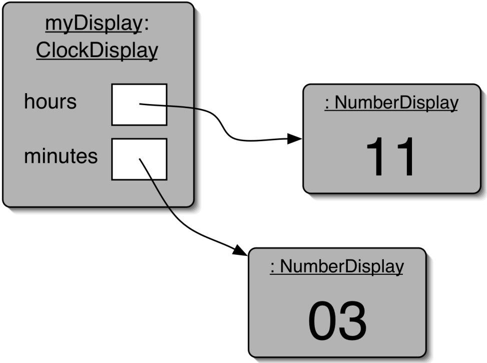


- A sensible approach if we go for 2*2-digit displays (‘NumberDisplays’) is to
    - write a NumberDisplays class which allows us to show any two digit number (i.e. a template for all NumberDisplays)
    - write a ClockDisplay class that creates two NumberDisplay instances
    - In other words, develop one class
    - that (in turn) creates two instances of another class (two Objects)

Class diagram



Object diagram



- Class Diagrams
  - Show the classes of an application and the relationships
  - between them
  - Give information about the source code
  - Static view of the program
- Object Diagrams
  - Show objects and their relationships at one moment in time during the execution of the program
  - Dynamic view of the program

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.
}
```

- We will use the incremental development approach described above:
- Starting with class comments and a class definition for the NumberDisplay class (no import statements are needed)
- all of the code needed to create a (very basic) DigitalClock will be available on Moodle

- But first a few words on Java Strings:
  - Java provides a String class
  - which in turn means that we can use String as a data type
  - Now that we know a little more about the construction of a Java class, we can guess that theString class provides us with at least one constructor
  - in fact it provides us with more than one
  - but the simplest is as follows:

```
public class HelloWorld {

    public static void main(String[] args) {

        //Call String constructor to create new String
        String greeting = "HelloWorld";

        System.out.println(greeting);
    }

}
```

- The String class also provides us with accessors
- e.g. length()

```java
public class HelloWorld {

    public static void main(String[] args) {

        String greeting = "HelloWorld";
        int strLength = greeting.length();

        // Concatenates strLength to an emptyString
           and prints result to the screen
        System.out.println(""+strLength);
    }
}
```

- : Note the . notation used to call an accessor method of String greeting
- : Note also the use of "+" to concatenate strLength to a String
- : Finally, note the possibility of concatenating ints (or floats or chars) to a String - much easier than C.

- the use of + to concatenate Strings is extremely common in java
- though a more formal concatente() method does exist
- being able to concatenate any object to a string is also a big help when printing
- this means that we do not need to use display formatting when printing combinations of Strings and values
- we can simply print a long concatenation expression (starting with a String)

```
public class HelloWorld {

    public static void main(String[] args) {

        System.out.println(""+4.0+"Hello"+5);
    }
}
```

- What is actually happening is that the non String data is being represented in a String before printing
- using the toString() method that is provided in each class
- you may not like the way that Java represents data of other types in a String
- but some representation is always possible.

- Importantly, however, we cannot change the contents of a Java String once it has been created (Unlike a C String)
- We cannot, for example, create a String and change the fourth letter.
- The Java compiler will simply return an error if we try

- We can describe this situation as one in which the Strings class does not provide us with mutator methods
- Java Strings are, therefore described as immutable
- definition: An immutable data type is a type whose state (contents) cannot be changed after creation.
- defintion: A mutable data type is a type whose data members, such as properties, data and fields, can be modified after its creation.

- Now we can return to developing a NumberDisplay class
- following the iterative development approach, we will start with a very simple version of the code
- i.e. a version which simply defines the class but provides neither fields nor methods

```
/**
*Add comments describing class here
 */
public class NumberDisplay{
}
```

- We can then add a definition of the fields needed by NumberDisplays

```
private int limit;
private int value;
```

- Next, we define constructors for the NumberDisplays

```
/**
 * Constructor for objects of class NumberDisplay
 */
public NumberDisplay(int rollOverLimit)
{
    limit = rollOverLimit;
    value = 0;
}
```

- We can also define the Accessors for the Number Display class

```java
/*
 * Return the current value.
 */
public int getValue()
{
    return value;
}
/*
 * Return current value as a two-digit String.
 * If value < 10, pad with leading zero.
 */
public String getDisplayValue()
{
    if(value < 10)
        return "0" + value;
    else
        return "" + value;
}
```

- ...and finally the mutators

```
/**
  * Set initial value.
  * If value<0 or over limit, do nothing.
  */
 public void setValue(int replacementValue)
 {
     if((replacementValue >= 0) && (
        replacementValue < limit))
         value = replacementValue;
 }
 /**
  * Increment the display value by one,
  * roll over if limit is reached.
  */
 public void increment()
 {
     value = (value + 1) % limit;
 }
```

- Having written code that defines the NumberDisplay class
- i.e. a template for all NumberDisplay Objects
- we can now write a ClockDisplay class
- That uses two NumberDisplay Objects

```
/*
* Add comments describing ClockDisplay
 */
public class ClockDisplay
{
}
```

Adding Detail: Fields, Constructors, Mutators, Accessors

```
private NumberDisplay hours;
private NumberDisplay minutes;
private String displayString;
```

```
 /*
 * Constructor for ClockDisplay objects, setting
    time to 00:00
 */
public ClockDisplay()
{
    hours = new NumberDisplay(24);
    minutes = new NumberDisplay(60);
    updateDisplay();
}
/*
 * Constructor for ClockDisplay objects,
    specifying time
 */
public ClockDisplay(int hour, int minute)
{
    hours = new NumberDisplay(24);
    minutes = new NumberDisplay(60);
    setTime(hour, minute);
}
```

```
/*
 * Mutator that sets time
 */
public void setTime(int hour, int minute)
{
    //some code
}
 /*
 * Mutator that updates clock by one minute every
    minute
 */
public void timeTick()
{
    //some code
}
```

```java
/*
 * Mutator that updates the internal string that
    represents the display.
 */
private void updateDisplay()
{
    //some code that calls String methods
}
```

```
 /*
 * Return the current time of this display in the format HH:MM.
 */
public String getTime()
{
    //some code
}

/*
 * Update the internal string that represents the display.
 */
private void updateDisplay()
{
    //some code
}
```

Access Modifiers 1

- Note:
  Public variables:
  - can be seen (accessed) and changed (mutated) externally
  - i.e. by Objects of this and other classes

  - e.g. private int value;

  - this usually turns out to be a terrible idea
  - use private variables and accessors/mutators instead
- Private methods
  - private void updateDisplay()
  - can only be called within the Class

Access Modifiers 2

- Note:
- Public methods:
  - e.g. public void increment()
  - can be called externally
  - i.e. by Objects of this and other classes

  - constructors: (Almost) always public
  - mutators: case by case
  - accessors: often public
- Private methods
  - e.g. private void updateDisplay()
  - can only be called within the Class

- Finally, to start our digital clock, we need a third class containing a main() method, which creates a new ClockDisplay
- NB we dont need to do any more work to create the NumberDisplays, the ClockDisplay will do that

```
public class DigitalClock {

    public static void main(String[] args) {

        //Call String constructor to create new
           ClockDisplay
        ClockDisplay cd = new ClockDisplay();

    }

}
```