Chapter 3 Object Interaction

Main Concepts

Abstraction	Object creation	Method goals
Modularization	Object diagrams	debuggers

Java Constructs:

Class types	Logical operators (&&,||)	String concatenation	Modulo operator (%)
Object construction (new)	Method calls (dot notation)	*this*	

Chapter 3.2 Abstraction and modularization

- *Abstraction :* Seeing complex programming problems as series of sub-problems but concentrating on the big picture.
- *Modularization* : process of dividing large, complex problems into smaller parts (modules).

Chapter 3.3 Abstraction in Software

- Sub-components of complex program are objects (and Classes).
- Complex program constructed from linking objects.

Chapter 3.4 Modularization in clock example

- Need to look at digital clock in abstract = single display with four digits.
- More abstract view = two separate two digit displays – hours, minutes.
- More abstract view = each display starts at 0 , rolls to a max number, returns to 0.
- Clock Plan:
    - o Create Class for two-digit number display
        - o Create accessor method to get value
        - o Create two mutator methods to set and increment value.
    - o Create two objects of class with different limits (hour, min) to make clock.

Chapter 3.5 Implementing Clock Display

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

- Variables used to hold limit of hour or minutes, and current value.
- Classes can define types (i.e. NumberDisplay = type for hours and minutes in Class ClockDisplay)

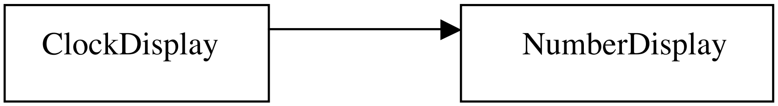
```
public class ClockDisplay
{
   private NumberDisplay hours;
   private NumberDisplay minutes;
```

- Field Declaration signature = private type nameoffield;

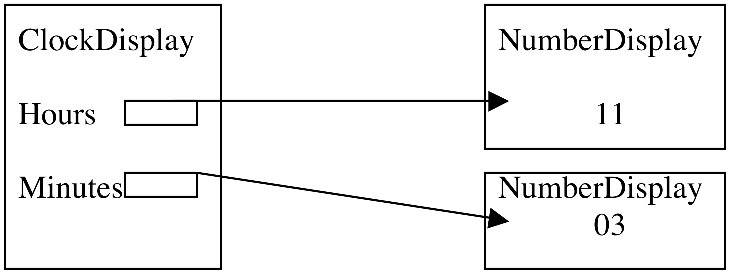
- Type tells what kind of data can be stored in field. If type is a class, then field can hold objects of that class (Objects in one class can hold data from other class.)

Chapter 3.6 Class Diagrams vs Object Diagrams.

- Class Diagram =
    - shows classes of an application and their relationships.
    - Gives information about source code (which class has other in code)
    - Presents static view of program



-
    - ClockDisplay makes use of Number Display. NumberDisplay appears in code of ClockDisplay. ClockDisplay *depends* on NumberDisplay.
- Object Diagram =
    - Shows objects and their relationship at time program is executed
    - Gives info about objects at runtime.
    - Presents dynamic view of program.



- When writing Java programs, important to use object diagram view in planning.

Chapter 3.7 Primitive Types and Object Types

- Primitive Types =
    - non-object types. I.e. int, Boolean, char, double, long.
    - Pre-defined in Java Language
    - See Appendix B for list of types.
    - Primitive type values are stored in variable

- Object Types =
    - Created in objects by programmer
    - Do not store a value but point to an object where value can be found.

Chapter 3.8 Source Code for Clock Display

Source Code or NumberDisplay

```
/**
 * 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
{
   private int limit;
   private int value;
```

Creates variables for value and limit of times to display

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

Constructor initializes *limit* and *value*

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

Accessor method allows other objects to read current 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;
   }
```

Adds “0” to digit to make 4 digit display

```
/**
 * 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;
}
```

Mutator method limits value >0 ,<limit

```
/**
 * Increment the display value by one, rolling over to zero if the
 * limit is reached.
 */
public void increment()
{
   value = (value + 1) % limit;
}
```

Uses modulo operator to set value to 0 when reach limit

```
}
```

- Logical operators: in conditional statememts:
    - && (and) -- if ((condtition 1) && (condition 2)) result;
        - is true if 1 and 2 are both true
    - || (or) -- if ((condtition 1) || (condition 2)) result;
        - is true if 1 or 2 are true
    - ! (not) – if !(condition 1) result;
        - true if 1 is false; false if 1 is true.

Source Code for Clock Display

```
/**
 * 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;    // simulates the actual display
```

Type NumberDisplay
refers to other class

```
   /**
    * 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 creates new objects.

- object (ClockDisplay()) creates object of type NumberDisplay and assigns value to called for parameter.
- Syntax = …. new ClassName (parameter value)
  … new NumberDisplay (24);
- Runs constructor for class NumberDisplay and sends value, 24, to parameter call (int rollOverLimit) → formal parameter; new NumberDisplay (24) → actual parameter

```
/**
 * 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);
}
```

- Second constructor sets alternate method of setting clock start up parameters, hour, minute.
- Overloading a constructor or method = good technique for adding alternative methods to accomplish a task.

```
/**
 * 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();
}
```

- Method Call:
    - Internal Method Call – calls method within same class.
        - Private void updateDisplay () – calls updateDisplay method from within class, ClockDisplay
        - Syntax = methodname (parameter – list)
        - Method call calls other method, returns to next line of calling method.
        - Important to match method name and parameter list because other methods may have same name due to method overloading.
    - External Method Call – can call methods of other class
        - Use ‘dot’ notation – minutes.increment();
            - Calls *minutes* object in *increment* method in NumberDisplay.
            - Syntax = object.methodName (parameter-list).
        - If(minutes.getValue() ==0{   ----- calls get.Value in NumberDisplay. If value = 0, then roll over the clock. Then update display by calling undateDisplay.

```
/**
 * 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();
}
```

- Calls setValue method for *minutes* and *hours* objects, then updates display through updateDisplay () call.

```
/**
    * 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();
   }
}
```

- Updates display by calling getDisplayValue from NumberDisplay for hours and minutes; concatenating “:” between hours and minutes and creating string called displayString.

Chapter 3.12 …object interaction

- Debugger = tool to find bugs in programs. Exams program one line of code at a time.
- Mail-system projet

Source Code for MailItem

```
/**
 * A class to model a simple mail item. The item has sender and recipient
 * addresses and a message string.
 * @author David J. Barnes and Michael Kolling
 * @version 2001.05.30
 */
public class MailItem
{
   // The sender of the item.
   private String from;
   // The intended recipient.
   private String to;
   // The text of the message.
   private String message;
```

→ *Creates "fields": from, to, message*

```
   /**
    * 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;
   }
```

→ *Creates "parameters": from, to, message*

- This.from = from; calls closest enclosing block for value of "from". I.e. this constructor.
- This is useful when need to use same name for both field and parameter. Calls closest use of word.
-

Souce Code for MailClient

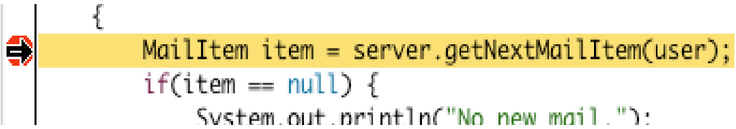
*/***

- *A class to model a simple email client. The client is run by a*
- *particular user, and sends and retrieves mail via a particular server.*
- *@author David J. Barnes and Michael Kolling*
- *@version 2001.05.30*

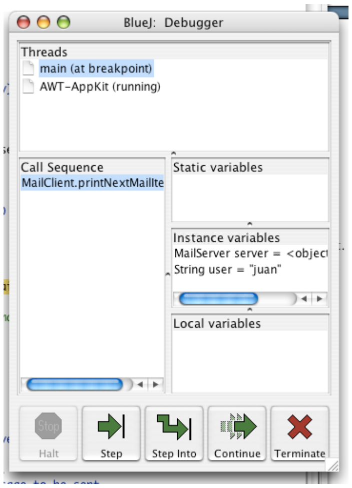
```
 */
public class MailClient
{
    // The server used for sending and receiving.
    Private MailServer server;
    // The user running this client.
    Private String user;

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

    /**
•    Return the next mail item (if any) for this 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);
    }
}
```

Chapter 3.13 debugger

- breakpoint = flag on line of code that stops execution at that point.



```
        {
            MailItem item = server.getNextMailItem(user);
            if(item == null) {
                System.out.println("No new mail.");
```



main (at breakpoint)
AWT-AppKit (running)
MailClient.printNextMailIte
MailServer server = <object
String user = "juan"

- when method from object in object window see degugger window:
- No local variables because line of code (MailItem..) was not run due to breakpoint.
- Use Step button to move one line at a time through program

Vocab: abstraction, modularization, classes defined types, class diagram, object diagram, object references, primitive type, object creation, overloading, internal method call, external method call, debugger.