M301: Software Systems & their Development

Block 1: Introduction to Java

Unit 5: Swing, AWT and Applets

Aims of the unit:

- investigate Java user-interface components, especially Swing components;
- introduce applets;
- study the classes for inputting and outputting data.

Block 1: Introduction to Java

Unit 5: Swing, AWT and Applets

Section 1: Swing and AWT Packages

The Abstract Windowing Toolkit (AWT) provides basic facilities for drawing graphics, e.g. drawing the lines and shapes as used in cannon and pinball worlds. AWT has been a core part of Java since the beginning, but Swing was only introduced with Java 2.

Like the AWT, Swing provides GUI components — in fact they have similar names and functions (e.g. `Frame` is similar to `JFrame` and `Button` similar to `JButton`). The main difference is that while the AWT components vary according to the underlying operating system, Swing components do not. For example, if you create an AWT button it will look like a Windows button on Windows based PC and a Macintosh button on a Macintosh. On the other hand, a Swing component will look the same on any platform. Swing is also larger and more comprehensive than AWT.

On the other hand, Swing components are much more computationally intensive, since they operate with only minimal operating system support (i.e. a Swing-based interface may take longer to draw than would an interface based on the earlier AWT components).

In general, given the superiority of Swing, you are not recommended to use AWT components in any Java 2 application.

Objectives:

On completing this section you should be able to:

- explain the importance of HCI design;
- describe some of the factors that contribute to successful HCI design;
- use a layout manager;
- understand the construction of, and extend, the Color Display case study which forms part of this section;
- describe what is meant by an anonymous class;
- include menus, text boxes and panels into your window applications;
- construct a generic ‘Quit’ menu.

IMPORTANT: Read Section 5.1.3 in Block1 which talks about HCI

IMPORTANT: Do practical activity in Section 6 of the *IDE Handbook*

Summary of Chapter 13

The AWT/Swing Class Hierarchy

See Figure 13.1 in Budd, pp. 212.

The class Object is the parent class of all classes in Java. Methods defined in this class include the following:

Method	**Description**
equals(anObject)	Return true if objects is equal to arguments
getClass()	Return the class of an object
hashCode()	Return a hash value for an object
toString()	Return a string representation of an object

A Component is something that can be displayed on a two-dimensional screen and with which the user can interact. Attributes of a component include a size, a location, foreground and background colors, whether or not it is visible, and a set of listeners for events. Methods defined in this class include the following:

Method	**Description**
setEnabled(boolean)	Enable/disable a component
setLocation(int,int), getLocation()	Set and get component location
setSize(int,int), getSize()	Set and get size of component
setVisible(boolean)	Show or hide the component
setForeground(Color), getForegrouns()	Set and get foreground colors
setBackground(Color), getBackgrouns()	Set and get background colors
setFont(Font), getFont()	Set and get font
repaint(Graphics)	Schedule component for repainting
paint(Graphics)	Repaint component appearance
addMouseListener(MouseListener)	Add a mouse listener for component
addKeyListener(KeyListener)	Add a keypress listener for component

In the AWT library graphical elements such as buttons, checkboxes, scroll bars and the like are declared as types of components.

A Container is a type of component that can nest other components within it. A container is the way that complex graphical interfaces are constructed. A container maintains a list of the components it manipulates, as well as a layout manager to determine how the components should be displayed. Methods defined in this class include the following:

Method	**Description**
setLayout(LayoutManager)	Set layout manager for display
Add(Component), remove(Component)	Add or remove component from display

A Window is a type of Container. A window is a two-dimensional drawing surface that can be displayed on an output device. A window can be stacked on top of other windows and can be moved either to the front or back of the visible window. Methods defined in this class include the following:

Method	**Description**
show()	Make the window visible
toFront()	Move window to front
toBack()	Move window to back

A Frame is a type of window with a title bar, a menu bar, a border, a cursor, and other properties. Methods defined in this class include the following:

Method	**Description**
setTitle(String), getTitle()	Set or get title
setCursor(int)	Set cursor
setResizable()	Make the window resizable
setMenuBar(MenuBar)	Set menu bar for window

Finally, the Swing class JFrame is a subclass of the earlier AWT class Frame.

A JFrame will maintain a Container, called the *contentPane*, that will hold all the graphical elements of the window. In addition, the method overrides several of the methods inherited from parent classes.

The Layout Manager

The layout manager places in position the list of components held in a container.

There are a variety of standard layout managers, each of which will place components in slightly different ways.

The container holds the layout manager as a data field. But in actual fact, the variable that holds the layout manager is polymorphic. While the Container thinks that it is maintaining a value of type LayoutManager, in fact it will be holding a value from some other type (e.g. BorderLayout) that is implementing the LayoutManager interface.

There are three different mechanisms at work here: inheritance, composition, and implementation of interface. Each is serving a slightly different purpose.

- *Inheritance* allows the application class to use the setLayout() method from Container, which enables an instance of the layout manager to be added to the application frame.
- *Composition* is involved because the class Container *has-a* LayoutManager (i.e. a data field that references a particular layout manager).
- *Implementing an interface* is required because the actual layout manager being used is an object of a class that implements the LayoutManager interface.

Layout Manager Types

There are five standard types of layout managers:

1) BorderLayout

The BorderLayout can manage no more than *five* different components.

This is the default layout manager for applications constructed by subclassing from JFrame.

The five locations in this layout manager are: North, West, East, South and Center.

Not all five locations need to be filled. If a location is not used, the space is allocated to the remaining components.

Example:

Add a button to the top of the display.

```
// In JFrame, we must first use the method getContentPane in order to
// access the container holding the window content

getContentPane().add("North", new JButton("quit"));
```

2) GridLayout

The GridLayout creates a rectangular array of components, each occupying the same size portion of the screen.

Using arguments with the constructor, the programmer specifies the number of rows and columns in the grid.

Two additional integer arguments can be used to specify a horizontal and vertical space between the components.

Example:

Make a 4 by 4 grid with 3 pixels between each element.

```
JPanel p = new JPanel();

p.setLayout(new GridLayout(4,4,3,3));

p.add(new ColorButton(Color.black, "black"));
...
```

3) FlowLayout

The FlowLayout places components in rows left to right, top to bottom.

Unlike the layout created by a GridLayout manager, the components managed by a flow layout manager need not all have the same size.

When a component cannot be completely placed on a row without truncation, a new row is created.

The flow manger is the default layout for the class JPanel.

4) CardLayout

The CardLayout stacks components vertically. Only one component is visible at any one time.

The components managed by a card layout manger can be named (using the add method). Subsequently, a named component can be made the visible component.

Example:

```
CardLayout lm = new CardLayout();

JPanel p = new Jpanel(lm);

p.add("One", new JLabel("Number One"));

p.add("Two", new JLabel("Number Two"));

p.add("Three", new JLabel("Number Three"));
.
.
.
lm.show(p, "Two"); //show component "Two"
…
```

5) GridBagLayout

This is the most general type of layout managers.

The GridBagLayout allows the programmer to create non-uniform grid of squares and place components in various positions within each square.

User Interface Components

In the Swing extension of the AWT, all the user interface components are subclass of the parent class JComponent which is a subclass of the Container class (see Figure 13.5 in Budd, pp. 218).

Containers assume only that the elements they will hold are instances of subclass of class JComponent. In fact, the values they maintain are polymorphic and represent more specialized values (e.g. buttons or scroll bars).

Thus, the design of the user interface construction system depends upon the mechanism of inheritance, polymorphism, and substitutability.

VERY IMPORTANT: Read the description of the different user interface components (e.g. Labels, Buttons, etc.) from Budd, pp. 218-225

Panels

A JPanel is both a Container and a Component. A panel represents a rectangular region of the display.

It can hold its own layout manager for inserting components into it. Like any component, it must in turn be inserted into the application display.

Example:

The three scroll bars on the left are placed on a panel. This panel is laid out using a BorderLayout manager. The method to create and return this panel is described as follows:

```
private JPanel makeScrollBars() {
            JPanel p = new JPanel();
            p.setLayout(new BorderLayout());
            p.add("West", redBar);
            p.add("Center", greenBar);
            p.add("East", blueBar);
            return p;
      }
```

The panel return as the result of this method is then placed on the left side of the application window.

```
Container p = getContentPane();
...
p.add("West", makeScrollBars());
...
```

JScrollPane

A JScrollPane is in many ways similar to a JPanel. Like a panel, it can hold another component. However, a JScrollPane can only hold one component, and it does not have a layout manger.

If the size of the component being held is larger than the size of the JScrollPane itself, scroll bars will be automatically generated to allow the user to move the underlying component.

Unnamed (or Anonymous) Classes

In several of the earlier case studies we have seen situations where inheritance is used to override an existing class, yet only one instance of the new class is created.

One way to do this is illustrated from the Cannon Game:

```
public class CannonWorld extends JFrame {
  ...

  public CannonWorld() {
    ...
    fire.addActionListener(new FireButtonListener());
    ...
 }

  ...

 private class FireButtonListener implements ActionListener{
   public void actionPerformed(ActionEvent evt){
     ...
   }
 }
```

Note how the constructor for the class CannonWorld creates an instance of the inner class FireButtonListener. This is the only instance of this class created.

An alternative way to achieve the same effect would be as follows:

```
public class CannonWorld extends JFrame {
  ...

  public CannonWorld() {
    ...
    fire.addActionListener(new ActionListener () {
          public void actionPerformed(ActionEvent evt){
              ...
          }
        });
  ...
  }
}
```

Notice that in this example the object being created is declared only as being an instance of ActionListener. However, a class definition follows immediately the ending parenthesis, indicating that a new and *unnamed* class is being defined. This class definition would have exactly the same form as before, ending with a closing curly brace. The parenthesis that follows this curly brace ends the arcument list for the addActionListener call.

An advantage to unnamed classes is that it avoids the need to introduce a new class name (in this case, the inner class FireButtonListener).

A disadvantage is that such expressions tend to be difficult to read.

IMPORTANT: Study the Color Display case study in Budd, pp. 226-231

Dialogs

A JDialog is a special purpose window that is displayed for a short period of time during the course of execution, disappearing thereafter.

Dialogs are often used to notify the user of certain events, or to ask simple questions.

A dialog must always be attached to an instance of JFrame, and disappears automatically when the frame is hidden.

Dialog windows can be *modal* or *nonmodal*.

- A modal dialog demands a response from the user, and it prevents the user from performing any further action until the dialog is dismissed.
- A nonmodal dialog (or modeless dialog), can be ignored by the user.

Whether or not a dialog is modal is determined when the dialog is created. The two arguments used in the constructor for the dialog are the application Frame and a Boolean value that is true if the dialog is modal.

```
// create a new nonmodal dialog in current application

JDialog dig = new JDialog (this, false);
```

Because a JDialog is a type of Window, graphical components can be placed in the dialog area, just as in a JFrame or JPanel.

The default layout manger for a dialog is BorderLayout.

The most common methods used with a dialog are inherited from parent classes. They include: setSize(int,int), show(), setVisible(false), setTitle(String) and getTitle().

The Menu Bar

A Swing bar is a graphical component, it is declared as a sub-class of JComponent, both menu bars and menus act like containers.

A menu bar contains a series of menus, and each menu contains a series of menu items.

An instance of JMenuBar can be attached to a Frame using the method setJMenuBar:

```
...
JMenuBar bar = new JMenuBar();
setJMenuBar(bar);
...
```

Individual menus are named, and are placed on the menu bar using the method add:

```
...
JMenu helpMenu = new JMenu("Help");
bar.add(helpMenu);
...
```

Menu items are created using the class JMenuItem. Each menu item maintains a list of ActionListener objects, the same class used to handle JButton events. The listeners will be notified when the menu item is selected.

```
...
JMenuItem quitItem = new JMenuItem("Quit");
quitItem.addActionListener(new QuitListener());
helpMenu.add(quitItem);
...
```

Block 1: Introduction to Java

Unit 5: Swing, AWT and Applets

Section 2: Applets

There is little doubt that much of Java's growing popularity derives from its networking capabilities, in particular, its applet mechanism.

Applets allow web browsers to execute applications, downloaded from other computers, in safety: that is, without compromising the safety of the client's machine.

This section provides a brief introduction to applets.

Objectives:

On completing this section you should be able to:

- describe what an applet is and how it differs from the Java applications you have met so far in this unit;
- run, under the control of your own Internet browser, a number of simple applets;
- create and run your own simple applet.

Summary of Chapter 21

IMPORTANT: Do practical activity in Section 7 of the *IDE Handbook*

Although Java is a general purpose programming language that can be used to create almost any type of computer program, much of the excitement surrounding Java has been generated by its employment as a language for creating programs intended for execution across the WWW.

Programs written for this purpose must follow certain conventions, and they differ slightly from programs designed to be executed directly on a computer. Herein, we will discuss these differences and how to create programs for the Web (i.e. applets).

Applets and HTML

Applications written for the WWW are commonly referred to as *applets*.

Applets are attached to documents distributed over the WWW. These documents are written using the HyperText Markup Language (HTML) protocol.

A Web browser that includes a Java processor will then automatically retrieve and execute the Java program.

Two HTML tags are used to describe the applet as part of an HTML document. These are the <applet> tag and the <param> tag. A typical sequence of instructions would be the following:

```
<applet
  codebase = "http://www.sun.com"
  code     = "MyApplet.class"
  width    = "400"
  height   = "300"
>
<param name=name1 value="value1">
You do not have a Java enabled browser
</applet>
```

- The <applet> tag indicates the address of the Java program.
- The codebase parameter gives the URL Web address where the Java program will be found.
- The code parameter provides the name of the class.
- The width and height attributes tell the browser how much space to allocate to the applet.
- Just as users can pass information into an application using command line arguments, applets can have information passed into them using the <param> tags. Within an applet, the values associated with parameters can be accessed using the method getParameter().
    - **Example:** String s = getParameter(name1); // s will contain "value1"
- If an applet cannot be loaded (perhaps it is no longer at the location specified in the HTML document), any text between <applet> and </applet> which is not part of a <param> clause, will be displayed by the browser. It is good practice always to include such text to alert the user.

Security Issues

Applets are designed to be loaded from a remote computer (the server) and then executed locally. Because most users will execute the applet without examining the code, the potential exists for malicious programmers to develop applets that can cause significant damage (i.e. erasing a hard drive).

To address this, applets are much more restricted than applications in the type of operations they can perform.

1. Applets are not permitted to run any local executable program.
2. Applets cannot read or write to the local computer's file system.
3. Applets can only communicate with the server from which they originate.
4. Applets can learn only a very restricted set of facts about the local computer.
5. Dialog windows that an applet creates are normally labeled with a special text.

6. In Java 1.2, it is possible to attach a digital signature to applets.

Applets and Applications

A program that is intended to run on the Web is subclasses from Applet (instead of JFrame).

The class Applet provides the necessary structure and resources needed to run a program on the Web.

The Swing class JApplet is a subclass of the AWT Applet class which is a subclass of Panel.

Hence, JApplet inherits the applet functionality of Applet and the graphical component attributes of Panel.

Rather than starting execution with a static method named main, as applications do, applets start execution at a method named init, which is defined in class Applet but can be overridden by users.

There are four methods defined in Applet that is available for overriding by the user:

Method name	**Description**
init()	- Invoked when an applet is first loaded. - Can be used for one-time initialization (similar to the code you would normally found in the constructor for an application).
start()	- Called to begin execution of the applets. - Called again each time the Web page containing the applet is exposed. - Can be used for further initialization or for restarting the applet.
stop()	- Called when a Web page containing an applet is hidden. - Can be used to halt extensive calculations when the page becomes covered.
destroy()	- Called when the applet is about to be terminated. - Should halt the application and free any used resources.

Example:

Suppose a Web page containing an applet as well as several other links is loaded.

- The applet will first invoke init(), then start().
- If the user clicks on one of the links, the Web page holding the applet is overwritten, the method stop() will be invoked to temporarily halt the applet.
- When the user returns to the Web page, the method start(), but not init(), will once again be executed.
- When the user finally exits the page containing the applet, the method destroy() will be called.

Note. The default layout manager for an Applet is a **flow layout** rather than the border layout that is default to applications.

In summary, the main differences between an application and an applet are:

1. Applets are created by subclassing from the class JApplet, rather than from the class JFrame.
2. Applets begin execution with the method init, rather than main.
3. Applets can be halted and restarted as the Web browser moves to a new page and returns.

Block 1: Introduction to Java

Unit 5: Swing, AWT and Applets

Section 3: Input and Output Streams

Objectives:

On completing this section you should be able to:

- describe the structure of the classes in the `java.io` package that provide the facilities to input and output data;
- use the facilities of the `java.io` package to read data from, and write data to, various kinds of sources and destinations respectively;
- use the facilities of the `java.io` package that support the communication between sources and sinks.

IMPORTANT: Read Section 5.3.3 in Block1

Summary of Chapter 14

Streams versus Readers and Writers

The term *stream* represents a programming abstraction and can be thought of as a communication channel between the program and a source (an input stream) or sink (an output stream).

This channel is represented as an object in a Java program, and has associated with it various methods for sending, receiving and manipulating data.

At the lowest level, a stream is a device for transmitting or receiving 8-bit values. Note the emphasis on the action of reading or writing, as opposed to the data itself.

Example:

A *file* is a collection of items stored on an external device (e.g. a floppy disk or CD-ROM). The Java object through which the file is accessed (e.g. `FileStream`) provides the means to access the data values in the file but does not actually hold the file contents.

Although we emphasize that at the lowest levels streams are always pipelines for transmitting 8-bit values, we also note that much of the functionality provided by the various different stream abstractions is intended to permit the programmer to think in terms of higher level units (e.g. transmitting strings or integers or object values).

Input and output can be divided into the stream abstractions (i.e. InputStream and OutputStream), which read and write 8-bit values, and the Reader/Writer classes, which manipulates 16-bit Unicode character values.

Input Streams

The class InputStream is an abstract class, parent to 10 subclasses in the Java library.

The methods common to all subclasses of InputStream are those inherited from InputStream. Budd describes the following main methods (there are others).

- read() reads a single byte from the input and returns it as a positive int (or –1 if the end of the stream was reached). This method will wait until data is available to be read. That is, the program that is executing will be blocked (stopped from executing) until the data is available to be read.
- read(byte[] buffer) reads a collection of values from the input, placing them into a byte array here named buffer, and returns a positive int containing the number of bytes read or –1 if the end of the stream was reached.
- skip(long n) skips bytes from the input and returns a long value containing a count of the number of bytes that were actually skipped.
- available() determines the number of bytes readable without blocking and returns it as an int.
- close closes the input stream.

All the above methods throws an exception, IOException, that will be generated if a file cannot be opened, or in other exceptional conditions.

Input streams can be divided into:

1. *Physical Input Streams:* those based on a physical input source (reading input from a byte array, file, a pipe (a buffered data area used for both reading and writing)),and
2. *Virtual Input Stream:* those based on adding new functionality to a logical input stream.

Physical Input Streams

There are three input stream classes that read from actual data areas:
ByteArrayInputStream
FileInputStream
PipedInputStream

Virtual Input Streams

The classes SequenceInputStream, ObjectInputStream, and the three subclasses of FilterInputStream can be thought of as virtual input classes.

None of these streams actually read characters from any input area, but instead they rely on one or more underlying input streams for their data source. Each adds useful functionality as values are passed through the class.

Example:

```
FileInputStream fs = new FileInputStream("README.ser");
ObjectInputStream os = new ObjectInputStream(fs);
```

One example of a virtual input stream is DataInputStream *is-a* InputStream and therefore implements all its methods. However, it also *has-a* InputStream. This means that it is a wrapper class providing additional functionality for reading an input stream. In DataInputStream, methods are provided to read bytes from the source and return them as a primitive data type.

Example:

To read an int will require reading four characters from the underlying input stream. In response to a call on readInt(), the four-byte values are read and combined together to form the new integer value, which is then returned as the result of the call on readInt().

Similar methods are provided for each of the primitive data types.

IMPORTANT: Read Budd, pp. 242-244 for details

Stream Tokenizer

Although not specifically an InputStream, the class StreamTokenizer provides facilities to break a textual file into single tokens, such as words and numbers. For an example refer to Budd, pp. 244-245.

Output Streams

Like InputStream, the abstract class OutputStream defines minimal functionality:

- write() writes a single byte to the output.
- write (byte[] buffer) writes an array of byte values.
- flush() flush all output from buffers.
- close closes the output stream.

Outputting data to a device one byte/character at a time is an inefficient process. Many subclasses of OutputStream (and Writer as well) will collect data in an internal buffer and only send it to the device when enough has been collected for it to be done efficiently. On occasions, however, the programmer may wish to force the stream to output what is currently in the buffer (even if the buffer has not been completely filled). The method flush() is used to do this. Closing a stream will automatically flush all pending operations.

As we did with input streams, we can divide the description of output streams into two major categories:

1. Those classes that characterize the physical location of the output. There are three classes in this category: ByteArrayOutputStream which writes values into an in-memory byte array; FileOutputStream which writes values to an external file, and PipedOutputStream which writes values to a pipe.
2. Those classes that adds more behavior to an output stream. This category is represented by the class ObjectOutputStream and by the class FilterOutputStream (which generally performs some tasks before sending the values to the underlying output stream) and its subclasses: BufferedOutputStream, DataOutputStream, PrintStream.

IMPORTANT: Read Budd, pp. 246-247 for details

Readers and Writers

Readers and writers manipulate 16-bit Unicode character values, rather than 8-bit bytes.

As with input streams, we can divide the various types of readers into those that directly manipulate a data area (CharArrayReader, StringReader, FileReader) and those that add functionality to data being generated to another reader (BufferedReader, LineNumberReader, FilterReader).

Readers and writers are useful whenever the input or output values are **purely textual**, as opposed to binary data such as colors or images.

Example:

The class BufferedReader is a filter, which must be built on top of another reader. To use it to read text from a file, one would first create a FileReader, and then use the file reader to construct the BufferedReader:

```
FileReader f = new FileReader("filename");
BufferedReader input = new BufferedReader(f);
...
// read line of text from a file
String text = input.readLine();
```

Readers and writers are linked to streams (i.e. InputStream and OutputStream) via the wrapper classes InputStreamReader and OutputStreamReader respectively. For example, the constructor for InputStreamReader takes an InputStream object as an argument and returns an object that responds to Reader methods. Similarly, the constructor for OutputStreamReader takes an OutputStream object as an argument and returns an object that responds to Writer methods.

Example:

The following could be used to read lines from a file that contained Cyrillic characters:

```
// first get access to the file

FileInputStream f = new FileInputStream("filename");

// then convert bytes to characters

InputStreamReader r = new InputStreamReader(f, "MacCyrillic");

// then buffer the input

BufferedReader input = new BufferedReader(r);

// now read text line by line

String text = input.readLine();

while (text != null) {
        .
        .
        .
        text = input.readLine();
}
```