Advanced Topics

4.1 Threads
What Is a Thread?

- A **thread** is a single sequential flow of control within a program.
- Like a program, a thread has a **beginning point**, a **sequence**, and an **end**.
- A thread cannot run on its own. It **runs inside a program** and **uses the resources provided by the program**.
- There can be **two or more threads inside a program**. Each can be run freely on its own (in the point of view of users).
Why threads?

- There are many cases that a program needs to wait for other resources, such as, harddisk, network, monitor, etc.
- It will be inefficient if the CPU just stays there to wait for the program to get the required resource.
- By implementing multi-threading, execution of a thread will be totally independent to the other threads.
- It can do its own work no matter whether the other threads are waiting or not.
- It improves the utilization of the CPU.
Create Threads in Java by subclassing

- The simplest way of create thread in Java is to subclass the `Thread` class.
- The `Thread` class contains a number of methods to help executing a thread. For example,
  - `start` - to start the thread as soon as possible
  - `stop` - to stop the thread as soon as possible
  - `sleep` - to ask the thread to sleep for some time
  - `run` - to specify what the thread should do
  - `yield` - to give up the right of the thread to execute
- The default implementation of `run` does nothing. The actual thread should override the `run` method.
public class SimpleThread extends Thread {
    // Define SimpleThread is a thread
    public SimpleThread (String str) {
        // Constructor - initialize the thread
        super(str);
    }
    public void run() {
        // specify the code of that thread
    }
}

• Since SimpleThread is a subclass, we can create a thread by using the new statement as follows:

public class TwoThreads {
    public static void main (String[] args) {
        new SimpleThread("A").start();
        new SimpleThread("B").start();
    }
}
Create Threads in Java - Runnable Interface

• When a class is already a subclass of other class (such as Applet), it is not possible to be a subclass of Thread, since Java does not support multi-class inheritance
• Need to implement Runnable interface to create threads
• The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread
• The class must define a method of no arguments called run
• A class that implements Runnable can run without subclassing Thread by instantiating a Thread instance and passing itself in as the target
public class Demo extends Group implements Runnable {
    // Define Demo is a subclass of Group but
    // implement Runnable interface

    protected void startGroup () {
        // create the thread named “A”. Indicate that
        // Demo is the target where the thread run
        aThread = new Thread(this, "A");
        // Start the thread. It will in turn call run
        aThread.start();
    }

    public void run() {
        // The run method must be implemented since the
        // class promises to implement the Runnable
        // :
        // Add code for the thread
    }
}
Example

- The **clock** display is a good example to demonstrate the necessity of using thread
- **Clock will only update its screen once per second**
- There is **no reason for the CPU to wait** the clock to update the display and do nothing
- The CPU can do something useful between every second
- A thread for the **clock** display can be implemented in **Java Workshop 2.0** using the **Runnable** interface
Step 1: Set the Layout and Name the Components

• Add a project called `clockThread`. Choose to use GUI Builder.
• From the `panel` created, delete until one cell remain.
• From the `component palette`, add the items `Label` to the cell.
• Name the component as `labelArea`.

Step 2: Set up the framework of `clockThread.java`

• Import the following classes and package:
  ```java
  import java.text.DateFormat;
  import java.awt.Graphics;
  import java.util.*;
  ```
• Add a member variable to `clockThread` class
  ```java
  private Thread clockThread = null;
  ```
• Modify the `Group` class to implement `Runnable` Interface
  ```java
  public class clockThread extends Group implements Runnable;
  ```
Step 3: Update `startGroup()`

```java
protected void startGroup() {
    if (clockThread == null) {
        clockThread = new Thread(this, "timeThread");
        clockThread.start();
        // If the clockThread not exist, create one and
        // start immediately
    }
}
```

Step 4: Update `stopGroup()`

```java
protected void stopGroup() {
    clockThread = null;
    // If the group stops, remove the thread
}
```
public void run() {
    Label labelBody = (Label) gui.labelArea.getBody();
    Thread myThread = Thread.currentThread();
    while (clockThread == myThread) {
        // clockThread will die if not equal myThread
        // see stopGroup()
        Calendar cal = Calendar.getInstance();
        cal.add(Calendar.HOUR, -8);  // set to H.K. time
        Date date = cal.getTime();
        DateFormat dateFormatter =
            DateFormat.getTimeInstance();
        labelBody.setText(dateFormatter.format(date));
        try {
            Thread.sleep(1000);  // update display once
            // per second
            } catch (InterruptedException e) {
            }
    }
}
The Life Cycle of a Thread

- **New Thread**
- **Runnable**
- **Running**
  - `start()`
  - `yield()`
  - `Run() terminate`
  - `stop()`
- **Not Runnable**
- **Dead**
  - `sleep()`, `wait()`, blocked by I/O
  - Awaken condition satisfied

THE HONG KONG POLYTECHNIC UNIVERSITY
Department of Electronic and Information Engineering
Understanding Thread Priority

• Most commercial computers usually have only one CPU, not all runnable threads can run at the same time
• A **scheduling policy** is required to determine which thread should run next if resource is available
Java supports a very simple, deterministic scheduling algorithm called **fixed priority scheduling**

This algorithm schedules threads based on their **priority** relative to other Runnable threads.

Each java thread is given a numeric priority between **MIN_PRIORITY** and **MAX_PRIORITY** (constants defined in Thread class).

When a thread is first created, it **inherits the priority** of the class that generates the thread.

Lower priority thread can execute only **if the higher ones stop or are suspended**.
• Higher priority thread can **preempt** the lower priority one if it is ready
• For threads of the same priority, no **preemption** will take place. The thread will **run until end**
• A thread can give up its right to execute by calling **yield**

**Selfish Threads - not recommended**

```java
public int tick = 1;
public void run() {
    while (tick < 400000)
        tick++;
    // No other same priority
    // thread can execute
    // until it finishes its loop
}
```

• **yield** gives other threads with the same priority a chance to run
• If no equal priority threads are **Runnable**, **yield** is ignored
Synchronizing Threads

• **Threading** will not be that useful if they can only run on its own
• **Threads can work together** to perform a task
• However, it introduces the problem of **resource sharing** since different threads may use the shared resource concurrently
• A typical problem: **producer and consumer problem**
Task: Producer generates number 1-10, Consumer gets these numbers sequentially

If the producer and consumer run freely:

<table>
<thead>
<tr>
<th>Problem 1:</th>
<th></th>
<th>Problem 2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Producer</td>
<td>Consumer</td>
<td>Producer</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>
• **Synchronization** is necessary to avoid incorrect results
  • Synchronization should be performed in two ways
    • To **lock** the shared resource such that if any thread using the shared resource, no other thread can use it
    • To provide **communication** between threads to indicate the status of the current thread

```
Producer       Pool       Consumer
  1  →  1  →  1
  2  →  2  →  2
  3  →  3  →  3
```
How Synchronization is Done in Java?

A. Example

Task:
- For the previous `clockThread` example, add another `soundThread` such that it will generate sound for every 15 seconds

How to do it:
- Every time `clockThread` updates the display, it will increase the value of a data pool
- The `soundThread` gets the value in the data pool and generates a sound if it is equal to 15
B. Data Pool - shared resource

- Both the `clockThread` and `soundThread` need to access the value in data pool
- Data pool should be locked if any one uses it
- Data pool in this case is also called the critical section
- In Java, a critical section is identified with the `synchronized` keyword
- The Java platform associates a lock with every object that has methods declared with the `synchronized` keyword
The implementation of the data pool is as follows:

```java
public class Pool {  // The data pool is named Pool
    private int count;  // A counter is installed to
        // indicate the current count
    private boolean available = false;

    public synchronized int get() {  // With the synchronized keyword, if anyone
        // calls get, the data pool is locked hence
        // other cannot access anything inside it
            return count;
    }

    public synchronized void put() {  // Similarly, if anyone call put, the data pool
        // is locked. Others cannot access anything in it
            count++;
        // Increase the value of count if put is called
    }
}
```
C. Notify Others

- After the `clockThread` (producer) modifies the count, it should notify the `soundThread` (consumer) to get the new value.
- After the `soundThread` gets the value of `count`, it should notify the `clockThread` that `count` is free to update.
- To achieve this, a private variable `available` is added to indicate if a new data is available.
- If the new count is not available, the `soundThread` should wait.
- If the `soundThread` has not got the new value, the `clockThread` should wait.
public synchronized int get() {
    while(available == false) { // Loop if not available
        try {
            wait(); // wait until someone wakes it up
            // The soundThread that calls get at this point
            // will go to the Not runnable state
        } catch (InterruptedException e) {};
    }
    // If clockThread updates count, it will call
    // notifyAll. soundThread will change back to
    // runnable state. ClockThread set available to
    // true. soundThread gets out from the while loop

    available = false;
    notifyAll(); // Wake the clockThread up
    return count; // return the new value of count
}
public synchronized void put() {
    while(available == true) {
        // Loop if previous result has not been got
        try {
            wait();   // wait until someone wakes it up
            // The clockThread that calls put at this point
            // will go to the Not runnable state
        } catch (InterruptedException e) { }}
    // If the soundThread gets count, it will call
    // notifyAll. The clockThread will change back to
    // runnable state. soundThread set available to
    // false. clockThread gets out from the while loop
    count++;
    if (count == 15) {count = 0;} 
    available = true;
    notifyAll();  // Wake the soundThread up
}
D. Create Threads

- To create multiple threads, the object class must implement the `Runnable` interface as in the example of `clockThread`.
- Multiple threads are created using codes like these:

```java
protected void startGroup() {
    if (clockThread == null) {
        clockThread = new Thread(this, "time");
        clockThread.start();
        // clockThread is created and started
    }
    soundThread = new Thread(this, "sound");
    soundThread.start();
    // soundThread is created and started
}
```
E. Define What Threads Should Do

- The `run` method defines what threads should do.
- Every created thread runs the same `run` method.
- Every created thread keeps a copy of the variables in `run`.
- Some `if-then` statements should be used in `run` to identify which thread should run which part of the codes in `run`.

```java
public void run() {
    Thread myThread = Thread.currentThread();
    if (clockThread == myThread) {
        :
    }
    if (soundThread == myThread) {
        :
    }
}
```

The code for the `clockThread`

The code for the `soundThread`
public void run() {
    Thread myThread = Thread.currentThread();
    if (clockThread == myThread) {
        dataPool.put();
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
        }
    }
    if (soundThread == myThread) {
        Label label2Body = (Label) gui.label2Area.getBody();
        while (soundThread == myThread) {
            int value = dataPool.get();
            label2Body.setText(""+value+"");
            if (value == 0)
                { ... Generate Sound ... }
        }
    }
}
Exercise: Develop the applet that implements the above example

Hints:
• Use the codes as shown above
• You may need to redesign the layout and add some codes to complete the task
• The Pool class needs to be contained in a separated file named Pool.java within the same directory of your applet
• Add the created Pool.java to your project using the Add File command of the Project Manager
• For the part of generating sound, see FirstGUI
Advanced Topics

4.2 Animation
Performing Animation

- All forms of **animation** have in common in that they create motion by **showing successive frames at high speed**
- Computer animation  **10 - 20 frames per second**
- To ask Java to perform animation, one needs to set up the **framework** correctly
- Animation usually performs by an **animation loop**
Creating the Animation Loop

- An animation loop draws images frame-by-frame in a loop to form a motion.
- Since this loop is often computational intensive, it should be performed in its own thread to avoid affecting the main program.
public class myclass extends Group implements Runnable{
    int frameNumber = -1;  // the current frame no.
    int delay=100;  // the delay between frames in millisec
    Thread animatorThread;  // the animation loop thread
    Image images[];

    protected void initGroup() {
        // ... initialize all variables and get images.../
    }

    protected void startGroup() {
        // ... create animation loop thread ... //
    }

    protected void run() {
        // ... implement the animation loop thread ... //
    }

    protected void stop() {
        // ... stop the animation loop thread ... //
    }
}
A. A Typical Animation Loop

```java
while ( ... ) {
    // test if the animation thread is still running
    frameNumber++;
    // advance the frame no.
    update();
    // clear the background of the component
    drawImage();
    // draw the image on the component
    ... // Delay depending on how far we are behind
}
```
B. Ensuring a Constant Frame Rate

- The most obvious way to implement the delay in the animation loop is to **sleep for delay milli-seconds**
- Can cause thread to **sleep too long** since losing time in executing the animation loop
- The better way is to **sleep until a particular time**

```java
long startTime = System.currentTimeMillis();
while (...) {
    ...try { startTime += delay;
    Thread.sleep(Math.max(0,startTime - System.currentTimeMillis()))
    } catch (InterruptedException e) { break; }
}
```

*Sleep until the time for displaying the next frame*
Exercise: From the attached program, build the applet that plays the animation. Your result should look like the following.
Eliminating Flashing

- It is seen that the previous animation has a problem of flashing.
- The reason is that the painting of the image is so complex that it takes longer to compute and draw than the video screen refreshes.

When the screen refreshes, the program can only draw to here.

Frame 4  Frame 5
A. Double Buffering

- To solve the problem, the approach of **double buffering** is suggested.
- It involves performing multiple graphics operations on an **undisplayed graphics buffer**.
- When the drawing of that buffer is finished, the result is put onto the screen.
B. Create the Buffer

- To create an off-screen buffer
  - create an off-screen image buffer of the proper size
  - get a graphics context to manipulate the image

```java
Dimension offDimension = null;
Image offImage = null;
Graphics offGraphics = null;
Applet ap = this.getApplet();
Dimension d = ap.getSize();
if ((offGraphics == null) ||
    (d.width != offDimension.width) ||
    (d.height != offDimension.height) ) {
    offDimension = d;
    offImage = canvasBody.createImage(d.width,d.height);
    offGraphics = offImage.getGraphics();
}
```
C. Update the Buffer

- We cannot use the `repaint` or `update` methods of the `VJCanvas` component to update the off-screen buffer.
- It is because they only update the drawing area of `VJCanvas`, not the buffer.
- A separated method should be added to
  - clear up the buffer before the painting of every frame
  - paint the frame
public void imageUpdate() {
    VJCanvas canvasBody=(VJCanvas)gui.canvasArea.getBody();
    Graphics g = canvasBody.getGraphics();
    Applet ap = this.getApplet();
    ... codes for creating the off-screen buffer ...
    offGraphics.setColor(canvasBody.getBackground());
    offGraphics.fillRect(0,0,d.width,d.height);
    try {
        offGraphics.drawImage(images[frameNumber%10],
            0, 0, canvasBody);
    } catch(ArrayIndexOutOfBoundsException e) {return;}
    g.drawImage(offImage,0,0,canvasBody);
}
Exercise: Modify your previous program to implement the double buffering technique. Use the code segments as listed in previous pages. Your result should look like the following
D. Loading the Images

• For some platforms, the execution of the previous program still suffers from the flashing problem at the very beginning

• **Reasons:** Images are not loaded if they are not needed
  - Loading images need much time
  - Not completely loaded images are still displayed

• To solve this problem:
  - Do not display the sequence if they are not completely loaded
  - The **MediaTracker** class provides methods to find out when the images are fully loaded
MediaTracker tracker;

protected void initGroup() {
    Applet ap = this.getApplet();
    tracker = new MediaTracker(ap);
    for (int i=1; i<=10; i++) {
        images[i-1] = ap.getImage(ap.getCodeBase(),
                                "T"+i+".gif");
        tracker.addImage(images[i-1], 0);
    }
}
public void run() {
    try {
        tracker.waitForAll();
    } catch (InterruptedException e) {} 
    : 
    ... previous run code ... 
    : 
}
public void imageUpdate() {
    VJCanvas canvasBody=(VJCanvas)gui.canvasArea.getBody();
    Graphics g = canvasBody.getGraphics();
    Applet ap = this.getApplet();
    Dimension d = ap.getSize();

    if (!tracker.checkAll()) {
        g.clearRect(0,0,d.width,d.height);
        g.drawString("Please wait...",0,d.height/2);
    } else {
        ...
        ...previous code...
        ...
    }
}
Exercise: Modify your previous program to implement the `MediaTracker` class. Use the code segments as listed in the previous pages. Your result should look like the following
Advanced Topics

4.3 Two Practical Considerations
A. GUI Interaction

- In practical situation, GUI will invoke other GUI to work in an application
- Java Workshop 2.0 allows each GUI to be developed individually and then executed altogether in an application
Example 1

- For GUIWin:
  - If the button “Next” is pressed, the GUI of GUIWin will be invisible and create the GUI of GUIWinNext
  - If the button “Exit” is pressed, the application quits

- For GUIWinNext:
  - If the list item “Back” is selected, the GUI of GUIWin will be shown again. The GUI of GUIWinNext will be destroyed
  - If the list item “Next” is selected, no action will be preformed (it is reserved for further development)
  - If the list item “Exit” is selected, the application quits
Step 1 - develop the GUIs

- Develop two projects (both are standalone applications) with names GUIWin and GUIWinNext. The GUI of both projects are as follows:
Step 2 - add operations to components

- For the project **GUIWin**,
  - Title the frame as “Main Page” and has the size “width=300;height=300”
  - For the button “Next”, add the following operation code with the Operation Editor
    ```java
    group.nextCallBack();
    ```
  - For the button “Exit”, add the following operation code with the Operation Editor
    ```java
    group.exitCallBack();
    ```
• For the project **GUIWinNext**,
  
  • Title the frame as "**Second Frame**" and has the size "**width=300;height=300**". Name the list component as "**itemList**" with text "**Back,Next,Exit**"
  
  • For "**itemList**", add the following operation code with the Operation Editor

```java
import java.awt.List;
List listBody = (List) gui.itemList.getBody();
String selectItem = listBody.getSelectedItem();
if (selectItem.equalsIgnoreCase("Back"))
    group.backCallback();
if (selectItem.equalsIgnoreCase("Next"))
    group.nextCallback();
if (selectItem.equalsIgnoreCase("Exit"))
    group.exitCallback();
```
Step 3 - add files to project GUIWin

• Add the following files from the project GUIWinNext to GUIWin:
  • GUIWinNext.java - describe the group of GUIWinNext
  • GUIWinNextRoot.java - describe the GUI designed in GUIWinNext
  • GUIWinNextOps.java - describe the operation associated with every component of GUIWinNext

• No need to add GUIWinNextMain.java since it describes the startup of the application (already given in GUIWinMain.java)
• Only one Main is required for an application
**Group (Container, components operations)**
Step 4 - implement callback functions

- To provide the connection between two GUIs, modify the following two functions of GUIWin.java as follows:

```java
public void nextCallBack() {
    Group nextGp = new GUIWinNext(this);
    nextGp.setEnvironmentInfo(null, null);
    nextGp.initialize();
    nextGp.create();
    nextGp.start();
    this.hide();
}
public void exitCallBack() {
    this.exit();
}
```

- When Next button is pressed, create the GUIWinNext group
- Pass GUIWin to GUIWinNext to show that it is the parent
- When GUIWinNext shows up, hide GUIWin
• The following functions of GUIWinNext.java should also be modified as follows:

```java
public void backCallback() {
    parentGp.show();
    this.destroy();
}

public void nextCallback() {
}

public void exitCallback() {
    this.exit();
}
```

When Back is selected in GUIWinNext, show the parent group (i.e. GUIWin) and destroy the GUIWinNext group.
The constructor of GUIWinNext group should also be modified as follows to keep a record of who is the parent:

```java
public class GUIWinNext extends Group {

    Group parentGp;
    private GUIWinNextRoot gui;

    public GUIWinNext(Group parent) {
        addForwardedAttributes();
        parentGp = parent;
    }
}
```

To record who is the parent group
B. Communication of GUIs: By File

- The simplest way to communicate between GUIs is by file
- However, may not work in Applet since some browsers do not allow Applets to use the file system of the destination machine
- Need to study the security issue of java applets
- For standalone java applications, file access is perfectly OK
Example 2

• The GUIs in Example 1 is modified such that two labels are added to show the number of accesses of the Next item in the list of GUIWinNext. A text file Try.txt serves as the bridge between the two GUIs.
• For **GUIWin:**
  • The text on the label is “Count = x”, where x is the number stored in Try.txt
  • At the beginning, the value of x is 0. Upon the modification by GUIWinNext, its value will be updated whenever GUIWin is shown

• For **GUIWinNext:**
  • The text on the label is “Count = x”, where x is the number stored in Try.txt
  • At the beginning, the value of x is 0. Whenever the Next item is selected, the value of x is increased by 1
Step 1 - modify the GUls

Modify the GUls in Example 1 as follows:

1. To modify the GUI in GUIWinNext, you need to go back to the project GUIWinNext and re-build it since you cannot see the GUI of GUIWinNext in GUIWin.

2. Since modifications have been made in GUIWinNext, errors may occur when you re-build the project GUIWinNext. You can ignore these errors and go back to GUIWin to debug the project.

Label name: countLabel

Label name: countLabel2
Step 2 - modify the constructor

- Modify the constructor of the group GUIWinNext as follows:

```java
int cnt;

public GUIWinNext(GroupName parent) {
    addForwardedAttributes();
    parentGp = parent;
    cnt = 0;
}
```

Add a member variable `cnt` for the group GUIWinNext

Whenever the group GUIWinNext is created, `cnt` is initialized to 0
Step 3 - modify the nextCallBack() of GUIWinNext

```java
public void nextCallBack() {
    Label showCount = (Label) gui.countLabel2.getBody();
    showCount.setText("Count = "+(++cnt));
    String cntString = "+cnt;
    try {
        File outputFile = new File("D:\jws\GUIWin2\Try.txt");
        FileWriter out = new FileWriter(outputFile);
        out.write(cntString, 0, cntString.length());
        out.close();
    } catch (IOException e) {}  
}
```

Write the no. of accesses to the Next item on the label

Open the file with full pathname

Write the value of count to file with 0 offset

Remember to close the file or the data will not be saved

Catch the IOException, necessary for file access
Step 4 - modify the initGroup() of GUIWin

• Objective: Whenever the group GUIWin is initialized, Try.txt is also reset

```java
protected void initGroup() {
    try {
        File outputFile = new File("D:\jws\GUIWin2\Try.txt");
        FileWriter out = new FileWriter(outputFile);
        out.write("0",0,1);
        out.close();
    } catch (IOException e) {} 
}
```

Note: “\” is a special character. If it has to be included in a string, use \\

Write a 0 to Try.txt to reset the count
Step 5 - modify the showGroup() of GUIWin

Objective: Whenever the group GUIWin is shown, read the content of Try.txt and write to the label countLabel

```java
protected void showGroup() {
    try {
        File inputFile = new File("D:\\jws\\GUIWin2\\Try.txt");
        FileReader in = new FileReader(inputFile);
        String cntString = "";
        int c;

        while ((c = in.read()) != -1)
            cntString = cntString + (char) c;
        in.close();
        Label cntLabelBody=(Label) gui.countLabel.getBody();
        cntLabelBody.setText("Count = "+cntString);
    } catch (IOException e) {} }```

Read the characters one-by-one from the file and concatenate to form a string
Step 6 - Run the application on DOS Prompt

- Visual GUI Builder requires some class libraries that are not part of Java JDK
- Need to be incorporated into the project
- Try to look for the following zip file:
  Drive:\Java-WorkShop20\Jws\lib\visualrt.zip
- Unzip it and extract to the directory where GUIWinMain is stored
- On DOS Prompt, add the path where JDK1.1.6\bin can be found
- Execute the following command:
  java GUIWinMain