Object Interaction

• You will learn about the design of a digital clock
  – You will study abstraction and modularization
  – You will learn the differences between class diagrams and object diagrams
  – There are fundamentally two categories of types: class types and primitive types
  – You will see how one object can create other objects

• You will learn more about constructors & methods

• You will study a second project, a Mail System

• You will learn to use a debugger
Abstracting and Modularizing

• Consider the display of a clock time, what are some decisions that will have to be made?
  – Will the display be analog or digital
  – Will the value of seconds be displayed
  – Will the display be 12 hour time or 24 hour time
  – Will the ticking of the clock be manual or automatic

• Our initial design will be digital, showing HH:MM where H is a digit for hours and M a digit for minutes, we will use 24 hour time, and ticking will be manual based on user interaction
Abstracting the Display

• Here is a typical display value 13:08

• What features are the same about the display of the hours value and the minutes value?
  – Both are two digits and positive
  – If the number value is a single digit, a mandatory leading zero is added to the left (padding)
  – As the clock runs both values are incremented by 1 to reach the next allowed value

• What are the difference
  – Minutes rollover at 60, so 58 → 59 → 00 → 01
  – Hours rollover at 24, so 22 → 23 → 00 → 01
Modularizing the NumberDisplay

- value and limit will be integer fields
- The constructor sets the limit value based on actual parameter and initializes value to zero
- getValue returns an int value while getDisplay value returns a two character String padded with 0 if needed
- setValue insures the supplied value is between 0 and limit-1
- Increment adds 1 to value and rolls over if necessary; it uses the modular operation %
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.

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

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

    /**
     * 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, pad with a leading zero).
 */
public String getDisplayValue()
{
    if(value < 10)
        return "0" + value;
    else
        return "" + value;
}
/**
 * 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 display value by 1, roll over to zero if the limit is reached.
 */
public void increment()
{
    value = (value + 1) % limit;
}
The ClockDisplay Class

• The ClockDisplay will create two instances of NumberDisplay for the hours and the minutes

• The fields in ClockDisplay are
  
  ```
  private NumberDisplay hours;
  private NumberDisplay minutes;
  private String displayString;
  ```

• Notice how this object diagram differs from the class diagram
ClockDisplay Constructors & Methods

• There are two constructors
  
  public ClockDisplay()
  public ClockDisplay(int hour, int minute)

• Every time the timeTick method is called the clock advances by one minute
  
  public void timeTick()

• Set the time of the display to a specified hour and minute
  
  public void setTime(int hour, int minute)

• Return the current time of this display in the format HH:MM.
  
  public String getTime()

• Update the internal string that represents the display.
  
  private void updateDisplay()
/**
 * The ClockDisplay class implements a digital clock display for a
 * 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.
 */

public class ClockDisplay{

    private NumberDisplay hours;
    private NumberDisplay minutes;
    private String displayString;  // simulates the actual display

    /**
     * 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()
{
    minutes.increment();
    if(minutes.getValue() == 0) {  // it just rolled over!
        hours.increment();
    }
    updateDisplay();
}
/**
 * Set the time of the display to the specified hour and minute.
 */
public void setTime(int hour, int minute)
{
    hours.setValue(hour);
    minutes.setValue(minute);
    updateDisplay();
}

/**
 * Return the current time of this display in the format HH:MM.
 */
public String getTime()
{
    return displayString;
}

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

• The operators are:  && (and),  || (or),  ! (not)

• Example from our program
  
  if((replacementValue >= 0) && (replacementValue < limit))

• Write logical expressions for the following
  
  – A person is a teenager (assume variable age)

  – True if an income is in the bottom five percent (less than $10,000) or in the top five percent (> $100,000)

  – A person is not a teenager (write in two different ways)
The Modulo Operator

• Integer division produces two answers, a quotient and a remainder
  – The quotient  $41/7$ results in 5
  – The remainder  $41\%7$ results in 6

• Example from our program
  
  \[\text{value} = (\text{value} + 1) \% \text{ limit};\]

• Other uses of modulo
  – To test whether an int is even (how would you do this)
  – To find the units digit of an int value
  – To find the hundreds digit of an int value
Internal and External Method Calls

• When you call a method in a different class
  – You give the instance variable, a dot, and the method as in minutes.increment()
  – Sometimes there is more than one dot, as in System.out.println("hello")
  – If a value is returned from a method, it can be part of a more complex expression, as in
    
    if(minutes.getValue() == 0)

• If the method is internal to the class, then it can be called directly without the prefix of an instance, as in
  
  setTime(hour, minute);
Lab 6: new clock program

• Copy Lab6 from the CLASSDAT directory to your local My Documents directory

• Add in the display of seconds, so the display would now look like **09:26:43**
  - Implementing this change will require many small changes, starting with private NumberDisplay seconds;

• Leave the clock running in 24 hour mode, but change the display String to show 12 hour mode and the suffix am or pm as appropriate
Lab 7: Making the Clock Tick

• In our drawing animation program, the Canvas class use a wait method to slow down the display

Here is the wait method

```java
/**
* Wait for a specified number of milliseconds before finishing.
* This provides an easy way to specify a small delay which can be
* used when producing animations.
* @param  milliseconds  the number
*/
public void wait(int milliseconds)
{
    try
    {
        Thread.sleep(milliseconds);
    }
    catch (Exception e)
    {
        // ignoring exception at the moment
    }
}
How Do We Introduce Wait?

• What should our parameter value be if we want the clock to tick every second?
• In which class, ClockDisplay or NumberDisplay, should we put the wait method?
• We will make the timeTick act like a clock start
• For simplicity we will print to the text terminal the value of the time every second
• We will start with a clock that already has the seconds added to the time measurement
The Basic Loop for Repetition

while (true) { // this is an infinite loop
    // print the current time to the text terminal
    // wait for one second
    // increment the value for seconds
    // if the value of seconds rolled over to zero,
    // increment the value of minutes
    // if the value of minutes rolled over to zero,
    // increment the value of hours
}

• This program has a subtle logical error; let’s see if we can find it by completing the program and running it
The Mail System

- Our mail system will consist of three classes: MailServer, MailClient, and MailItem
- When this program is run, you should create a MailServer and give it a name
- You should create two mail clients and give them different names
- Each client can send the other client a message by specifying the name of the server, the name of the recipient, and the message itself
- Run this program and experiment with how it works
The MailItem Class - 1

- This class is primarily used to transfer data; it is used internally so the user does not have to instantiate instances
- The from, to and message fields are all Strings

```java
/**
 * Create a mail item from sender to the given recipient, containing the given message.
 * @param from The sender of this item.
 * @param to The intended recipient of this item.
 * @param message The text of the message to be sent.
 */

public MailItem(String from, String to, String message) {
    this.from = from;
    this.to = to;
    this.message = message;
}
```
• There are three accessor methods
  
  ```java
  public String getFrom()
  {
      return from;
  }
  public String getTo()
  {
      return to;
  }
  public String getMessage()
  {
      return message;
  }
  ```

• There is a single print method
  
  ```java
  public void print()
  {
      System.out.println("From: "+ from);
      System.out.println("To: "+ to);
      System.out.println("Message: "+ message);
  }
  ```
The MailClient Class - 1

// The server used for sending and receiving.
private MailServer server;
// The user running this client.
private String user;

/**
 * Create mail client run by user and attached to a given server.
 */
public MailClient(MailServer server, String user)
{
    this.server = server;
    this.user = user;
}

public MailItem getNextMailItem()
{
    return server.getNextMailItem(user);
}
/**
 * Print the next mail item (if any) for this user to the text terminal.
 */
public void printNextMailItem()
{
    MailItem item = server.getNextMailItem(user);
    if(item == null) {
        System.out.println("No new mail.");
    } else {
        item.print();
    }
}

/**
 * Send the given message to the given recipient via the attached mail server.
 * @param to The intended recipient.
 * @param mess A fully prepared message to be sent.
 */
public void sendMessage(String to, String message)
{
    MailItem mess = new MailItem(user, to, message);
    server.post(mess);
}
The MailServer

• The class uses many features that we have not yet discussed, so we will defer discussion until a later time

• It is important to learn the tools available in a program development environment

• So far you have learned to use the inspector to check data values associated with an object

• Now you will learn how to use a simple debugger that will allow you to control the execution of a program either by single stepping through instructions or running until a breakpoint is encountered
Setting up Program Execution

• You must create the test conditions that you want to investigate in the debugger

• For this program,
  – Create an instance of a MailServer
  – Create a MailClient named sophie
  – Create a MailClient named juan
  – Have sophie send juan a message
  – You are now ready to use the debugger
Setting and Clearing Breakpoints

- Open the source code file where you want the breakpoint and position the cursor on the desired line
- We want to set a breakpoint at the first line of executable code in `printNextMailItem`
- To set (or clear) a breakpoint, select Tools, then set/clear breakpoint; the previous value will be toggled
- You can have multiple breakpoints but remember breakpoints can only be set after compiling the file
Encountering the Breakpoint

• Have Juan call getNextMailItem; the editor screen now looks like this

• Notice that the stop sign has a right pointing arrow threw it indicating we are stopped before this line executes

• On the main BlueJ screen, under view check the show debugger option
Viewing the Debugger Results

- The debugger screen is shown to the right
- We are at a breakpoint in the program
- print NextMailItem has just been called
- The instance variables are the MailServer Juan is using and the user name is “Juan”
- Now click on “Step” to execute the current instruction
Inspecting the MailItem

- A new local variable appears:
  `mailItem item = <object reference>`
- By clicking on this line in the debugger the information to the right appears
- We can see that this is the mail sent from Sophie to Juan
- If we single step again, the test in the if statement executes; that test was `(item == null)` which we know must be false since we just inspected this item
- Therefore program execution stops before the first executable statement in the “else” part of the “if”
Finishing the Method

• If you click Step now, item.print() will be executed:
  From: Sophie
  To: Juan
  Message: Hello

  appears in the terminal window

• The current line to be executed in the editor is the ending brace } in printNextMailItem

• Other debugger option
  - “Step” performs method calls like a single instruction;
    “Step Into” will go into the called method and execute line by line
  – “Continue” will resume program execution until a breakpoint is encountered

• Remember, recompiling resets all breakpoints
Lab 8: Modifying the Mail System

- This lab will involve changing the mail system in two ways: add a subject line to each mail item and add a method that allows you to print all mail waiting for you.

- Adding the subject line
  - In MailItem add a String field for subject, change the constructor, add an accessor method, and change the print so subject appears right before the message.
  - In the other classes, add subject to the constructor parameter list or other method calls where it is appropriate.
Printing All Your Mail

• Write a new method printAllMailItems() that fetches each item from the server, prints it, and continues until there are no items remaining.
• This requires a “while” loop that we will learn about formally in the next chapter.
• Here is the pseudocode:
  get a mail item
  while (the mail item is not null) {
    print the mail item
    get a mail item
  }
  print there is no more mail