

MỤC LỤC

CHƯƠNG I: Bắt đầu với Swing	6
1. Giới thiệu về JFC và Swing	6
2. Các gói Swing	7
3. Hệ thống đồ hoạ của Java.....	7
3.1. Giới thiệu chung.....	7
3.2. Một số phương thức của lớp Component	8
3.3. Lớp Container.....	8
3.4. Tạo ra Frame	9
3.5. JApplet	9
3.6. JLabel	10
3.7. JTextComponent	11
3.8. JButton	13
3.9. Checkbox – checkbox group – radio button – radio group	15
3.10. JList.....	17
3.11. JComboBox.....	20
CHƯƠNG II: Layout Management và Listener	23
1. Bố trí các thành phần bên trong các đối tượng chứa.....	23
2. Sử dụng Layout Managers	24
3. Tạo một Custom Layout Manager	34
4. Khai báo sự kiện tới Component (Listeners)	37
4.1. Tổng quan về khai báo sự kiện (Event Listeners)	37
4.2. Hỗ trợ Listeners của các thành phần Swing	40
4.3. Thực hiện Listeners cho các Handled Events thông thường.....	42
CHƯƠNG III: LẬP TRÌNH CƠ SỞ DỮ LIỆU	46
1. GIỚI THIỆU	46
2. JDBC Java Database Connectivity API	46
3. Kiến trúc của JDBC	47
3.1. Kiểu 1 (Cầu nối JDBC-ODBC).....	48
3.2. Kiểu 2 (điều khiển gốc).....	48
3.3. Kiểu 3 (trình điều khiển JDBC trên client)	49
3.4. Kiểu 4 (trình điều khiển kết nối trực tiếp tới CSDL)	49
4. Kết nối cơ sở dữ liệu	50
4.1. DriverManager	51

4.2. Connection	51
4.3. Statement.....	52
4.4. ResultSet	52
5. Lớp DatabaseMetaData.....	54
6. Lớp ResultSetMetaData	54
7. Các bước cơ bản để kết nối với cơ sở dữ liệu từ một ứng dụng Java.....	56
8. Các ví dụ về kết nối cơ sở dữ liệu từ ứng dụng Java.....	56
9. Sử dụng PreparedStatement	58
10. CallableStatement	60
11. Thực thi nhóm lệnh đồng thời (Batch)	62
12. Sử dụng Transaction.....	62
CHƯƠNG IV: Collections và cấu trúc dữ liệu trong Java.....	67
1. Giới thiệu.....	67
2. Cấu trúc dữ liệu trong Java	67
2.1. LinkedList	67
2.2. Stack.....	73
2.3. Queue	74
3. Collections Framework	75
3.1. Các thành phần của Java Collection.....	75
3.2. Các thao tác chính trên Collection	75
3.3. Set (tập hợp).....	75
3.4. List (danh sách)	77
3.5. Các lớp ArrayList, Vector	78
3.6. Map (ánh xạ)	80
3.7. Các lớp HashMap và Hashtable.....	81
3.8. SortedSet (tập được sắp) và SortedMap (ánh xạ được sắp)	82
3.9. TreeSet và TreeMap.....	83
CHƯƠNG V: LUỒNG I/O (I/O Streams).....	85
1. Giới thiệu.....	85
2. Luồng (Streams).....	85
2.1. Khái niệm luồng.....	85
2.2. Luồng byte (Byte Streams)	86
2.3. Luồng ký tự (Character Streams)	87
2.4. Những luồng được định nghĩa trước (The Predefined Streams)	88
3. Sử dụng luồng Byte.....	88

3.1. Lớp InputStream.....	88
3.2. Lớp OutputStream	89
4. Nhập và xuất mảng byte.....	89
5. Nhập và xuất tập tin	91
5.1. Lớp File	91
5.2. Lớp FileDescriptor	93
5.3. Lớp FileInputStream	94
5.4. Lớp FileOutputStream.....	94
6. Nhập xuất đã lọc.....	95
6.1. Lớp FilterInputStream.....	95
6.2. Lớp FilterOutputStream	95
7. I/O có lập vùng đệm.....	95
7.1. Lớp BufferedInputStream	96
8. Lớp Reader và Writer.....	98
8.1. Lớp Reader	98
8.2. Lớp Writer.....	98
9. Nhập/ xuất chuỗi và xâu ký tự	98
10. Lớp PrintWriter	100
11. Giao diện DataInput	101
12. Giao diện DataOutput.....	101
13. Lớp RandomAccessFile	102
CHƯƠNG VI: ĐA LUỒNG (MULTITHREADING).....	105
1. Giới thiệu.....	105
2. Đa luồng	105
3. Tạo và quản lý luồng.....	106
4. Vòng đời của Luồng.....	108
5. Phạm vi của luồng và các phương thức của lớp luồng	110
6. Thời gian biểu luồng	111
7. Luồng chạy nền	113
8. Đa luồng với Applets	113
9. Nhóm luồng.....	115
10. Sự đồng bộ luồng	116
11. Kỹ thuật “wait-notify” (đợi – thông báo).....	120
12. Sự bế tắc (Deadlocks)	123
13. Thu dọn “rác” (Garbage collection)	125

PHỤ LỤC I: LISTENER.....	130
--------------------------	-----

PHÂN BỐ CHƯƠNG TRÌNH

BUỔI			NỘI DUNG
STT	Lý thuyết	Thực hành	
1	Swing pI		
2		Lab 1	
3	Swing pII		
4		Lab 2	
5	JDBC pI		
6		Lab 3	
7	JDBC pII		
8		Lab 4	
9	I/O Streams		
10		Lab 5	
11	Collections		
12		Lab 6	
13	Thread		
14		Lab 7	
15	Thi kết thúc môn		

CHƯƠNG I: Bắt đầu với Swing

Mục tiêu:

Sau khi kết thúc chương này, bạn có thể:

- Có kiến thức cơ bản về Swing và JFC
- Hiểu rõ về quan hệ phân cấp Component trong Swing
- Hiểu rõ về các thành phần chính (Component) trong Swing như JFrame, JApplet, JPanel, ...

1. Giới thiệu về JFC và Swing

JFC (Java Foundation Classes) là một tập hợp các chức năng giúp người dùng xây dựng giao diện đồ họa (GUIs). JFC được công bố lần đầu tiên vào năm 1997 trong hội nghị về những nhà phát triển JavaOne, bao gồm những chức năng sau:

1. Các thành phần Swing
2. Hỗ trợ Pluggable Look and Feel
3. Truy cập API
4. Java 2D API (phiên bản Java 2 trở lên)
5. Hỗ trợ Drag and Drop (phiên bản Java 2 trở lên)

3 chức năng đầu của JFC được thực hiện mà không cần bảng mã quốc gia mà dựa vào các hàm API có trong JDK.

Một khả năng của Java là cho phép ta xây dựng các ứng dụng có giao diện đồ họa hay còn gọi là GUI (Graphical User Interface). Khi Java được phát hành, các thành phần đồ họa được tập trung vào thư viện mang tên Abstract Window Toolkit (AWT). Đối với mỗi hệ nền, thành phần AWT sẽ được ánh xạ sang một thành phần nền cụ thể, bằng cách sử dụng trực tiếp mã native của hệ nền, chính vì vậy nó phụ thuộc rất nhiều vào hệ nền và nó còn gây lỗi trên một số hệ nền. Với bản phát hành Java 2, các thành phần giao diện được thay bằng tập hợp các thành phần linh hoạt, đa năng, mạnh mẽ, độc lập với hệ nền thuộc thư viện Swing. Phần lớn các thành phần trong thư viện Swing đều được tô vẽ trực tiếp trên canvas bằng mã lệnh của Java, ngoại trừ các thành phần là lớp con của lớp `java.awt.Window` hoặc `Java.awt.Panel` vốn phải được vẽ bằng GUI trên nền cụ thể.

Thành phần Swing ít phụ thuộc vào hệ nền hơn do vậy ít gặp lỗi hơn và đặc biệt nó sử dụng ít tài nguyên của hệ thống hơn các thành phần trong thư viện AWT. Mặc dù các thành phần AWT vẫn được hỗ trợ trong Java 2 nhưng, tuy nhiên Sun khuyên bạn nên sử dụng các thành phần Swing thay cho các thành phần AWT, tuy nhiên các thành phần trong thư viện

Swing không thể thay tất cả các thành phần trong thư viện AWT. Chúng chỉ thay thế một phần của AWT như: Button, Panel, TextField, v.v.

Các lớp trợ giúp khác trong AWT như : Graphics, Color, Font, FontMetrics, v.v. vẫn không thay đổi. Bên cạnh đó các thành phần Swing còn sử dụng mô hình sự kiện của AWT.

2. Các gói Swing

Swing API vừa lớn mạnh, lại vừa mềm dẻo. Trong phiên bản 1.1 của API có 15 gói dùng chung: **javax.accessibility**, **javax.swing**, **javax.swing.border**, **javax.swing.colorchooser**, **javax.swing.event**, **javax.swing.filechooser**, **javax.swing.plaf**, **javax.swing.plaf.basic**, **javax.swing.plaf.metal**, **javax.swing.plaf.multi**, **javax.swing.table**, **javax.swing.text**, **javax.swing.text.html**, **javax.swing.tree**, and **javax.swing.undo**.

Trong hầu hết các chương trình chỉ sử dụng một phần nhỏ của API, chủ yếu là các gói Swing **javax.swing**, **javax.swing.event**.

3. Hệ thống đồ hoạ của Java

3.1. Giới thiệu chung

Thiết kế API cho lập trình đồ hoạ của Java là một ví dụ hoàn hảo về cách dùng lớp, sự kế thừa và giao diện. API cho lập trình đồ hoạ bao gồm một tập rất nhiều lớp nhằm trợ giúp xây dựng các thành phần giao diện khác nhau như: cửa sổ, nút ấn, ô văn bản, menu, hộp kiểm, v.v. Mỗi quan hệ kế thừa giữa các thành phần này được mô tả dưới đây:

- **Component** Đây là lớp (trừu tượng) cha của mọi lớp giao diện người dùng. Lớp này cung cấp các thuộc tính, hành vi cơ bản nhất của tất cả các thành phần giao diện.
- **Container** Là một vật chứa dùng để ghép nhóm các thành phần giao diện khác. Mỗi vật chứa có một lớp quản lý hiển thị, lớp quản lý hiển thị có trách nhiệm bố trí cách thức hiển thị các thành phần bên trong. Hai vật chứa hay được sử dụng nhất là JFrame và JPanel.
- **JComponent** Là lớp cha của mọi thành phần Swing lightweight, được vẽ trực tiếp lên canvas bằng mã lệnh Java.
- **Window** Được sử dụng để tạo ra một cửa sổ, Thông thường ta hay sử dụng hai lớp con của nó là JFrame và JDialog.
- **JFrame** là cửa sổ không lồng bên trong cửa sổ khác.
- **JDialog** là một cửa sổ được hiển thị dưới dạng modal.
- **JApplet** là lớp cha của mọi lớp ứng dụng applet.
- **JPanel** là một vật chứa, lưu giữ các thành phần giao diện người dùng.
- **Graphics** là lớp trừu tượng cung cấp ngữ cảnh đồ hoạ để vẽ các đối tượng đồ hoạ như: Đường thẳng, đường tròn, hình ảnh...

- **Color** lớp này biểu diễn một màu sắc.
- **Font** lớp này biểu thị cho một font đồ họa.
- **FontMetrics** là một lớp trừu tượng dùng để xác định các thuộc tính của Font.

Tất cả các thành phần đồ họa trong thư viện Swing được nhóm trong gói `javax.swing`.

3.2. Một số phương thức của lớp Component

Lớp Component cung cấp các thuộc tính, phương thức chung cho các lớp con của nó. Sau đây là một số phương thức của lớp Component :

- **Dimension getSize():** trả về đối tượng thuộc lớp Dimension gồm width (chiều rộng), height (chiều cao) xác định kích thước của một thành phần tính theo pixel.
- **void setSize(int width, int height)** và **void setSize(Dimension d)** thiết lập kích thước của thành phần.
- **Point getLocation():** trả về tọa độ (kiểu Point) trên cùng bên trái (tọa độ gốc) của thành phần đang xét.
- **void setLocation(int x, int y)** và **void setLocation(Point p)** thiết lập các tọa độ được chỉ định cho một thành phần.
- **Rectangle getBounds():** trả về đường biên là hình chữ nhật Rectangle bao gồm tọa độ gốc và chiều dài, chiều rộng của hình chữ nhật.
- **void setBounds(int x, int y)** và **void setBounds(Rectangle r)**: thiết lập đường biên cho một thành phần.
- **void setForeground(Color c)**: được sử dụng để đặt màu vẽ cho thành phần đồ họa
- **void setBackground(Color c)**: đặt màu nền cho thành phần đồ họa. Các tham số của hai hàm này là đối tượng của lớp Color sẽ được giới thiệu ở phần sau.
- **Font getFont():** được sử dụng để biết được font của các chữ đang xử lý trong thành phần đồ họa.
- **void setFont(Font f)**: thiết lập font chữ cho một thành phần.
- **void setEnabled(boolean b)**: Nếu đối số b của hàm getEnabled() là true thì thành phần đang xét hoạt động bình thường, nghĩa là có khả năng kích hoạt (enable), có thể trả lời các yêu cầu của người sử dụng và sinh ra các sự kiện như mong muốn. Ngược lại, nếu là false thì thành phần tương ứng sẽ không kích hoạt được, nghĩa là không thể trả lời được các yêu cầu của người sử dụng (Chú ý: Tất cả các thành phần giao diện khi khởi tạo đều được kích hoạt)
- **void setVisible(boolean b)**: Một thành phần đồ họa có thể được hiển thị lên màn hình (nhìn thấy được) hoặc bị che giấu tùy thuộc vào đối số của hàm setVisible() là true hay false.

3.3. Lớp Container

Lớp Container là lớp con của lớp trừu tượng Component. Các lớp chứa (lớp con của Container) cung cấp tất cả các chức năng để xây dựng các giao diện đồ họa ứng dụng, trong đó có phương thức `add()` được nạp chồng dùng để bổ sung một thành phần vào vật chứa và phương thức `remove()` cũng được nạp chồng để gỡ bỏ một thành phần ra khỏi vật chứa.

3.4. Tạo ra Frame

Lớp `JFrame` là lớp con của lớp `Frame` (`Frame` là lớp con của lớp `Window`) được sử dụng để tạo ra những cửa sổ cho các giao diện ứng dụng GUI.

Kịch bản chung để tạo ra một cửa sổ là:

- Tạo ra một frame có tiêu đề gì đó, ví dụ “My Frame” :

```
JFrame myWindow= new JFrame(“Design Global”);
```

- Xây dựng một cấu trúc phân cấp các thành phần bằng cách sử dụng hàm `myWindow.getContentPane().add()` để bổ sung thêm `JPanel` hoặc những thành phần giao diện khác vào `Frame`:

```
Ví dụ: myWindow.getContentPane().add(new JButton(“OK”));// Đưa vào  
một nút (JButton) có tên “OK” vào frame
```

- Thiết lập kích thước cho frame sử dụng hàm `setSize()`:

```
myWindow.setSize(200, 300);// Thiết lập khung frame là 200, 300
```

- Gói khung frame đó lại bằng hàm `pack()`:

```
myWindow.pack();
```

- Cho hiện frame:

```
myWindow.setVisible(true);
```

3.5. JApplet

`JApplet` kế thừa từ lớp `Java Applet`, đồng thời kế thừa 4 phương thức để quản lý vòng đời của mình. Bao gồm:

- **init()**, sự kiện xảy ra khi lần đầu applet load bởi trình duyệt
- **start()**, sự kiện xảy ra khi applet bắt đầu thực thi
- **stop()**, sự kiện xảy ra khi applet dừng thực thi
- **destroy()**, sự kiện xảy ra khi applet không cần tới

Thêm mới Component tới Applet, ta khai báo:

```
getContentPane( ).add(component);
```

Ví dụ :

```
import javax.swing.JApplet;  
import javax.swing.JButton;  
import javax.swing.JPanel;
```

```
import javax.swing.JTextField;
public class JAppletSample extends JApplet {
    JPanel panel;
    JButton button;
    JTextField text;
    @Override
    public void init() {
        panel = new JPanel();
        button = new JButton("OK");
        text = new JTextField("Design Global");
        panel.add(button);
        panel.add(text);
        this.setSize(250, 50);
        this.getContentPane().add(panel);
    }
}
```

Kết quả hiển thị như sau:



3.6. JLabel

JLabel biểu diễn cho đối tượng Label thông thường, dùng để chứa các tiêu đề, bên cạnh đó JLabel có thể chứa các ảnh. Tuy nhiên đối tượng JLabel không thể phản hồi các sự kiện nhập vào nên ta không thể khai báo các sự kiện trên đó.

Tạo đối tượng JLabel như sau hiển thị chuỗi ký tự căn giữa hàng:

```
JLabel label = new JLabel("Text on Label", SwingConstants.CENTER);
```

JLabel có thể căn dọc khi sử dụng các hàm sau **setVerticalAlignment()**. Hoặc có thể chỉ định khi khởi tạo như sau **SwingConstants.TOP**, **SwingConstants.BOTTOM** and **SwingConstants.CENTER**.

```
import java.awt.Color;
import java.awt.Container;
import javax.swing.BorderFactory;
import javax.swing.ImageIcon;
```

```
import javax.swing.JApplet;
import javax.swing.JLabel;
public class TJApplet extends JApplet {
    public void init() {
        this.setSize(500, 250);
        Container contentPane = getContentPane();
        // 2. Create a label with an icon and text.
        JLabel label = new JLabel("Design Global Label",
            new ImageIcon("Design_Global.jpg"),
            JLabel.CENTER);
        label.setBorder(BorderFactory.createMatteBorder(
            5, // top inset
            5, // left inset
            5, // bottom inset
            5, // right inset
            new Color(0x000000)));
        contentPane.add(label);
    }
}
```

Kết quả hiển thị như sau



3.7. JTextComponent

JTextComponent là thành phần cơ bản trong Swing, cung cấp tính năng nâng cao trong việc xử lý chuỗi văn bản. Các lớp con gồm có:

- JTextField: cho phép hiển thị, chỉnh sửa văn bản trên 1 dòng
- JPasswordField: cho phép hiển thị, chỉnh sửa văn bản trên 1 dòng, hiển thị kèm các ký tự * thay thế
- JTextArea: cho phép hiển thị, chỉnh sửa văn bản trên nhiều dòng
- JEditorPane: loại thành phần soạn thảo nâng cao, cho phép hiển thị, chỉnh sửa nhiều định dạng văn bản khác nhau
- JTextPane: thành phần cho phép đánh dấu

Tạo đối tượng JTextComponent như sau

```
JTextField label = new JTextField ("Text on Text field");  
JTextArea txt = new JTextArea(5,15);
```

Ví dụ:

```
import java.awt.Container;  
import java.awt.FlowLayout;  
import java.awt.event.ActionEvent;  
import java.awt.event.ActionListener;  
import javax.swing.JApplet;  
import javax.swing.JLabel;  
import javax.swing.JPasswordField;  
public class Password extends JApplet {  
    String correctpassword = "DesignGlobal";  
    JPasswordField jpasswordfield = new JPasswordField(10);  
    public void init() {  
        Container contentPane = getContentPane();  
        contentPane.setLayout(new FlowLayout());  
        contentPane.add(new JLabel("Design Global password: "));  
        contentPane.add(jpasswordfield);  
        jpasswordfield.setEchoChar('*');  
        jpasswordfield.addActionListener(new ActionListener() {  
            public void actionPerformed(ActionEvent e) {  
                String input = new String(jpasswordfield.getPassword());  
                if (correctpassword.equals(input)) {  
                    showStatus("Correct");  
                } else {  
                    showStatus("Incorrect");  
                }  
            }  
        })  
    }  
}
```

```
});  
}  
}
```

Kết quả hiển thị như sau



3.8. JButton

Lớp JButton cung cấp cho giao diện người dùng nút bấm quen thuộc. JButton kế thừa từ lớp AbstractButton. Khi nút được bấm, sự kiện gắn liền với nút sẽ được kích hoạt như hiển thị cửa sổ thông báo (Jdialog) hiển thị lựa chọn chức năng “Yes”, “No”, “Okay”, “Cancel”.

Khai báo khởi tạo JButton như sau:

```
JButton() –tạo nút bấm không thiết lập tên hiển thị  
JButton(Icon icon) – tạo nút bấm với Icon thiết lập sẵn  
JButton(String text) –tạo nút bấm với tên hiển thị thiết lập sẵn  
JButton(String text, Icon icon) - tạo nút bấm với tên hiển thị và Icon thiết lập sẵn
```

Ví dụ:

```
import java.awt.Color;  
import java.awt.event.ActionEvent;  
import java.awt.event.ActionListener;  
import javax.swing.JApplet;  
import javax.swing.JButton;  
import javax.swing.JPanel;  
import javax.swing.JTextField;  
public class JButtonExample extends JApplet implements ActionListener {
```

```
JButton btn, btn1;
JTextField txt;
JPanel jp;
public void init() {
    btn = new JButton("Design");
    btn1 = new JButton("Global");
    txt = new JTextField("Button click", 20);
    jp = new JPanel();
    jp.add(btn);
    jp.add(btn1);
    jp.add(txt);
    getContentPane().add(jp);
    btn.addActionListener(this);
    btn1.addActionListener(this);
}
public void actionPerformed(ActionEvent a) {
    if (a.getSource() == btn) {
        txt.setBackground(Color.yellow);
        txt.setText("Button named  DESIGN  was clicked");
    }
    if (a.getSource() == btn1) {
        txt.setBackground(Color.orange);
        txt.setText("Button named  GLOBAL  was clicked");
    }
}
}
```

Kết quả hiển thị như sau:



3.9. Checkbox – checkbox group – radio button – radio group

Checkbox dùng để chuyển đổi trạng thái (state) giữa yes/no hay true/false. Khi state là true thì ô đã được đánh dấu. Có 3 instructor thường dùng là: `Checkbox()` `Checkbox(String label)` `Checkbox(String label,boolean state)` với label hiển thị nhãn còn state là true/false

Khai báo khởi tạo đối tượng Checkbox như sau

`JCheckBox()` –tạo checkbox chưa được chọn, không có tiêu đề và Icon

`JCheckBox(Icon icon)` - tạo checkbox chưa được chọn, không có tiêu đề và thiết lập sẵn Icon

`JCheckBox(Icon icon, boolean selected)` - tạo checkbox đã được chọn, không có tiêu đề và thiết lập sẵn Icon

`JCheckBox(String text)` - tạo checkbox chưa được chọn, có tiêu đề `JCheckBox(String text, boolean selected)` - tạo checkbox đã được chọn, có tiêu đề

`JCheckBox(String text, Icon icon)` - tạo checkbox chưa được chọn, có tiêu đề và thiết lập sẵn Icon

`JCheckBox(String text, Icon icon, boolean selected)` - tạo checkbox đã được chọn, có tiêu đề và thiết lập sẵn Icon

Ví dụ:

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;
import javax.swing.ButtonGroup;
import javax.swing.JApplet;
import javax.swing.JCheckBox;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JRadioButton;
```

```
import javax.swing.JTextField;

public class Checkbox extends JApplet implements ActionListener, ItemListener {
    JLabel l;
    JTextField t;
    JPanel jp;
    JCheckBox c1, c2, c3, c4;
    JRadioButton r1, r2;
    ButtonGroup grp;
    public void init() {
        this.setSize(550, 75);
        l = new JLabel("Please tick the games you play");
        t = new JTextField("-----", 20);
        t.setEditable(false);
        c1 = new JCheckBox("Are you a teenager ?");
        c2 = new JCheckBox("Cricket");
        c3 = new JCheckBox("Hockey");
        c4 = new JCheckBox("Baseball");
        r1 = new JRadioButton("Male");
        r2 = new JRadioButton("Female");
        grp = new ButtonGroup();
        grp.add(r1);
        grp.add(r2);
        jp = new JPanel();
        jp.add(t);
        jp.add(c1);
        jp.add(r1);
        jp.add(r2);
        jp.add(l);
        jp.add(c2);
        jp.add(c3);
        jp.add(c4);
        getContentPane().add(jp);
        r1.addActionListener(this);
        r2.addActionListener(this);
        c1.addItemListener(this);
    }
    public void actionPerformed(ActionEvent a) {
```

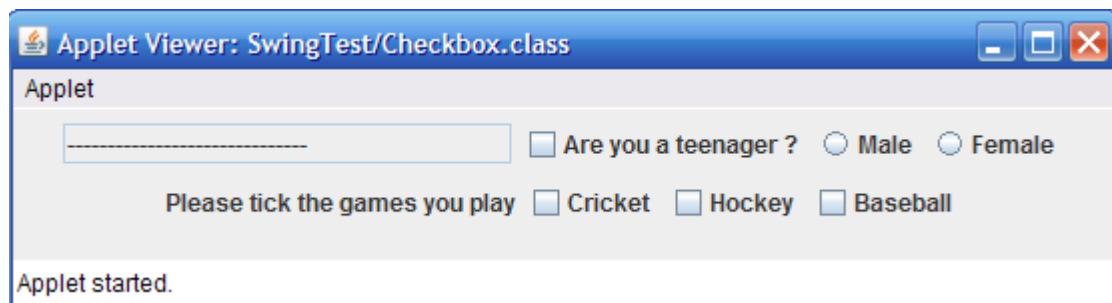


```

        if (a.getSource() == r1) {
            t.setText("Male");
        }
        if (a.getSource() == r2) {
            t.setText("Female");
        }
    }
    public void itemStateChanged(ItemEvent e) {
        if (e.getSource() == c1) {
            if (e.getStateChange() == ItemEvent.SELECTED) {
                t.setText("You are less than 20 years");
            } else if (e.getStateChange() == ItemEvent.DESELECTED) {
                t.setText("You may be more than 20 years");
            }
        }
    }
}

```

Kết quả hiển thị như sau:



3.10. JList

JList được mở rộng từ lớp List của AWT. JList chứa tập hợp các phần tử sắp xếp liên tiếp, có thể lựa chọn từng phần tử hoặc nhiều phần tử cùng lúc. JList ngoài việc hiển thị danh sách dạng chuỗi văn bản, còn có thể hiển thị dạng Icon.

Khai báo khởi tạo JList như sau:

```

public JList( ) - khởi tạo danh sách rỗng
public JList(ListModel dataModel) – khởi tạo danh sách xác định trước
public JList(Object [] listData) – khởi tạo danh sách từ giá trị mảng

```

JList không hỗ trợ thanh cuộn, nên cần phải thực thi mở rộng từ đối tượng JScrollPane.

```

JScrollPane myScrollPane = new JScrollPane();

```

```
myScrollPane.getViewport().setView(dataList);
```

Ví dụ

```
import javax.swing.JList;
import javax.swing.JPanel;
import javax.swing.JFrame;
import javax.swing.DefaultListModel;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
public class PlainList {
    public static void main(String args[]) {
        JFrame frame = new JFrame("Design Global");
        frame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
        // Use a list model that allows for updates
        DefaultListModel model = new DefaultListModel();
        JList statusList = new JList(model);
        statusList.setSize(120, 60);
        // Create some dummy list items
        model.addElement("Thoi Trang");
        model.addElement("Do hoa");
        model.addElement("Phan mem");
        // Display the list
        JPanel panel = new JPanel();
        panel.add(statusList);
        frame.getContentPane().add("Center", panel);
        frame.pack();
        frame.setVisible(true);
    }
}
```

Kết quả hiển thị như sau:



Ví dụ khác

```
import java.awt.Container;
import java.awt.FlowLayout;
import javax.swing.JApplet;
import javax.swing.JList;
import javax.swing.JScrollPane;
import javax.swing.ListSelectionModel;
import javax.swing.event.ListSelectionEvent;
import javax.swing.event.ListSelectionListener;

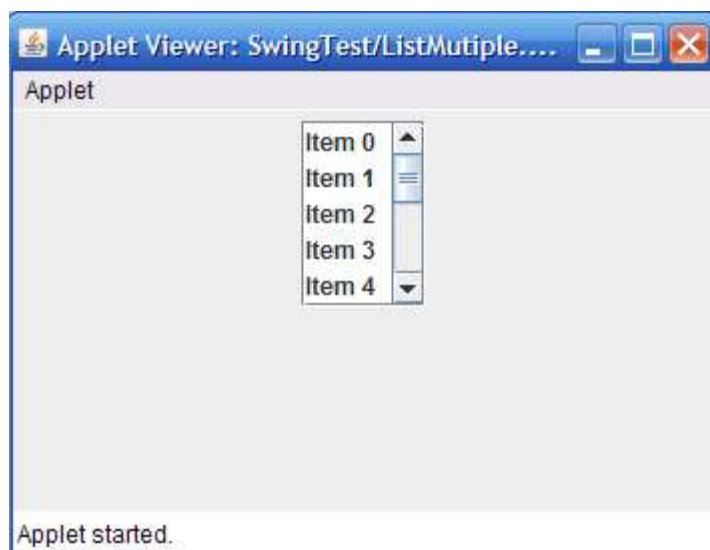
public class ListMutiple extends JApplet implements ListSelectionListener {
    JList jlist;

    public void init() {
        Container contentPane = getContentPane();
        String[] items = new String[12];
        for (int loop_index = 0; loop_index <= 11; loop_index++) {
            items[loop_index] = "Item " + loop_index;
        }
        jlist = new JList(items);
        JScrollPane jscrollpane = new JScrollPane(jlist);
        jlist.setVisibleRowCount(5);
        jlist.setSelectionModel(ListSelectionModel.MULTIPLE_INTERVAL_SELECTION);
        jlist.addListSelectionListener(this);
        contentPane.setLayout(new FlowLayout());
        contentPane.add(jscrollpane);
    }

    public void valueChanged(ListSelectionEvent e) {
        int[] indexes = jlist.getSelectedIndices();
        String outString = "You chose:";
        for (int loop_index = 0; loop_index < indexes.length; loop_index++) {
            outString += " item " + indexes[loop_index];
        }
        showStatus(outString);
    }
}
```

```
}  
}
```

Kết quả hiển thị như sau



3.11. JComboBox

ComboBox là sự kết hợp giữa text field và drop-down list để cho phép người dùng nhập vào giá trị hoặc chọn từ danh sách cho trước.

JComboBox trong Swing đã ẩn đi tính năng cho phép chỉnh sửa nội dung, và chỉ còn lại tính năng lựa chọn từ danh sách cho trước.

Khai báo khởi tạo JComboBox như sau

JComboBox () –tạo thành phần combobox mặc định không chứa phần tử nào

JComboBox (ComboBoxModel asModel) – tạo thành phần combobox với danh sách phần tử được thiết lập trước

public JComboBox (Object [] items) - tạo thành phần combobox với danh sách phần tử được nhận từ mảng tham số đầu vào

Ví dụ:

```
import java.awt.Container;  
import java.awt.FlowLayout;  
import javax.swing.JApplet;  
import javax.swing.JComboBox;  
public class ComboBox extends JApplet {  
    private JComboBox jcombobox = new JComboBox();  
    public void init() {  
        Container contentPane = getContentPane();  
        jcombobox.addItem("Thoi trang");
```

```
jcombobox.addItem("Do hoa");
jcombobox.addItem("Lap trinh");
contentPane.setLayout(new FlowLayout());
contentPane.add(jcombobox);
}
}
```

Kết quả hiển thị như sau



Tóm tắt bài học

JFC (Java Foundation Classes) là một tập hợp các chức năng giúp người dùng xây dựng giao diện đồ họa (GUIs).

Lớp JComponent là lớp cha của mọi thành phần Swing lightweight, được vẽ trực tiếp lên canvas bằng mã lệnh Java.

JFrame là cửa sổ không lồng bên trong cửa sổ khác.

JDialog là một cửa sổ được hiển thị dưới dạng modal.

JApplet là lớp cha của mọi lớp ứng dụng applet.

JPanel là một vật chứa, lưu giữ các thành phần giao diện người dùng.

JTextComponent là các thành phần cho phép người dùng thao tác với chuỗi văn bản thông thường và các định dạng nâng cao.

JButton cung cấp cho giao diện người dùng nút bấm quen thuộc và đón bắt sự kiện nhấn nút của người dùng.

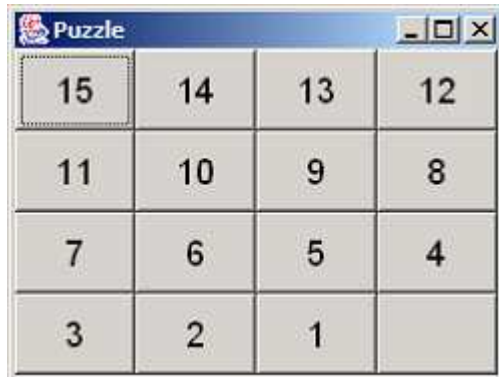
JCheckBox, JRadioButton, CheckboxGroup, RadioGroup cung cấp cho giao diện người dùng thêm các lựa chọn.

JList cung cấp cho giao diện người dùng chức năng hiển thị văn bản, hình ảnh dạng phân nhóm, liệt kê.

JCombobox là một thành phần thừa kế tính năng của một drop-down list.

Bài tập

Viết chương trình trong Java bằng Swing, xây dựng trò chơi Puzzle như sau:



15	14	13	12
11	10	9	8
7	6	5	4
3	2	1	

Gợi ý xây dựng trò chơi:

- Người chơi sử dụng chuột, nhấp vào 1 ô trống và 1 ô chứa số thì hai ô này sẽ đổi chỗ cho nhau.
- Người chơi chiến thắng khi các ô được sắp giảm từ 15 về 1.
- Chương trình cho phép random vị trí các số bất kỳ.

CHƯƠNG II: Layout Management và Listener

Mục tiêu:

Sau khi kết thúc chương này, bạn có thể:

- Hiểu về các chế độ bố cục trong Swing
- Tự thiết kế giao diện ứng dụng theo yêu cầu bài toán
- Học cách khai báo sự kiện trên các component của Swing
- Hiểu về các sự kiện Low-level và Semantic Listener

1. Bố trí các thành phần bên trong các đối tượng chứa

Các đối tượng chứa sử dụng **layout managers** để xác lập kích thước và vị trí của các thành phần chứa trong nó. Borders sẽ ảnh hưởng đến layout của Swing GUIs bằng cách làm cho các thành phần lớn lên.

Layout management là quá trình xác định kích thước và vị trí của các thành phần. Mặc định, mỗi đối tượng chứa sẽ có một **layout manager**.

Java platform hỗ trợ sử dụng 5 layout managers thông thường nhất: **BorderLayout**, **BoxLayout**, **FlowLayout**, **GridBagLayout**, và **GridLayout**. Những layout managers được thiết kế để hiển thị đa thành phần trong cùng một thời điểm. Và lớp thứ 6, **CardLayout**, là một trường hợp đặc biệt. Nó được sử dụng để kết hợp các layout managers với nhau.

Bảng sau cung cấp bốn lớp quản lý layout (cách bố trí và sắp xếp) các thành phần GUI.

Tên lớp	Mô tả
FlowLayout	Xếp các thành phần giao diện trước tiên theo hàng từ trái qua phải, sau đó theo cột từ trên xuống dưới. Cách sắp xếp này là mặc định đối với Panel, JPanel, Applet và JApplet.
GridLayout	Các thành phần giao diện được sắp xếp trong các ô lưới hình chữ nhật lần lượt theo hàng từ trái qua phải và theo cột từ trên xuống dưới trong một phần tử chứa. Mỗi thành phần giao diện chứa trong một ô.
BorderLayout	Các thành phần giao diện (ít hơn 5) được đặt vào các vị trí theo các hướng: north (bắc), south (nam), west (tây), east (đông) và center (trung tâm). Cách sắp xếp này là mặc định đối với lớp

	Window, Frame, JFrame, Dialog và JDialog.
GridBagLayout	Cho phép đặt các thành phần giao diện vào lưới hình chữ nhật, nhưng một thành phần có thể chiếm nhiều nhiều hơn một ô.
null	Các thành phần bên trong vật chứa không được sắp lại khi kích thước của vật chứa thay đổi.
CardLayout	Nhiều card có thể khai báo chồng lên nhau và mỗi card xuất hiện tại 1 thời điểm

Các phương pháp thiết đặt layout

Để lấy về layout hay để thiết lập layout cho vật chứa, chúng ta có thể sử dụng hai phương thức của lớp Container:

```
LayoutManager getLayout();
void setLayout(LayoutManager mgr);
```

Các thành phần giao diện sau khi đã được tạo ra thì phải được đưa vào một phần tử chứa nào đó. Hàm add() của lớp Container được nạp chồng để thực hiện nhiệm vụ đưa các thành phần vào phần tử chứa.

```
Component add(Component comp)
Component add(Component comp, int index)
Component add(Component comp, Object constraints)
Component add(Component comp, Object constraints, int index)
```

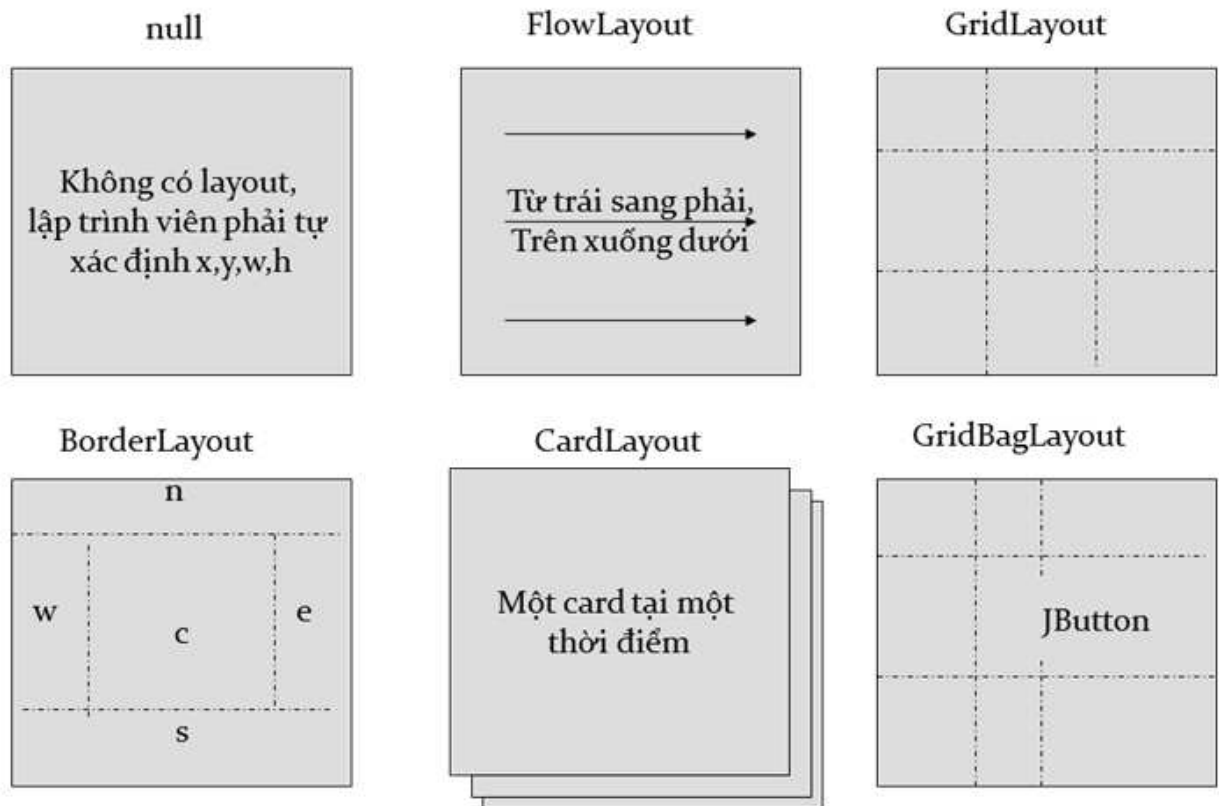
Trong đó, đối số index được sử dụng để chỉ ra vị trí của ô cần đặt thành phần giao diện comp vào. Đối số constraints xác định các hướng để đưa comp vào phần tử chứa.

Ngược lại, khi cần loại ra khỏi phần tử chứa một thành phần giao diện thì sử dụng các hàm sau:

```
void remove(int index)
void remove(Component comp)
void removeAll()
```

2. Sử dụng Layout Managers

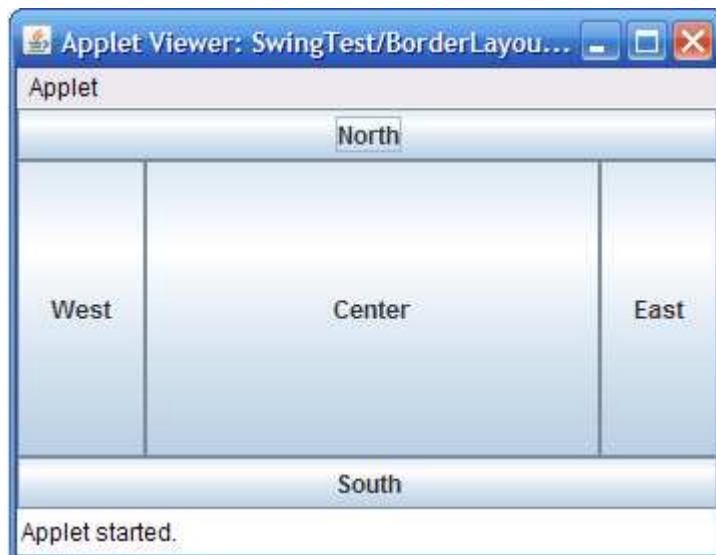
Phần này cung cấp các qui tắc tổng quan và chi tiết lệnh trong việc sử dụng việc quản lý bố trí mà Java platform cung cấp. Tham khảo: <http://www.java2s.com/Code/Java/Swing-JFC/CardLayoutDemo.htm>



a) Sử dụng BorderLayout

Sau đây là một Applet cho thấy BorderLayout làm việc như thế nào .

```
setLayout(new BorderLayout());
setFont(new Font("Helvetica", Font.PLAIN, 14));
add("North", new Button("North"));
add("South", new Button("South"));
add("East", new Button("East"));
add("West", new Button("West"));
add("Center", new Button("Center"));
```



Quan trọng: khi thêm một thành phần vào một Container sử dụng BorderLayout, bạn nên dùng phương thức **add()** hai thông số, và thông số thứ nhất phải là **"North"**, **"South"**, **"East"**, **"West"**, hoặc **"Center"**. Nếu bạn sử dụng phương thức **add()** một thông số hay bạn không xác lập thông số thứ nhất thì thành phần đó sẽ không hiển thị.

Theo mặc định, BorderLayout không đặt khoảng trống giữa các thành phần. Muốn vậy, bạn phải xác lập nó bằng cách dùng cấu trúc sau:

```
public BorderLayout(int horizontalGap, int verticalGap)
```

b) Sử dụng CardLayout

Sau đây là một ví dụ cho thấy CardLayout làm việc như thế nào.

```
import java.awt.BorderLayout;
import java.awt.CardLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
public class CardLayouts extends JFrame {
    private int currentCard = 1;
    private JPanel cardPanel;
    private CardLayout cl;
    public CardLayouts() {
        setTitle("Design Global Example");
        setSize(300, 150);
```

```

cardPanel = new JPanel();
cl = new CardLayout();
cardPanel.setLayout(cl);
JPanel p1 = new JPanel();
JPanel p2 = new JPanel();
JPanel p3 = new JPanel();
JLabel lab1 = new JLabel("Thoi trang");
JLabel lab2 = new JLabel("Do hoa");
JLabel lab3 = new JLabel("Lap trinh");
p1.add(lab1);
p2.add(lab2);
p3.add(lab3);
cardPanel.add(p1, "1");
cardPanel.add(p2, "2");
cardPanel.add(p3, "3");
JPanel buttonPanel = new JPanel();
JButton b1 = new JButton("Previous");
JButton b2 = new JButton("Next");
buttonPanel.add(b1);
buttonPanel.add(b2);
b1.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent arg0) {
        if (currentCard > 1) {
            currentCard -= 1;
            cl.show(cardPanel, "" + (currentCard));
        }
    }
});
b2.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent arg0) {
        if (currentCard < 3) {
            currentCard += 1;
            cl.show(cardPanel, "" + (currentCard));
        }
    }
});
getContentPane().add(cardPanel, BorderLayout.NORTH);

```

```

        getContentPane().add(buttonPanel, BorderLayout.SOUTH);
    }
    public static void main(String[] args) {
        CardLayouts cl = new CardLayouts();
        cl.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        cl.setVisible(true);
    }
}

```

Khi bạn thêm một thành phần vào một Container mà có sử dụng CardLayout , bạn phải sử dụng phương thức **add()** hai thông số: **add(String name, Component comp)**. Thông số thứ nhất có thể bất kì chuỗi nào để nhận ra thành phần được thêm vào.



Như đoạn mã trên , bạn có thể sử dụng phương thức **show()** của CardLayout để xác lập thành phần hiển thị hiện tại . Thông số thứ nhất của phương thức **show()** là Container mà CardLayout điều khiển. thông số thứ hai là chuỗi để xác định thành phần hiển thị . Chuỗi này giống như chuỗi của thành phần thêm vào Container.

Theo sau là tất cả các phương thức của CardLayout mà có thể cho phép chọn một thành phần . cho mỗi phương thức , thông số thứ nhất Container cho CardLayout là một Layout Manager.

```

public void first(Container parent)
public void next(Container parent)
public void previous(Container parent)
public void last(Container parent)
public void show(Container parent, String name)

```

c) Sử dụng FlowLayout

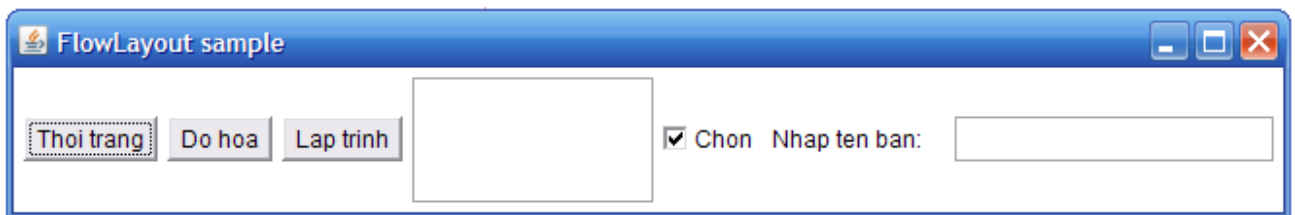
Sau đây là một Applet cho thấy FlowLayout hoạt động như thế nào.

```

import java.awt.Button;
import java.awt.Checkbox;
import java.awt.FlowLayout;
import java.awt.Frame;

```

```
import java.awt.Label;
import java.awt.List;
import java.awt.TextField;
public class FlowLayouts {
    public static void main(String[] args) {
        Frame f = new Frame("FlowLayout sample");
        f.setLayout(new FlowLayout());
        f.add(new Button("Thoi trang"));
        f.add(new Button("Do hoa"));
        f.add(new Button("Lap trinh"));
        List list = new List();
        for (int i = 0; i < args.length; i++) {
            list.add(args[i]);
        }
        f.add(list);
        f.add(new Checkbox("Chon", true));
        f.add(new Label("Nhap ten ban:"));
        f.add(new TextField(20));
        f.pack();
        f.setVisible(true);
    }
}
```



Lớp FlowLayout có ba cấu trúc:

```
public FlowLayout()
public FlowLayout(int alignment)
public FlowLayout(int alignment, int horizontalGap, int verticalGap)
```

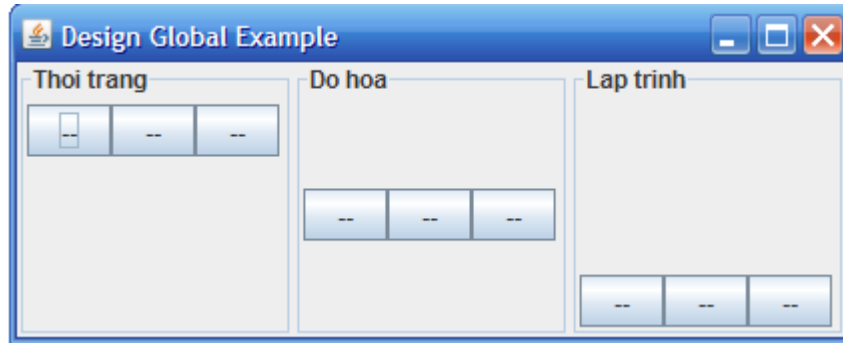
Thông số **alignment** phải là các giá trị **FlowLayout.LEFT**, **FlowLayout.CENTER**, hoặc **FlowLayout.RIGHT**. Thông số **horizontalGap** và **verticalGap** xác định số Pixel đặc giữa các thành phần. Nếu bạn không xác lập giá trị này, FlowLayout sẽ mặc định giá trị 5 cho mỗi thông số.

d) Sử dụng GridLayout

Sau đây là một ví dụ cho thấy GridLayout làm việc như thế nào.

```
import java.awt.Component;
import java.awt.Container;
import java.awt.GridLayout;
import javax.swing.BorderFactory;
import javax.swing.BoxLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
public class GridLayouts {
    private static Container makeIt(String title, float alignment) {
        String labels[] = { "--", "--", "--" };
        JPanel container = new JPanel();
        container.setBorder(BorderFactory.createTitledBorder(title));
        BoxLayout layout = new BoxLayout(container, BoxLayout.X_AXIS);
        container.setLayout(layout);
        for (int i = 0, n = labels.length; i < n; i++) {
            JButton button = new JButton(labels[i]);
            button.setAlignmentY(alignment);
            container.add(button);
        }
        return container;
    }
    public static void main(String args[]) {
        JFrame frame = new JFrame("Design Global Example");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container panel1 = makeIt("Thời trang", Component.TOP_ALIGNMENT);
        Container panel2 = makeIt("Đồ họa", Component.CENTER_ALIGNMENT);
        Container panel3 = makeIt("Lập trình", Component.BOTTOM_ALIGNMENT);
        Container contentPane = frame.getContentPane();
        contentPane.setLayout(new GridLayout(1, 3));
        contentPane.add(panel1);
        contentPane.add(panel2);
        contentPane.add(panel3);
        frame.setSize(423, 171);
        frame.setVisible(true);
    }
}
```

}



Cấu trúc trên cho thấy lớp GridLayout tạo một đối tượng có hai cột và nhiều hàng. Đây là một trong hai cấu trúc cho GridLayout. Sau đây là cách khai báo cho cả hai cấu trúc này:

```
public GridLayout(int rows, int columns)
public GridLayout(int rows, int columns, int horizontalGap, int verticalGap)
```

e) Sử dụng GridBagLayout

Theo sau là một vài đoạn lệnh tiêu biểu trong một Container có sử dụng GridBagLayout.

```
GridBagLayout gridbag = new GridBagLayout();
GridBagConstraints c = new GridBagConstraints();
setLayout(gridbag);
//For each component to be added to this container:
//...Create the component...
//...Set instance variables in the GridBagConstraints instance...
gridbag.setConstraints(theComponent, c);
add(theComponent);
```

Bạn có thể sử dụng lại một đối tượng của GridBagConstraints cho nhiều thành phần khác nhau, ngay cả khi các thành phần đó có sự ràng buộc khác nhau. GridBagLayout rút ra một giá trị ràng buộc và không dùng lại GridBagConstraints. Bạn phải cẩn thận, tuy nhiên, để khởi tạo lại giá trị của một đối tượng GridBagConstraints làm giá trị mặc định khi cần thiết.

Bạn có thể xác lập các giá trị sau:

gridx, gridy

Xác định hàng và cột tại vị trí trên bên trái của thành phần. Hầu hết cột trên bên trái có tọa độ **gridx=0**, và hàng trên cùng có tọa độ **gridy=0**. Sử dụng **GridBagConstraints.RELATIVE** (giá trị mặc định) để xác định rằng thành phần đó chỉ ở bên phải hay ở phía dưới.

gridwidth, gridheight

Xác lập số cột hoặc số hàng trong vùng hiển thị của thành phần. những giá trị này xác định số Cell mà thành phần sử dụng, không phải số Pixel nó sử dụng. Mặc định là 1. Sử dụng **GridBagConstraints.REMAINDER** để xác định thành phần đang ở hàng cuối cùng hay cột cuối cùng. Sử dụng **GridBagConstraints.RELATIVE** để xác định bước kế tiếp của thành phần là hàng cuối hay cột cuối cùng.

fill

Được sử dụng khi vùng hiển thị của thành phần lớn hơn kích thước thành phần đòi hỏi để quyết định khi nào hoặc thay đổi kích thước như thế nào. các giá trị thích hợp là **GridBagConstraints.NONE** (mặc định), **GridBagConstraints.HORIZONTAL**, **GridBagConstraints.VERTICAL** và **GridBagConstraints.BOTH**.

ipadx, ipady

Xác định phần phụ ở bên trong: bao nhiêu để thêm vào kích thước tối thiểu của thành phần. giá trị mặc định là 0. Chiều rộng của thành phần tối thiểu nhất là bằng chiều rộng tối thiểu của nó cộng với **ipadx*2**. Similarly, chiều cao của thành phần tối thiểu nhất là bằng chiều cao tối thiểu của nó cộng với **ipady*2**.

insets

Xác định phần phụ bên ngoài của thành phần. mặc định, mỗi thành phần không có phần phụ bên ngoài.

anchor

Sử dụng khi thành phần nhỏ hơn vùng hiển thị để quyết định khi nào đặt thành phần. giá trị thích hợp là **GridBagConstraints.CENTER** (mặc định), **GridBagConstraints.NORTH**, **GridBagConstraints.NORTHEAST**, **GridBagConstraints.EAST**, **GridBagConstraints.SOUTHEAST**, **GridBagConstraints.SOUTH**, **GridBagConstraints.SOUTHWEST**, **GridBagConstraints.WEST**, và **GridBagConstraints.NORTHWEST**.

Ví dụ :

Sau đây là một đoạn lệnh tạo một GridBagLayout và các thành phần nó quản lý

```
import java.awt.*;
import javax.swing.JButton;
import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JTextField;
public class GridBagLayouts {
```



```

public static void addComponentsToPane(Container pane) {
    JButton jbnButton;
    pane.setLayout(new GridBagLayout());
    GridBagConstraints gBC = new GridBagConstraints();
    gBC.fill = GridBagConstraints.HORIZONTAL;
    jbnButton = new JButton("Design");
    gBC.weightx = 0.5;
    gBC.gridx = 0;
    gBC.gridy = 0;
    pane.add(jbnButton, gBC);
    JTextField jtf = new JTextField("Global");
    gBC.gridx = 2;
    gBC.gridy = 0;
    jtf.setEditable(false);
    pane.add(jtf, gBC);
    jbnButton = new JButton("Thoi trang");
    gBC.gridx = 2;
    gBC.gridy = 0;
    pane.add(jbnButton, gBC);
    jbnButton = new JButton("Do hoa");
    gBC.ipady = 40;    //This component has more breadth compared to other buttons
    gBC.weightx = 0.0;
    gBC.gridwidth = 3;
    gBC.gridx = 0;
    gBC.gridy = 1;
    pane.add(jbnButton, gBC);
    JComboBox jcmbSample = new JComboBox(new String[]{"Lap trinh C#", "Lap trinh Java",
    "Lap trinh CSDL"});
    gBC.ipady = 0;
    gBC.weighty = 1.0;
    gBC.anchor = GridBagConstraints.PAGE_END;
    gBC.insets = new Insets(10, 0, 0, 0); //Padding
    gBC.gridx = 1;
    gBC.gridwidth = 2;
    gBC.gridy = 2;
    pane.add(jcmbSample, gBC);
}

```

```

}

private static void createAndShowGUI() {
    JFrame.setDefaultLookAndFeelDecorated(true);
    JFrame frame = new JFrame("Design Global Source Demo");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    //Set up the content pane.
    addComponentsToPane(frame.getContentPane());
    frame.pack();
    frame.setVisible(true);
}

public static void main(String[] args) {
    javax.swing.SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            createAndShowGUI();
        }
    });
}
}

```



3. Tạo một Custom Layout Manager

Thay vì sử dụng cách quản lý mà Java platform cung cấp, ta có thể viết một chương trình quản lý của chính mình. Quản lý bố trí phải thực thi **LayoutManager** interface, nơi chỉ định năm phương thức phải được định nghĩa. Việc quản lý cách bố trí có thể thực thi **LayoutManager2**, là một giao diện con của **LayoutManager**.

5 phương thức được thực thi là:

void addLayoutComponent(String, Component)

void removeLayoutComponent(Component)

Dimension preferredLayoutSize(Container)

Dimension minimumLayoutSize(Container)

void layoutContainer(Container)

```
public void addLayoutComponent(String name, Component comp)
```

Chỉ được gọi bằng phương thức `add(name, component)` của `Container`.

```
public void removeLayoutComponent(Component comp)
```

Gọi bởi những phương thức `remove()` và `removeAll()` của `Container`.

```
public Dimension preferredLayoutSize(Container parent)
```

Gọi bởi phương thức `preferredSize()` của `Container`, có thể tự gọi dưới mọi tình huống.

```
public Dimension minimumLayoutSize(Container parent)
```

Gọi bởi phương thức `minimumSize()` của `Container`, có thể tự gọi dưới mọi tình huống.

```
public void layoutContainer(Container parent)
```

Gọi khi `Container` hiển thị lần đầu, và mọi lúc nó thay đổi kích thước.

a) Làm việc không có Layout Manager(Absolute Positioning)

Mặc dù có thể làm việc mà không cần `Layout Manager`, bạn nên dùng `Layout Manager` nếu có thể. `Layout managers` dễ thay đổi kích thước của `Container` và điều chỉnh hình dạng của các thành phần phụ thuộc vào `Platform`. Nó cũng có thể được sử dụng lại bởi các `Container` và các chương trình khác. nếu `Custom Container` sẽ không tái sử dụng, không thể thay đổi kích thước, và hoàn toàn có thể điều khiển được các thông số phụ thuộc vào hệ thống như `Font` và hình dạng các thành phần.

Ví dụ:

```
import java.awt.Button;
import java.awt.Event;
import java.awt.Font;
import java.awt.Frame;
import java.awt.Graphics;
import java.awt.Insets;
public class NoneWindow extends Frame {
    private boolean inAnApplet = true;
    private boolean laidOut = false;
    private Button b1, b2, b3;
    public NoneWindow() {
```

```

super();
setLayout(null);
setFont(new Font("Helvetica", Font.PLAIN, 14));
b1 = new Button("Thoi trang");
add(b1);
b2 = new Button("Do hoa");
add(b2);
b3 = new Button("Lap trinh");
add(b3);
}

public void paint(Graphics g) {
    if (!laidOut) {
        Insets insets = insets();
        b1.reshape(20 + insets.left, 5 + insets.top, 100, 20);
        b2.reshape(100 + insets.left, 35 + insets.top, 100, 20);
        b3.reshape(180 + insets.left, 15 + insets.top, 100, 20);
        laidOut = true;
    }
}

public boolean handleEvent(Event e) {
    if (e.id == Event.WINDOW_DESTROY) {
        if (inAnApplet) {
            dispose();
            return false;
        } else {
            System.exit(0);
        }
    }
    return super.handleEvent(e);
}

public static void main(String args[]) {
    NoneWindow window = new NoneWindow();
    Insets insets = window.insets();
    window.inAnApplet = false;
    window.setTitle("Design Global Sample");
    window.resize(350 + insets.left + insets.right,
        90 + insets.top + insets.bottom);
}

```

```

window.show();
}
}

```

Kết quả hiển thị như sau



4. Khai báo sự kiện tới Component(Listeners)

Trong bài học này sẽ trình bày một cách chi tiết để làm thế nào viết một sự kiện listeners cho một thành phần component.

4.1. Tổng quan về khai báo sự kiện(Event Listeners)

Cung cấp thông tin cần thiết về tất cả các kiểu của sự kiện. Một trong những tiêu đề trong phần này là trình bày cách làm sao để giảm bớt công sức và sự không cần thiết của việc viết code cho chương trình bằng cách sử dụng các lớp trong để thực thi các sự kiện.

Để có thể nắm bắt phần này một cách dễ dàng, xem như bạn đã có những kiến thức cơ bản về các sự kiện listener trong phần Event Handling trong các ví dụ trước đây. Chẳng hạn như ta có thể gắn một listeners vào nguồn của một đơn sự kiện. Nhưng quan trọng hơn hết là các phương thức event-listener sẽ được xử lý một cách nhanh chóng. Bởi vì tất cả các event-handling và các phương thức vẽ đều được thực hiện trong cùng một tiến trình.

Trong phần này, chúng ta sẽ bàn về **EventObject**, một lớp cơ bản cho tất cả các sự kiện AWT và Swing.

a) Lấy thông tin sự kiện: Event Objects

Mỗi một phương thức event-listener đều có một đối số, một đối tượng thừa kế từ lớp **EventObject**. Mặc dù đối số luôn xuất phát từ **EventObject**, với kiểu tổng quan để có thể thực hiện chính xác hơn. Ví dụ như khi nắm bắt sự kiện của chuột, đối số cho phương thức này sẽ lấy từ **MouseEvent**, một lớp con của **EventObject**.

Lớp **EventObject** định nghĩa một phương thức rất hữu ích như sau:

Object getSource()

Phương thức này trả về một đối tượng nắm bắt sự kiện.

Chú ý rằng phương thức **getSource** cũng trả về một đối tượng. Lớp Event đôi khi cũng định nghĩa một phương thức giống như **getSource**, nhưng kiểu của giá trị trả về hơi bị hạn chế. Ví dụ như lớp **ComponentEvent** định nghĩa phương thức **getComponent**, giống như

getSource, trả về đối tượng nắm bắt sự kiện. Cái khác nhau ở đây là **getComponent** luôn luôn trả về một **Component**.

Thường thì một lớp sự kiện nào đó định nghĩa một phương thức và trả về thông tin của sự kiện.

b) Khái niệm: Low-Level Events and Semantic Events

Các sự kiện có thể được phân chia thành 2 loại: **low-level** events và **semantic** events. Low-level events mô tả window -system xảy ra hoặc dữ liệu vào ở mức thấp (low-level input). Tất cả các sự kiện còn lại thuộc loại semantic event.

Sự kiện mouse và key, cả hai đều là kết quả trực tiếp từ phía người dùng, là những sự kiện low-level. Những sự kiện low-level khác bao gồm component, container, focus, và window events. Sự kiện component cho phép thay đổi vị trí, kích thước và sự hiển thị của thành phần. Sự kiện container quản lý để nắm bắt được thành phần nào khi được thêm vào hay gỡ bỏ khỏi các đối tượng chứa. Focus events sẽ báo cho biết khi nào một thành phần là có hoặc không **keyboard focus**, khả năng nhận biết ký tự được gõ tại bàn phím. Window events giúp để nắm bắt những trạng thái căn bản nhất của bất kỳ Window nào, chẳng hạn như Dialog hay một Frame.

Semantic events bao gồm **action events**, **item events**, và **list selection** events. Hành động của mỗi semantic event có thể khác nhau do thành phần. Ví dụ như một button có thể nắm bắt sự kiện khi người dùng kích chuột lên nó. Nhưng một text field nắm bắt sự kiện khi người dùng nhấn Return.

c) Sử dụng Adapters and Inner Classes để nắm bắt các sự kiện

Phần này hướng dẫn bạn sử dụng các lớp adapters và inner để làm giảm bớt sự lộn xộn trong đoạn mã của chương trình bạn.

Hầu hết các giao diện AWT listener, không như **ActionListener**, chứa nhiều hoặc một phương thức. Ví dụ, giao diện **MouseListener** chứa năm phương thức: **mousePressed**, **mouseReleased**, **mouseEntered**, **mouseExited**, và **mouseClicked**. Dù là bạn chỉ quan tâm về nhấn chuột, nếu lớp bạn đang sử dụng thực thi **MouseListener** thì bạn phải thực thi tất cả 5 phương thức.

Ví dụ :

```
//An example with cluttered but valid code.
public class MyClass implements MouseListener {
    ...
    someObject.addMouseListener(this);
    ...
    /* Empty method definition. */
}
```

```

public void mousePressed(MouseEvent e) {
}

/* Empty method definition. */
public void mouseReleased(MouseEvent e) {
}

/* Empty method definition. */
public void mouseEntered(MouseEvent e) {
}

/* Empty method definition. */
public void mouseExited(MouseEvent e) {
}

public void mouseClicked(MouseEvent e) {
    ...//Event handler implementation goes here...
}
}

```

Đáng tiếc là kết quả của sự lựa chọn các phương thức rỗng có thể khó đọc và duy trì. Để giúp bạn tránh được các lộn xộn với những phương thức rỗng trong chương trình, AWT cung cấp lớp **adapter** class cho mỗi **listenerinterface** với nhiều hơn một phương thức.

Để sử dụng adapter, bạn tạo một lớp con cho nó, thay vì phải thực thi một listener interface.

```

public class MyClass extends MouseAdapter {
    ...
    someObject.addMouseListener(this);
    ...
    public void mouseClicked(MouseEvent e) {
        ...//Event handler implementation goes here...
    }
}

```

Giả dụ bạn muốn viết một applet, và bạn muốn **Applet** của bạn chứa vài đoạn mã để nắm bắt các sự kiện của chuột. Từ khi ngôn ngữ Java không cho phép đa thừa kế thì bạn không thể mở rộng cả 2 lớp **Applet** and **MouseAdapter**. Giải pháp là định nghĩa một lớp **inner** -- một lớp nằm trong Applet -- that extends the **MouseAdapter** class,

```

//An example of using an inner class.
public class MyClass extends Applet {
    ...

```

```

someObject.addMouseListener(new MyAdapter());
...
class MyAdapter extends MouseAdapter {
    public void mouseClicked(MouseEvent e) {
        ...//Event handler implementation goes here...
    }
}
}

```

4.2. Hỗ trợ Listeners của các thành phần Swing

Có thể nói rằng loại của sự kiện một thành phần có thể được phân loại bằng cách dựa vào loại của sự kiện listeners mà ta đăng ký trên thành phần đó. Ví dụ như lớp Component định nghĩa phương thức listener như sau:

- **addComponentListener**
- **addFocusListener**
- **addKeyListener**
- **addMouseListener**
- **addMouseMotionListener**

Do vậy, mỗi thành phần hỗ trợ component, focus, key, mouse, và mouse-motion listeners. Tuy nhiên, một thành phần khởi động những sự kiện mà listeners có đăng ký trên nó. Ví dụ, một mouse listener được đăng ký trên một thành phần riêng biệt, nhưng thành phần ấy không có listeners khác, thì thành phần đó sẽ khởi động chỉ mỗi sự kiện mouse events, không có các sự kiện component, focus, key, or mouse-motion.

a) Các sự kiện được các thành phần Swing hỗ trợ

Vì tất cả các thành phần Swing đều xuất phát từ lớp AWT Component, cho nên ta phải khai báo những listeners sau trên bất kỳ thành phần Swing nào:

component listener

Nắm bắt sự thay đổi về kích thước, vị trí và sự hiển thị của thành phần.

focus listener

Nắm bắt các thành phần có nhận hay không tác động từ bàn phím.

key listener

Nắm bắt động tác ấn phím; sự kiện key chỉ khởi tạo bởi các thành phần đang có trạng thái mặc định của bàn phím.

mouse events

Nắm bắt sự kiện kích chuột và di chuyển chuột trên thành phần.

mouse-motion events

Nắm bắt sự thay đổi về vị trí của con trỏ trên thành phần.

b) Các Listeners khác mà các thành phần Swing hỗ trợ

Bảng sau đây liệt kê các thành phần Swing và listeners được hỗ trợ. Trong nhiều trường hợp, các sự kiện được khởi động trực tiếp từ thành phần. Những trường hợp khác, các sự kiện được khởi động từ dữ liệu của thành phần hoặc từ các kiểu mẫu được chọn.

COMPONENT	LISTENER							
	action	caret	change	document, undoable edit	item	list selection	wind ow	other
button	X		X		X			
check box	X		X		X			
color chooser			X					
combo box	X				X			
dialog							X	
editor pane		X		X				hyperlink
file chooser	X							
frame							X	
internal frame								internal frame
list						X		list data
menu								menu
menu item	X		X		X			menu key, menu drag mouse
option pane								
password field	X	X		X				
popup menu								popup menu
progress bar			X					
radio button	X		X		X			
slider			X					
tabbed pane			X					
table						X		table model, table column model, cell editor
text area		X		X				
text field	X	X		X				
text pane		X		X				hyperlink
toggle button	X		X		X			

tree								tree expansion, tree will expand, tree model, tree selection
viewport (used by scrollpane)			x					

4.3. Thực hiện Listeners cho các Handled Events thông thường

Phần này sẽ bao gồm các chi tiết về ví dụ và thông tin của việc viết những sự kiện listener thông thường.

a) Viết một Action Listener

Khi người sử dụng kích chuột vào Button , đúp chuột vào ListItem , chọn MenuItem, hoặc nhấn phím trong Text Field, một sự kiện sẽ xảy ra . Kết quả đó là một thông báo **actionPerformed** được gọi đi đến tất cả các action listener và nó đăng kí với các thành phần có liên quan.

Các phương thức, sự kiện của hành động

Giao diện **ActionListener** chứa một phương thức đơn , và do đó nó không có lớp adapter tương ứng. Đây là phương thức **ActionListener** cô độc:

void actionPerformed(ActionEvent)

Một ví dụ về nắm bắt các sự kiện của hành động

```
public class Beeper implements ActionListener {
//where initialization occurs:
    button.addActionListener(this);
    public void actionPerformed(ActionEvent e) {
        ...//Make a beep sound...
    }
}
```

b) Viết một Adjustment Listener

Các sự kiện Adjustment thông báo cho bạn biết sự thay đổi giá trị trong các thành phần. Đối tượng **Adjustable** có một giá trị nguyên, và nó trả về các các sự kiện adjustment bất cứ khi nào giá trị đó thay đổi. Chỉ có một lớp của AWT thực thi **Adjustable** là lớp **Scrollbar**.

Có 5 loại sự kiện adjustment:

track

Người sử dụng hoàn toàn thay đổi giá trị của thành phần.

unit increment, unit decrement

Người sử dụng chỉ biểu thị sự thay đổi nhỏ về giá trị của thành phần.

block increment, block decrement

Người sử dụng biểu thị sự thay đổi giá trị của thành phần với số lượng lớn.

Các phương thức sự kiện của Adjustment

Giao diện **AdjustmentListener** chứa một phương thức đơn, và vì thế nó không có lớp mô phỏng tương ứng. Sau đây là phương thức đó:

void adjustmentValueChanged(AdjustmentEvent)

Được gọi bởi AWT vừa sau khi thay đổi giá trị của thành phần.

Ví dụ về Handling Adjustment Events

```
class ConversionPanel implements AdjustmentListener {
    Scrollbar slider;
    ConversionPanel() {
        slider.addAdjustmentListener(this);
    }
    /** Respond to the slider. */
    public void adjustmentValueChanged(AdjustmentEvent e) {
        textField.setText(String.valueOf(e.getValue()));
        controller.convert(this);
    }
}
```

Lớp AdjustmentEvent

Phương thức **adjustmentValueChanged** có một thông số: một đối tượng **AdjustmentEvent**. Lớp **AdjustmentEvent** định nghĩa các phương thức sau:

Adjustable getAdjustable()

Trả về thành phần mà sinh ra sự kiện đó. Bạn có thể dùng nó thay vì dùng phương thức **getSource**.

int getAdjustmentType()

Trả về kiểu của adjustment được tìm thấy. giá trị trả về là một trong những giá trị sau được định nghĩa trong lớp **AdjustmentEvent**: **UNIT_INCREMENT**, **UNIT_DECREMENT**, **BLOCK_INCREMENT**, **BLOCK_DECREMENT**, **TRACK**.

int getValue()

Trả về giá trị của thành phần ngay sau khi adjustment được tìm thấy.

c) window listener, Text Listener, Mouse Motion Listener, Mouse Listener, Key Listener, Item Listener, Focus Listener, Contain Listener, Component Listener

Các sự kiện trên được viết chi tiết trong phần Phụ lục 1 của tài liệu này.

Tóm tắt bài học

FlowLayout sắp xếp các thành phần giao diện trước tiên theo hàng từ trái qua phải, sau đó theo cột từ trên xuống dưới. Cách sắp xếp này là mặc định đối với Panel, JPanel, Applet và JApplet.

GridLayout sắp xếp các thành phần giao diện được sắp xếp trong các ô lưới hình chữ nhật lần lượt theo hàng từ trái qua phải và theo cột từ trên xuống dưới trong một phần tử chứa. Mỗi thành phần giao diện chứa trong một ô.

BorderLayout sắp xếp các thành phần giao diện (ít hơn 5) được đặt vào các vị trí theo các hướng: north (bắc), south (nam), west (tây), east (đông) và center (trung tâm)). Cách sắp xếp này là mặc định đối với lớp Window, Frame, JFrame, Dialog và JDialog.

GridBagLayout sắp xếp cho phép đặt các thành phần giao diện vào lưới hình chữ nhật, nhưng một thành phần có thể chiếm nhiều nhiều hơn một ô.

null sắp xếp các thành phần bên trong vật chứa không được sắp lại khi kích thước của vật chứa thay đổi.

Customer Layout cho phép người dùng tùy biến sắp xếp bố cục các thành phần trên giao diện

EventListener cung cấp thông tin cần thiết về tất cả các kiểu của sự kiện.

Mỗi một phương thức event-listener đều có một đối số đơn, một đối tượng thừa kế từ lớp EventObject.

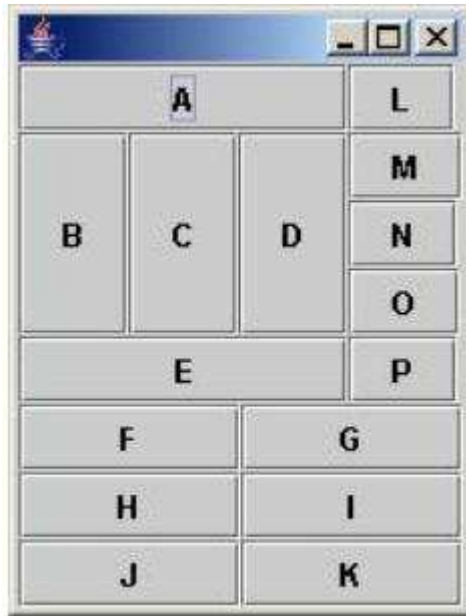
Các sự kiện có thể được phân chia thành 2 loại: low-level events và semantic events. Low-level events mô tả window -system xảy ra hoặc dữ liệu vào ở mức thấp (low-level input). Tất cả các sự kiện còn lại thuộc loại semantic event.

- Sự kiện mouse và key khả năng nhận biết ký tự được gõ tại bàn phím hoặc chuột được sử dụng
- Sự kiện component cho phép thay đổi vị trí, kích thước và sự hiển thị của thành phần
- Sự kiện container quản lý để nắm bắt được thành phần nào khi được thêm vào hay gỡ bỏ khỏi các đối tượng chứa
- Sự kiện focus sẽ báo cho biết khi nào một thành phần là có hoặc không keyboard focus

- Sự kiện window events giúp để nắm bắt những trạng thái căn bản nhất của bất kỳ Window nào, chẳng hạn như Dialog hay một Frame

Bài tập

Viết chương trình hiển thị giao diện như hình dưới :



Tại mỗi nút bấm tương ứng, hãy viết sự kiện click chuột hoặc bấm phím tương ứng sẽ chuyển màu nền của nút tương ứng.

CHƯƠNG III: LẬP TRÌNH CƠ SỞ DỮ LIỆU

Mục tiêu:

Sau khi kết thúc chương này, bạn có thể:

- Hiểu về lập trình với cơ sở dữ liệu sử dụng JDBC trong Java
- Hiểu về kiến trúc JDBC trong Java
- Hiểu các thiết lập Driver manager, Connection để kết nối tới các Hệ quản trị cơ sở dữ liệu khác nhau
- Hiểu cách sử dụng các đối tượng Statement, PreparedStatement để tạo các truy vấn SQL và thao tác với dữ liệu trả về
- Hiểu cách sử dụng đối tượng CallableStatement để thao tác với thủ tục
- Hiểu cách sử dụng nhóm lệnh (Batch) để nâng cao hiệu suất chương trình
- Hiểu về Transaction và sử dụng Transaction trong JDBC

1. GIỚI THIỆU

Các ứng dụng Internet ngày nay thường được dựa trên các cơ sở dữ liệu lớn được cài đặt bằng cách sử dụng công nghệ cơ sở dữ liệu quan hệ. Kể từ khi xuất hiện từ năm 1995, Java được yêu cầu cần cung cấp khả năng kết nối với các cơ sở dữ liệu quan hệ hiện có như Ingres, Oracle, Access, và SQL Server,... Các tiện ích cho phép truy xuất cơ sở dữ liệu nằm trong gói java.sql.

Ngày nay các thông tin với dung lượng lớn đều được lưu trữ trong các kho dữ liệu lớn. Khả năng truy xuất tới các cơ sở dữ liệu là điều không thể thiếu đối với các ứng dụng. Điều này lại càng đúng với các ứng dụng chạy trên mạng máy tính nói chung và Internet nói riêng. Trong chương này chúng ta sẽ đi vào tìm hiểu giao diện lập trình ứng dụng JDBC của Java và cách thức để kết nối với một cơ sở dữ liệu từ một ứng dụng Java thông qua JDBC.

2. JDBC Java Database Connectivity API

SUN đã phát triển một giao diện lập trình ứng dụng API để truy xuất cơ sở dữ liệu-JDBC. Mục tiêu đặt ra của SUN là:

- JDBC là một giao diện lập trình ứng dụng mức SQL.
- JDBC cần có được những kinh nghiệm làm việc với các API cơ sở dữ liệu hiện có.
- JDBC cần đơn giản

Giao diện lập trình ứng dụng mức SQL nghĩa là JDBC cho phép ta xây dựng các lệnh SQL và nhúng các lệnh SQL bên trong các lời gọi Java API. Nói tóm lại, về cơ bản ta vẫn sử dụng SQL nhưng JDBC cho phép ta dịch một cách trôi chảy giữa thế giới cơ sở dữ liệu và thế giới ứng dụng Java. Kết quả của bạn từ cơ sở dữ liệu, được trả về dưới dạng các đối tượng Java và nếu có vấn đề khi truy xuất nó sẽ đưa ra các ngoại lệ.

JDBC API đã chuẩn hóa:

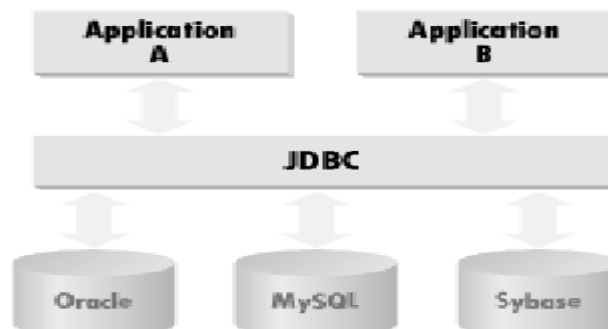
- Cách thiết lập tới cơ sở dữ liệu
- Cách tiếp cận để khởi tạo các truy vấn
- Cách thức để tạo ra các truy vấn có tham số
- Chuẩn hóa cấu trúc dữ liệu của kết quả truy vấn
 - Xác định số cột
 - Tra tìm các metadata.

JDBC API chưa chuẩn hóa cú pháp SQL. JDBC không phải là SQL nhưng. Lớp JDBC nằm trong gói java.sql. Nó bao gồm hai phần:

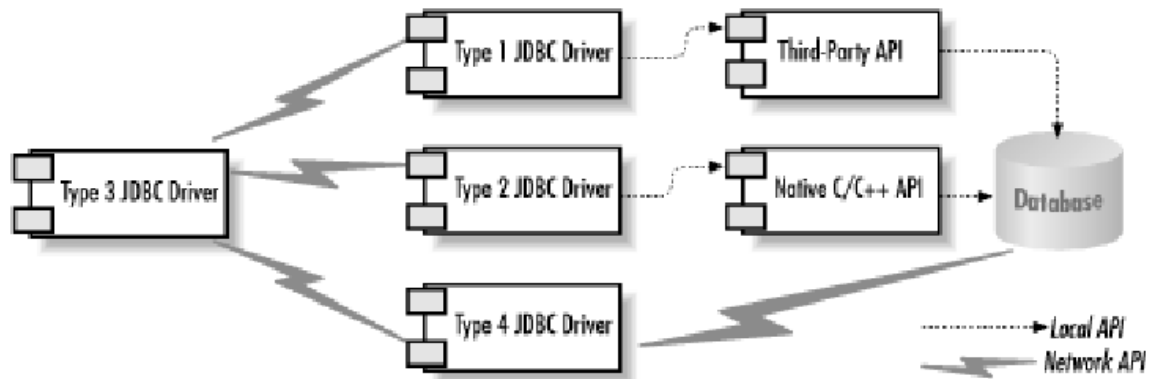
- JDBC API là một giao diện lập trình ứng dụng viết bằng ngôn ngữ Java thuần túy.
- Trình quản lý Driver JDBC truyền tin với các trình điều khiển cụ thể của nhà sản xuất, các trình điều khiển cơ sở dữ liệu của nhà sản xuất truyền tin với cơ sở dữ liệu

3. Kiến trúc của JDBC

JDBC thực hiện các mục tiêu của nó thông qua một tập hợp các giao tiếp JDBC, mỗi giao tiếp thực được thực hiện bởi từng nhà sản xuất. Tập hợp các lớp thực thi các giao tiếp JDBC cho một mô tơ cơ sở dữ liệu cụ thể được gọi là một trình điều khiển JDBC. Khi xây dựng một ứng dụng cơ sở dữ liệu, ta không phải xem xét đến tất cả các lớp cơ sở. JDBC che dấu các chi tiết của từng cơ sở dữ liệu và như vậy ta chỉ cần quan tâm đến ứng dụng của mình.



Các cơ sở dữ liệu và các trình điều khiển



3.1. Kiểu 1 (Cầu nối JDBC-ODBC)

Các trình điều khiển này sử dụng một công nghệ cầu nối để truy xuất tới một cơ sở dữ liệu. Cầu nối JDBC-ODBC được bắt đầu đưa vào từ JDK 1.2 là một ví dụ điển hình cho kiểu driver này. Nó cung cấp một gateway tới API ODBC. Cài đặt của API này thực hiện truy xuất tới cơ sở dữ liệu thực tế. Giải pháp cầu nối thường yêu cầu phần mềm phải được cài đặt trên hệ thống client.

Cầu nối JDBC-ODBC cung cấp cách truy xuất thông qua một hay nhiều trình điều khiển ODBC.

Ưu điểm:

- Đây là một cách tiếp cận tốt để học JDBC.
- Hữu ích cho các công ty đã cài đặt trình điều khiển ODBC trên từng máy client.
- Đây là cách duy nhất để truy xuất được tới các cơ sở dữ liệu trên máy tính để bàn mức thấp.

Nhược điểm:

- Không phù hợp với các ứng dụng quy mô lớn. Hiệu năng thấp vì có cần nhiều công đoạn cần thực hiện để chuyển từ JDBC sang ODBC.
- Không hỗ trợ tất cả các đặc trưng của Java.
 - Người sử dụng bị hạn chế bởi chức năng do trình điều khiển ODBC cung cấp.

3.2. Kiểu 2 (điều khiển gốc)

Các trình điều khiển kiểu 2 là các trình điều khiển API-trình điều khiển gốc. Điều này nghĩa là mã Java gọi các phương thức C hoặc C++ được cung cấp bởi từng nhà sản xuất hệ quản trị cơ sở dữ liệu để thực hiện truy xuất tới cơ sở dữ liệu. Giải pháp này vẫn yêu cầu phải có phần mềm trên hệ thống client. JDBC chuyển các lời gọi tới JDBC API thành các lời gọi kết nối với giao diện lập trình ứng dụng của máy khác cho một cơ sở dữ liệu cụ thể như IBM, Informix, Oracle, hoặc Sybase.

Ưu điểm:

- Hiệu năng tốt hơn kiểu 1, vì trình điều khiển kiểu 2 chứa các mã lệnh đã được biên dịch được tối ưu hóa cho hệ điều hành của server có sở dữ liệu hoạt động ở chế độ hậu trường,

Nhược điểm

- Người sử dụng cần đảm bảo rằng trình điều khiển JDBC của nhà sản xuất cơ sở dữ liệu có trên từng máy khách.
- Phải có chương trình đã được biên dịch cho mỗi hệ điều hành mà ứng dụng sẽ chạy.
- Chỉ sử dụng có hiệu quả trong các môi trường có kiểm soát như một mạng intranet

3.3. Kiểu 3 (trình điều khiển JDBC trên client)

Các trình điều khiển kiểu 3 cung cấp cho client một API mạng chung, API này sau đó chuyển thành thao tác truy xuất cơ sở dữ liệu mức server. Mặt khác, trình điều khiển JDBC trên client sử dụng các socket để gọi một ứng dụng trung gian (middleware) trên server để chuyển các yêu cầu của client thành một API cụ thể đối với từng server. Kết quả là trình điều khiển này đặc biệt linh hoạt, vì nó không cần phải có phần mềm cài đặt trên client và một trình điều khiển có thể cung cấp khả năng truy xuất tới nhiều cơ sở dữ liệu.

Trình điều khiển Java thuần túy cho các chương trình trung gian cơ sở dữ liệu để dịch các lời gọi JDBC cho giao thức của nhà sản xuất phần mềm trung gian, trình điều khiển này sau đó được chuyển cho một giao thức gắn với cơ sở dữ liệu cụ thể bởi phần mềm server trung gian.

Ưu điểm:

- Được sử dụng khi một công ty có nhiều cơ sở dữ liệu và muốn sử dụng một trình điều khiển JDBC để kết nối với tất cả các cơ sở dữ liệu.
- Trình điều khiển nằm trên server, vì thế không cần trình điều khiển JDBC trên từng máy client
- Thành phần server được tối ưu hóa cho hệ điều hành đang chạy ở chế độ hậu trường

Nhược điểm:

- Cần mã lệnh cho cơ sở dữ liệu cụ thể trên server trung gian

3.4. Kiểu 4 (trình điều khiển kết nối trực tiếp tới CSDL)

Sử dụng các giao thức mạng được tích hợp sẵn vào engine cơ sở dữ liệu, các driver kiểu 4 truyền tin trực tiếp với cơ sở dữ liệu bằng cách sử dụng socket Java. Đây là trình điều khiển Java thuần túy nhất. Kiểu trình điều khiển này thường do nhà sản xuất cơ sở dữ liệu cung cấp.

Trình điều khiển Java thuần túy tới kết nối trực tiếp với cơ sở dữ liệu chuyển các lời gọi JDBC thành các gói tin được truyền đi trên mạng theo một khuôn dạng được sử dụng bởi cơ sở dữ liệu cụ thể. Cho phép một lời gọi trực tiếp từ máy client tới cơ sở dữ liệu.

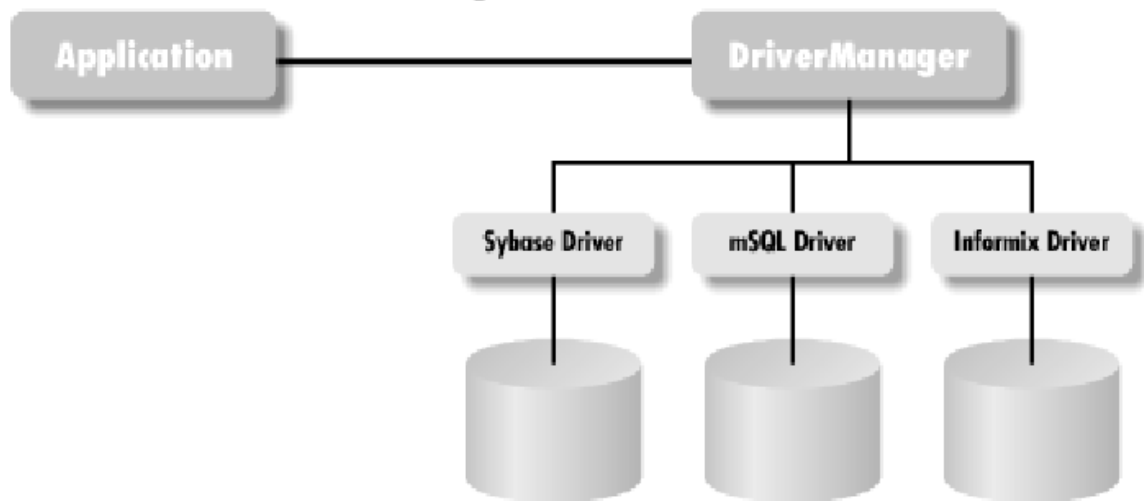
Ưu điểm:

- Không cần cài phần mềm đặc biệt nào trên client hoặc server. Có thể được tải về một cách linh hoạt

Nhược điểm

- Không tối ưu cho hệ điều hành server vì vậy trình điều khiển không thể tận dụng các đặc trưng ưu việt của hệ điều hành

4. Kết nối cơ sở dữ liệu



Hình vẽ trên cho thấy cách thức mà một ứng dụng JDBC truyền tin với một hoặc nhiều cơ sở dữ liệu mà không cần biết đến các chi tiết có liên quan đến cài đặt driver cho cơ sở dữ liệu đó. Một ứng dụng sử dụng JDBC như là một giao tiếp, thông qua đó nó truyền tất cả các yêu cầu liên quan đến cơ sở dữ liệu của nó.

Khi ta viết các applet hay ứng dụng cơ sở dữ liệu, ta có thể cung cấp các thông tin cụ thể về trình điều khiển JDBC là URL cơ sở dữ liệu. Thậm chí ta có thể nhập vào URL cơ sở dữ liệu cho ứng dụng và applet vào thời gian chạy dưới dạng các tham số.

JDBC là gói kết nối cơ sở dữ liệu bao gồm giao diện lập trình ứng dụng căn bản Java API. Java cung cấp một interface độc lập với cơ sở dữ liệu để mở một kết nối tới cơ sở dữ liệu, bằng cách phát ra các lời gọi SQL tới cơ sở dữ liệu và nhận về kết quả là một tập hợp các dữ liệu. Ở góc độ kỹ thuật, JDBC đóng vai trò như là một chương trình cài đặt giao tiếp mức lời gọi SQL được định nghĩa bởi X/Open và được hỗ trợ bởi hầu hết các nhà cung cấp cơ sở dữ liệu quan hệ. Để thực hiện giao tác với một kiểu cơ sở dữ liệu cụ thể, ta cần phải có

một trình điều khiển JDBC đóng vai trò như là một cầu nối giữa các lời gọi phương thức JDBC và interface cơ sở dữ liệu.

4.1. DriverManager

DriverManager cung cấp phương tiện để nạp các trình điều khiển cơ sở dữ liệu vào một ứng dụng Java hoặc một applet; nó chính là cách để JDBC thiết lập một liên kết với cơ sở dữ liệu. Ứng dụng Java, trước tiên tạo một đối tượng DriverManager, kết nối với cơ sở dữ liệu bằng cách gọi phương thức tĩnh getConnection() của lớp DriverManager, với tham chiếu truyền vào giống như một URL được gọi là URL cơ sở dữ liệu. DriverManager tìm kiếm một driver hỗ trợ việc kết nối trong tập hợp các driver hiện có. Nếu tìm thấy driver nó truyền địa chỉ cơ sở dữ liệu cho driver và yêu cầu driver tạo ra một kết nối. Kết nối tới cơ sở dữ liệu được trả về dưới dạng một đối tượng Connection.

Tất cả các driver JDBC cung cấp một cài đặt giao tiếp java.sql.Driver. Khi một DriverManager được tạo ra, nó tải một tập hợp các driver được xác định bởi thuộc tính của java.sql.Driver. Driver được nạp vào thời gian chạy Java, nó có nhiệm vụ tạo ra một đối tượng và đăng ký đối tượng với DriverManager. Các driver cần cho ứng dụng có thể được nạp bởi phương thức Class.forName()

```
Driver myDriver=(Driver)Class.forName("specialdb.Driver");
```

4.2. Connection

Mỗi khi các driver cần thiết được nạp bởi DriverManager, sẽ có một liên kết với một cơ sở dữ liệu được tạo ra nhờ phương thức getConnection() của lớp DriverManager. Cơ sở dữ liệu cần làm việc được xác định thông qua một tham số String đóng vai trò như là địa chỉ tham chiếu tới cơ sở dữ liệu. Không có một khuôn dạng chuẩn nào cho địa chỉ xâu cơ sở dữ liệu; DriverManager truyền xâu địa chỉ cho từng driver JDBC đã được nạp và xem nó có hiểu và hỗ trợ kiểu cơ sở dữ liệu đã được xác định.

Jdbc:odbc:financedata

Trong đó financedata là nguồn cơ sở dữ liệu cục bộ. Để truy xuất tới một cơ sở dữ liệu từ xa từ một máy client ta có thể dùng cú pháp sau:

Jdbc:odbc:drv://dataserver.foobar.com:500/financedata.

Đặc tả JDBC API khuyến cáo một URL cơ sở dữ liệu nên có dạng như sau:

Jdbc:<sub-protocol>:<sub-name>

Trong đó <sub-protocol> xác định dịch vụ kết nối cơ sở dữ liệu và <sub-name> cung cấp tất cả các thông tin cần thiết để dịch vụ tìm cơ sở dữ liệu và kết nối tới nó.

Phương thức getConnection() trên DriverManager hoặc là trả về một đối tượng Connection biểu diễn liên kết tới cơ sở dữ liệu đã được chỉ ra, hoặc là đưa ra ngoại lệ nếu liên kết không được thiết lập.

4.3. Statement

Giao tiếp Connection cho phép người sử dụng tạo ra một câu lệnh truy vấn tới cơ sở dữ liệu. Các lệnh truy vấn được biểu diễn dưới dạng các đối tượng Statement hoặc các lớp con của nó. Giao tiếp Connection cung cấp ba phương thức để tạo ra các lệnh truy vấn cơ sở dữ liệu là: **createStatement()**, **prepareStatement()**, và **prepareCall()**. **createStatement()** được sử dụng cho các lệnh SQL đơn giản không liên quan đến các tham số. Phương thức này trả về một đối tượng Statement được sử dụng để phát ra các truy vấn SQL tới cơ sở dữ liệu, bằng cách sử dụng phương thức **executeQuery()**. Phương thức này chấp nhận một lệnh SQL như là một chuỗi và các kết quả trả về là ở dưới dạng một đối tượng ResultSet. Các phương thức khác có trong giao tiếp Statement để phát ra các lệnh SQL tới các cơ sở dữ liệu là phương thức **execute()**, phương thức này được sử dụng cho các truy vấn SQL và trả về nhiều resultset và phương thức **executeUpdate()** được sử dụng để phát ra các lệnh INSERT, UPDATE, hoặc DELETE.

Ngoài giao tiếp Statement cơ bản, một đối tượng Connection có thể được sử dụng để tạo ra một đối tượng **PreparedStatement** và các **CallableStatement** biểu diễn các thủ tục stored procedure trong cơ sở dữ liệu. Một lệnh SQL có thể liên quan đến nhiều tham số đầu vào, hoặc một lệnh mà ta muốn xử lý nhiều lần, có thể được tạo ra bằng cách sử dụng lệnh **prepareStatement()** trên đối tượng Connection, phương thức này trả về đối tượng PreparedStatement. Lệnh SQL được truyền cho phương thức **prepareStatement()** là một lệnh được biên dịch trước vì vậy việc xử lý nhiều lần một lệnh sẽ hiệu quả hơn. Lớp con của lớp Statement hỗ trợ việc thiết lập các giá trị của các tham số đầu vào được biên dịch trước thông qua các phương thức setXXX(). Đối tượng PreparedStatement có phương thức **executeQuery()** không cần tham số, thay vào đó nó xử lý các lệnh SQL được biên dịch trước trên cơ sở dữ liệu. Chú ý rằng không phải tất cả các nhà sản xuất cơ sở dữ liệu hoặc các driver JDBC đều hỗ trợ các lệnh được biên dịch trước.

4.4. ResultSet

Các dòng dữ liệu được trả về từ việc xử lý một lệnh được biểu diễn bằng một ResultSet trong JDBC. Ví dụ, phương thức **executeQuery()** của Statement trả về một đối tượng ResultSet. Đối tượng ResultSet cung cấp các cách để duyệt qua các dòng dữ liệu được trả về từ việc xử lý câu lệnh truy vấn SQL thông qua phương thức **next()** của nó; các trường dữ liệu trong mỗi hàng có thể được tìm kiếm thông qua các tên hoặc chỉ mục cột bằng cách sử dụng phương thức getXXX(). Người dùng cần phải biết kiểu dữ liệu trong mỗi cột dữ liệu được trả về, vì mỗi mục dữ liệu được tìm kiếm thông qua các phương thức getXXX() có kiểu cụ thể.

Tùy thuộc vào kiểu trình điều khiển JDBC được cài đặt, việc duyệt qua các hàng dữ liệu trong đối tượng ResultSet có thể tạo ra hiệu ứng lấy dữ liệu từ cơ sở dữ liệu, hoặc đơn giản là trả về từng hàng dữ liệu từ cache. Nếu hiệu năng của các giao dịch là vấn đề đối với

ứng dụng, ta cần xác định dữ liệu trả về được quản lý như thế nào bởi các trình điều khiển của nhà sản xuất.

Lưu ý: Giá trị trả lại của hàm getXXX(args) là dữ liệu của trường có tên là args của các dòng dữ liệu đã được chọn ra. Ngoài ra cũng cần phân biệt các kiểu của Java với các kiểu dữ liệu của SQL. Bảng dưới đây mô tả các kiểu dữ liệu tương ứng của Java, SQL và các hàm getXXX().

Kiểu của SQL	Kiểu của Java	Hàm getXXX()
CHAR	String	getString()
VARCHAR	String	getString()
LONGVARCHAR	String	getString()
NUMERIC	java.math.BigDecimal	getBigDecimal()
DECIMAL	java.math.BigDecimal	getBigDecimal()
BIT	Boolean (boolean)	getBoolean()
TINYINT	Integer (byte)	getByte()
SMALLINT	Integer (short)	getShort()
INTEGER	Integer (int)	getInt()
BIGINT	Long (long)	getLong()
REAL	Float (float)	getFloat()
FLOAT	Double (double)	getDouble()
DOUBLE	Double (double)	getDouble()
BINARY	byte[]	getBytes()
VARBINARY	byte[]	getBytes()
LONGVARBINARY	byte[]	getBytes()
DATE	java.sql.Date	getDate()
TIME	java.sql.Time	getTime()
TIMESTAMP	java.sql.Timestamp	getTimestamp()

5. Lớp DatabaseMetaData

Muốn xử lý tốt các dữ liệu của một CSDL thì chúng ta phải biết được những thông tin chung về cấu trúc của CSDL đó như: hệ QTCSDL, tên của các bảng dữ liệu, tên gọi của các trường dữ liệu, v.v .

Để biết được những thông tin chung về cấu trúc của một hệ CSDL, chúng ta có thể sử dụng giao diện `java.sql.DatabaseMetaData` thông qua hàm `getMetaData()`.

```
DatabaseMetaData dbmeta = con.getMetaData();
```

Trong đó, `con` là đối tượng kết nối đã được tạo ra bởi lớp **Connection**.

Lớp `DatabaseMetaData` cung cấp một số hàm được nạp chồng để xác định được những thông tin về cấu hình của một CSDL. Một số hàm trả về đối tượng của `String` (`getURL()`), một số trả lại giá trị logic (**`nullsAreSortedHigh()`**) hay trả lại giá trị nguyên như hàm `getMaxConnection()`. Những hàm khác trả về kết quả là các đối tượng của **ResultSet** như: `getColumns()`, `getTableType()`, `getPrivileges()`, v.v.

6. Lớp ResultSetMetaData

Giao diện **ResultSetMetaData** cung cấp các thông tin về cấu trúc cụ thể của **ResultSet**, bao gồm cả số cột, tên và giá trị của chúng. Dưới đây là một chương trình hiển thị các kiểu và giá trị của từng trường của một bảng dữ liệu.

Ví dụ Chương trình hiển thị một bảng dữ liệu.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
import java.sql.Statement;
public class MySqlSample {
    final static String jdbcDriver = "com.mysql.jdbc.Driver";
    final static String jdbcURL = "jdbc:mysql://localhost:3306/wordpress";
    final static String table = "wp_posts";
    final static String user = "root";
    final static String pass = "";

    public static void main(java.lang.String[] args) {
        try {
            Class.forName(jdbcDriver);
            Connection con =
```

```
        DriverManager.getConnection(jdbcURL, user, pass);
    Statement stmt = con.createStatement();
    // Đọc ra cả bảng Student và đưa vào đối tượng rs
    ResultSet rs = stmt.executeQuery("SELECT * FROM " + table);
    // Đọc ra các thông tin về rs
    ResultSetMetaData rsmd = rs.getMetaData();
    // Xác định số cột của rsmd
    int colCount = rsmd.getColumnCount();
    for (int col = 1; col <= colCount; col++) {
        // In ra tên và kiểu của từng trường dữ liệu trong rsmd
        System.out.print(rsmd.getColumnLabel(col));
        System.out.print(" (" + rsmd.getColumnTypeName(col) + ")");
        if (col < colCount) {
            System.out.print(", ");
        }
    }
    System.out.println();
    while (rs.next()) {
        // In ra dòng dữ liệu trong rsmd
        for (int col = 1; col <= colCount; col++) {
            System.out.print(rs.getString(col));
            if (col < colCount) {
                System.out.print(" ");
            }
        }
        System.out.println();
    }
    rs.close();
    stmt.close();
    con.close();
} catch (ClassNotFoundException e) {
    System.out.println("Unable to load database driver class");
} catch (SQLException se) {
    System.out.println("SQL Exception: " + se.getMessage());
}
}
```

7. Các bước cơ bản để kết nối với cơ sở dữ liệu từ một ứng dụng Java

a) Bước 1: Nạp trình điều khiển

```
try{
Class.forName("com.mysql.jdbc.Driver");
}
catch(ClassNotFoundException e) {
System.out.println("Unable to load database driver class");
}
```

b) Bước 2: Xác định URL cơ sở dữ liệu

```
String URL="jdbc:mysql://localhost:3306/wordpress";
```

c) Bước 3: Thiết lập liên kết

```
String username="design_global";
String password="design_global";
Connection con=DriverManager.getConnection(URL,username,password);
```

d) Bước 4: Tạo ra một đối tượng Statement

```
Statement s=con.createStatement();
```

e) Bước 5: Xử lý truy vấn

```
ResultSet rs = stmt.executeQuery("SELECT * FROM wp_posts");
```

f) Bước 6: Xử lý kết quả

```
while (rs.next()) {
    for (int col = 1; col <= colCount; col++) {
        System.out.print(rs.getString(col));
        if (col < colCount) {
            System.out.print(" ");
        }
    }
}
```

Cột đầu tiên có chỉ mục là 1 chứ không phải là 0.

g) Bước 7: Đóng liên kết

```
con.close();
```

8. Các ví dụ về kết nối cơ sở dữ liệu từ ứng dụng Java

a) Ví dụ 1:

```
import java.sql.PreparedStatement;
import java.sql.DriverManager;
```



```
import java.sql.SQLException;
import java.sql.Connection;

public class PreparedStatementSample {
    final static String jdbcDriver = "com.mysql.jdbc.Driver";
    final static String jdbcURL = "jdbc:mysql://localhost:3306/wordpress";
    final static String table = "wp_posts";
    final static String user = "root";
    final static String pass = "";

    public static void main(java.lang.String[] args) {
        try {
            Class.forName(jdbcDriver);
            Connection con = DriverManager.getConnection(jdbcURL, user, pass);
            String query = "insert into emp(name,salary) values(?,?)";
            PreparedStatement prSt = con.prepareStatement(query);
            prSt.setString(1, "John");
            prSt.setInt(2, 10000);
            //count will give you how many records got updated
            int count = prSt.executeUpdate();
            //Run the same query with different values
            prSt.setString(1, "Cric");
            prSt.setInt(2, 5000);
            count = prSt.executeUpdate();
            prSt.close();
            con.close();
        } catch (ClassNotFoundException e) {
            System.out.println("Unable to load database driver class");
            e.printStackTrace();
        } catch (SQLException se) {
            System.out.println("SQL Exception: " + se.getMessage());
            se.printStackTrace();
        }
    }
}
```

9. Sử dụng PreparedStatement

Đôi khi việc sử dụng một đối tượng PreparedStatement hiệu quả và tiện lợi hơn nhiều so với việc sử dụng đối tượng Statement. Kiểu lệnh đặc biệt này là lớp con của lớp Statement.

a) Khi nào cần sử dụng đối tượng PreparedStatement

Nếu ta muốn xử lý một đối tượng Statement nhiều lần, ta có thể sử dụng đối tượng PreparedStatement để giảm thời gian xử lý.

Đặc trưng chính của một đối tượng PreparedStatement là nó được cung cấp trước một lệnh SQL trước khi tạo ra đối tượng. Đối tượng PreparedStatement là một lệnh SQL đã được biên dịch trước. Điều này nghĩa là khi đối tượng PreparedStatement được xử lý, hệ quản trị cơ sở dữ liệu chỉ cần xử lý lệnh SQL của PreparedStatement mà không phải biên dịch nó.

Mặc dù PreparedStatement có thể được sử dụng với các lệnh SQL không có tham số nhưng ta thường hay sử dụng các lệnh SQL có tham số. Ưu điểm của việc sử dụng lệnh SQL có tham số là ta có thể sử dụng cùng một lệnh và cung cấp cho nó các giá trị khác nhau mỗi khi xử lý. Ta sẽ thấy điều này trong ví dụ ở phần sau.

b) Tạo một đối tượng PreparedStatement

Giống như các đối tượng Statement, bạn đọc có thể tạo ra các đối tượng PreparedStatement với một phương thức Connection. Sử dụng một kết nối mở trong ví dụ trước là con, có thể tạo ra đối tượng PreparedStatement nhận hai tham số đầu vào như sau:

```
PreparedStatement updateSales = con.prepareStatement(
"UPDATE COFFEES SET SALES = ? WHERE COF_NAME LIKE ?");
```

c) Cung cấp các giá trị cho các tham số của đối tượng PreparedStatement

Ta cần cung cấp các giá trị được sử dụng thay cho vị trí của các dấu hỏi nếu có trước khi xử lý một đối tượng PreparedStatement. Ta có thể thực hiện điều này bằng cách gọi một trong các phương thức setXXX đã được định nghĩa trong lớp PreparedStatement. Nếu giá trị ta muốn thay thế cho dấu hỏi (?) là kiểu int trong Java, ta có thể gọi phương thức setInt. Nếu giá trị ta muốn thay thế cho dấu (?) là kiểu String trong Java, ta có thể gọi phương thức setString,... Một cách tổng quát, ứng với mỗi kiểu trong ngôn ngữ lập trình Java sẽ có một phương thức setXXX tương ứng.

Ví dụ:

```
import java.sql.*;

public class PrepareStmt {
    public static void main(String args[]) {
        int empid;
        String LastName;
```

```
String FirstName;
String query = "SELECT * FROM Employees where EmployeeID=?";
try {
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    Connection con = DriverManager.getConnection("jdbc:odbc:MyData");
    PreparedStatement pstmt = con.prepareStatement(query);
    pstmt.setInt(1, 2);
    ResultSet rs = pstmt.executeQuery();
    while (rs.next()) {
        empid = rs.getInt("EmployeeID");
        LastName = rs.getString("LastName");
        FirstName = rs.getString("FirstName");
        System.out.println(empid + ", " + LastName + "\t" + FirstName + "\t");
    }
} catch (ClassNotFoundException e) {
    e.printStackTrace();

} catch (SQLException e) {
    e.printStackTrace();
}
}
```

Ta có thể sử dụng vòng lặp để thiết lập các giá trị cho các tham số đầu vào.

```
PreparedStatement updateSales;
String updateString = "update COFFEES "
    + "set SALES = ? where COF_NAME like ?";
updateSales = con.prepareStatement(updateString);
int[] salesForWeek = {175, 150, 60, 155, 90};
String[] coffees = {"Colombian", "French_Roast", "Espresso",
    "Colombian_Decaf", "French_Roast_Decaf"};
int len = coffees.length;
for (int i = 0; i < len; i++) {
    updateSales.setInt(1, salesForWeek[i]);
    updateSales.setString(2, coffees[i]);
    updateSales.executeUpdate();
}
```

Các giá trị trả về của phương thức executeUpdate

Phương thức **executeQuery** trả về một đối tượng ResultSet chứa các kết quả của truy vấn được gửi tới hệ quản trị cơ sở dữ liệu, giá trị trả về khi xử lý phương thức executeUpdate là một số nguyên int chỉ ra số hàng trong bảng đã được cập nhật.

```
updateSales.setInt(1, 50);
updateSales.setString(2, "Espresso");
int n = updateSales.executeUpdate();
```

10. CallableStatement

Callable Statement cung cấp câu lệnh gọi thi hành các thủ tục đã cài đặt sẵn trên các hệ quản trị cơ sở dữ liệu.

Cú pháp như sau :

```
Call procedure_name (arg1, arg2, ...)
? = Call procedure _nam arg1, arg2, ...
```

Dấu “?” thay cho các đối số. Đối số có thể là input (IN parameter), output (OUT parameter) hoặc cả 2 (INOUT).

Sử dụng như sau:

```
CallableStatement callableStatement = con.prepareStatement("{call getDBUSERByUserId(?,?,?,?)}");
```

Truyền đối số IN bằng hàm setXXX() kế thừa từ PreparedStatement, đăng ký đối số OUT trước khi thi hành thủ tục

```
RegisterOutParmeter(1, Type.VARCHAR);
Stmt1.setString(1, "00000");
Stmt.registerOutParameter(1, Type.VARCHAR);
```

Ví dụ :

```
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.Date;
public class CallableStatementSample {

    final static String jdbcDriver = "com.mysql.jdbc.Driver";
    final static String jdbcURL = "jdbc:mysql://localhost:3306/wordpress";
    final static String table = "wp_posts";
    final static String user = "root";
```

```

    final static String pass = "";
//  CREATE OR REPLACE PROCEDURE getDBUSERByUserId(
//      p_userid IN DBUSER.USER_ID%TYPE,
//      o_username OUT DBUSER.USERNAME%TYPE,
//      o_createdby OUT DBUSER.CREATED_BY%TYPE,
//      o_date OUT DBUSER.CREATED_DATE%TYPE)
//  IS
//  BEGIN
//
//      SELECT USERNAME , CREATED_BY, CREATED_DATE
//      INTO o_username, o_createdby, o_date
//      FROM DBUSER WHERE USER_ID = p_userid;
//
//  END;

public static void main(java.lang.String[] args) {
    String getDBUSERByUserIdSql = "{call getDBUSERByUserId(?,?,?,?)}";
    try {
        Class.forName(jdbcDriver);
        Connection con = DriverManager.getConnection(jdbcURL, user, pass);

        CallableStatement callableStatement = con.prepareCall(getDBUSERByUserIdSql);

        callableStatement.setInt(1, 10);
        callableStatement.registerOutParameter(2, java.sql.Types.VARCHAR);
        callableStatement.registerOutParameter(3, java.sql.Types.VARCHAR);
        callableStatement.registerOutParameter(4, java.sql.Types.DATE);
        callableStatement.executeUpdate();

        String userName = callableStatement.getString(2);
        String createdBy = callableStatement.getString(3);
        Date createdDate = callableStatement.getDate(4);
        System.out.println("UserName : " + userName);
        System.out.println("CreatedBy : " + createdBy);
        System.out.println("CreatedDate : " + createdDate);
    } catch (ClassNotFoundException e) {
        System.out.println("Unable to load database driver class");
    } catch (SQLException se) {
        System.out.println("SQL Exception: " + se.getMessage());
    }
}

```

```
}  
}  
}
```

11. Thực thi nhóm lệnh đồng thời (Batch)

Batch Update là một kỹ thuật mà Java cho phép chương trình có thể thực thi đồng thời nhiều lệnh để tác động lên Database chỉ với 1 Connection. Trong chương trình, bạn có thể tạo ra 1 **Connection** gắn liền với 1 **Statement** Object (Hoặc **PreparedStatement**, **CallableStatement**), sau đó tích hợp các lệnh cần thiết trước khi tác động lên Database. Khi đã hội đủ điều kiện thì tiến hành chọn chức năng lệnh phù hợp, lúc này Connection mới thực sự tạo ra và chuyển thông tin đến cho Database Server.

Ta thấy kỹ thuật này thể hiện ưu điểm đầu tiên chính là giúp lập trình viên có thể định hướng và triển khai chức năng lệnh của chương trình cho mục đích làm giảm lưu lượng truyền thông đến Database tốt hơn, hơn nữa việc thực thi và xử lý thông tin khi có nhu cầu điều chỉnh ngay tại Client với dữ liệu đang có trong bộ nhớ sẽ làm hiệu xuất thực thi ứng dụng tại Client tốt hơn, thuật toán phục vụ cho khả năng hỗ trợ hoàn tác trong quá trình làm việc đối với người dùng sẽ đơn giản hơn rất nhiều.

```
import java.sql.Connection;  
import java.sql.Statement;  
//...  
Connection connection = new getConnection();  
Statement statement = connection.createStatement();  
for(Employee employee: employees) {  
    String query = "insert into employee (name, city) values("  
        + employee.getName() + ", " + employee.getCity() + ")";  
    statement.addBatch(query);  
}  
statement.executeBatch();  
statement.close();  
connection.close();
```

12. Sử dụng Transaction

a) Quản lý giao tác

Một giao tác là một tập hợp một hoặc nhiều lệnh được xử lý cùng với nhau như một chỉnh thể thống nhất (đơn vị). Khi xử lý một giao tác hoặc tất cả các lệnh được xử lý hoặc không lệnh nào được xử lý. Nhiều trường hợp ta không muốn một lệnh có hiệu lực ngay nếu lệnh khác không thành công.

Điều này có thể được thực hiện nhờ phương thức `setAutoCommit()` của đối tượng `Connection`. Phương thức này nhận một giá trị boolean làm tham số..

Ngăn chế độ Auto-commit

Khi một liên kết được tạo ra, thì liên kết đó ở chế độ auto-commit.

Mỗi lệnh SQL được xem như là một giao tác và sẽ được tự động hoàn thành ngay khi nó được xử lý.

Cách để cho phép hai hoặc nhiều lệnh được nhóm cùng với nhau thành một giao tác là cấm chế độ auto-commit.

Ví dụ:

```
con.setAutoCommit(false);
```

b) Xác nhận hoàn thành một giao tác

Mỗi khi chế độ auto-commit bị cấm, không có lệnh SQL nào sẽ được xác nhận hoàn thành cho tới khi ta gọi phương thức **`commit()`**.

Ta có thể thực hiện điều này bằng cách gọi phương thức **`commit()`** của các đối tượng liên kết.

Nếu ta cố gắng xử lý một hay nhiều lệnh trong một giao tác và nhận được một ngoại lệ `SQLException`, ta cần gọi phương thức **`rollback()`** để hủy bỏ giao tác và khởi động lại toàn bộ giao tác.

Ví dụ:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;

public class PrepareUpdate {
    public static void main(String args[]) throws Exception {
        int rows = 0;
        String query = "insert into EMP " + "(EmployeeID, LASTNAME, FIRSTNAME) " + "values "
            + "(?, ?, ?)";
        try {
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            Connection con = DriverManager.getConnection("jdbc:odbc:MyData");
            con.setAutoCommit(false);
            PreparedStatement pstmt = con.prepareStatement(query);
            pstmt.setInt(1, Integer.parseInt(args[0]));
```

```
pstmt.setString(2, args[1]);
    pstmt.setString(3, args[2]);
    rows = pstmt.executeUpdate();
    pstmt.close();
pstmt = null;
    System.out.println(rows + " rows inserted");
    System.out.println("");
    con.commit();
} catch (ClassNotFoundException e) {
    e.printStackTrace();
} catch (SQLException e) {
    e.printStackTrace();
}
}
```

Sau đó thực hiện các lệnh: Các chương trình Java chỉ thực hiện được các lệnh trên CSDL thông qua đối tượng **Statement**.

Các câu lệnh SQL có thể được thực hiện tức thì thông qua đối tượng **Statement**, có thể là một câu lệnh biên dịch trước (đối tượng **PreparedStatement**) hay có thể là một lệnh gọi các thủ tục cài sẵn (Stored Procedure) trong CSDL (đối tượng **CallableStatement**).

Các câu lệnh SQL có thể được thực hiện thông qua phương thức **executeQuery()** – kết quả là một đối tượng **ResultSet**, hay phương thức **executeUpdate()** – kết quả là một số nguyên cho biết tổng số các record chịu ảnh hưởng của câu lệnh vừa thực hiện (thường là các câu lệnh sửa đổi dữ liệu Update - Delete).

Trong trường hợp có sử dụng trình quản lý transaction, các phương thức **rollback()** được dùng để phục hồi trạng thái trước đó và **commit()** để xác nhận việc thực hiện lệnh.

Để chấm dứt cần xóa kết nối, xóa các đối tượng để giải phóng tài nguyên của hệ thống.

Tóm tắt bài học

JDBC API đã chuẩn hóa:

- Cách thiết lập tới cơ sở dữ liệu
- Cách tiếp cận để khởi tạo các truy vấn
- Cách thức để tạo ra các truy vấn có tham số
- Chuẩn hóa cấu trúc dữ liệu của kết quả truy vấn
 - Xác định số cột

- Tra tìm các metadata.

JDBC API chưa chuẩn hóa cú pháp SQL. JDBC không phải là SQL nhưng. Lớp JDBC nằm trong gói `java.sql`. Nó bao gồm hai phần:

- JDBC API là một giao diện lập trình ứng dụng viết bằng ngôn ngữ Java thuần túy.
- Trình quản lý Driver JDBC truyền tin với các trình điều khiển cụ thể của nhà sản xuất, các trình điều khiển cơ sở dữ liệu của nhà sản xuất truyền tin với cơ sở dữ liệu

Các trình điều khiển JDBC có các kiểu sau:

- Kiểu 1 (Cầu nối JDBC-ODBC)
- Kiểu 2 (điều khiển gốc)
- Kiểu 3 (trình điều khiển JDBC trên client)
- Kiểu 4 (trình điều khiển kết nối trực tiếp tới CSDL)

Tất cả các driver JDBC cung cấp một cài đặt giao tiếp `java.sql.Driver`.

Mỗi khi các driver cần thiết được nạp bởi `DriverManager`, sẽ có một liên kết với một cơ sở dữ liệu được tạo ra nhờ phương thức `getConnection()` của lớp `DriverManager`. Cơ sở dữ liệu cần làm việc được xác định thông qua một tham số String đóng vai trò như là địa chỉ tham chiếu tới cơ sở dữ liệu.

Giao tiếp `Connection` cho phép người sử dụng tạo ra một câu lệnh truy vấn tới cơ sở dữ liệu. Các lệnh truy vấn được biểu diễn dưới dạng các đối tượng `Statement` hoặc các lớp con của nó. Giao tiếp `Connection` cung cấp ba phương thức để tạo ra các lệnh truy vấn cơ sở dữ liệu là: `createStatement()`, `prepareStatement()`, và `prepareCall()`. `createStatement()` hoặc tạo ra một đối tượng `PreparedStatement` và các `CallableStatement` biểu diễn các thủ tục stored procedure trong cơ sở dữ liệu.

Các dòng dữ liệu được trả về từ việc xử lý một lệnh được biểu diễn bằng một `ResultSet` trong JDBC.

Bài tập

Viết chương trình quản lý thông tin sinh viên. Chương trình cho phép thêm mới, cập nhật và xóa bản ghi từ bảng `Student`. Cấu trúc bảng `Student` như sau:

Column Name	Data Type
Name	Varchar
Rollno	Numeric
Class	Varchar

Giao diện chương trình như sau:



CHƯƠNG IV: Collections và cấu trúc dữ liệu trong Java

Mục tiêu:

Sau khi kết thúc chương này, bạn có thể:

- Tạo dựng kiểu cấu trúc dữ liệu Linked List
- Nắm bắt cách thao tác với Stack và Queue
- Hiểu về Collection cũng như các thực thi của các lớp trong Collection Framework
- Ứng dụng để nâng cao hiệu năng thực thi cho ứng dụng, tái sử dụng mã nguồn

1. Giới thiệu

Mảng cũng tốt, nhưng làm việc với chúng cũng có đôi chút bất tiện. Nạp giá trị cho mảng cũng mất công, và một khi khai báo mảng, bạn chỉ có thể nạp vào mảng những phần tử đúng kiểu đã khai báo và với số lượng phần tử đúng bằng số lượng mà mảng có thể chứa. Mảng chắc chắn là không có vẻ hướng đối tượng lắm. Mảng có trong mọi phần mềm, bởi vậy không có mảng sẽ khiến cho ngôn ngữ khó mà tồn tại trong thế giới thực, đặc biệt khi bạn phải tương tác với các hệ thống khác có dùng mảng.

Bên cạnh mảng, Java cung cấp cho bạn nhiều công cụ để quản lý uyển chuyển và hướng đối tượng hơn như Stack, Queue, Linked List, Tree và Collection Framework như List, Set, Map, ...

Khi bạn cần một số lượng cố định các phần tử có cùng kiểu, bạn có thể dùng mảng. Khi bạn cần các phần tử có kiểu khác nhau hoặc số lượng các phần tử có thể thay đổi linh hoạt, bạn dùng các Collection của Java.

Các Collection (Collection, Set, List, Map, ArrayList, Vector, Hashtable, HashSet, HashMap). Các collection được đặt trong gói java.util. JCF là một kiến trúc hợp nhất để biểu diễn và thao tác trên các Collection.

2. Cấu trúc dữ liệu trong Java

2.1. LinkedList

Linked list là cấu trúc gồm các node liên kết với nhau thông qua các mối liên kết. Node cuối linked list được đặt là null để đánh dấu kết thúc danh sách.

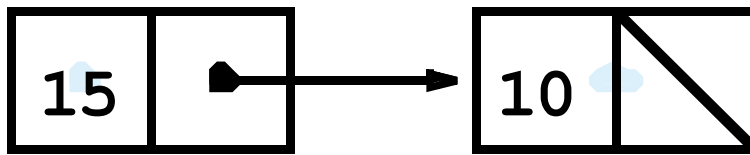
Linked list giúp tiết kiệm bộ nhớ so với mảng trong các bài toán xử lý danh sách.

Khi chèn/xoá một node trên linked list, không phải dẫn/dồn các phần tử như trên mảng.

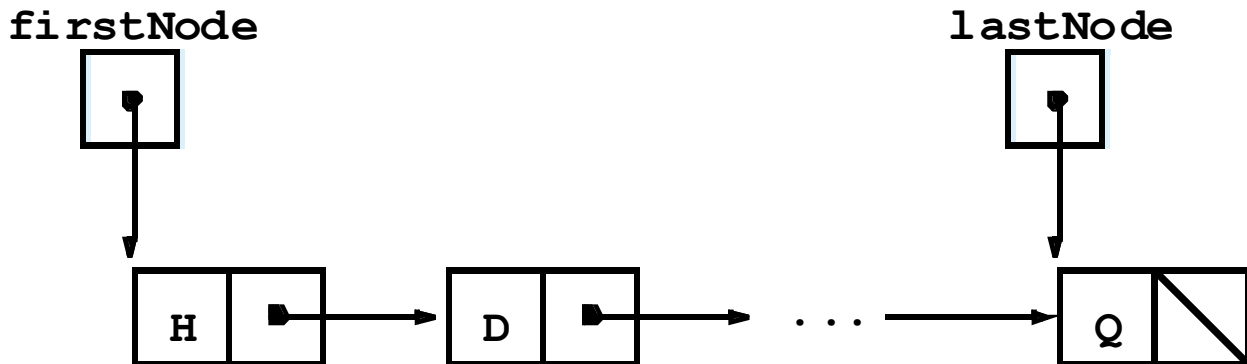
Việc truy nhập trên linked list luôn phải tuần tự.

Thể hiện Node thông qua lớp tự tham chiếu (self-referential class).

```
class Node
{
    private int data;
    private Node nextNode;
    // constructors and methods ...
}
```



Một linked list được quản lý bởi tham chiếu tới node đầu và node cuối.



a) Cài đặt LinkedList như sau:

```
class ListNode {
    int data;
    ListNode nextNode;

    ListNode(int value) {
        this(value, null);
    }
    ListNode(int value, ListNode node) {
        data = value;
        nextNode = node;
    }
    int getData() {
```

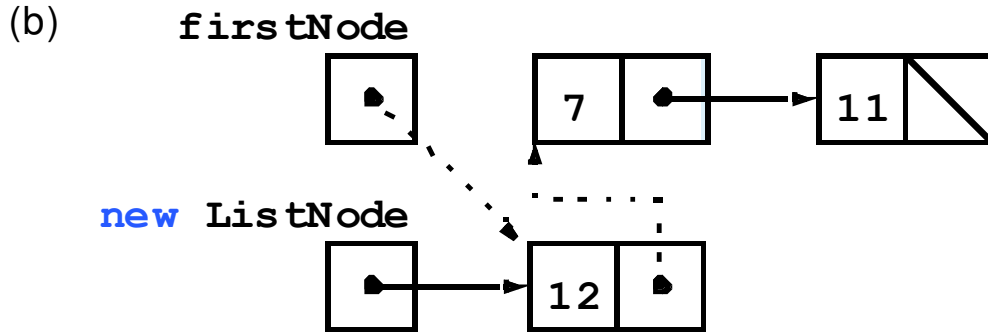
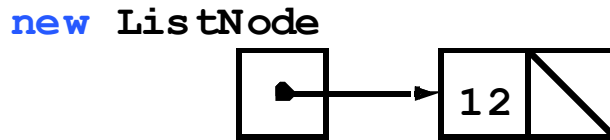
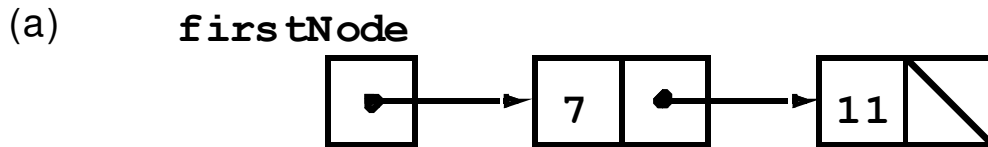
```
        return data;
    }
    ListNode getNext() {
        return nextNode;
    }
}

// Định nghĩa lớp LinkedList
class LinkedList {
    private ListNode firstNode;
    private ListNode lastNode;

    public LinkedList() {
        firstNode = lastNode = null;
    }
    public void insertAtFront(int insertItem) {
        if (isEmpty()) {
            firstNode = lastNode = new ListNode(insertItem);
        } else {
            firstNode = new ListNode(insertItem, firstNode);
        }
    }
    public void insertAtBack(int insertItem) {
        if (isEmpty()) {
            firstNode = lastNode = new ListNode(insertItem);
        } else {
            lastNode = lastNode.nextNode = new ListNode(insertItem);
        }
    }
    public int removeFromFront() {
        int removeItem = -1;
        if (!isEmpty()) {
            removeItem = firstNode.data;
            if (firstNode == lastNode) {
                firstNode = lastNode = null;
            } else {
                firstNode = firstNode.nextNode;
            }
        }
    }
}
```

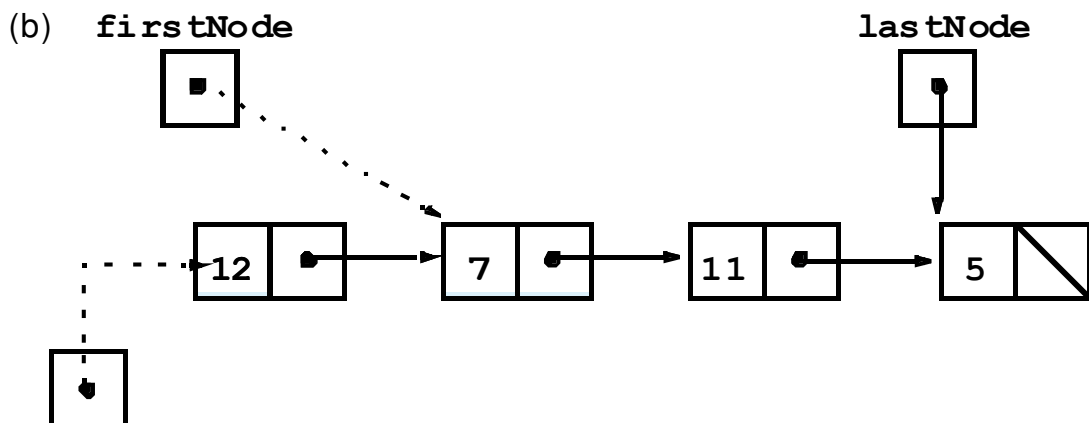
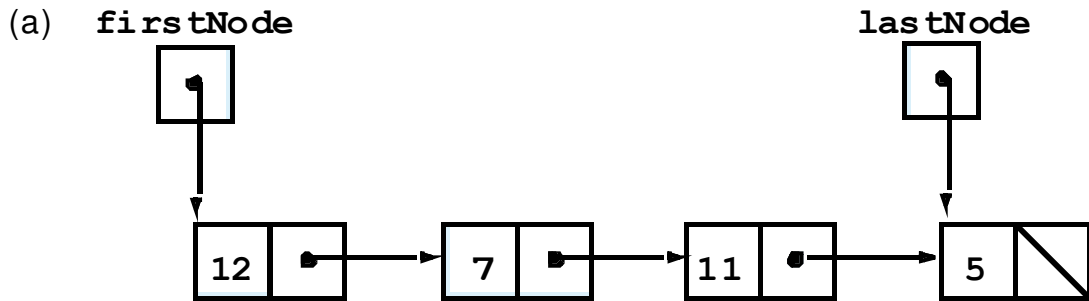
```
    }  
    }  
    return removeItem;  
}  
public int removeFromBack() {  
    int removeItem = -1;  
    if (!isEmpty()) {  
        removeItem = lastNode.data;  
        if (firstNode == lastNode) {  
            firstNode = lastNode = null;  
        } else {  
            ListNode current = firstNode;  
            while (current.nextNode != lastNode) {  
                current = current.nextNode;  
            }  
            lastNode = current;  
            current.nextNode = null;  
        }  
    }  
    return removeItem;  
}  
public boolean isEmpty() {  
    return (firstNode == null);  
}  
public void print() {  
    ListNode node = firstNode;  
    while (node != null) {  
        System.out.print(node.data + " ");  
        node = node.nextNode;  
    }  
    System.out.println("\n");  
}  
}
```

b) Mô tả insertAtFront



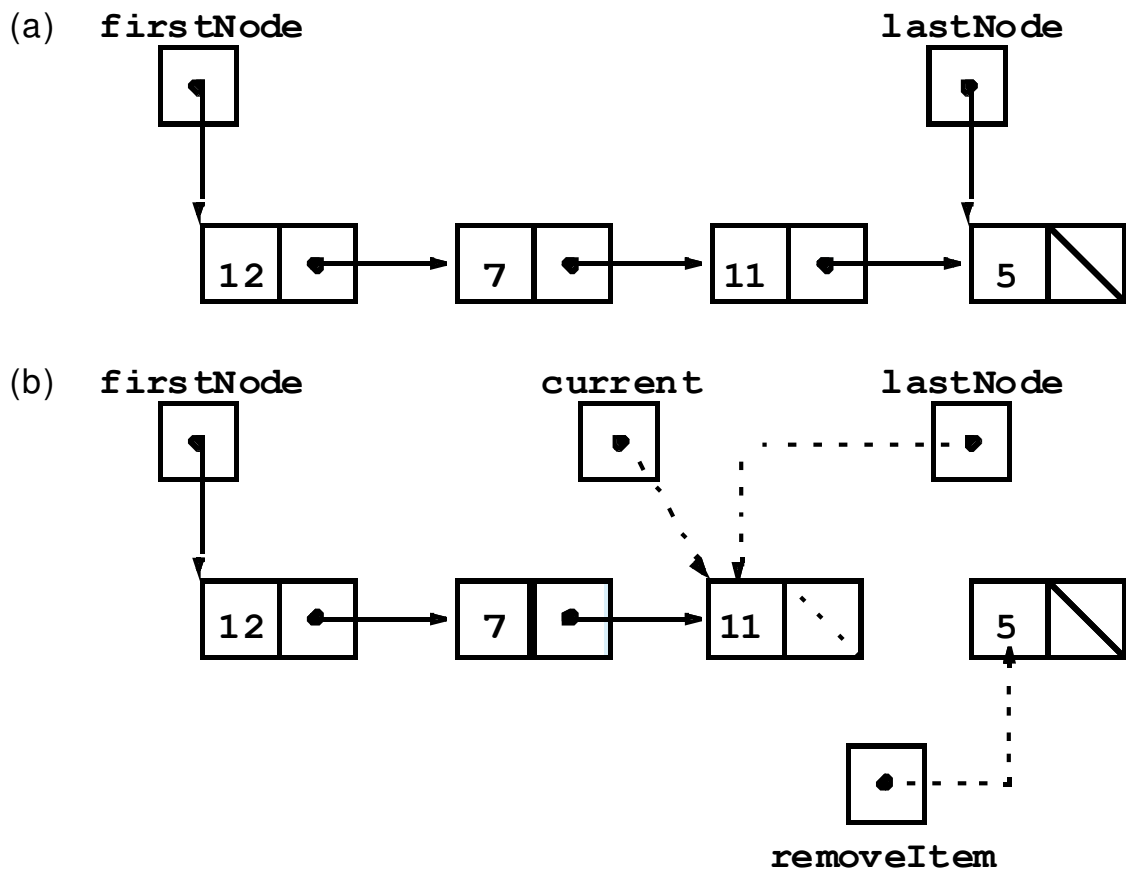
c)

d) Mô tả **removeFromFront**



e) **removeItem**

f) Mô tả **removeFromBack**



g)

h) Sử dụng LinkedList

```
public class ListTest
{
    public static void main( String args[] )
    {
        LinkedList list = new LinkedList();
        list.insertAtFront( 5 );
        list.insertAtFront( 7 );
        list.insertAtBack( 9 );
        list.insertAtBack( 8 );
        list.insertAtBack( 4 );
        list.print();
        list.removeFromFront();
        list.removeFromBack();
        list.print();
    }
}
```

Kết quả hiển thị như sau:

7 5 9 8 4
5 9 8

i)

2.2. Stack

Stack là một cấu trúc theo kiểu LIFO (Last In First Out), phần tử vào sau cùng sẽ được lấy ra trước.

Các thao tác cơ bản trên Stack

- Push : thêm 1 phần tử vào đỉnh Stack
- Pop : lấy 1 phần tử từ đỉnh Stack
- Peek: trả về phần tử đầu Stack mà không loại bỏ nó ra khỏi Stack
- isEmpty: Kiểm tra Stack có rỗng hay không
- Search: trả về vị trí phần tử trong Stack tính từ đỉnh stack nếu không thấy trả về -1

Ví dụ

```
class Stack
{
    private LinkedList stackList;
    public Stack(){
        stackList = new LinkedList();
    }
    public void push( int value ) {
        stackList.insertAtFront( value );
    }
    public int pop() { return stackList.removeFromFront(); }
    public boolean isEmpty() { return stackList.isEmpty(); }
    public void print() { stackList.print(); }
}

public class StackTest
{
    public static void main(String[] args) {
        Stack stack = new Stack();
        stack.push(5);
        stack.push(7);
        stack.push(4);
        stack.push(8);
```

```
        stack.print();
        stack.pop();
        stack.pop();

        stack.print();
    }
}
```

Kết quả hiển thị như sau:

```
8 4 7 5
7 5
```

2.3. Queue

Queue (Hàng đợi) là cấu trúc theo kiểu FIFO (First In First Out), phần tử vào trước sẽ được lấy ra trước.

Hai thao tác cơ bản trên hàng đợi

- Chèn phần tử: Luôn chèn vào cuối hàng đợi (enqueue)
- Lấy ra phần tử: Lấy ra từ đầu hàng đợi (dequeue)

Ví dụ:

```
class Queue {
    private LinkedList queueList;
    public Queue() {
        queueList = new LinkedList();
    }
    public void enqueue(int value) {
        queueList.insertAtBack(value);
    }
    public int dequeue() {
        return queueList.removeFromFront();
    }
    public boolean isEmpty() {
        return queueList.isEmpty();
    }
    public void print() {
        queueList.print();
    }
}
```

```
}
```

Kết quả hiển thị như sau:

```
5 7 4 8
```

```
4 8
```

3. Collections Framework

3.1. Các thành phần của Java Collection

- **Interfaces:** Là các giao tiếp thể hiện tính chất của các kiểu collection khác nhau như List, Set, Map.
- **Implementations:** Là các lớp collection có sẵn được cài đặt các collection interfaces.
- **Algorithms:** Là các phương pháp tính đã được xử lý trên collection, ví dụ : sắp xếp danh sách, dùng phần tử

3.2. Các thao tác chính trên Collection

Collection cung cấp các tính năng chính để thao tác với nó như là thêm/xóa/tìm kiếm các phần tử.

- boolean add(Object element);
- boolean remove(Object element);
- boolean contains(Object element);
- int size();
- boolean isEmpty();
- void clear();
- Object[] toArray();

Nếu lớp được cài đặt kế thừa từ Collection mà không hỗ trợ các thao tác làm việc với các phần tử thêm/xóa/tìm kiếm sẽ bị xuất ra ngoại lệ UnsupportedOperationException.

3.3. Set (tập hợp)

Tập hợp Set là cấu trúc dữ liệu, trong đó không có sự lặp lại và không có sự sắp xếp của các phần tử. Giao diện Set không định nghĩa thêm các hàm mới mà chỉ giới hạn lại các hàm của Collection để không cho phép các phần tử của nó được lặp lại.

Giả sử a, b là hai tập hợp (hai đối tượng của các lớp cài đặt Set). Kết quả thực hiện trên a, b có thể mô tả như trong bảng sau:

Các phép hợp tương ứng trong **Set**

```
a.containsAll(b)  b ⊆ a (Tập con)
```

```
a.addAll(b)  a = a ∪ b (Hợp tập hợp)
```

```
a.removeAll(b)  a = a - b (Hiệu tập hợp)
```

`a.retainAll(b)` $a = a \cap b$ (Giao tập hợp)

`a.clear()` $a = \emptyset$ (Tập rỗng)

Sau đây chúng ta xét một số lớp thực thi cài đặt giao diện Set.

HashSet

Một dạng cài đặt nguyên thủy của Set là lớp HashSet, trong đó các phần tử của nó là không được sắp xếp. Lớp này có các toán tử khởi tạo:

`HashSet()`

Tạo ra một tập mới không có phần tử nào cả (tập rỗng).

`HashSet(Collection c)`

Tạo ra một tập mới chứa các phần tử của tập hợp c.

`HashSet(int initCapacity)`

Tạo ra một tập mới rỗng có kích thước (khả năng chứa) là `initCapacity`.

`HashSet(int initCapacity, float loadFactor)`

Tạo ra một tập mới rỗng có kích thước (khả năng chứa) là `initCapacity` và yếu tố được nạp vào là `loadFactor`.

Ví dụ Khi thực hiện các đối số được đưa vào sau tên chương trình theo dòng lệnh. Chương trình bắt đầu với `tap1` là rỗng và lấy các ký tự của đối số đầu tiên để tạo ra `tap2`. So sánh hai tập đó, thông báo kết quả ra màn hình, sau đó cộng dồn `tap2` vào `tap1` và lại tiếp tục như thế đối với đối số tiếp theo cho đến hết.

```
import java.util.HashSet;
public class HashSetExample {
    public static void main(String args[]) {
        HashSet set1 = new HashSet();
        int iNumberArgument = args.length;
        for (String tmp : args) {
            HashSet setTmp = new HashSet();
            for (int i = 0; i < tmp.length(); i++) {
                setTmp.add(tmp.charAt(i));
            }
            HashSet setChung = (HashSet) set1.clone();
            setChung.retainAll(setTmp);
            if (setChung.size() == 0) {
                System.out.println("Hai tap " + set1 + " va " + setTmp + " khong co phan tu chung");
            } else {
```

```

        boolean bolSub = setTmp.containsAll(set1);
        boolean bolSuper = set1.containsAll(setTmp);
        if (bolSub && bolSuper){
            System.out.println("Hai tap " + set1 + " va " + setTmp + " la bang nhau");
        } else if (bolSub) {
            System.out.println("tap " + setTmp + " chua " + set1);

        } else if (bolSuper) {
            System.out.println("tap " + set1 + " chua " + setTmp);
        } else {
            System.out.println("tap " + setTmp + " va " + set1 + " co cac phan tu chung la " +
setChung);
        }
    }
    set1.addAll(setTmp);
}
}
}

```

Dịch và thực hiện chương trình với các đối số như sau:

```
java HashSetExample hoc sinh hoc vien design global
```

Kết quả thực thi chương trình:

```

Hai tap [] va [c, o, h] khong co phan tu chung
tap [s, n, h, i] va [c, o, h] co cac phan tu chung la [h]
tap [s, c, n, o, h, i] chua [c, o, h]
tap [v, e, n, i] va [s, c, n, o, h, i] co cac phan tu chung la [n, i]
tap [g, d, e, s, n, i] va [v, e, s, c, n, o, h, i] co cac phan tu chung la [e, s, n, i]
tap [g, b, a, o, l] va [g, v, d, e, s, c, n, o, h, i] co cac phan tu chung la [g, o]

```

3.4. List (danh sách)

Cấu trúc List là dạng tập hợp các phần tử được sắp theo thứ tự (còn được gọi là dãy tuần tự) và trong đó cho phép lặp (hai phần tử giống nhau). Ngoài những hàm mà nó được kế thừa từ Collection, List còn bổ sung thêm những hàm như:

```
Object get(int index)
```

Trả về phần tử được xác định bởi index.

```
Object set(int index, Object elem) // Tùy chọn
```

Thay thế phần tử được xác định bởi index bằng elem

```
void add(int index, Object elem) // Tùy chọn
```

Chèn elem vào sau phần tử được xác định bởi index.

```
Object remove(int index) // Tùy chọn
```

Bỏ đi phần tử được xác định bởi index.

```
boolean addAll(int index, Collection c) // Tùy chọn
```

Chèn các phần tử của tập hợp c vào vị trí được xác định bởi index.

```
int indexOf(Object elem)
```

Cho biết vị trí lần xuất hiện đầu tiên của elem trong danh sách.

```
int lastIndexOf(Object elem)
```

Cho biết vị trí lần xuất hiện cuối cùng của elem trong danh sách.

```
List subList(int fromIndex, int toIndex)
```

Lấy ra một danh sách con từ vị trí fromIndex đến toIndex .

```
ListIterator listIterator()
```

Trả về các phần tử liên tiếp bắt đầu từ phần tử đầu tiên.

```
ListIterator listIterator(int index)
```

Trả về các phần tử liên tiếp bắt đầu từ phần tử được xác định bởi index. Trong đó ListIterator là giao diện mở rộng giao diện Iterator đã có trong java.lang.

3.5. Các lớp ArrayList, Vector

Hai lớp này có những toán tử khởi tạo để tạo ra những danh sách mới rỗng hoặc có các phần tử lấy theo các tập hợp khác.

Vector và ArrayList là hai lớp kiểu mảng động (kích thước thay đổi được). Hiệu suất sử dụng hai lớp này là tương đương nhau, tuy nhiên nếu xét theo nhiều khía cạnh khác thì ArrayList là cấu trúc hiệu quả nhất để cài đặt cấu trúc danh sách

List

Ví dụ Hệ thống có một dãy N_DIGIT (5) chữ số bí mật. Hãy viết chương trình nhập vào N_DIGIT chữ số để đoán xem có bao nhiêu chữ số trùng và có bao nhiêu vị trí các chữ số trùng với dãy số cho trước.

```
import java.util.*;

public class NhapDoanSo {
    final static int N_DIGIT = 5;

    public static void main(String args[]){
        if(args.length != N_DIGIT) {
```

```

        System.err.println("Hay doan " + N_DIGIT + " chu so!");
        return;
    }
    List biMat = new ArrayList();// Tạo danh sách biMat là rỗng
    biMat.add("5");    // Bỏ sung các số vào dãy biMat
    biMat.add("3");
    biMat.add("2");
    biMat.add("7");
    biMat.add("2");

    List doan = new ArrayList();// Tạo danh sách doan là rỗng
    for(int i = 0; i < N_DIGIT; i++)
        doan.add(args[i]); // Đưa các số từ đối số chương trình vào doan
    List lap = new ArrayList(biMat);// Lưu biMat sang lap
    int nChua = 0;
    // Đếm số các chữ số trùng nhau, nghĩa là thực hiện được phép bỏ đi remove()
    for(int i = 0; i < N_DIGIT; i++)
        if (lap.remove(doan.get(i)))
            ++nChua;
    int nViTri = 0;
    ListIterator kiemTra = biMat.listIterator();
    ListIterator thu = doan.listIterator();
    // Tìm những vị trí đoán trùng trong hai dãy có lặp
    while (kiemTra.hasNext())// Khi còn phần tử tiếp theo
        // Kiểm tra xem lần lượt các vị trí của hai dãy có trùng nhau hay không
        if (kiemTra.next().equals(thu.next()))
            nViTri++;
    // Thông báo kết quả ra màn hình
    System.out.println(nChua + " chu so doan trung.");
    System.out.println(nViTri + " vi tri doan trung.");
}
}

```

Dịch và thực hiện chương trình:

```
java NhapDoanSo 3 2 2 2 7
```

Kết quả hiển thị

4 chu so doan trung

1 vi tri doan trung

3.6. Map (ánh xạ)

Map định nghĩa các ánh xạ từ các khoá (keys) vào các giá trị. Lưu ý: các khoá phải là duy nhất (không cho phép lặp). Mỗi khoá được ánh xạ sang nhiều nhất một giá trị, được gọi là ánh xạ đơn.

Các ánh xạ không phải là các tập hợp, giao diện Map cũng không phải là mở rộng của các Collection. Song, phép ánh xạ có thể xem như là một loại tập hợp theo nghĩa: các khoá (key) tạo thành tập hợp và tập hợp các giá trị (value) hoặc tập các cặp <key, value>.

Giao diện Map khai báo những hàm sau:

```
Object put(Object key, Object value); // Tùy chọn
```

Chèn vào một cặp <key, value>

```
Object get(Object key);
```

Đọc giá trị được xác định bởi key nếu có ánh xạ, ngược lại cho null nếu không có ánh xạ ứng với key.

```
Object remove(Object key); // Tùy chọn
```

Loại bỏ ánh xạ được xác định bởi key.

```
boolean containsKey(Object key);
```

Cho giá trị true nếu key được ánh xạ sang một giá trị nào đó, ngược lại là false.

```
boolean containsValue(Object value);
```

Cho giá trị true nếu value được ánh xạ bởi một key nào đó, ngược lại là false.

```
int size();
```

Cho số các cặp ánh xạ <key, value>.

```
boolean isEmpty();
```

Cho giá trị true nếu ánh xạ rỗng, ngược lại là false.

Một số phép toán thường dùng

```
void putAll(Map t); // Tùy chọn
```

Sao lại các ánh xạ từ t.

```
void clear(); // Tùy chọn
```

Xoá đi tất cả các ánh xạ.

```
Set keySet();
```

Xác định tập các khoá.

```
Collection values();
```

Xác định tập hợp các giá trị.

```
Set entrySet();
```

Xác định tập các ánh xạ <key, value>.

3.7. Các lớp HashMap và HashTable

Hai lớp này cài đặt giao diện Map và được xây dựng trong java.lang. Chúng cho phép tạo ra các ánh xạ mới có thể rỗng hoặc có những kích thước tùy ý.

Ví dụ

Viết chương trình nhập vào các trọng lượng và in ra tần suất của các trọng lượng đó trong các nhóm cách nhau 5 đơn vị (kg).

```
import java.util.*;
public class NhomTrongLuong {
    public static void main(String args[]) {
        int iCount = args.length;
        Map mResult = new HashMap<Long, Integer>();
        for (String strTemp : args) {
            double dInput = Double.parseDouble(strTemp);
            long iRound = Math.round(dInput / 5) * 5;
            if (mResult.containsKey(iRound)){
                Integer iTemp = (Integer) mResult.get(iRound);
                mResult.remove(iRound);
                mResult.put(iRound, iTemp + 1);
            } else {
                mResult.put(iRound, 1);
            }
        }
        List<Long> lstResultKey = new ArrayList<Long>(mResult.keySet());
        Collections.sort(lstResultKey);
        for(Long lTempKey : lstResultKey){
            Integer iTemValue = (Integer) mResult.get(lTempKey);
            char[] arrCharPrint = new char[iTemValue];
            Arrays.fill(arrCharPrint, '*');
            System.out.println(lTempKey + "\t" + new String(arrCharPrint));
        }
    }
}
```

Dịch và chạy chương trình NhomTrongLuong với các tham số:

```
java NhomTrongLuong 75 72 93 12 34
```

Kết quả hiển thị như sau:

10 *
35 *
70 *
75 *
95 *

Như vậy, nhóm 10 kg có 1, 35 kg có 1, 70 kg có 1, 75 kg có 1 và 95 kg có 1.

3.8. SortedSet (tập được sắp) và SortedMap (ánh xạ được sắp)

Các cấu trúc tập hợp (set) và ánh xạ (map) có giao diện đặc biệt là SortedSet và SortedMap để cài đặt những cấu trúc có các phần tử được sắp theo thứ tự chỉ định.

a) Giao diện SortedSet

SortedSet là giao diện mở rộng của Set cung cấp các hàm để xử lý các tập được sắp.

SortedSet headSet(Object toElem);

Trả về tập được sắp xếp gồm những phần tử đứng trước **toElem**.

SortedSet tailSet(Object fromElem);

Trả về tập được sắp gồm những phần tử cuối đứng sau **fromElem**.

SortedSet subSet(Object fromElem, Object toElem);

Trả về tập được sắp gồm những phần tử kể từ **fromElem** đến **toElem**.

Object **first()**; Trả về phần tử đầu tiên (cực tiểu) của tập được sắp.

Object **last()**; Trả về phần tử cuối cùng (cực đại) của tập được sắp.

Comparator comparator();

Trả về thứ tự so sánh của cấu trúc được sắp, cho null nếu các phần tử được sắp theo thứ tự tự nhiên (tăng dần)

b) Giao diện SortedMap

SortedMap là giao diện mở rộng của Map cung cấp các hàm để xử lý các ánh xạ được sắp theo thứ tự của khoá (key).

SortedMap headMap(Object toKey);

Trả về ánh xạ được sắp gồm những phần tử đứng trước toKey.

SortedMap tailMap(Object fromKey);

Trả về ánh xạ được sắp gồm những phần tử cuối đứng sau fromKey.

SortedMap subMap(Object fromKey, Object toKey);

Trả về ánh xạ được sắp gồm những phần tử kể từ fromKey đến toKey.

Object **firstKey()**; Trả về phần tử đầu tiên (cực tiểu) của ánh xạ được sắp.

Object **lastKey()**; Trả về phần tử cuối cùng (cực đại) của ánh xạ được sắp.

3.9. TreeSet và TreeMap

Hai lớp này cài đặt hai giao diện SortedSet và SortedMap tương ứng. Chúng có bốn loại toán tử khởi tạo như sau:

```
TreeSet()
```

```
TreeMap()
```

Tạo ra những tập hoặc ánh xạ mới và rỗng, được sắp theo thứ tự tăng dần của các phần tử hoặc của khoá.

```
TreeSet(Comparator c)
```

```
TreeMap(Comparator c)
```

Tạo ra những tập hoặc ánh xạ mới được sắp và xác định thứ tự so sánh theo c.

```
TreeSet(Collection c)
```

```
TreeMap(Map m)
```

Tạo ra những tập hoặc ánh xạ mới được sắp và có các phần tử lấy từ c hoặc từ m tương ứng.

```
TreeSet(SortedSet s)
```

```
TreeMap(SortedMap m)
```

Tạo ra những tập hoặc ánh xạ mới được sắp và có các phần tử lấy từ s hoặc từ m tương ứng.

Tóm tắt bài học

Linked list là cấu trúc gồm các node liên kết với nhau thông qua các mối liên kết. Node cuối linked list được đặt là null để đánh dấu kết thúc danh sách. Việc truy nhập trên linked list luôn phải tuần tự.

Stack là một cấu trúc theo kiểu LIFO (Last In First Out), phần tử vào sau cùng sẽ được lấy ra trước. Các thao tác cơ bản trên Stack là Push() và Pop().

Queue (Hàng đợi) là cấu trúc theo kiểu FIFO (First In First Out), phần tử vào trước sẽ được lấy ra trước. Các thao tác cơ bản trên Stack là Queue() và EnQueue().

Tập hợp Set là cấu trúc dữ liệu, trong đó không có sự lặp lại và không có sự sắp xếp của các phần tử. Giao diện Set không định nghĩa thêm các hàm mới mà chỉ giới hạn lại các hàm của Collection để không cho phép các phần tử của nó được lặp lại.

Cấu trúc List là dạng tập hợp các phần tử được sắp theo thứ tự (còn được gọi là dãy tuần tự) và trong đó cho phép lặp (hai phần tử giống nhau).

Vector và ArrayList là hai lớp kiểu mảng động (kích thước thay đổi được).

Map định nghĩa các ánh xạ từ các khoá (keys) vào các giá trị. Mỗi khoá được ánh xạ sang nhiều nhất một giá trị, được gọi là ánh xạ đơn.

Các bạn cũng đã được làm quen với các lớp khác hỗ trợ Collection như HashMap, HashTable, SortedSet, SortedMap, TreeSet và TreeMap.

Bài tập

Sử dụng Collection Framework viết chương trình quản lý sinh viên gồm những chức năng sau:

1. Tạo mới sinh viên
2. Cập nhật thông tin sinh viên
3. Xóa sinh viên
4. Tìm kiếm sinh viên
5. Hiển thị danh sách sinh viên
6. Lưu ra file
7. Đọc từ file
8. Thoát

Mời chọn:

Thông tin sinh viên bao gồm Name, Age, Mark.

CHƯƠNG V: LUỒNG I/O (I/O Streams)

Mục tiêu

Kết thúc chương, bạn có thể :

- Đề cập đến các khái niệm về luồng
- Mô tả các lớp InputStream và OutputStream
- Mô tả I/O mảng Byte
- Thực hiện các tác vụ đệm I/O và lọc
- Dùng lớp RandomAccessFile.
- Mô tả các tác vụ chuỗi I/O và ký tự
- Dùng lớp PrintWriter

1. Giới thiệu

Lớp ‘java.lang.System’ định nghĩa các luồng nhập và xuất chuẩn chúng là các lớp chính của các luồng byte mà java cung cấp. Chúng ta cũng đã sử dụng các luồng xuất để xuất dữ liệu và hiển thị kết quả trên màn hình. Luồng I/O bao gồm các luồng chuẩn và luồng byte, luồng ký tự:

- Lớp System.out: Luồng xuất chuẩn dùng để hiển thị kết quả trên màn hình.
- Lớp System.in: Luồng nhập chuẩn thường đến từ bàn phím và được dùng để đọc các ký tự dữ liệu.
- Lớp System.err: Đây là luồng lỗi chuẩn.

Các lớp ‘InputStream’ và ‘OutputStream’ cung cấp nhiều khả năng I/O khác nhau. Cả hai lớp này có các lớp con để thực hiện I/O thông qua các vùng đệm bộ nhớ, các tập tin và ống dẫn. Các lớp con của lớp InputStream thực hiện đầu vào, trong khi các lớp con của lớp OutputStream thực hiện kết xuất.

2. Luồng (Streams)

2.1. Khái niệm luồng

Tất cả những hoạt động nhập/xuất dữ liệu (nhập dữ liệu từ bàn phím, lấy dữ liệu từ mạng về, ghi dữ liệu ra đĩa, xuất dữ liệu ra màn hình, máy in, ...) đều được quy về một khái niệm gọi là luồng (stream). Luồng là nơi có thể “sản xuất” và “tiêu thụ” thông tin. Luồng thường được hệ thống xuất nhập trong java gắn kết với một thiết bị vật lý. Tất cả các luồng đều có chung một nguyên tắc hoạt động ngay cả khi chúng được gắn kết với các thiết bị vật lý khác nhau. Vì vậy cùng một lớp, phương thức xuất nhập có thể dùng chung cho các thiết bị vật lý khác nhau. Chẳng hạn cùng một phương

thức có thể dùng để ghi dữ liệu ra console, đồng thời cũng có thể dùng để ghi dữ liệu xuống một file trên đĩa. Java hiện thực luồng bằng tập hợp các lớp phân cấp trong gói java.io.

Java định nghĩa hai kiểu luồng: byte và ký tự (phiên bản gốc chỉ định nghĩa kiểu luồng byte, và sau đó luồng ký tự được thêm vào trong các phiên bản về sau).

- Luồng byte (hay luồng dựa trên byte) hỗ trợ việc xuất nhập dữ liệu trên byte, thường được dùng khi đọc ghi dữ liệu nhị phân.
- Luồng ký tự được thiết kế hỗ trợ việc xuất nhập dữ liệu kiểu ký tự (Unicode). Trong một vài trường hợp luồng ký tự sử dụng hiệu quả hơn luồng byte, nhưng ở mức hệ thống thì tất cả những xuất nhập đều phải quy về byte. Luồng ký tự hỗ trợ hiệu quả chỉ đối với việc quản lý, xử lý các ký tự.

2.2. Luồng byte (Byte Streams)

Các luồng byte được định nghĩa dùng hai lớp phân cấp. Mức trên cùng là hai lớp trừu tượng InputStream và OutputStream. InputStream định nghĩa những đặc điểm chung cho những luồng nhập byte. OutputStream mô tả cách xử lý của các luồng xuất byte.

Các lớp con dẫn xuất từ hai lớp InputStream và OutputStream sẽ hỗ trợ chi tiết tương ứng với việc đọc ghi dữ liệu trên những thiết bị khác nhau. Đừng choáng ngợp với hàng loạt rất nhiều các lớp khác nhau. Đừng quá lo lắng, mỗi khi bạn nắm vững, sử dụng thành thạo một luồng byte nào đó thì bạn dễ dàng làm việc với những luồng còn lại.

Lớp luồng byte	Ý nghĩa
BufferedInputStream	Buffered input stream
BufferedOutputStream	Buffered output stream
ByteArrayInputStream	Input stream đọc dữ liệu từ một mảng byte
ByteArrayOutputStream	Output stream ghi dữ liệu đến một mảng byte
DataInputStream	Luồng nhập có những phương thức đọc những kiểu dữ liệu chuẩn trong java
DataOutputStream	Luồng xuất có những phương thức ghi những kiểu dữ liệu chuẩn trong java
FileInputStream	Luồng nhập cho phép đọc dữ liệu từ file
FileOutputStream	Luồng xuất cho phép ghi dữ liệu xuống file
FilterInputStream	Hiện thực lớp trừu tượng InputStream
FilterOutputStream	Hiện thực lớp trừu tượng OutputStream
InputStream	Lớp trừu tượng, là lớp cha của tất cả các lớp luồng nhập kiểu Byte
OutputStream	Lớp trừu tượng, là lớp cha của tất cả các lớp xuất nhập kiểu Byte
PipedInputStream	Luồng nhập byte kiểu ống (piped) thường phải được gắn với

	một luồng xuất kiểu ống.
PipedOutputStream	Luồng nhập byte kiểu ống (piped) thường phải được gắn với một luồng nhập kiểu ống để tạo nên một kết nối trao đổi dữ liệu kiểu ống.
PrintStream	Luồng xuất có chứa phương thức print() và println()
PushbackInputStream	Là một luồng nhập kiểu Byte mà hỗ trợ thao tác trả lại (push back) và phục hồi thao tác đọc một byte (unread)
RandomAccessFile	Hỗ trợ các thao tác đọc, ghi đối với file truy cập ngẫu nhiên.
SequenceInputStream	Là một luồng nhập được tạo nên bằng cách nối kết logic các luồng nhập khác.

2.3. Luồng ký tự (Character Streams)

Các luồng ký tự được định nghĩa dùng hai lớp phân cấp. Mức trên cùng là hai lớp trừu tượng Reader và Writer. Lớp Reader dùng cho việc nhập dữ liệu của luồng, lớp Writer dùng cho việc xuất dữ liệu của luồng. Những lớp dẫn xuất từ Reader và Writer thao tác trên các luồng ký tự Unicode.

Lớp luồng byte	Ý nghĩa
BufferedReader	Luồng nhập ký tự đọc dữ liệu vào một vùng đệm.
BufferedWriter	Luồng xuất ký tự ghi dữ liệu tới một vùng đệm.
CharArrayReader	Luồng nhập đọc dữ liệu từ một mảng ký tự
CharArrayWriter	Luồng xuất ghi dữ liệu tới một mảng ký tự
FileReader	Luồng nhập ký tự đọc dữ liệu từ file
FileWriter	Luồng xuất ký tự ghi dữ liệu đến file
FilterReader	Lớp đọc dữ liệu trung gian (lớp trừu tượng)
FilterWriter	Lớp xuất trung gian trừu tượng
InputStreamReader	Luồng nhập chuyển bytes thành các ký tự
LineNumberReader	Luồng nhập đếm dòng
OutputStreamWriter	Luồng xuất chuyển những ký tự thành các bytes
PipedReader	Luồng đọc dữ liệu bằng cơ chế đường ống
PipedWriter	Luồng ghi dữ liệu bằng cơ chế đường ống
PrintWriter	Luồng ghi văn bản ra thiết bị xuất (chứa phương thức print() và println())
PushbackReader	Luồng nhập cho phép đọc và khôi phục lại dữ liệu
Reader	Lớp nhập dữ liệu trừu tượng
StringReader	Luồng nhập đọc dữ liệu từ chuỗi
StringWriter	Luồng xuất ghi dữ liệu ra chuỗi
Writer	Lớp ghi dữ liệu trừu tượng

2.4. Những luồng được định nghĩa trước (The Predefined Streams)

Tất cả các chương trình viết bằng java luôn tự động import gói java.lang. Gói này có định nghĩa lớp System, bao gồm một số đặc điểm của môi trường run-time, nó có ba biến luồng được định nghĩa trước là in, out và err, các biến này là các fields được khai báo static trong lớp System.

System.out: luồng xuất chuẩn, mặc định là console. System.out là một đối tượng kiểu PrintStream.

System.in: luồng nhập chuẩn, mặc định là bàn phím. System.in là một đối tượng kiểu InputStream.

System.err: luồng lỗi chuẩn, mặc định cũng là console. System.err cũng là một đối tượng kiểu PrintStream giống System.out.

3. Sử dụng luồng Byte

3.1. Lớp InputStream

Lớp InputStream là một lớp trừu tượng. Nó định nghĩa cách nhận dữ liệu. Điểm quan trọng không nằm ở chỗ dữ liệu đến từ đâu, mà là nó có thể truy cập. Lớp InputStream cung cấp một số phương pháp để đọc và dùng các luồng dữ liệu để làm đầu vào. Các phương thức này giúp ta tạo, đọc và xử lý các luồng đầu vào. Các phương thức được hiện trong bảng

Tên phương thức	Mô tả
read()	Đọc các byte dữ liệu từ một luồng. Nếu như không dữ liệu nào là hợp lệ, nó khoá phương thức. Khi một phương thức được khoá, các dòng thực hiện được chờ cho đến khi dữ liệu hợp lệ.
read (byte [])	trả về byte được ‘đọc’ hay ‘-1’, nếu như kết thúc của một luồng đã đến. nó kích hoạt IOException nếu lỗi xảy ra.
read (byte [], int, int)	Nó cũng đọc vào mảng byte. Nó trả về số byte thực sự được đọc. Khi kết thúc của một luồng đã đến. nó kích hoạt IOException nếu lỗi xảy ra.
available()	Phương pháp này trả về số lượng byte có thể được đọc mà không bị phong tỏa. Nó trả về số byte hợp lệ. Nó không phải là phương thức hợp lệ đáng tin cậy để thực hiện tiến trình xử lý đầu vào.
close()	Phương thức này đóng luồng. Nó dùng để phóng thích mọi tài nguyên kết hợp với luồng. Luôn luôn đóng luồng để chắc chắn rằng luồng xử lý được kết thúc. Nó kích hoạt IOException nếu lỗi xảy ra.
mark()	Đánh dấu vị trí hiện tại của luồng.

markSupporte()	trả về giá trị boolean nêu rõ luồng có hỗ trợ các khả năng mark và reset hay không. Nó trả về đúng nếu luồng hỗ trợ nó bằng không là sai.
reset()	Phương thức này định vị lại luồng theo vị trí được đánh dấu chót. Nó kích hoạt IOException nếu lỗi xảy ra.
skip()	Phương thức này bỏ qua 'n' byte đầu vào. 'n' chỉ định số byte được bỏ qua. Nó kích hoạt IOException nếu lỗi xảy ra. Phương thức này sử dụng để di chuyển tới vị trí đặc biệt bên trong luồng đầu vào.

3.2. Lớp OutputStream

Lớp OutputStream cũng là lớp trừu tượng. Nó định nghĩa cách ghi các kết xuất đến luồng. Nó cung cấp tập các phương thức trợ giúp tạo ra, ghi và xử lý kết xuất các luồng. Các phương thức bao gồm:

Tên phương thức	Mô tả
write(int)	Phương thức này ghi một byte
write(byte[])	Phương thức này phong toả cho đến khi một byte được ghi. luồng chờ cho đến khi tác vụ ghi hoàn tất. Nó kích hoạt IOException nếu lỗi xảy ra.
write(byte[],int,int)	Phương thức này cũng ghi mảng các byte. Lớp OutputStream định nghĩa ba dạng quá tải của phương thức này để cho phép phương thức write() ghi một byte riêng lẻ, mảng các byte, hay một đoạn của một mảng.
flush()	Phương thức này xả sạch luồng đệm dữ liệu được ghi ra luồng kết xuất. Nó kích hoạt IOException nếu lỗi xảy ra.
close()	Phương thức đóng luồng. Nó được dùng để giải phóng mọi tài nguyên kết hợp với luồng. Nó kích hoạt IOException nếu lỗi xảy ra.

4. Nhập và xuất mảng byte

Các lớp 'ByteArrayInputStream' và 'ByteArrayOutputStream' sử dụng các đệm bộ nhớ. Không cần thiết phải dùng chúng với nhau.

Lớp ByteArrayInputStream:

Lớp này tạo luồng đầu vào từ bộ nhớ đệm. Nó là mảng các byte. Lớp này không hỗ trợ các phương thức mới. Ngược lại nó chạy đề các phương thức của lớp InputStream như 'read()', 'skip()', 'available()' và 'reset()'.

Lớp ByteArrayOutputStream:

Lớp này tạo ra luồng kết suất trên một mảng các byte. Nó cũng cung cấp các khả năng bổ sung để mảng kết suất tăng trưởng nhằm mục đích chứa chỗ cho mảng được ghi.

Lớp này cũng cung cấp các phương thức ‘toArray()’ và ‘toString()’. Chúng được dùng để chuyển đổi luồng thành một mảng byte hay đối tượng chuỗi.

Lớp `ByteArrayOutputStream` cũng cung cấp hai phương thức thiết lập. Một chấp nhận một đối số số nguyên dùng để ấn định mảng byte kết xuất theo một kích cỡ ban đầu. và thứ hai không chấp nhận đối số nào, và thiết lập đệm kết xuất với kích thước mặc định. lớp này cung cấp vài phương thức bổ sung, không được khai báo trong `OutputStream`:

- **reset()**: Thiết lập lại kết xuất vùng đệm nhằm cho phép tiến trình ghi khởi động lại tại đầu vùng đệm.
- **size()**: Trả về số byte hiện tại đã được ghi tới vùng đệm.
- **writeto()**: Ghi nội dung của vùng đệm kết xuất ra luồng xuất đã chỉ định. Để thực hiện, nó chấp nhận một đối tượng của lớp `OutputStream` làm đối số.

Chương trình sử dụng lớp ‘`ByteArrayInputStream`’ và ‘`ByteArrayOutputStream`’ để nhập và xuất:

```
import java.lang.System;
import java.io.*;
public class byteexam
{
    public static void main(String args[]) throws IOException
    {
        ByteArrayOutputStream os = new ByteArrayOutputStream();
        String s = "Welcome to Byte Array Input Outputclasses";
        for(int i=0; i<s.length(); i++)
            os.write(s.charAt(i));
        System.out.println("Output Stream is: " + os);
        System.out.println("Size of output stream is: " + os.size());
        ByteArrayInputStream in;
        in = new ByteArrayInputStream(os.toByteArray());
        int ib = in.available();

        System.out.println("Input Stream has : " + ib + "available bytes");
        byte ibuf[] = new byte[ib];
        int byrd = in.read(ibuf, 0, ib);

        System.out.println("Number of Bytes read are : " + byrd);
        System.out.println("They are: " + new String(ibuf));
    }
}
```

5. Nhập và xuất tập tin

Java hỗ trợ các tác vụ nhập và xuất tập tin với sự trợ giúp các lớp sau đây:

- **File**
- **FileDescriptor**
- **FileInputStream**
- **FileOutputStream**

Java cũng hỗ trợ truy cập nhập và xuất ngẫu nhiên hoặc trực tiếp bằng các lớp ‘File’, ‘FileDescriptor’, và ‘RandomAccessFile’.

5.1. Lớp File

Lớp này được sử dụng để truy cập các đối tượng tập tin và thư mục. Các tập tin đặt tên theo qui ước đặt tên tập tin của hệ điều hành chủ. Các qui ước này được gói riêng bằng các hằng lớp File. Lớp này cung cấp các thiết lập các tập tin và các thư mục. Các thiết lập chấp nhận các đường dẫn tập tin tuyệt đối lẫn tương đối cùng các tập tin và thư mục. Tất cả các tác vụ thư mục và tập tin chung được thực hiện thông qua các phương thức truy cập của lớp File.

Các phương thức:

- Cho phép bạn tạo, xoá, đổi tên các file.
- Cung cấp khả năng truy cập tên đường dẫn tập tin.
- Xác định đối tượng có phải tập tin hay thư mục không.
- Kiểm tra sự cho phép truy cập đọc và ghi.

Giống như các phương thức truy cập, các phương thức thư mục cũng cho phép tạo, xoá, đặt tên lại và liệt kê các thư mục. Các phương pháp này cho phép các cây thư mục đang chéo bằng cách cung cấp khả năng truy cập các thư mục cha và thư mục anh em.

Lớp File không phục vụ cho việc nhập/xuất dữ liệu trên luồng. Lớp File thường được dùng để biết được các thông tin chi tiết về tập tin cũng như thư mục (tên, ngày giờ tạo, kích thước, ...)

Các Constructor:

Tạo đối tượng File từ đường dẫn tuyệt đối

public File(String pathname)

```
File f = new File("C:\\Java\\vd1.java");
```

Tạo đối tượng File từ tên đường dẫn và tên tập tin tách biệt

public File(String parent, String child)

```
File f = new File("C:\\Java", "vd1.java");
```

Tạo đối tượng File từ một đối tượng File khác

public File(File parent, String child)

```
File dir = new File ("C:\\Java");
File f = new File(dir, "vd1.java");
```

Một số phương thức thường gặp của lớp File (chi tiết về các phương thức đọc thêm trong tài liệu J2SE API Specification)

Tên phương thức	Mô tả
public String getName()	Lấy tên của đối tượng File
public String getPath()	Lấy đường dẫn của tập tin
public boolean isDirectory()	Kiểm tra xem tập tin có phải là thư mục không?
public boolean isFile()	Kiểm tra xem tập tin có phải là một file không?
...	
public String[] list()	Lấy danh sách tên các tập tin và thư mục con của đối tượng File đang xét và trả về trong một mảng.

Chương trình

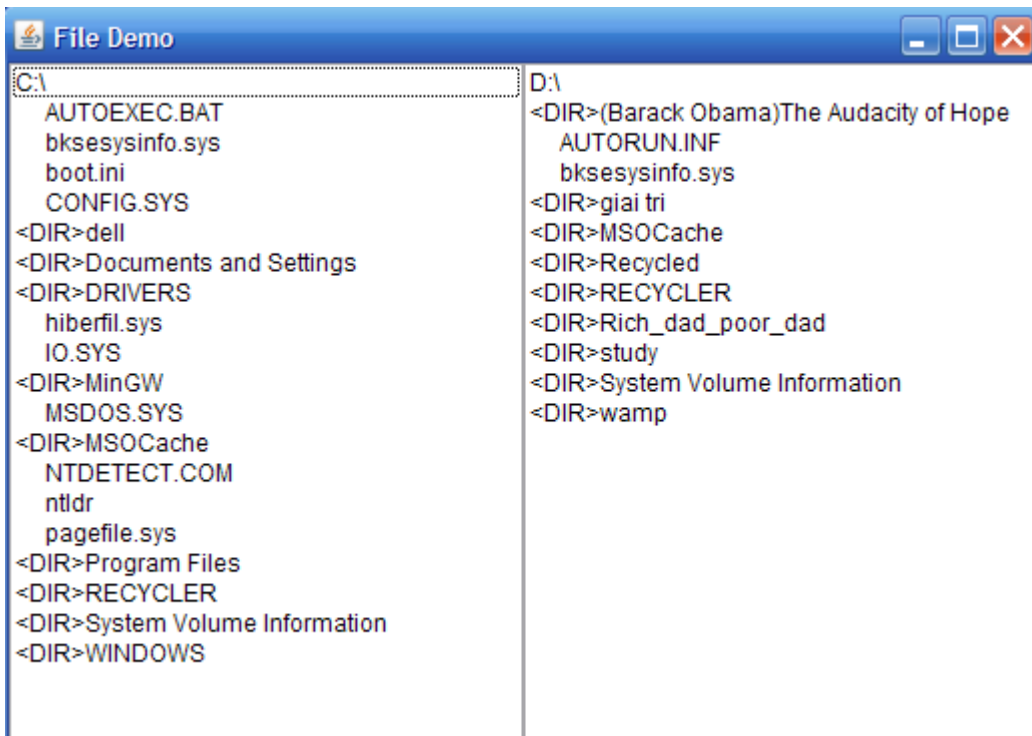
```
import java.awt.*;
import java.io.*;
public class FileDemo
{
    public static void main(String args[])
    {
        Frame fr = new Frame ("File Demo");
        fr.setBounds(10, 10, 300, 200);
        fr.setLayout(new BorderLayout());
        Panel p = new Panel(new GridLayout(1,2));
        List list_C = new List();
        list_C.add("C:\\");
        File driver_C = new File ("C:\\");
        String[] dirs_C = driver_C.list();
        for (int i=0;i<dirs_C.length;i++)
        {
            File f = new File ("C:\\\" + dirs_C[i]);
            if (f.isDirectory())
                list_C.add("<DIR>" + dirs_C[i]);
            else
                list_C.add("  " + dirs_C[i]);
        }
    }
}
```

```

    }
    List list_D = new List();
    list_D.add("D:\\");
    File driver_D = new File ("D:\\");
    String[] dirs_D = driver_D.list();
    for (int i=0;i<dirs_D.length;i++)
    {
        File f = new File ("D:\\\" + dirs_D[i]);
        if (f.isDirectory())
            list_D.add("<DIR>" + dirs_D[i]);
        else
            list_D.add("  " + dirs_D[i]);
    }
    p.add(list_C);
    p.add(list_D);
    fr.add(p, BorderLayout.CENTER);
    fr.setVisible(true);
}
}

```

Kết quả thực thi chương trình



5.2. Lớp FileDescriptor

Lớp này cung cấp khả năng truy cập các mô tả tập tin mà hệ điều hành duy trì khi các tập tin và thư mục đang được truy cập. Lớp này không cung cấp tầm nhìn đối với thông tin cụ thể do hệ điều hành duy trì. Nó cung cấp chỉ một phương thức có tên 'valid()', giúp xác định một đối tượng mô tả tập tin hiện có hợp lệ hay không.

5.3. Lớp FileInputStream

Lớp này cho phép đọc đầu vào từ một tập tin dưới dạng một luồng. Các đối tượng của lớp này được tạo ra nhờ dùng một tập tin String, File, hoặc một đối tượng FileDescriptor làm một đối số. Lớp này chồng lên các phương thức của lớp InputStream. Nó cũng cung cấp các phương thức 'finalize()' và 'getFD()'.

Phương thức 'finalize()' được dùng để đóng luồng khi đang được bộ gom rác Java xử lý. Phương thức 'getFD()' trả về đối tượng FileDescriptor biểu thị sự kết nối đến tập tin thực tế trong hệ tập tin đang được 'FileInputStream' sử dụng.

5.4. Lớp FileOutputStream

Lớp này cho phép ghi kết xuất ra một luồng tập tin. Các đối tượng của lớp này cũng tạo ra sử dụng các đối tượng chuỗi tên tập tin, tập tin, FileDescriptor làm tham số. Lớp này chồng lên phương thức của lớp OutputStream và cung cấp phương thức 'finalize()' và 'getFD()'.

```
import java.io.FileOutputStream;
import java.io.FileInputStream;
import java.io.File;
import java.io.IOException;
public class fileioexam
{
    public static void main(String args[ ]) throws IOException
    {
        // creating an output file abc.txt
        FileOutputStream os = new FileOutputStream("abc.txt");
        String s = "Welcome to File Input Output Stream " ;
        for(int i = 0; i< s.length( ); ++i) .
            os. write(s.charAt(i));
        os.close();
        FileInputStream is = new FileInputStream("abc.txt");
        int ibyts = is.available( );
        System.out.println("Input Stream has " + ibyts + " available bytes");
        byte ibuf[ ] = new byte[ibyts];
        int byrd = is.read(ibuf, 0, ibyts);
    }
}
```

```
        System.out.println("Number of Bytes read are: " + byrd);
        System.out.println("They are: " + new String(ibuf));
        is.close();
        File fl = new File("abc.txt");
        fl.delete();
    }
}
```

6. Nhập xuất đã lọc

Một ‘Filter’ là một kiểu luồng sửa đổi cách điều quản một luồng hiện tồn tại. Các lớp, các luồng nhập xuất đã lọc của java sẽ giúp ta lọc I/O theo một số cách. Về cơ bản, các bộ lọc này dùng để thích ứng các luồng theo các nhu cầu của chương trình cụ thể.

Bộ lọc nằm giữa một luồng nhập và một luồng xuất. Nó thực hiện xử lý một tiến trình đặc biệt trên các byte được truyền từ đầu vào đến kết xuất. Các bộ lọc có thể phối hợp thực hiện dãy tuần tự các tùy chọn lọc ở đó mọi bộ lọc tác động như kết xuất của một bộ lọc khác.

6.1. Lớp FilterInputStream

Đây là lớp trừu tượng. Nó là cha của tất cả các lớp luồng nhập đã lọc. Lớp này cung cấp khả năng tạo ra một luồng từ luồng khác. Một luồng có thể được đọc và cung cấp dưới dạng kết xuất cho luồng khác. Biến ‘in’ được sử dụng để làm điều này. Biến này được dùng để duy trì một đối tượng tách biệt của lớp InputStream. Lớp FilterInputStream được thiết kế sao cho có thể tạo nhiều bộ lọc kết xích [chained filters]. Để thực hiện điều này chúng ta dùng vài tầng lồng ghép. đến lượt mỗi lớp sẽ truy cập kết xuất của lớp trước đó với sự trợ giúp của biến ‘in’.

6.2. Lớp FilterOutputStream

Lớp này là một dạng hỗ trợ cho lớp FilterInputStream. Nó là lớp cha của tất cả các lớp luồng xuất đã lọc. Lớp này tương tự như lớp FilterInputStream ở chỗ nó duy trì đối tượng của lớp OutputStream làm một biến ‘out’. Dữ liệu ghi vào lớp này có thể sửa đổi theo nhu cầu để thực hiện tác vụ lọc và sau đó được chuyển gửi tới đối tượng OutputStream.

7. I/O có lập vùng đệm

Vùng đệm là kho lưu trữ dữ liệu. Chúng ta có thể lấy dữ liệu từ vùng đệm thay vì quay trở lại nguồn ban đầu của dữ liệu. Java sử dụng cơ chế nhập/xuất có lập vùng đệm để tạm thời lập cache dữ liệu được đọc hoặc ghi vào/ra một luồng. Nó giúp các chương trình đọc/ghi các lượng dữ liệu nhỏ mà không tác động ngược lên khả năng thực hiện của hệ thống.

Trong khi thực hiện nhập có lập vùng đệm, số lượng byte lớn được đọc tại thời điểm này, và lưu trữ trong một vùng đệm nhập. khi chương trình đọc luồng nhập, các byte dữ liệu được đọc từ vùng đệm nhập.

Tiến trình lập vùng đệm kết xuất cũng thực hiện tương tự. khi dữ liệu được một chương trình ghi ra một luồng, dữ liệu kết xuất được lưu trữ trong một vùng đệm xuất. Dữ liệu được lưu trữ đến khi vùng đệm trở nên đầy hoặc các luồng kết xuất được xả trống. Cuối cùng kết xuất có lập vùng đệm được chuyển gửi đến đích của luồng xuất.

Các bộ lọc hoạt động trên vùng đệm. Vùng đệm được phân bố nằm giữa chương trình và đích của luồng có lập vùng đệm.

7.1. Lớp `BufferedInputStream`

Lớp này tự động tạo ra và chứa đựng vùng đệm để hỗ trợ vùng đệm nhập. Nhờ đó chương trình có thể đọc dữ liệu từng luồng theo byte một mà không ảnh hưởng đến khả năng thực hiện của hệ thống. Bởi lớp '`BufferedInputStream`' là một bộ lọc, nên có thể áp dụng nó cho một số đối tượng nhất định của lớp `InputStream` và cũng có thể phối hợp với các tập tin đầu vào khác.

Lớp này sử dụng vài biến để thực hiện các cơ chế lập vùng đệm đầu vào. Các biến này được khai báo là `protected` và do đó chương trình không thể truy cập trực tiếp. Lớp này định nghĩa hai phương thức thiết lập. Một cho phép chỉ định kích cỡ của vùng đệm nhập trong khi đó phương thức thiết lập kia thì không. Nhưng cả hai phương thức thiết lập đều tiếp nhận đối tượng của lớp `InputStream` và `OutputStream` làm đối số. lớp này chồng lên các phương thức truy cập mà `InputStream` cung cấp và không làm nảy sinh bất kì phương thức mới nào.

Lớp `BufferedInputStream`. Lớp này cũng định nghĩa hai phương thức thiết lập. nó cho phép chỉ định kích cỡ của vùng đệm xuất trong một phương thức thiết lập cũng như cung cấp một kích cỡ vùng đệm ngầm định. Nó chồng lên tất cả các phương thức của `OutputStream` và không làm nảy sinh bất kì phương thức nào.

Chương trình dưới đây mô tả cách dùng các luồng nhập/xuất có lập vùng đệm:

```
import javaJang. * ;
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.FileInputStream;
import java.io.SequenceInputStream;
import java.io.IOException;
publicI class buff exam
{
    public static void main(String args[ ]) throws IOException
```



```
{
    // defining sequence input stream
    SequenceInputStream Seq3;
    FileInputStream Fis1 ;
    Fis1 = new FileInputStream("byteexam.java");
    FileInputStream Fis2;
    Fis2= new FileInputStream("fileioexam.java");
    Seq3 = new SequenceInputStream(Fis1, Fis2);
    // create buffered input and output streams
    BufferedInputStream inst;
    inst := new BufferedInputStream(Seq3);
    BufferedOutputStream oust;
    oust= new BufferedOutputStream(System.out);
    inst.skip(1000);
    boolean eof = false;
    int bytcnt = 0;
    while(!eof)
    {
        int num = inst.read();
        if(num == -1)
        {
            eof =true;
        }
        else
        {
            oust.write((char) num);
            ++ bytcnt;
        }
    }
    String bytrd := String.valueOf(bytcnt);
    bytrd += "bytes were read";
    oust.write(bytrd.getBytes(), 0, bytrd.length());

    // close all streams.
    inst.close();
    oust.close();
    Fis1.close();
}
```

```
        Fis2.close();  
    }  
}
```

8. Lớp *Reader* và *Writer*

Đây là các lớp trừu tượng. Chúng nằm tại đỉnh của hệ phân cách lớp, hỗ trợ việc đọc và ghi các luồng ký tự unicode.

8.1. Lớp *Reader*

Lớp này hỗ trợ các phương thức:

- **read()**
- **reset()**
- **skip()**
- **mark()**
- **markSupported()**
- **close()**

Lớp này cũng hỗ trợ phương thức gọi 'ready()'. Phương thức này trả về giá trị kiểu boolean nếu rõ tác vụ đọc kế tiếp có tiếp tục mà không phong tỏa hay không.

8.2. Lớp *Writer*

Lớp này hỗ trợ các phương thức:

- **write()**
- **flush()**
- **close()**

9. Nhập/ xuất chuỗi và xâu ký tự

Các lớp 'CharArrayReader' và 'CharArrayWriter' cũng tương tự như các lớp `ByteArrayInputStream` và `ByteArrayOutputStream` ở chỗ chúng hỗ trợ nhập/xuất từ các vùng đệm nhớ. Các lớp `CharArrayReader` và `CharArrayWriter` hỗ trợ nhập/ xuất ký tự 8 bit.

`CharArrayReader` không hỗ trợ bổ sung các phương pháp sau đây vào các phương thức của lớp `Reader` cung cấp. Lớp `CharArrayWriter` bổ sung các phương thức sau đây vào các phương thức của lớp `Writer`.

- **reset()**: thiết lập lại vùng đệm
- **size()**: trả về kích cỡ hiện hành của vùng đệm
- **toCharArray()**: Trả về bản sao mảng ký tự của vùng đệm xuất
- **toString()**: Chuyển đổi vùng đệm xuất thành một đối tượng `String`
- **writeTo()**: Ghi vùng đệm ra một luồng xuất khác.

Lớp `StringReader` trợ giúp luồng nhập ký tự từ một chuỗi. Nó không bổ sung phương thức nào vào lớp `Reader`.

Lớp `StringWriter` trợ giúp ghi luồng kết xuất ký tự ra một đối tượng `StringBuffer`. Lớp này bổ sung hai phương thức có tên là `'getBuffer()'` và `'toString()'`. Phương thức `'getBuffer()'` trả về đối tượng `StringBuffer` tương ứng với vùng đệm xuất, trong khi đó phương thức `toString()` trả về một bản sao chuỗi của vùng đệm xuất.

Chương trình dưới đây thực hiện các tác vụ nhập/xuất mảng ký tự:

```
import java.io.CharArrayReader;
import java.io.CharArrayWriter;
import java.io.IOException;
public class test1 {
    public static void main(String args[]) throws IOException {
        CharArrayWriter ost = new CharArrayWriter();
        String s = "Welcome to Design Global";
        for (int i = 0; i < s.length(); ++i)
            ost.write(s.charAt(i));
        System.out.println("Output Stream is: " + ost);
        System.out.println("Size is: " + ost.size());
        CharArrayReader inst;
        inst = new CharArrayReader(ost.toCharArray());
        int a = 0;
        StringBuffer sbI = new StringBuffer(" ");
        while ((a = inst.read()) != -1) {
            sbI.append((char) a);
        }
        s = sbI.toString();
        System.out.println(s.length() + "characters were read");
        System.out.println("They are:" + s);
    }
}
```

Kết quả hiển thị như sau:

```
Output Stream is: Welcome to Design Global
Size is: 24
25characters were read
They are: Welcome to Design Global
```

Chương trình Mô tả tiến trình nhập/xuất chuỗi.

```
import java.io.IOException;
import java.io.StringReader;
import java.io.StringWriter;
public class IOSample {
    public static void main(String args[]) throws IOException {
        StringWriter ost = new StringWriter();
        String s = "Welcome to Design Global";
        for (int i = 0; i < s.length(); ++i) {
            ost.write(s.charAt(i));
        }
        System.out.println("Output Stream is: " + ost);
        StringReader inst;
        inst = new StringReader(ost.toString());
        int a = 0;
        StringBuffer sb1 = new StringBuffer(" ");
        while ((a = inst.read()) != -1)
            sb1.append((char) a);
        s = sb1.toString();
        System.out.println("No of characters read: " + s.length());
        System.out.println("They are: " + s);
    }
}
```

Kết quả hiển thị như sau:

```
Output Stream is: Welcome to Design Global
No of characters read: 25
They are: Welcome to Design Global
```

10. Lớp **PrintWriter**

Lớp ‘PrintStream’ thực hiện việc kết xuất dữ liệu. Lớp này có các phương thức bổ sung, trợ giúp cho việc in ấn dữ liệu cơ bản.

Lớp **PrintWriter** là một thay thế của lớp **PrintStream**. Nó thực tế cải thiện lớp **PrintStream** bằng cách dùng dấu tách dòng phụ thuộc nền tảng để in các dòng thay vì ký tự ‘\n’. Lớp này cũng cấp hỗ trợ các ký tự Unicode so với **PrintStream**. Phương thức ‘checkError()’ được sử dụng kiểm tra kết xuất được xả sạch và và được kiểm tra các lỗi. Phương thức **setError()** được sử dụng để thiết lập lỗi điều kiện. Lớp **PrintWriter** cung cấp việc hỗ trợ in ấn các kiểu dữ liệu nguyên thủy, các mảng ký tự, các chuỗi và các đối tượng.

11. Giao diện DataInput

Giao diện DataInput được sử dụng để đọc các byte từ luồng nhị phân và xây dựng lại các kiểu dữ liệu dạng nguyên thủy trong Java. DataInput cũng cho phép chúng ta chuyển đổi dữ liệu từ định dạng sửa đổi UTF-8 tới dạng chuỗi. Chuẩn UTF cho định dạng chuyển đổi Unicode. Nó là kiểu định dạng đặt biệt giải mã các giá trị Unicode 16 bit. UTF lạc quan ở mức thấp giả lập trong hầu hết các trường hợp, mức cao 8 bit Unicode sẽ là 0. Giao diện DataInput được định nghĩa là số các phương thức, các phương thức bao gồm việc đọc các kiểu dữ liệu nguyên thủy trong java.

Bảng tóm lược vài phương thức. Tất cả các phương thức được kính hoạt IOException trong trường hợp lỗi:

Tên phương thức	Mô tả
boolean readBoolean()	Đọc một byte nhập, và trả về đúng nếu byte đó không phải là 0, và sai nếu byte đó là 0.
byte readByte()	Đọc một byte
char readChar()	Đọc và trả về một giá trị ký tự
short readShort()	Đọc 2 byte và trả về giá trị short
long readLong()	Đọc 8 byte và trả về giá trị long.
float readFloat()	đọc 4 byte và trả về giá trị float
int readInt()	Đọc 4 byte và trả về giá trị int
double readDouble()	Đọc 8 byte và trả về giá trị double
String readUTF()	Đọc một chuỗi
String readLine()	Đọc một dòng văn bản

Bảng Các phương thức của giao diện DataInput

12. Giao diện DataOutput

Giao diện DataOutput được sử dụng để xây dựng lại các kiểu dữ liệu nguyên thủy trong java vào trong dãy các byte. nó ghi các byte này lên trên luồng nhị phân.

Giao diện DataOutput cũng cho phép chúng ta chuyển đổi một chuỗi vào trong java được sửa đổi theo định dạng UTF-8 và ghi nó vào luồng. Giao diện DataOutput định nghĩa số phương thức được tóm tắt trong bảng. Tất cả các phương thức sẽ kích hoạt IOException trong trường hợp lỗi.

Tên phương thức	Mô tả
void writeBoolean(boolean b)	Ghi một giá trị Boolean vào luồng
void writeByte(int value)	Ghi giá trị 8 bit thấp
void writeChar(int value)	Ghi 2 byte giá trị kiểu ký tự vào luồng
void writeShort(int value)	Ghi 2 byte, biểu diễn lại giá trị dạng short

<code>void writeLong(long value)</code>	Ghi 8 byte, biểu diễn lại giá trị dạng long
<code>void writeFloat(float value)</code>	Ghi 4 byte, biểu diễn lại giá trị dạng float
<code>void writeInt(int value)</code>	ghi 4 byte
<code>void writeDouble(double value)</code>	Ghi 8 byte, biểu diễn lại giá trị dạng double
<code>void writeUTF(String value)</code>	Ghi một sâu dạng UTF tới luồng.

Bảng Các phương thức của giao diện `DataOutput`

13. Lớp `RandomAccessFile`

Lớp `RandomAccessFile` cung cấp khả năng thực hiện I/O theo một vị trí cụ thể bên trong một tập tin. Trong lớp này, dữ liệu có thể đọc hoặc ghi ở vị trí ngẫu nhiên bên trong một tập tin thay vì một kho lưu trữ thông tin liên tục. Hơn thế nữa lớp này có tên `RandomAccess`. Phương thức `'seek()'` hỗ trợ truy cập ngẫu nhiên. Kết quả là, biến trở tương ứng với tập tin hiện hành có thể ấn định theo vị trí bất kỳ trong tập tin.

Lớp `RandomAccessFile` thực hiện cả hai việc nhập và xuất. Do vậy, có thể thực hiện I/O bằng các kiểu dữ liệu nguyên thủy. Lớp này cũng hỗ trợ cho phép đọc hoặc ghi tập tin cơ bản, điều này cho phép đọc tập tin theo chế độ chỉ đọc hoặc đọc-ghi. Tham số `'r'` hoặc `'rw'` được gán cho lớp `RandomAccessFile` chỉ định truy cập `'chỉ đọc'` và `'đọc-ghi'`. Lớp này giới thiệu vài phương thức mới khác với phương pháp đã thừa kế từ các lớp `DataInput` và `DataOutput`.

Các phương thức bao gồm:

`seek()`

Thiết lập con trỏ tập tin tới vị trí cụ thể bên trong tập tin.

`getFilePointer()`

Trả về vị trí hiện hành của con trỏ tập tin.

`length()`

Trả về chiều dài của tập tin tính theo byte.

Chương trình dưới đây minh họa cách dùng lớp `RandomAccessFile`. Nó ghi một giá trị boolean, một int, một char, một double tới một file có tên `'abc.txt'`. Nó sử dụng phương pháp `seek()` để tìm vị trí định vị 1 bên trong tập tin. Sau đó nó đọc giá trị số nguyên, ký tự và double từ tập tin và hiển thị chúng ra màn hình.

```
import java.io.IOException;
import java.io.RandomAccessFile;
public class RandomAccessSample {
    public static void main(String args[]) throws IOException {
        RandomAccessFile rf;
        rf = new RandomAccessFile("DesignGlobal.txt", "rw");
```

```
rf.writeBoolean(true);
rf.writeInt(67868);
rf.writeChars("J");
rf.writeDouble(678.68);
    rf.seek(1);
System.out.println(rf.readInt());
System.out.println(rf.readChar());
System.out.println(rf.readDouble());
rf.seek(0);
System.out.println(rf.readBoolean());
rf.close();
}
}
```

Kết quả hiển thị như sau:

```
67868
J
678.68
true
```

Tóm tắt bài học

Một luồng là một lộ trình qua đó dữ liệu di chuyển trong một chương trình java.

Khi một luồng dữ liệu được gửi hoặc nhận. Chúng ta xem nó như đang ghi và đọc một luồng theo thứ tự nêu trên.

Luồng nhập/xuất bao gồm các lớp sau đây:

- **Lớp System.out**
- **Lớp System.in**
- **Lớp System.err**

Lớp InputStream là một lớp trừu tượng định nghĩa cách nhận dữ liệu.

Lớp OutputStream cũng là lớp trừu tượng. Nó định nghĩa ghi ra các luồng được kết xuất như thế nào.

Lớp ByteArrayInputStream tạo ra một luồng nhập từ vùng đệm bộ nhớ trong khi ByteArrayOutputStream tạo một luồng xuất trên một mảng byte.

Java hỗ trợ tác vụ nhập/xuất tập tin với sự trợ giúp của các File, FileDescriptor, FileInputStream và FileOutputStream.

Các lớp Reader và Writer là lớp trừu tượng hỗ trợ đọc và ghi các luồng ký tự Unicode.

CharArrayReader, CharArrayWriter khác với ByteArrayInputStream, ByteArrayOutputStream hỗ trợ định dạng nhập/xuất 8 bit, Trong khi ByteArrayInputStream, ByteArrayOutputStream hỗ trợ nhập/xuất 16bit.

Lớp PrintStream thực thi một kết xuất. lớp này có phương thức bổ sung, giúp ta in các kiểu dữ liệu cơ bản.

Lớp RandomAccessFile cung cấp khả năng thực hiện I/O tới vị trí cụ thể trong một tập tin.

Bài tập

Tạo lớp Student gồm có: 3 thuộc tính là name, age, mark và các phương thức getter/setter tương ứng. Sử dụng FileWriter, FileReader và BufferedReader để viết chương trình với các chức năng trên menu như sau:

“MENU

1.Lưu thông tin Sinh viên vào File
2. Đọc thông tin Sinh viên từ File
3. Thoát
Mời chọn: “

Chức năng “**Lưu thông tin Sinh viên vào File**”: sẽ yêu cầu người dùng nhập thông tin Sinh viên và lưu thông tin sinh viên vào file SV.txt. Mỗi sinh viên lưu trên một dòng, các trường thông tin sẽ ngăn nhau bởi 1 dấu TAB.

Chức năng “**Đọc thông tin Sinh viên từ File**” sẽ đọc thông tin sinh viên trong file SV.txt và lưu vào một mảng, sau đó sẽ hiển thị lên màn hình.

CHƯƠNG VI: ĐA LUỒNG (MULTITHREADING)

Mục tiêu:

Sau khi kết thúc chương này, bạn có thể:

- Định một luồng (thread)
- Mô tả đa luồng
- Tạo và quản lý luồng
- Hiểu được vòng đời của luồng
- Mô tả một luồng chạy nền (daemon thread)
- Giải thích thiết lập các luồng ưu tiên như thế nào
- Giải thích được sự cần thiết của sự đồng bộ
- Hiểu được cách áp dụng vào các từ khoá đồng bộ như thế nào
- Liệt kê những điểm yếu của sự đồng bộ
- Giải thích vai trò của các phương thức wait() (đợi), notify() (thông báo) và notifyAll().
- Mô tả một điều kiện bế tắc (deadlock condition)

1. Giới thiệu

Một luồng là một thuộc tính duy nhất của Java. Nó là đơn vị nhỏ nhất của đoạn mã có thể thi hành được mà thực hiện một công việc riêng biệt. Ngôn ngữ Java và máy ảo Java cả hai là các hệ thống được phân luồng

2. Đa luồng

Java hỗ trợ đa luồng, mà có khả năng làm việc với nhiều luồng. Một ứng dụng có thể bao hàm nhiều luồng. Mỗi luồng được đăng ký một công việc riêng biệt, mà chúng được thực thi đồng thời với các luồng khác.

Đa luồng giữ thời gian nhàn rỗi của hệ thống thành nhỏ nhất. Điều này cho phép bạn viết các chương trình có hiệu quả cao với sự tận dụng CPU là tối đa. Mỗi phần của chương trình được gọi một luồng, mỗi luồng định nghĩa một đường dẫn khác nhau của sự thực hiện. Đây là một thiết kế chuyên dùng của sự đa nhiệm.

Trong sự đa nhiệm, nhiều chương trình chạy đồng thời, mỗi chương trình có ít nhất một luồng trong nó. Một vi xử lý thực thi tất cả các chương trình. Cho dù nó có thể xuất hiện mà các chương trình đã được thực thi đồng thời, trên thực tế bộ vi xử lý nhảy qua lại giữa các tiến trình.

3. Tạo và quản lý luồng

Khi các chương trình Java được thực thi, luồng chính luôn luôn đang được thực hiện. Đây là 2 nguyên nhân quan trọng đối với luồng chính:

- Các luồng con sẽ được tạo ra từ nó.
- Nó là luồng cuối cùng kết thúc việc thực hiện. Trong chốc lát luồng chính ngừng thực thi, chương trình bị chấm dứt.

Cho dù luồng chính được tạo ra một cách tự động với chương trình thực thi, nó có thể được điều khiển thông qua một luồng đối tượng.

Các luồng có thể được tạo ra từ hai con đường:

- Trình bày lớp như là một lớp con của lớp luồng, nơi mà phương thức run() của lớp luồng cần được ghi đè. Lấy ví dụ:

```
Class Mydemo extends Thread {  
    //Class definition  
    public void run()  
    {  
        //thực thi  
    }  
}
```

- Trình bày một lớp mà lớp này thực hiện lớp Runnable. Rồi thì định nghĩa phương thức run().

```
Class Mydemo implements Runnable {  
    //Class definition  
    public void run() {  
        //thực thi  
    }  
}
```

Chương trình sau sẽ chỉ ra sự điều khiển luồng chính như thế nào

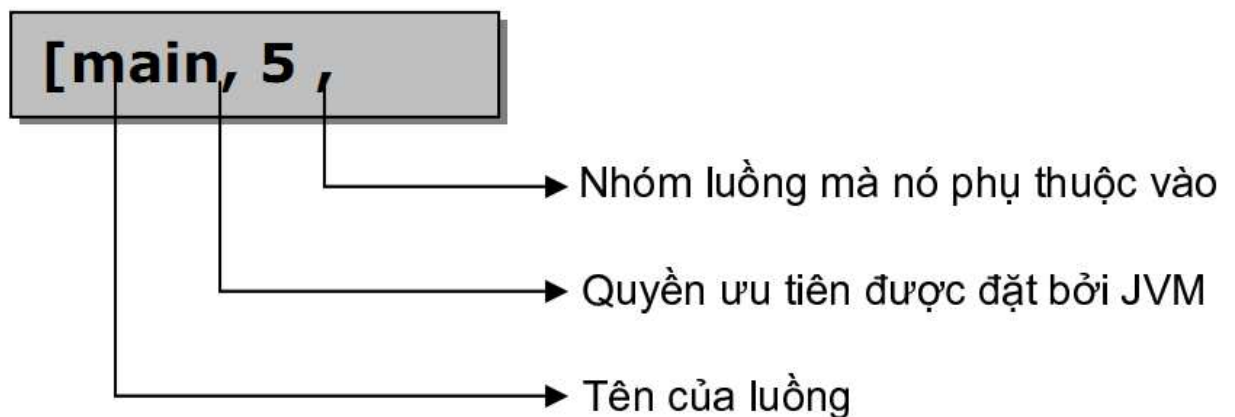
```
public class ThreadSample1 extends Thread {  
  
    public static void main(String[] args) {  
        Thread mainThread = Thread.currentThread();  
        System.out.println("Thread is: " + mainThread);  
  
        mainThread.setName("Design Global thread");  
    }  
}
```

```
System.out.println("Thread is: " + mainThread);
ThreadSample1 objThread = new ThreadSample1();
objThread.start();
while(true) {}
}
public void run() {
    for (int i = 0; i < 4; i++) {
        System.out.println(i);
        try {
            Thread.sleep(1000);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    System.exit(0);
}
```

Kết quả hiển thị như sau:

```
Thread is: Thread[main,5,main]
Thread is: Thread[Test thread,5,main]
0
1
2
3
```

Trong kết quả xuất ra ở trên



Mỗi luồng trong chương trình Java được đăng ký cho một quyền ưu tiên. Máy ảo Java không bao giờ thay đổi quyền ưu tiên của luồng. Quyền ưu tiên vẫn còn là hằng số cho đến khi luồng bị ngắt.

Mỗi luồng có một giá trị ưu tiên nằm trong khoảng của một thread.MIN_PRIORITY của 1, và một Thread.MAX_PRIORITY của 10. Mỗi luồng phụ thuộc vào một nhóm và mỗi nhóm luồng có quyền ưu tiên của chính nó. Mỗi luồng được nhận một hằng số ưu tiên của phương thức Thread.PRIORITY là 5. Mỗi luồng mới thừa kế quyền ưu tiên của luồng mà tạo ra nó.

Lớp luồng có vài phương thức khởi dựng, hai trong số các phương thức khởi dựng được đề cập đến dưới đây:

```
public Thread(String threadname)
```

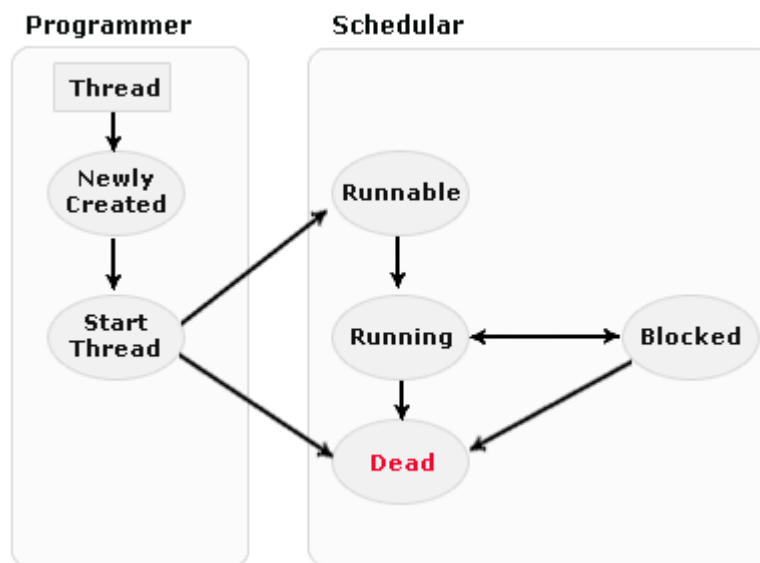
Cấu trúc một luồng với tên là “threadname”

```
public Thread()
```

Cấu trúc một luồng với tên “Thread, được ràng buộc với một số; lấy ví dụ, Thread-1, Thread-2, v.v...

Chương trình bắt đầu thực thi luồng với việc gọi phương thức start(), mà phương thức này phụ thuộc vào lớp luồng. Phương thức này, lần lượt, viện dẫn phương thức run(), nơi mà phương thức định nghĩa tác vụ được thực thi. Phương thức này có thể viết đè lên lớp con của lớp luồng, hoặc với một đối tượng Runnable.

4. Vòng đời của Luồng



a) Trạng thái NEW

Là trạng thái trước khi gọi hàm start().

b) Runnable: Sẵn sàng chạy

Trạng thái này bắt đầu vòng đời của một thread, tuy nhiên trạng thái này rồi mới đến hàm start() được triệu gọi.

Tuy nhiên, trong khi chạy, lúc ngủ, lúc bị khóa thì thread vẫn có thể về trạng thái này. Đây là trạng thái “chờ đến lượt dùng CPU”.

c) **Running: Đang chạy**

Đây là trạng thái các thread được thực thi. Có nhiều cách để về trạng thái Runnable nhưng chỉ có một cách đến trạng thái Running, đó là khi bộ scheduler chọn thread này.

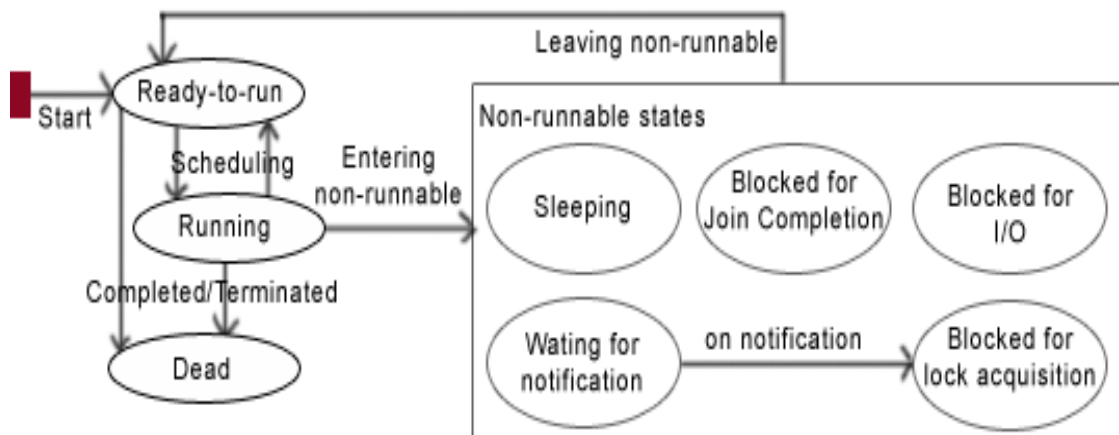
d) **Dead - Chết**

Khi hàm run() chạy xong, thread được coi là ở trạng thái DEAD. Lúc này thread không thể được chạy lại nữa.

e) **Blocked – Khóa**

Đây là trạng thái mà thread chờ đến lượt được dùng tài nguyên (đang bị chiếm dụng bởi thread khác).

Sơ đồ các trạng thái của **Multi-Threads**:



Các trạng thái:

Sleeping : Vẫn sống nhưng ko sẵn sàng để chạy, được tạo bởi hàm sleep()

Waiting for Notification: Chờ được đánh thức bởi thread khác, tạo bởi hàm wait()

Blocked on I/O : Chờ đến sự đánh thức bởi tài nguyên IO

Các phương thức:

Phương thức	Kiểu trả về	Mô tả
-------------	-------------	-------

currentThread()	Thread	Trả về thread hiện thời nơi mà hàm này được gọi (giống như biến this)
getName()	String	Lấy tên của thread
start()	void	Bắt đầu thread bằng cách gọi hàm run()
run()	void	Phương thức này là nơi bắt đầu của thread (giống như main là nơi bắt đầu của chương trình)
sleep()	void	Tạm treo thread trong một thời gian nhất định (tính theo mili giây)
isAlive()	boolean	Xác định xem thread còn sống hay đã chết
interrupt()	void	ngắt thread

5. Phạm vi của luồng và các phương thức của lớp luồng

Một luồng đã được tạo mới gần đây là trong phạm vi “sinh”. Luồng không bắt đầu chạy ngay lập tức sau khi nó được tạo ra. Nó đợi phương thức start() của chính nó được gọi. Cho đến khi, nó là trong phạm vi “sẵn sàng để chạy”. Luồng đi vào phạm vi “đang chạy” khi hệ thống định rõ vị trí luồng trong bộ vi xử lý.

Bạn có thể sử dụng phương thức sleep() để tạm thời treo sự thực thi của luồng. Luồng trở thành sẵn sàng sau khi phương thức sleep kết thúc thời gian. Luồng Sleeping không sử dụng bộ vi xử lý. luồng đi vào phạm vi “waiting” (đợi) luồng đang chạy gọi phương thức wait() (đợi).

Khi các luồng khác liên kết với các đối tượng, gọi phương thức notify(), luồng đi vào trở lại phạm vi “ready” (sẵn sàng) Luồng đi vào phạm vi “blocked” (khối) khi nó đang thực thi các phép toán vào/ra (Input/output). Nó đi vào phạm vi “ready” (sẵn sàng) khi các phương thức vào/ra nó đang đợi cho đến khi được hoàn thành. Luồng đi vào phạm vi “dead” (chết) sau khi phương thức run() đã được thực thi hoàn toàn, hoặc khi phương thức stop() (dừng) của nó được gọi.

Thêm vào các phương thức đã được đề cập trên, Lớp luồng cũng có các phương thức sau:

Phương thức	Mục đích
Enumerate(Thread t)	Sao chép tất cả các luồng hiện hành vào mảng được chỉ định từ nhóm của các luồng, và các nhóm con của nó.
getName()	Trả về tên của luồng
isAlive()	Trả về Đúng, nếu luồng vẫn còn tồn tại (sống)
getPriority()	Trả về quyền ưu tiên của luồng
setName(String name)	Đặt tên của luồng là tên mà luồng được truyền như là một tham số.

<code>join()</code>	Đợi cho đến khi luồng chết.
<code>isDaemon(Boolean on)</code>	Kiểm tra nếu luồng là luồng một luồng hiếm.
<code>resume()</code>	Đánh dấu luồng như là luồng hiếm hoặc luồng người sử dụng phụ thuộc vào giá trị được truyền vào.
<code>sleep()</code>	Hoãn luồng một khoảng thời gian chính xác.
<code>start()</code>	Gọi phương thức <code>run()</code> để bắt đầu một luồng.

Bảng Các phương thức của một lớp luồng

Bảng kế hoạch Round-robin (bảng kiến nghị ký tên vòng tròn) liên quan đến các luồng với cùng quyền ưu tiên được chiếm hữu quyền ưu tiên của mỗi luồng khác. Chúng chia nhỏ thời gian một cách tự động trong theo kiểu kế hoạch xoay vòng này.

Phiên bản mới nhất của Java không hỗ trợ các phương thức `Thread.suspend()` (tri hoãn), `Thread.resume()` (phục hồi) và `Thread.stop()` (dừng).

6. Thời gian biểu luồng

Hầu hết các chương trình Java làm việc với nhiều luồng. CPU chứa đựng cho việc chạy chương trình chỉ một luồng tại một khoảng thời gian. Hai luồng có cùng quyền ưu tiên trong một chương trình hoàn thành trong một thời gian CPU. Lập trình viên, hoặc máy ảo Java, hoặc hệ điều hành chắc chắn rằng CPU được chia sẻ giữa các luồng. Điều này được biết như là bảng thời gian biểu luồng.

Không có máy ảo Java nào thực thi rành mạch cho bảng thời gian biểu luồng. Một số nền Java hỗ trợ việc chia nhỏ thời gian. Ở đây, mỗi luồng nhận một phần nhỏ của thời gian bộ vi xử lý, được gọi là định lượng. Luồng có thể thực thi tác vụ của chính nó trong suốt khoảng thời gian định lượng đấy. Sau khoảng thời gian này được vượt qua, luồng không được nhận nhiều thời gian để tiếp tục thực hiện, ngay cả nếu nó không được hoàn thành việc thực hiện của nó. Luồng kế tiếp của luồng có quyền ưu tiên bằng nhau này sẽ lấy khoảng thời gian thay đổi của bộ vi xử lý. Java là người lập thời gian biểu chia nhỏ tất cả các luồng có cùng quyền ưu tiên cao.

Phương thức `setPriority()` lấy một số nguyên (integer) như là một tham số có thể hiệu chỉnh quyền ưu tiên của một luồng. Đây là giá trị có phạm vi thay đổi từ 1 đến 10, mặc khác, phương thức đưa ra một ngoại lệ (bẫy lỗi) được gọi là `IllegalArgumentException` (Chấp nhận tham số trái luật)

Phương thức `yield()` đưa ra các luồng khác một khả năng để thực thi. Phương thức này được thích hợp cho các hệ thống không chia nhỏ thời gian (non-time-sliced), nơi mà các luồng hiện thời hoàn thành việc thực hiện trước khi các luồng có quyền ưu tiên ngang nhau kế tiếp tiếp quản. Ở đây, bạn sẽ gọi phương thức `yield()` tại những khoản thời gian riêng biệt để có thể tất cả các luồng có quyền ưu tiên ngang nhau chia sẻ thời gian thực thi CPU.

Chương trình chứng minh quyền ưu tiên của luồng:

```
public class PrioritySample {
    Priority t1, t2, t3;
    public PrioritySample() {
        t1 = new Priority("Design Global 1");
        t1.start();
        t2 = new Priority("Design Global 2");
        t2.start();
        t3 = new Priority("Design Global 3");
        t3.start();
    }
    public static void main(String args[]) {
        new PrioritySample();
    }
    class Priority extends Thread implements Runnable {
        String name;
        int sleep;
        int prio = 3;
        public Priority() {
            sleep += 100;
            prio++;
            setPriority(prio);
        }
        public Priority(String name) {
            super(name);
            this.name = name;
        }
        public void run() {
            try {
                Thread.sleep(sleep);
                System.out.println("Name " + getName() + " Priority = " + getPriority() + "/" + this.prio);
            } catch (InterruptedException e) {
                System.out.println(e.getMessage());
            }
        }
    }
}
```

Kết quả hiển thị như hình sau

Name Design Global 1 Priority = 5/3

Name Design Global 2 Priority = 5/3

Name Design Global 3 Priority = 5/3

Hình Quyền ưu tiên luồng

7. Luồng chạy nền

Một chương trình Java bị ngắt chỉ sau khi tất cả các luồng bị chết. Có hai kiểu luồng trong một chương trình Java:

- Các luồng người sử dụng
- Luồng chạy nền

Người sử dụng tạo ra các luồng người sử dụng, trong khi các luồng được chỉ định như là luồng “background” (nền). Luồng chạy nền cung cấp các dịch vụ cho các luồng khác.

Máy ảo Java thực hiện tiến trình thoát, khi và chỉ khi luồng chạy nền vẫn còn sống. Máy ảo Java có ít nhất một luồng chạy nền được biết đến như là luồng “garbage collection” (thu lượm những dữ liệu vô nghĩa - dọn rác). Luồng dọn rác thực thi chỉ khi hệ thống không có tác vụ nào. Nó là một luồng có quyền ưu tiên thấp. Lớp luồng có hai phương thức để thỏa thuận với các luồng chạy nền:

```
public void setDaemon(boolean on)
```

```
public boolean isDaemon()
```

8. Đa luồng với Applets

Trong khi đa luồng là rất hữu dụng trong các chương trình ứng dụng độc lập, nó cũng đáng được quan tâm với các ứng dụng trên Web. Đa luồng được sử dụng trên web, cho ví dụ, trong các trò chơi đa phương tiện, các bức ảnh đầy sinh khí, hiển thị các dòng chữ chạy qua lại trên biểu ngữ, hiển thị đồng hồ thời gian như là một phần của trang Web v.vv... Các chức năng này cấu thành các trang web làm quyến rũ và bắt mắt.

Chương trình Java dựa trên Applet thường sử dụng nhiều hơn một luồng. Trong đa luồng với Applet, lớp `java.applet.Applet` là lớp con được tạo ra bởi người sử dụng định nghĩa applet. Từ đó, Java không hỗ trợ nhiều kế thừa với các lớp, nó không thể thực hiện được trực tiếp lớp con của lớp luồng trong các applet. Tuy nhiên, chúng ta sử dụng một đối tượng của luồng người sử dụng đã định nghĩa, mà các luồng này, lần lượt, dẫn xuất từ lớp luồng. Một luồng đơn giản xuất hiện sẽ được thực thi tại giao diện (Interface) `Runnable`

Chương trình chỉ ra điều này thực thi như thế nào:

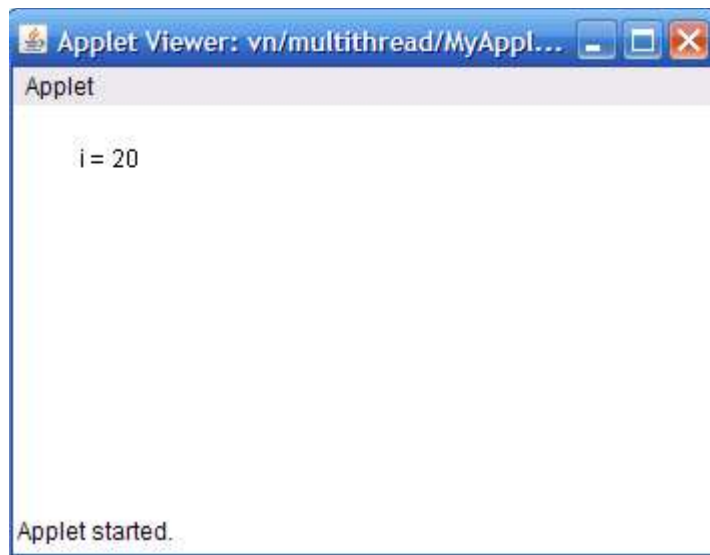
```
import java.applet.Applet;
```

```
import java.awt.Graphics;
```

```
public class MyApplet extends Applet implements Runnable {
    int i;
    Thread t;
    public void init() {
t = new Thread(this);
        t.start();
    }
    public void paint(Graphics g) {
        g.drawString(" i = " + i, 30, 30);
    }
    public void run() {
        for (i = 1; i <= 20; i++) {
            try {
                repaint();
                Thread.sleep(500);
            } catch (InterruptedException e) {
                System.out.println(e.getMessage());
            }
        }
    }
}
```

Trong chương trình này, chúng ta tạo ra một Applet được gọi là MyApplet, và thực thi giao diện Runnable để cung cấp khả năng đa luồng cho applet. Sau đó, chúng ta tạo ra một thể nghiệm (instance) cho lớp luồng, với thể nghiệm applet hiện thời như là một tham số để thiết lập (khởi dựng). Rồi thì chúng ta viện dẫn phương thức start() của luồng thể nghiệm này. Lần lượt, rồi sẽ viện dẫn phương thức run(), mà phương thức này thực sự là điểm bắt đầu cho phương thức này. Chúng ta in số từ 1 đến 20 với thời gian kéo trễ là 500 miligiây giữa mỗi số. Phương thức sleep() được gọi để hoàn thành thời gian kéo trễ này. Đây là một phương thức tĩnh được định nghĩa trong lớp luồng. Nó cho phép luồng nằm yên (ngủ) trong khoản thời gian hạn chế.

Xuất ra ngoài có dạng như sau:



Hình Đa luồng với Applet

9. Nhóm luồng

Một lớp nhóm luồng (ThreadGroup) nắm bắt một nhóm của các luồng. Lấy ví dụ, một nhóm luồng trong một trình duyệt có thể quản lý tất cả các luồng phụ thuộc vào một đơn thể applet. Tất cả các luồng trong máy ảo Java phụ thuộc vào các nhóm luồng mặc định. Mỗi nhóm luồng có một nhóm nguồn cha. Vì thế, các nhóm từ một cấu trúc dạng cây. Nhóm luồng “hệ thống” là gốc của tất cả các nhóm luồng. Một nhóm luồng có thể là thành phần của cả các luồng, và các nhóm luồng.

Hai kiểu nhóm luồng thiết lập (khởi dựng) là:

```
public ThreadGroup(String str)
```

Ở đây, “str” là tên của nhóm luồng mới nhất được tạo ra.

```
public ThreadGroup(ThreadGroup tgroup, String str)
```

Ở đây, “tgroup” chỉ ra luồng đang chạy hiện thời như là luồng cha, “str” là tên của luồng đang được tạo ra.

Một số các phương thức trong nhóm luồng (ThreadGroup) được cho như sau:

```
public synchronized int activeCount()
```

Trả về số lượng các luồng kích hoạt hiện hành trong nhóm luồng

```
public synchronized int activeGroupCount()
```

Trả về số lượng các nhóm hoạt động trong nhóm luồng

```
public final String getName()
```

Trả về tên của nhóm luồng

```
public final ThreadGroup getParent()
```

Trả về cha của nhóm luồng

10. Sự đồng bộ luồng

Trong khi đang làm việc với nhiều luồng, nhiều hơn một luồng có thể muốn thâm nhập cùng biến tại cùng thời điểm. Lấy ví dụ, một luồng có thể cố gắng đọc dữ liệu, trong khi luồng khác cố gắng thay đổi dữ liệu.

Trong trường hợp này, dữ liệu có thể bị sai lệch. Trong những trường hợp này, bạn cần cho phép một luồng hoàn thành trọn vẹn tác vụ của nó (thay đổi giá trị), và rồi thì cho phép các luồng kế tiếp thực thi. Khi hai hoặc nhiều hơn các luồng cần thâm nhập đến một tài nguyên được chia sẻ, bạn cần chắc chắn rằng tài nguyên đó sẽ được sử dụng chỉ bởi một luồng tại một thời điểm. Tiến trình này được gọi là “sự đồng bộ” (synchronization) được sử dụng để lưu trữ cho vấn đề này. Phương thức “đồng bộ” (synchronized) báo cho hệ thống đặt một khóa vòng một tài nguyên riêng biệt.

Mấu chốt của sự đồng bộ hóa là khái niệm “monitor” (sự quan sát, giám sát), cũng được biết như là một bảng mã “semaphore” (bảng mã). Một “monitor” là một đối tượng mà được sử dụng như là một khóa qua lại duy nhất, hoặc “mutex”. Chỉ một luồng có thể có riêng nó một sự quan sát (monitor) tại mỗi thời điểm được đưa ra. Tất cả các luồng khác cố gắng thâm nhập vào monitor bị khóa sẽ bị trì hoãn, cho đến khi luồng đầu tiên thoát khỏi monitor. Các luồng khác được báo chờ đợi monitor. Một luồng mà monitor của riêng nó có thể thâm nhập trở lại cùng monitor.

Mã đồng bộ

Tất cả các đối tượng trong Java được liên kết với các monitor (sự giám sát) của riêng nó. Để đăng nhập vào monitor của một đối tượng, lập trình viên sử dụng từ khóa synchronized (đồng bộ) để gọi một phương thức hiệu chỉnh (modified). Khi một luồng đang được thực thi trong phạm vi một phương thức đồng bộ (synchronized), bất kỳ luồng khác hoặc phương thức đồng bộ khác mà cố gắng gọi nó trong cùng thể nghiệm sẽ phải đợi.

Chương trình chứng minh sự làm việc của từ khóa synchronized (sự đồng bộ). Ở đây, lớp “Target” (mục tiêu) có một phương thức “display()” (hiển thị) mà phương thức này lấy một tham số kiểu số nguyên (int). Số này được hiển thị trong phạm vi các cặp ký tự “<> # số # <>”. Phương thức “Thread.sleep(1000)” tạm dừng luồng hiện tại sau khi phương thức “display()” được gọi.

Thiết lập (khởi dựng) của lớp “Source” lấy một tham chiếu đến một đối tượng “t” của lớp “Target”, và một biến số nguyên (integer). Ở đây, một luồng mới cũng được tạo ra. Luồng này gọi phương thức run() của đối tượng “t”. Lớp chính “Synch” thể nghiệm lớp “Target” như là “target (mục tiêu), và tạo ra 3 đối tượng của lớp “Source” (nguồn). Cùng đối tượng “target” được truyền cho mỗi đối tượng “Source”. Phương thức “join()” (gia nhập) làm luồng được gọi đợi cho đến khi việc gọi luồng bị ngắt.

```
class Target {
    synchronized void display(int num) {
        System.out.print("<> " + num);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            System.out.println("Interrupted");
        }
        System.out.println(" <>");
    }
}

class Source implements Runnable {
    int number;
    Target target;
    Thread t;
    public Source(Target targ, int n) {
        target = targ;
        number = n;
        t = new Thread(this);
        t.start();
    }
    public void run() {
        synchronized (target) {
            target.display(number);
        }
    }
}

class SyncSample {
    public static void main(String args[]) {
        Target target = new Target();
        int digit = 10;
        Source s1 = new Source(target, digit++);
        Source s2 = new Source(target, digit++);
        Source s3 = new Source(target, digit++);
        try {
            s1.t.join();
            s2.t.join();
        }
    }
}
```

```
s3.t.join();
} catch (InterruptedException e) {
    System.out.println("Interrupted");
}
}
}
```

Kết quả hiển thị như hình cho dưới đây:

```
<> 10 <>
<> 12 <>
<> 11 <>
```

Hình Kết quả hiển thị của chương trình

Trong chương trình trên, có một “dãy số” đăng nhập được hiển thị “display()”. Điều này có nghĩa là việc thâm nhập bị hạn chế một luồng tại mỗi thời điểm. Nếu từ khóa synchronized đặt trước bị bỏ quên trong phương thức “display()” của lớp “Target”, tất cả luồng trên có thể cùng lúc gọi cùng phương thức, trên cùng đối tượng. Điều kiện này được biết đến như là “loại điều kiện” (race condition). Trong trường hợp này, việc xuất ra ngoài sẽ được chỉ ra như sau:

```
<> 11<> 12<> 10 <>
<>
<>
```

Hình Kết quả hiển thị của chương trình không có sự đồng bộ

Sử dụng khối đồng bộ (Synchronized Block)

Tạo ra các phương thức synchronzed (đồng bộ) trong phạm vi các lớp là một con thường dễ dàng và có hiệu quả của việc thực hiện sự đồng bộ. Tuy nhiên, điều này không làm việc trong tất cả các trường hợp.

Hãy xem một trường hợp nơi mà lập trình viên muốn sự đồng bộ được xâm nhập vào các đối tượng của lớp mà không được thiết kế cho thâm nhập đa luồng. Tức là, lớp không sử dụng các phương thức đồng bộ. Vì thế từ khoá synchronized không thể được thêm vào các phương thức thích hợp trong phạm vi lớp.

Để đồng bộ thâm nhập một đối tượng của lớp này, tất cả chúng gọi các phương thức mà lớp này định nghĩa, được đặt bên trong một khối đồng bộ. Tất cả chúng sử dụng chung một câu lệnh đồng bộ được cho như sau:

```
synchronized(object)
{
    // các câu lệnh đồng bộ
```

```
}
```

Ở đây, “object” (đối tượng) là một tham chiếu đến đối tượng được đồng bộ. Dấu ngoặc móc không cần thiết khi chỉ một câu lệnh được đồng bộ. Một khối đồng bộ bảo đảm rằng nó gọi đến một phương thức (mà là thành phần của đối tượng) xuất hiện chỉ sau khi luồng hiện hành đã được tham nhập thành công vào monitor (sự quan sát) của đối tượng.

Chương trình chỉ ra câu lệnh đồng bộ sử dụng như thế nào:

```
class Target {
    void display(int num) {
        System.out.print("<> " + num);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            System.out.println("Interrupted");
        }
        System.out.println(" <>");
    }
}

class Source implements Runnable {
    int number;
    Target target;
    Thread t;
    public Source(Target targ, int n) {
        target = targ;
        number = n;
        t = new Thread(this);
        t.start();
    }
    public void run() {
        synchronized (target) {
            target.display(number);
        }
    }
}

class SyncSample {
    public static void main(String args[]) {
        Target target = new Target();
```

```
int digit = 10;
Source s1 = new Source(target, digit++);
Source s2 = new Source(target, digit++);
Source s3 = new Source(target, digit++);
}
}
```

Ở đây, từ khóa `synchronized` không hiệu chỉnh phương thức `display()`. Từ khóa này được sử dụng trong phương thức `run()` của lớp `“Target”` (mục tiêu). Kết quả xuất ra màn hình tương tự với kết quả chỉ ra ở ví dụ trước.

Sự không thuận lợi của các phương thức đồng bộ

Người lập trình thường viết các chương trình trên các đơn thể luồng. Tất nhiên các trạng thái này chắc chắn không lợi ích cho đa luồng. Lấy ví dụ, luồng không tận dụng việc thực thi của trình biên dịch. Trình biên dịch Java từ Sun không chứa nhiều phương thức đồng bộ.

Các phương thức đồng bộ không thực thi tốt như là các phương thức không đồng bộ. Các phương thức này chậm hơn từ ba đến bốn lần so với các phương thức tương ứng không đồng bộ. Trong trạng thái nơi mà việc thực thi là có giới hạn, các phương thức đồng bộ bị ngăn ngừa.

11. Kỹ thuật “*wait-notify*” (đợi – thông báo)

Luồng chia các tác vụ thành các đơn vị riêng biệt và logic (hợp lý). Điều này thay thế các trường hợp (sự kiện) chương trình lặp. Các luồng loại trừ “polling” (kiểm soát vòng).

Một vòng lặp mà lặp lại việc một số điều kiện thường thực thi “polling” (kiểm soát vòng). Khi điều kiện nhận giá trị là `True` (đúng), các câu lệnh phức tạp được thực hiện. Đây là tiến trình thường bỏ phí thời gian của CPU. Lấy ví dụ, khi một luồng sinh ra một số dữ liệu, và các luồng khác đang chờ đợi nó, luồng sinh ra phải đợi cho đến khi các luồng sử dụng nó hoàn thành, trước khi phát sinh ra dữ liệu.

Để tránh trường hợp kiểm soát vòng, Java bao gồm một thiết kế tốt trong tiến trình kỹ thuật truyền thông sử dụng các phương thức `“wait()”` (đợi), `“notify()”` (thông báo) và `“notifyAll()”` (thông báo hết). Các phương thức này được thực hiện như là các phương thức cuối cùng trong lớp đối tượng (Object), để mà tất cả các lớp có thể thâm nhập chúng. Tất cả 3 phương thức này có thể được gọi chỉ từ trong phạm vi một phương thức đồng bộ (`synchronized`).

Các chức năng của các phương thức `“wait()”`, `“notify()”`, và `“notifyAll()”` là:

- Phương thức `wait()` nói cho việc gọi luồng trao cho monitor (sự giám sát), và nhập trạng thái “sleep” (chờ) cho đến khi một số luồng khác thâm nhập cùng monitor, và gọi phương thức “`notify()`”.
- Phương thức `notify()` đánh thức, hoặc thông báo cho luồng đầu tiên mà đã gọi phương thức `wait()` trên cùng đối tượng.
- Phương thức `notifyAll()` đánh thức, hoặc thông báo tất cả các luồng mà đã gọi phương thức `wait()` trên cùng đối tượng. Quyền ưu tiên cao nhất luồng chạy đầu tiên.

Cú pháp của 3 phương thức này như sau:

```
final void wait() throws IOException  
final void notify()  
final void notifyAll()
```

Các phương thức `wait()` và `notify()` cho phép một đối tượng được chia sẻ để tạm ngừng một luồng, khi đối tượng trở thành không còn giá trị cho luồng. Chúng cũng cho phép luồng tiếp tục khi thích hợp.

Các luồng bản thân nó không bao giờ kiểm tra trạng thái của đối tượng đã chia sẻ.

Một đối tượng mà điều khiển các luồng khách (client) của chính nó theo kiểu này được biết như là một monitor (sự giám sát). Trong các thuật ngữ chặt chẽ của Java, một monitor là bất kỳ đối tượng nào mà có một số mã đồng bộ. Các monitor được sử dụng cho các phương thức `wait()` và `notify()`. Cả hai phương thức này phải được gọi trong mã đồng bộ.

Một số điểm cần nhớ trong khi sử dụng phương thức `wait()`:

- Luồng đang gọi đưa vào CPU
- Luồng đang gọi đưa vào khóa
- Luồng đang gọi đi vào vùng đợi của monitor.

Các điểm chính cần nhớ về phương thức `notify()`:

- Một luồng đưa ra ngoài vùng đợi của monitor, và vào trạng thái sẵn sàng.
- Luồng mà đã được `notify()` phải thu trở lại khóa của monitor trước khi nó có thể bắt đầu.
- Phương thức `notify()` không thể chỉ ra được luồng mà phải được thông báo. Trong một số trường hợp, các phương thức của monitor đưa ra 2 sự đề phòng:
 - Trạng thái của monitor sẽ được kiểm tra trong một vòng lặp “while” tốt hơn là câu lệnh if
 - Sau khi thay đổi trạng thái của monitor, phương thức `notifyAll()` sẽ được sử dụng, tốt hơn phương thức `notify()`.

Chương trình biểu thị cho việc sử dụng các phương thức `notify()` và `wait()`:

```
import java.applet.Applet;
import java.awt.Graphics;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
public class mouseApplet extends Applet implements MouseListener {
    boolean click;
    int count;
    public void init() {
        super.init();
        add(new clickArea(this)); //doi tuong ve duoc tao ra va them vao
        add(new clickArea(this)); //doi tuong ve duoc tao ra va them vao
        addMouseListener(this);
    }
    public void mouseClicked(MouseEvent e) {
    }
    public void mouseEntered(MouseEvent e) {
    }
    public void mouseExited(MouseEvent e) {
    }
    public void mousePressed(MouseEvent e) {
        synchronized (this) {
            click = true;
            notify();
        }
        count++; //dem viec click
        Thread.currentThread().yield();
        click = false;
    }
    public void mouseReleased(MouseEvent e) {
    }
} //kết thúc Applet
class clickArea extends java.awt.Canvas implements Runnable {
    mouseApplet myapp;
    clickArea(mouseApplet mapp) {
        this.myapp = mapp;
        setSize(40, 40);
        new Thread(this).start();
    }
}
```

```
}  
public void paint(Graphics g) {  
    g.drawString(new Integer(myapp.count).toString(), 15, 20);  
}  
public void run() {  
    while (true) {  
        synchronized (myapp) {  
            while (!myapp.click) {  
                try {  
                    myapp.wait();  
                } catch (InterruptedException ie) {  
                }  
            }  
        }  
        repaint(250);  
    }  
} //end run  
}
```

Không cần các phương thức wait() và notify(), luồng bức vẽ (canvas) không thể biết khi nào cập nhập hiển thị. Kết quả xuất ra ngoài của chương trình được đưa ra như sau:



Hình kết quả sau mỗi lần kích chuột

12. Sự bế tắc (Deadlocks)

Một “deadlock” (sự bế tắc) xảy ra khi hai luồng có một phụ thuộc vòng quanh trên một cặp đối tượng đồng bộ; lấy ví dụ, khi một luồng thâm nhập vào monitor trên đối tượng “ObjA”, và một luồng khác thâm nhập vào monitor trên đối tượng “ObjB”. Nếu luồng trong “ObjA” cố gắng gọi phương thức đồng bộ trên “ObjB”, một bế tắc xảy ra.

Khó để gỡ lỗi một bế tắc bởi những nguyên nhân sau:

- Luồng được chạy nền khi xảy ra khó để theo dõi thêm nữa là hai luồng chia sẻ thời gian thực thi
- Đôi khi có thể bao hàm nhiều hơn hai luồng và hai đối tượng đồng bộ

Chương trình tạo ra điều kiện bế tắc. Lớp chính (main) bắt đầu 2 luồng. Mỗi luồng gọi phương thức đồng bộ run(). Khi luồng “t1” đánh thức, nó gọi phương thức “syncIt()” của đối tượng deadlock “dlk1”. Từ đó luồng “t2” một mình giám sát cho “dlk2”, luồng “t1” bắt đầu đợi monitor. Khi luồng “t2” đánh thức, nó cố gắng gọi phương thức “syncIt()” của đối tượng Deadlock “dlk2”. Bây giờ, “t2” cũng phải đợi, bởi vì đây là trường hợp tương tự với luồng “t1”. Từ đó, cả hai luồng đang đợi lẫn nhau. Đây là điều kiện bế tắc.

```
public class Deadlock implements Runnable {
    Deadlock grabIt;
    public static void main(String args[]) {
        Deadlock dlk1 = new Deadlock();
        Deadlock dlk2 = new Deadlock();
        Thread t1 = new Thread(dlk1);
        Thread t2 = new Thread(dlk2);
        dlk1.grabIt = dlk1;
        dlk2.grabIt = dlk2;
        t1.start();
        t2.start();
        System.out.println("Started");
    }
    public synchronized void run() {
        try {
            System.out.println("Sync");
            Thread.sleep(1500);
        } catch (InterruptedException e) {
            System.out.println("error occured");
        }
        grabIt.syncIt();
    }
    public synchronized void syncIt() {
        try {
            Thread.sleep(1500);
            System.out.println("Sync");
        } catch (InterruptedException e) {
```

```

        System.out.println("error occurred");
    }
    System.out.println("In the syncIt() method");
}
}

```

Kết quả của chương trình này được hiển thị như sau:

```

Started
Sync
In the syncIt() method
Sync
In the syncIt() method

```

13. Thu dọn “rác” (*Garbage collection*)

Thu dọn “rác” (*Garbage collection*) cải tạo hoặc làm trống bộ nhớ đã định vị cho các đối tượng mà các đối tượng này không sử dụng trong thời gian dài. Trong ngôn ngữ lập trình hướng đối tượng khác như C++, lập trình viên phải làm cho bộ nhớ trống mà đã không được yêu cầu trong thời gian dài. Tình trạng không hoạt động để bộ nhớ trống có thể là kết quả trong một số vấn đề. Java tự động tiến hành thu dọn rác để cung cấp giải pháp duy nhất cho vấn đề này. Một đối tượng trở nên thích hợp cho sự dọn rác nếu không có tham chiếu đến nó, hoặc nếu nó đã đăng ký rỗng.

Sự dọn rác thực thi như là một luồng riêng biệt có quyền ưu tiên thấp. Bạn có thể viện dẫn một phương thức `gc()` của thể nghiệm để viện dẫn sự dọn rác. Tuy nhiên, bạn không thể dự đoán hoặc bảo đảm rằng sự dọn rác sẽ thực thi một cách trọn vẹn sau đó. Sử dụng câu lệnh sau để tắt đi sự dọn rác trong ứng dụng:

```
Java -noasyncgc ....
```

Phương thức `finalize()` (hoàn thành)

Java cung cấp một con đường để làm sạch một tiến trình trước khi điều khiển trở lại hệ điều hành. Điều này tương tự như phương thức hủy của C++ Phương thức `finalize()`, nếu hiện diện, sẽ được thực thi trên mỗi đối tượng, trước khi sự dọn rác.

Câu lệnh của phương thức `finalize()` như sau:

protected void finalize() throws Throwable

Tham chiếu không phải là sự dọn rác; chỉ các đối tượng mới được dọn rác

Lấy thể nghiệm:

```

Object a = new Object();
Object b = a;

```

```
a = null;
```

Ở đây, nó sẽ sai khi nói rằng “b” là một đối tượng. Nó chỉ là một đối tượng tham chiếu. Hơn nữa, trong đoạn mã trích trên mặc dù “a” được đặt là rỗng, nó không thể được dọn rác, bởi vì nó vẫn còn có một tham chiếu (b) đến nó. Vì thế “a” vẫn còn với đến được, thật vậy, nó vẫn còn có phạm vi sử dụng trong phạm vi chương trình. Ở đây, nó sẽ không được dọn rác.

Tuy nhiên, trong ví dụ cho dưới đây, giả định rằng không có tham chiếu đến “a” tồn tại, đối tượng “a” trở nên thích hợp cho garbage collection.

Một ví dụ khác:

```
Object m = new Object();  
Object m = null;
```

Đối tượng được tạo ra trong sự bắt đầu có hiệu lực cho garbage collection

```
Object m = new Object();  
M = new Object();
```

Bây giờ, đối tượng căn nguyên có hiệu lực cho garbage collection, và một đối tượng mới tham chiếu bởi “m” đang tồn tại.

Bạn có thể chạy phương thức garbage collection, nhưng không có bảo đảm rằng nó sẽ xảy ra.

Chương trình điển hình cho garbage collection.

```
public class GCDemo {  
    public static void main(String args[]) {  
        int i;  
        long a;  
        Runtime r = Runtime.getRuntime();  
        Long[] values = new Long[200];  
        System.out.println("Amount of free memory is" + r.freeMemory());  
        r.gc();  
        System.out.println("Amount of free memory after garbage collection is " + r.freeMemory());  
        for (a = 0, i = 0; i < 200; a++, i++) {  
            values[i] = new Long(a);  
        }  
        System.out.println("Amount of free memory after creating the array " + r.freeMemory());  
        for (i = 0; i < 200; i++) {  
            values[i] = null;  
        }  
    }  
}
```

```

        System.out.println("Amount of free memory after garbage collection is " + r.freeMemory());
    }
}

```

Chúng ta khai một mảng gồm 200 phần tử, trong đó kiểu dữ liệu là kiểu Long. Trước khi mảng được tạo ra, chúng ta phải xác định rõ số lượng bộ nhớ trống, và hiển thị nó. Rồi thì chúng ta viện dẫn phương thức gc() của thể nghiệm Runtime (thời gian thực thi) hiện thời. Điều này có thể hoặc không thể thực thi garbage collection. Rồi thì chúng ta tạo ra mảng, và đăng ký giá trị cho các phần tử của mảng. Điều này sẽ giảm bớt số lượng bộ nhớ trống. Để làm các mảng phần tử thích hợp cho garbage collection, chúng ta đặt chúng rỗng. Cuối cùng, chúng ta sử dụng phương thức gc() để viện dẫn garbage collection lần nữa.

Kết quả xuất ra màn hình của chương trình trên như sau:

```

Amount of free memory is 15874472
Amount of free memory after garbage collection is 16058960
Amount of free memory after creating the array 16058960
Amount of free memory after garbage collection is 16058960

```

Tổng kết

Một luồng là đơn vị nhỏ nhất của đoạn mã thực thi được mà một tác vụ riêng biệt.

Đa luồng giữ cho thời gian rồi là nhỏ nhất. Điều này cho phép bạn viết các chương trình có khả năng sử dụng tối đa CPU.

Luồng bắt đầu thực thi sau khi phương thức start() được gọi

Lập trình viên, máy ảo Java, hoặc hệ điều hành bảo đảm rằng CPU được chia sẻ giữa các luồng.

Có hai loại luồng trong một chương trình Java:

- Luồng người dùng
- Luồng chạy nền.

Một nhóm luồng là một lớp mà nắm bắt một nhóm các luồng.

Đồng bộ cho phép chỉ một luồng thâm nhập một tài nguyên được chia sẻ tại một thời điểm.

Để tránh kiểm soát vòng, Java bao gồm một thiết kế tốt trong tiến trình kỹ thuật truyền thông sử dụng các phương thức “wait()” (đợi), “notify()” (thông báo) và “notifyAll()” (thông báo hết).

Một “bế tắc” xảy ra khi hai luồng có một phụ thuộc xoay vòng trên một phần của các đối tượng đồng bộ

Garbage collection là một tiến trình nhờ đó bộ nhớ được định vị để các đối tượng mà không sử dụng trong thời gian dài, có thể cải tạo hoặc làm rảnh bộ nhớ.

Bài tập

Viết chương trình sử dụng `wait()` và `notify()` để thực hiện giao dịch nạp tiền và rút tiền tại ngân hàng. Ngân hàng có nhiều tài khoản khách hàng sẽ được tạo thông qua chức năng nhập thông tin gồm Id, Name, Balance.

Tạo ra Menu như sau:

« MENU

1. Tạo tài khoản
2. Rút tiền
3. Nạp tiền
4. Thoát

Mời chọn : «

Sau khi tạo ra danh sách tài khoản, khi chọn chức năng « Rút tiền » hoặc « Nạp tiền » chương trình sẽ yêu cầu chọn tài khoản thực hiện và số tiền. Nếu số tiền trong tài khoản và số tiền thực hiện là hợp lệ, giao dịch sẽ được thực thi.

CÁC CHỦ ĐỀ KHÁC CẦN QUAN TÂM TRONG JAVA

- **CustomNetworking**: xây dựng các tính năng lập trình giao tiếp mạng nâng cao
- **Generics** : nâng cao tính khái quát hóa trong thiết kế lập trình. Là chủ đề rất quan trọng trong Java cần phải nắm bắt.
- **RMI**: bộ công cụ thư viện lập trình hỗ trợ xây dựng các ứng dụng phân tán.
- **Reflection**: kỹ thuật nâng cao của ngôn ngữ Java, hỗ trợ người lập trình xây dựng các tính năng cao cấp, thông qua việc truy cập vào lớp, đối tượng không thông qua việc nạp lớp, đối tượng truyền thống.
- **Security**: tập hợp bộ thư viện hỗ trợ lập trình các tính năng yêu cầu bảo mật mã hóa
- **2DGraphics**: xây dựng các ứng dụng đẹp giao diện đồ họa 2 chiều
- **SocketsDirectProtocol**: xây dựng các module giao tiếp mạng với các ứng dụng, service từ xa thông qua môi trường Internet
- **Java mail** : tập hợp bộ thư viện hỗ trợ lập trình xây dựng ứng dụng quản lý mail

PHỤ LỤC I: LISTENER

Thực hiện Listeners cho các Handled Events

a) Viết một Component Listener

Là một trong những sự kiện của thành phần được phát ra bởi đối tượng Component ngay sau khi thành phần đó mất đi, làm ẩn đi, chuyển vị trí hoặc thay đổi kích thước

Các phương thức, sự kiện của thành phần

Giao diện ComponentListener và lớp mô phỏng tương ứng, ComponentAdapter, chứa 4 phương thức:

void componentHidden(ComponentEvent)

Được gọi bởi AWT sau khi thành phần biến mất bởi phương thức setVisible.

void componentMoved(ComponentEvent)

Được gọi bởi AWT sau khi thành phần di chuyển, nó quan hệ với đối tượng chứa nó.

void componentResized(ComponentEvent)

Được gọi bởi AWT sau khi thành phần thay đổi kích thước.

void componentShown(ComponentEvent)

Được gọi bởi AWT sau khi thành phần xuất hiện bởi phương thức setVisible.

Ví dụ về Handling Component Events

```
public class ComponentEventDemo ... implements ComponentListener {  
    ...  
    //where initialization occurs:  
    aFrame = new Frame("A Frame");  
    ComponentPanel p = new ComponentPanel(this);  
    aFrame.addComponentListener(this);  
    p.addComponentListener(this);  
    ...  
  
    public void componentHidden(ComponentEvent e) {  
        displayMessage("componentHidden event from "  
            + e.getComponent().getClass().getName());  
    }  
}
```

```
public void componentMoved(ComponentEvent e) {
    displayMessage("componentMoved event from "
        + e.getComponent().getClass().getName());
}

public void componentResized(ComponentEvent e) {
    displayMessage("componentResized event from "
        + e.getComponent().getClass().getName());
}

public void componentShown(ComponentEvent e) {
    displayMessage("componentShown event from "
        + e.getComponent().getClass().getName());
}
}

class ComponentPanel extends Panel ... {
    ...
    ComponentPanel(ComponentEventDemo listener) {
    ...//after creating the label and checkbox:
        label.addComponentListener(listener);
        checkbox.addComponentListener(listener);
    }
    ...
}
```

Lớp ComponentEvent

Mỗi một phương thức của sự kiện các thành phần có một thông số đơn: đối tượng **ComponentEvent**. Lớp **ComponentEvent** định nghĩa một phương thức hay dùng, **getComponent**, trả về thành phần mà phát ra sự kiện.

b) Viết một Container Listener

Những sự kiện của Container được phát ra ngay sau khi một thành phần được thêm vào Container hoặc chuyển đi khỏi Container.

Các phương thức, sự kiện của Container

Giao diện **ContainerListener** và lớp mô phỏng tương ứng, **ContainerAdapter** chứa hai phương thức:

void componentAdded(ContainerEvent)

Được gọi sau khi một thành phần được thêm vào Container.

void componentRemoved(ContainerEvent)

Được gọi sau khi một thành phần được chuyển đi khỏi Container.

Ví dụ về Handling Container Events

```
public class ContainerEventDemo ... implements ContainerListener ... {
...//where initialization occurs:
    buttonPanel = new Panel();
    buttonPanel.addContainerListener(this);
...
    public void componentAdded(ContainerEvent e) {
        displayMessage(" added to ", e);
    }

    public void componentRemoved(ContainerEvent e) {
        displayMessage(" removed from ", e);
    }

    void displayMessage(String action, ContainerEvent e) {
        display.append(((Button)e.getChild()).getLabel()
            + " was"
            + action
            + e.getContainer().getClass().getName()
            + "\n");
    }
...
}
```

Lớp ContainerEvent

Mỗi phương thức của Container Event có một thông số đơn : đối tượng ContainerEvent. Lớp ContainerEvent định nghĩa hai phương thức thường dùng sau:

Component getChild()

Trả về thành phần được thêm hay chuyển khỏi Container trong sự kiện này.

Container getContainer()

Trả về Container sinh ra sự kiện này.

c) Viết một Focus Listener

Các sự kiện Focus được phát ra khi một thành phần có hoặc mất đi sự tập trung vào nó.

Các phương thức, sự kiện của Focus

Giao diện `FocusListener` và lớp mô phỏng tương ứng `FocusAdapter`, chứa hai phương thức:

`void focusGained(FocusEvent)`

Được gọi sau khi thành phần có sự tập trung.

`void focusLost(FocusEvent)`

Được gọi sau khi thành phần mất sự tập trung.

Ví dụ về Handling Focus Events

```
public class FocusEventDemo ... implements FocusListener ... {
    ...//where initialization occurs
    window = new FocusWindow(this);
    ...
    public void focusGained(FocusEvent e) {
        displayMessage("Focus gained", e);
    }
    public void focusLost(FocusEvent e) {
        displayMessage("Focus lost", e);
    }
    void displayMessage(String prefix, FocusEvent e) {
        display.append(prefix
            + ": "
            + e.getSource() //XXX
            + "\n");
    }
    ...
}
class FocusWindow extends Frame {
    ...
    public FocusWindow(FocusListener listener) {
        super("Focus Demo Window");
        this.addFocusListener(listener);
    }
    ...
}
```

```

Label label = new Label("A Label");
label.addFocusListener(listener);
...
Choice choice = new Choice();
...
choice.addFocusListener(listener);
...
Button button = new Button("A Button");
button.addFocusListener(listener);
...
List list = new List();
...
list.addFocusListener(listener);
}
}

```

Lớp FocusEvent

Mỗi phương thức Focus Event có một thông số đơn : đối tượng FocusEvent. Lớp FocusEvent định nghĩa một phương thức, isTemporary, trả về giá trị True khi sự kiện mất sự tập trung đó là tạm thời.

Mọi thông báo thông thường mà bạn gọi tới đối tượng FocusEvent là getComponent (được định nghĩa trong ComponentEvent), nó trả về thành phần gây ra sự kiện này.

d) Viết một Item Listener

Các sự kiện của Item được phát ra khi thực thi giao diện ItemSelectable.

Các phương thức, sự kiện của Item

Giao diện ItemListener chỉ có một phương thức , vì vậy nó không có lớp mô phỏng tương ứng:

void itemStateChanged(ItemEvent)

Được gọi sau khi thay đổi trạng thái của thành phần.

Ví dụ về Handling Item Events

```

public void itemStateChanged(ItemEvent e) {
    if (e.getStateChange() == ItemEvent.SELECTED) {
        label.setVisible(true);
    } else {
        label.setVisible(false);
    }
}

```

```
}
}
```

Lớp ItemEvent

Mỗi phương thức của Item event có một thông số đơn : đối tượng ItemEvent. Lớp ItemEvent định nghĩa các phương thức sau:

Object getItem()

Trả về Item được tập trung trong sự kiện này.

ItemSelectable getItemSelectable()

Trả về thành phần phát ra sự kiện.

int getStateChange()

Trả về trạng thái mới của Item . Lớp ItemEvent định nghĩa hai trạng thái : SELECTED và DESELECTED.

e) Viết một Key Listener

Được phát ra khi người sử dụng đánh phím . Đặc biệt Key events phát ra bởi đối tượng mà đang được tập trung khi người dùng nhấn hoặc nhả phím.

Các phương thức sự kiện của Key

Giao diện KeyListener và lớp mô phỏng tương ứng, KeyAdapter, chứa ba phương thức:

void keyTyped(KeyEvent)

Được gọi sau khi phím được đánh.

void keyPressed(KeyEvent)

Được gọi sau khi một phím được ấn.

void keyReleased(KeyEvent)

Được gọi sau khi một phím được nhả.

Ví dụ về Handling Key Events

```
public class KeyEventDemo ... implements KeyListener ... {
...//where initialization occurs:
    typingArea = new TextField(20);
    typingArea.addKeyListener(this);
...
/** Handle the key typed event from the text field. */
    public void keyTyped(KeyEvent e) {
        displayInfo(e, "KEY TYPED: ");
    }
}
```

```
/** Handle the key pressed event from the text field. */
public void keyPressed(KeyEvent e) {
    displayInfo(e, "KEY PRESSED: ");
}
/** Handle the key released event from the text field. */
public void keyReleased(KeyEvent e) {
    displayInfo(e, "KEY RELEASED: ");
}
...
protected void displayInfo(KeyEvent e, String s){
    ...
    char c = e.getKeyChar();
    int keyCode = e.getKeyCode();
    int modifiers = e.getModifiers();
    ...
    tmpString = KeyEvent.getKeyModifiersText(modifiers);

    ...//display information about the KeyEvent...
}
}
```

Lớp KeyEvent

Mỗi phương thức Key Event có một thông số đơn : đối tượng KeyEvent. Lớp KeyEvent định nghĩa những phương thức thường dùng sau:

int getKeyChar()

void setKeyChar(char)

Nhận hoặc xác lập kí tự liên quan với sự kiện này.

int getKeyCode()

void setKeyCode(int)

Nhận hoặc xác lập mã của phím liên quan với sự kiện này.

void setModifiers(int)

Xác lập trạng thái của phím liên quan tới sự kiện này

int getModifiers()

Trả về trạng thái của phím trong sự kiện này.

f) Viết một Mouse Listener

Các sự kiện được phát ra khi người sử dụng dùng chuột tác động đến một thành phần.

Các phương thức, sự kiện của Mouse

Giao diện `MouseListener` và lớp mô phỏng tương ứng , `MouseAdapter`, chứa ba phương thức:

`void mouseClicked(MouseEvent)`

Được gọi sau khi người sử dụng kích hoạt chuột vào một thành phần.

`void mouseEntered(MouseEvent)`

Được gọi sau khi con trỏ chuột nằm trong địa phận của thành phần.

`void mouseExited(MouseEvent)`

Được gọi sau khi con trỏ chuột ra khỏi địa phận của thành phần.

`void mousePressed(MouseEvent)`

Được gọi sau khi con chuột được ấn trên địa phận của thành phần.

`void mouseReleased(MouseEvent)`

Được gọi sau khi con chuột được nhả trên địa phận của thành phần.

Ví dụ về Handling Mouse Events

```
public class MouseEventDemo ... implements MouseListener {
    ...//where initialization occurs:
    //Register for mouse events on blankArea and applet (panel).
    blankArea.addMouseListener(this);
    addMouseListener(this);
    ...
    public void mousePressed(MouseEvent e) {
        saySomething("Mouse button press", e);
    }
    public void mouseReleased(MouseEvent e) {
        saySomething("Mouse button release", e);
    }
    public void mouseEntered(MouseEvent e) {
        saySomething("Cursor enter", e);
    }
    public void mouseExited(MouseEvent e) {
        saySomething("Cursor exit", e);
    }
    public void mouseClicked(MouseEvent e) {
        saySomething("Mouse button click", e);
    }
}
```

```

    }
    void saySomething(String eventDescription, MouseEvent e) {
        textArea.append(eventDescription + " detected on "
            + e.getComponent().getClass().getName()
            + ".\n");
        textArea.setCaretPosition(maxInt); //scroll to bottom
    }
}

```

Lớp MouseEvent

Mỗi phương thức Mouse Event có một thông số đơn : đối tượng MouseEvent. Lớp MouseEvent định nghĩa các phương thức thường dùng sau:

int getClickCount()

Trả về số lần nhấn liên tiếp của người sử dụng.

int getX()

int getY()

Point getPoint()

Trả về vị trí của con trỏ chuột, vị trí này phụ thuộc vào thành phần.

boolean isPopupTrigger()

Trả về giá trị True khi sự kiện này làm xuất hiện Popup Menu.

g) Viết một Mouse Motion Listener

Các sự kiện Mouse motion phát ra khi người sử dụng dùng chuột di chuyển trên màn hình.

Các phương thức, sự kiện của Mouse Motion

Giao diện MouseMotionListener và lớp mô phỏng tương ứng , MouseMotionAdapter, chứa hai phương thức:

void mouseDragged(MouseEvent)

Được gọi sau khi người sử dụng di chuyển chuột trong khi chuột đang được nhấn.

void mouseMoved(MouseEvent)

Được gọi sau khi người sử dụng di chuyển chuột khi con chuột chưa bị nhấn.

Ví dụ về Handling Mouse Motion Events

```

...//where initialization occurs:
MyListener myListener = new MyListener();
addMouseListener(myListener);

```

```
addMouseMotionListener(myListener);
...
class MyListener extends MouseAdapter
    implements MouseMotionListener {
public void mousePressed(MouseEvent e) {
    int x = e.getX();
    int y = e.getY();
    currentRect = new Rectangle(x, y, 0, 0);
    repaint();
}
public void mouseDragged(MouseEvent e) {
    updateSize(e);
}
public void mouseMoved(MouseEvent e) {
    //Do nothing.
}
public void mouseReleased(MouseEvent e) {
    updateSize(e);
}
void updateSize(MouseEvent e) {
int x = e.getX();
    int y = e.getY();
    currentRect.setSize(x - currentRect.x,
                        y - currentRect.y);
    repaint();
}
}
```

Các phương thức, sự kiện được sử dụng bởi Mouse-Motion Listeners

Mỗi phương thức Mouse Motion Event có một thông số đơn , và nó không được gọi là MouseMotionEvent! Thay vào đó , phương thức Mouse Motion Event sử dụng đối tượng MouseEvent.

h) Viết một Text Listener

Các sự kiện Text trả về sau khi chuỗi trong thành phần Text có sự thay đổi .

Các phương thức, sự kiện của Text

Giao diện TextListener chỉ có một phương thức nên không có lớp mô phỏng tương ứng:

void textValueChanged(TextEvent)

được gọi sau khi chuỗi trong thành phần Text thay đổi.

Ví dụ về Handling Text Events

```
public class TextEventDemo ... {
    TextField textField;
    TextArea textArea;
    TextArea displayArea;
    ...
//where initialization occurs:
    textField = new TextField(20);
    ...
    textField.addTextListener(new MyTextListener("Text Field"));

    textArea = new TextArea(5, 20);
    textArea.addTextListener(new MyTextListener("Text Area"));
    ...
}
class MyTextListener implements TextListener {
    String preface;
    public MyTextListener(String source) {
        preface = source
            + " text value changed.\n"
            + " First 10 characters: \";
    }
    public void textValueChanged(TextEvent e) {
        TextComponent tc = (TextComponent)e.getSource();
        String s = tc.getText();
        ...//truncate s to 10 characters...

        displayArea.append(preface + s + "\"\n");
        ...
    }
}
...
}
```

Lớp TextEvent

Mỗi phương thức Text Event có một thông số đơn : đối tượng TextEvent. Lớp TextEvent định nghĩa một phương thức. Phương thức getSource mà TextEvent thừa kế từ EventObject, bạn có thể nhận được thành phần Text liên quan đến sự kiện này và gửi thông điệp cho nó.

i) Viết một Window Listener

Các sự kiện của Window được phát ra sau khi Window mở, đóng, thu nhỏ, phóng to, hoạt động và không hoạt động.

Các phương thức, sự kiện của Window

Giao diện WindowListener và lớp mô phỏng tương ứng, WindowAdapter, chứa các phương thức sau:

void windowOpened(WindowEvent)

Được gọi sau khi Window được mở lần đầu.

void windowClosing(WindowEvent)

Được gọi sau khi người sử dụng đóng Window.

void windowClosed(WindowEvent)

Được gọi sau khi Window đóng lại.

void windowIconified(WindowEvent)

void windowDeiconified(WindowEvent)

Được gọi sau khi Window phóng to hay thu nhỏ.

void windowActivated(WindowEvent)

void windowDeactivated(WindowEvent)

Được gọi sau khi Window hoạt động hay không hoạt động.

Ví dụ về Handling Window Events

```
public class WindowEventDemo ... implements WindowListener {
...//where initialization occurs:
    //Create but don't show window.
    window = new Frame("Window Event Window");
    window.addWindowListener(this);
    window.add("Center",
        new Label("The applet listens to this window"
            " for window events."));
    window.pack();
}
```

```
public void windowClosing(WindowEvent e) {
    window.setVisible(false);
    displayMessage("Window closing", e);
}
public void windowClosed(WindowEvent e) {
    displayMessage("Window closed", e);
}
public void windowOpened(WindowEvent e) {
    displayMessage("Window opened", e);
}
public void windowIconified(WindowEvent e) {
    displayMessage("Window iconified", e);
}
public void windowDeiconified(WindowEvent e) {
    displayMessage("Window deiconified", e);
}
public void windowActivated(WindowEvent e) {
    displayMessage("Window activated", e);
}
public void windowDeactivated(WindowEvent e) {
    displayMessage("Window deactivated", e);
}
void displayMessage(String prefix, WindowEvent e) {
    display.append(prefix
        + ": "
        + e.getWindow()
        + newline);
}
...
}
```

Lớp WindowEvent

Mỗi phương thức Window Event có một thông số đơn : đối tượng WindowEvent. Lớp WindowEvent định nghĩa một phương thức, getWindow, trả về Window phát ra sự kiện này.