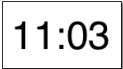


## Developping Interactive Classes: A digital clock

11:03

# Modularizing the clock display



11:03

One four-digit display?

Or two two-digit  
displays



11

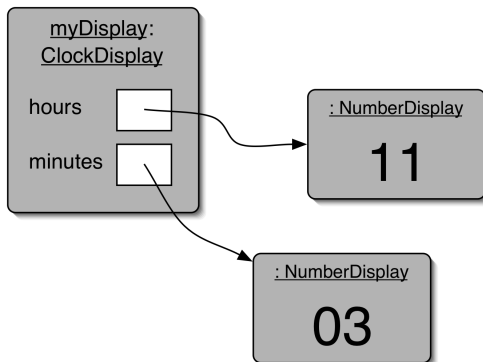


03

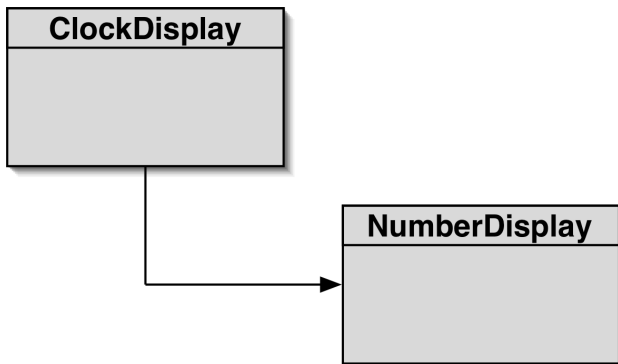
- As we do so, we will consider challenges of abstraction and modularization:
  - ▶ Abstraction is the ability to ignore details of parts to focus attention on a higher level of a problem.
  - ▶ Modularization is the process of dividing a whole into well-defined parts, which can be built and examined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

- 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)

# Object diagram



# Class diagram



## 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.
}
```



- 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

- 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 run a 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 the String 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) {  
  
        String greeting = "HelloWorld";  
  
        // Prints "Hello, World" to the terminal window.  
        System.out.println(greeting);  
    }  
  
}
```

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

```
public class HelloWorld {  
  
    public static void main(String[] args) {  
  
        String greeting = "HelloWorld";  
        int strLength = greeting.length();  
  
        // Concatenates strLength to an emptyString and prints it  
        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 concatenate() 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 converted to a String before printing
- using the toString() method that is provided in each class
- you may not like the way that Java converts data of other types to a String
- but some conversion 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.
- definition: 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

```
/**  
 * The NumberDisplay class represents a digital number display that can hold  
 * values from zero to a given limit. The limit can be specified when creating  
 * the display. The values range from zero (inclusive) to limit-1. If used,  
 * for example, for the seconds on a digital clock, the limit would be 60,  
 * resulting in display values from 0 to 59. When incremented, the display  
 * automatically rolls over to zero when reaching the limit.  
 *  
 * @author Michael Kolling and David J. Barnes  
 * @version 2001.05.26  
 */  
public class NumberDisplay{  
}
```



- followed by 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;  
}
```

- Next, we can define the Accessors for the Number Display class

```
/**
 * Return the current value.
 */
public int getValue()
{
    return value;
}

/**
 * Return the display value (that is, the current value
 * as a two-digit String. If the value is less than ten,
 * it will be padded with a leading zero).
 */
public String getDisplayValue()
{
    if (value < 10)
        return "0" + value;
    else
        return "" + value;
}
```

- ...and finally the mutators

```
/**
 * Set the value of the display to the new specified value.
 * If the new
 * value is less than zero or over the limit, do nothing.
 */
public void setValue(int replacementValue)
{
    if ((replacementValue >= 0) && (replacementValue < limit))
        value = replacementValue;
}

/**
 * Increment the display value by one, rolling over
 * to zero if the 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

```
/**
 * The ClockDisplay class implements a digital clock display for a
 * European-style 24 hour clock. The clock shows hours and minutes. The
 * range of the clock is 00:00 (midnight) to 23:59 (one minute before
 * midnight).
 *
 * The clock display receives "ticks" (via the timeTick method) every minute
 * and reacts by incrementing the display. This is done in the usual clock
 * fashion: the hour increments when the minutes roll over to zero.
 *
 * @author Michael Kolling and David J. Barnes
 * @version 2001.05.26
 */
public class ClockDisplay
{
}
```

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

```

/**
 * Constructor for ClockDisplay objects. This constructor
 * creates a new clock set at 00:00.
 */
public ClockDisplay()
{
    hours = new NumberDisplay(24);
    minutes = new NumberDisplay(60);
    updateDisplay();
}

/**
 * Constructor for ClockDisplay objects. This constructor
 * creates a new clock set at the time specified by the
 * parameters.
 */
public ClockDisplay(int hour, int minute)
{
    hours = new NumberDisplay(24);
    minutes = new NumberDisplay(60);
    setTime(hour, minute);
}

```



```
/**
 * This method should get called once every minute – it makes
 * the clock display go one minute forward.
 */
public void timeTick()
{
    //some code
}

/**
 * Set the time of the display to the specified hour and
 * minute.
 */
public void setTime(int hour, int minute)
{
    //some code
}
```

```
/**
 * 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
}
```

- Note: The ClockDisplay class creates and uses two NumberDisplay Objects
  - ▶ new NumberDisplay(parameter-list)
    - ★ creates a new NumberDisplay Object
    - ★ i.e. we execute the constructor of that class
    - ★ this involves creating sufficient memory to store the values of primitive instance variables and references to object instance variables.

- Public methods:

- ▶ `public void increment()` can be called externally

- Private methods

- ▶ `private void updateDisplay()`
- ▶ can only be called internally used for auxiliary methods

```
public class Point {  
    public int x = 0;  
    public int y = 0;  
  
    //constructor  
    public Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
}
```

- Implementing a test program:

- ▶ The purpose of a test program is to verify that one or more methods have been implemented correctly
- ▶ A test program calls methods and checks that they return the expected results.

- A test program contains the following steps:
  - ▶ Provide a tester class
  - ▶ Supply a main method
  - ▶ Inside the main method, create one or more objects
  - ▶ Apply methods to the objects
  - ▶ Display the results of the method calls - if needed
  - ▶ Display the value that you expect to get - if possible

```

/**
 * This cTest class instantiates a Java digital clock from Kolling
 * & Barnes ClockDisplay and NumberDisplay Classes.
 * It prints the time generated by those classes to the screen
 * approximately every second.
 */
public class cTest{

    static boolean isActive=true;

    //main method
    public static void main(String args[]){

        ClockDisplay myClock = new ClockDisplay();
        do{
            myClock.timeTick();
            System.out.println(myClock.getTime());
            wait(60);
        }
        while(isActive);
    }
}

```

- Note the use of do while (covered in more depth later in the course but similar to C)



- note the use of `System.currentTimeMillis()` which returns the current system time in milliseconds

```
//crude pausing method from http://blog.qarea.com/development/java-de  
public static void wait (int k){
```

```
    long time0=0;
```

```
    long time1=0;
```

```
    time0 = System.currentTimeMillis();
```

```
    do{
```

```
        time1 = System.currentTimeMillis();
```

```
    }
```

```
    while ((time1-time0) < k * 1000);
```

```
    }
```

```
}
```

- Exercise:

- ▶ How would you adjust the DigitalClock code to hold, update and display the time in hours, minutes and seconds (HH:MM:SS) rather than just hours and minutes?
- ▶ add comments to the code provided that describe each of the changes that you would have to make
- ▶ where you are able, also provide the Java code to make these changes