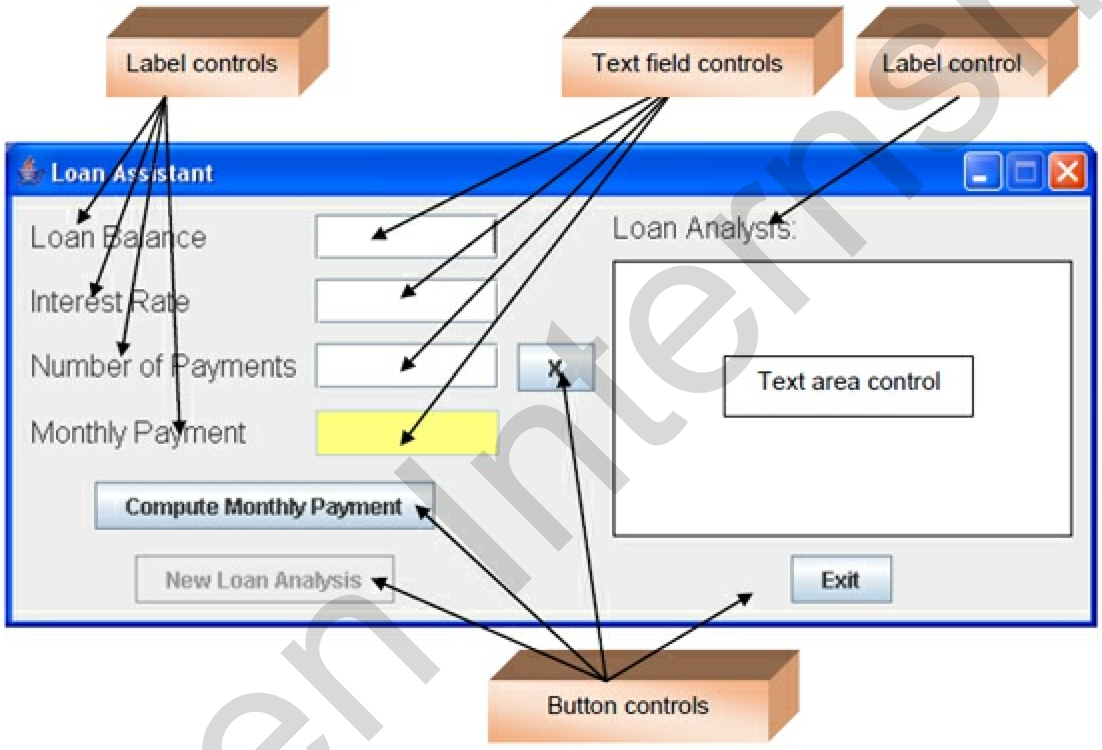
Consumer Loan Assistant Project Preview

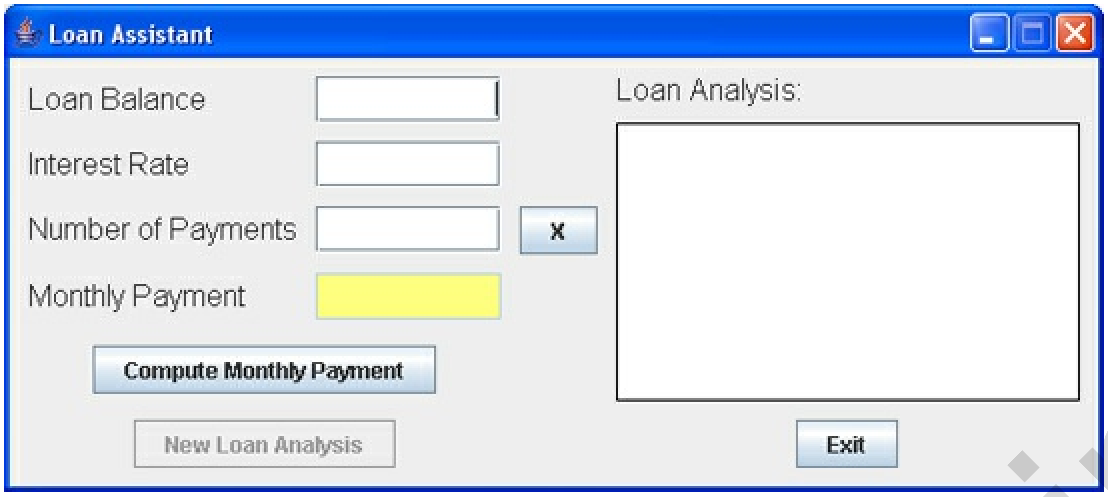
In this project, we will build a **consumer loan assistant**. You input a loan balance and yearly interest rate. You then have two options: (1) enter the desired number of payments and the loan assistant computes the monthly payment, or (2) enter the desired monthly payment and the loan assistant determines the number of payments you will make. An analysis of your loan, including total of payments and interest paid is also provided.

The finished project is saved as **LoanAssistant in** the **\HomeJava\HomeJava Projects** project group. Start NetBeans (or your IDE). Open the specified project group. Make **LoanAssistant** the main project. Run the project. You will see:

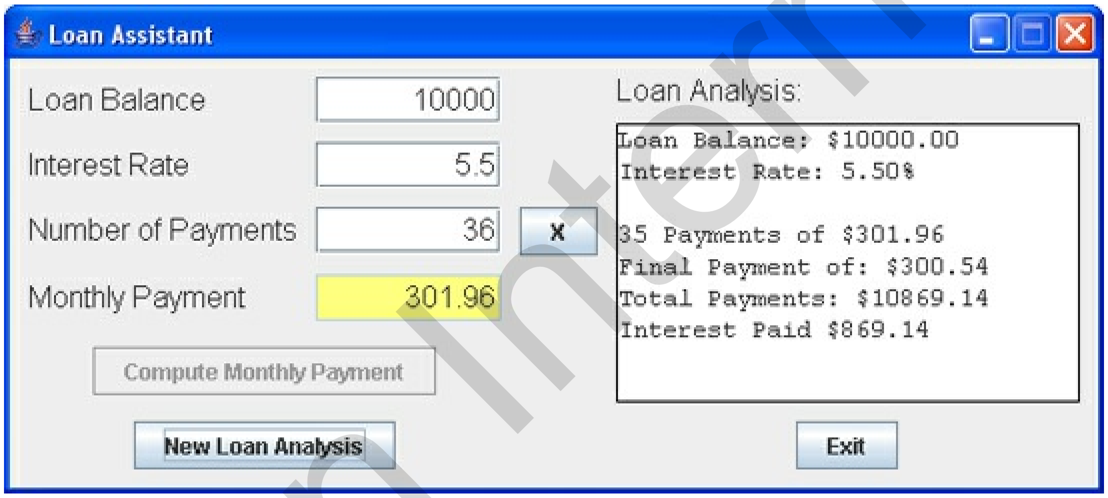


All label controls are used for title information. Two button controls are used to compute results and to start a new analysis. Two small button controls (marked with X; only one is seen at a time) control whether you compute the number of payments or the payment amount. One button exits the project. Four text field controls are used for inputs and a large text area is used to present the loan analysis results.

The loan assistant appears as:



In this initial configuration, you enter a **Loan Balance**, an **Interest Rate** (annual rate as a percentage) and a **Number of Payments** value. Click **Compute Monthly Payment**. The payment will appear in the ‘yellow’ text field and a complete loan analysis will appear in the large text field. Here are some numbers I tried:



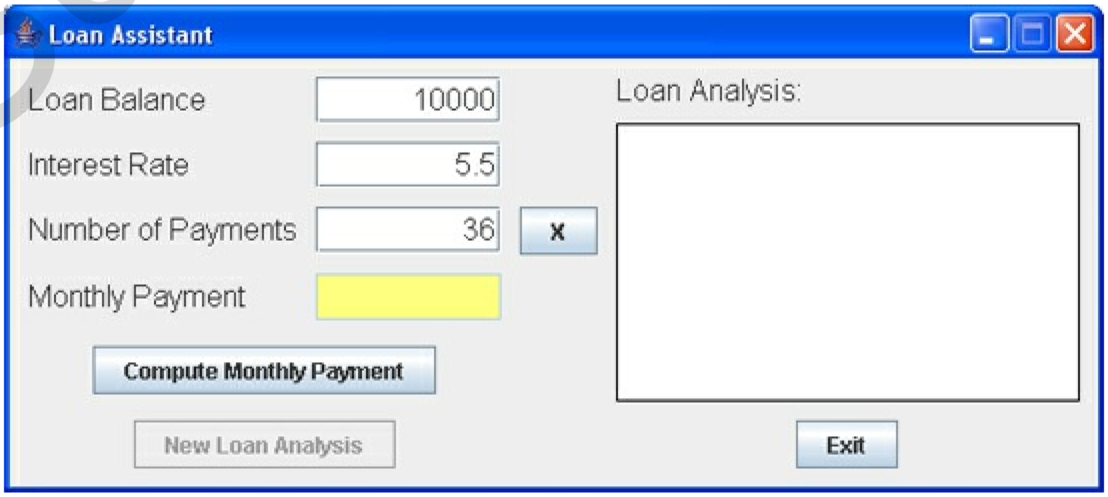
Loan Balance 10000
Interest Rate 5.5
Number of Payments 36
Monthly Payment 301.96

Loan Analysis:

```
Loan Balance: $10000.00
Interest Rate: 5.50%

35 Payments of $301.96
Final Payment of: $300.54
Total Payments: $10869.14
Interest Paid $869.14
```

So, if I borrow $10,000 at 5.5% interest, I will pay $301.96 for three years (36 months). More specific details on exact payment amounts, including total interest paid, is shown under **Loan Analysis**.

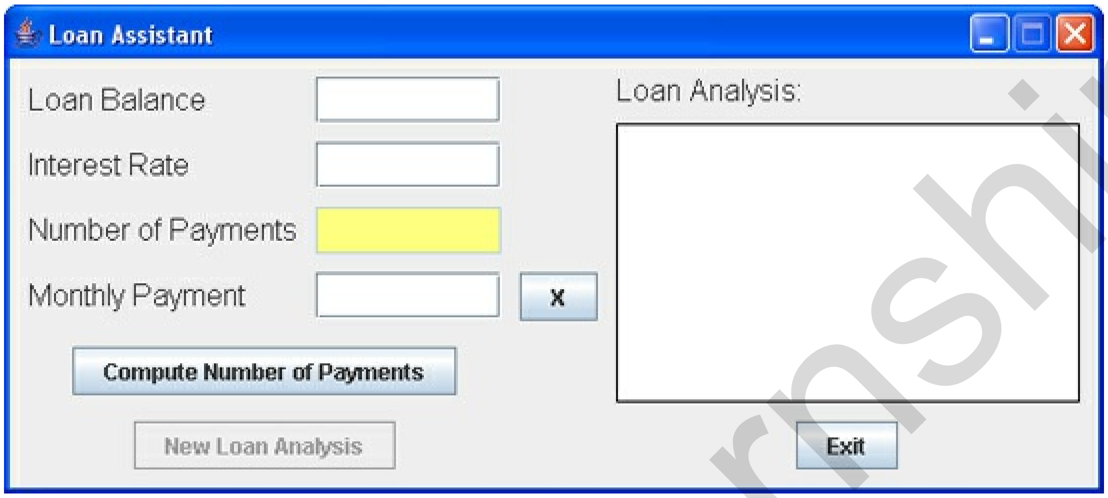
At this point, you can click **New Loan Analysis** to try some new values:



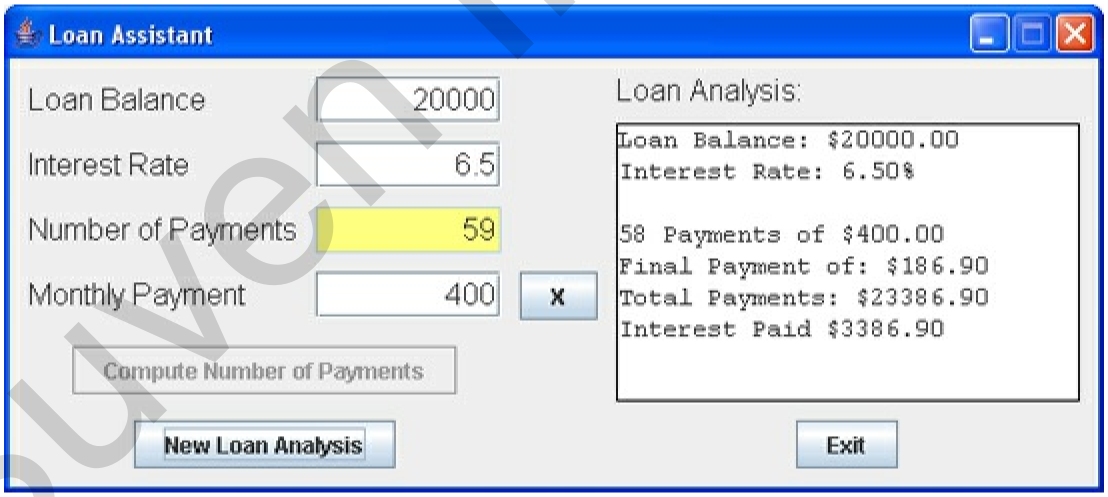
Loan Balance 10000
Interest Rate 5.5
Number of Payments 36

Note the **Loan Balance**, **Interest Rate**, and **Number of Payments** entries remain. Only the **Monthly Payment** and the **Loan Analysis** have been cleared. This lets you try different values with minimal typing of new entries. Change any entry you like to see different results – or even change them all. Try as many combinations as you like.

At some point, clear the text fields and click the button with an **X** next to the **Number of Payments** text field. You will see:



Notice the **Number of Payments** box is now yellow. The button with an **X** has moved to the **Monthly Payment** text field. In this configuration, you enter a **Loan Balance**, an **Interest Rate** and a **Monthly Payment**. The loan assistant will determine how many payments you need to pay off the loan. Here are some numbers I tried:



Loan Balance 20000
Interest Rate 6.5
Number of Payments 59
Monthly Payment 400

Loan Analysis:

```
Loan Balance: $20000.00
Interest Rate: 6.50%

58 Payments of $400.00
Final Payment of: $186.90
Total Payments: $23386.90
Interest Paid $3386.90
```

It will take 59 payments (the last one is smaller) to pay off this particular loan. Again, you can click **New Loan Analysis** to try other values and see the results.

That's all you do with the loan assistant project – there's a lot going on behind the scenes though. The loan assistant has two modes of operation. It can compute the monthly payment, given the balance, interest and number of payments. Or, it can compute the number of payments, given the balance, interest, and payment. The text field representing the computed value is yellow. The button marked **X** is used to switch from one mode to the next. To exit the project, click the **Exit** button.

You will now build this project in several stages. We first address **frame design**. We discuss the controls used to build the form, establish initial properties, and discuss switching from one mode to the next. And, we address **code design** in detail. We cover the mathematics behind the financial computations. We also discuss validation of the input values, making sure the user only types valid entries.

Loan Assistant Frame Design

We begin building the **Loan Assistant Project.** Let's build the frame. Start a new project in your Java project group – name it **LoanAssistant**. Delete default code in file named **LoanAssistant.java**. Once started, we suggest you immediately save the project with the name you chose. This sets up the folder and file structure needed for your project. Build the basic frame with these properties:

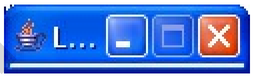
LoanAssistant Frame:

title	Loan Assistant
resizable	false

The code is:

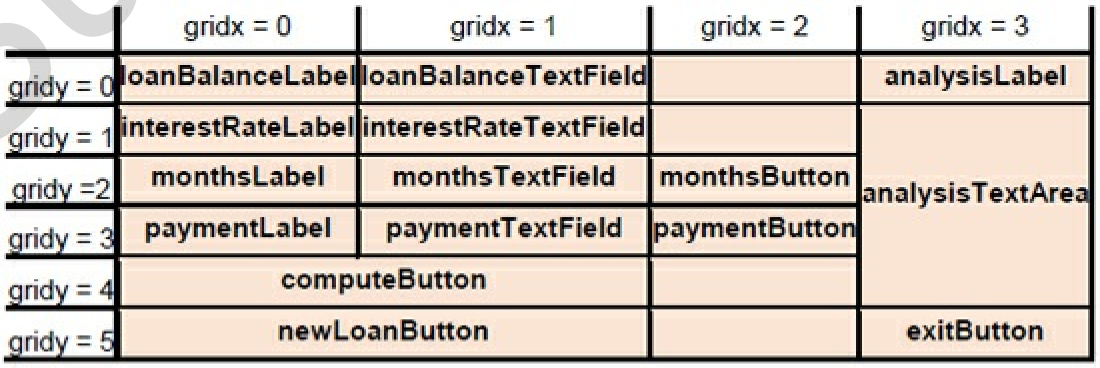
```
/ *
* LoanAssistant.java
*/
package loanassistant;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class LoanAssistant extends JFrame
{
    public static void main(String args[])
    {
        // create frame
        new LoanAssistant().show();
    }

    public LoanAssistant()
    {
        // frame constructor
        setTitle("Loan Assistant");
        setResizable(false);
        addWindowListener(new WindowAdapter()
        {
            public void windowClosing(WindowEvent evt)
            {
                exitForm(evt);
            }
        });
        getContentPane().setLayout(new GridBagLayout());
        GridBagConstraints gridConstraints;

        pack();
        Dimension screenSize =
Toolkit.getDefaultToolkit().getScreenSize();
        setBounds((int) (0.5 * (screenSize.width - getWidth())), (int) (0.5 * (screenSize.height
- getHeight())), getWidth(), getHeight());
    }

    private void exitForm(WindowEvent evt)
    {
        System.exit(0);
    }
}
```

This code builds the frame, sets up the layout manager and includes code to exit the application. Run the code to make sure the frame (at least, what there is of it at this point) appears and is centered on the screen:



Let's populate our frame with other controls. All code for creating the frame and placing controls (except declarations) goes in the **LoanAssistant** constructor.

All controls go directly on the frame. The **GridBagLayout** for the frame is:



	gridx = 0	gridx = 1	gridx = 2	gridx = 3
gridy = 0	loanBalanceLabel	loanBalanceTextField		analysisLabel
gridy = 1	interestRateLabel	interestRateTextField		analysisTextArea
gridy =2	monthsLabel	monthsTextField	monthsButton	
gridy = 3	paymentLabel	paymentTextField	paymentButton	
gridy = 4	computeButton			
gridy = 5	newLoanButton			exitButton

The label controls (**loanBalanceLabel**, **interestRateLabel**, **monthsLabel**, **paymentLabel**, **analysisLabel**) are used for title information. Four text fields (**loanBalanceTextField**,

interestRateTextField, **monthsTextField**, **paymentTextField**) are for user input. A text area (**analysisTextArea**) will display the loan analysis. Three buttons (**computeButton**, **newLoanButton** and **exitButton**) are used to compute loan results, redo analysis, and/or exit the project. Two other button controls (**monthsButton** and **paymentButton**) are used to switch from one calculation mode to the next. We'll add a few controls at a time. Let's add the four label/text field pairs.

The control properties are:

balanceLabel:

text	Loan Balance
font	Arial, Plain, Size 16
gridx	0
gridy	0
anchor	WEST
insets	10, 10, 0, 0

balanceTextField:

size	100, 25
font	Arial, Plain, Size 16
gridx	1
gridy	0
insets	10, 10, 0, 10

interestLabel:

text	Interest Rate
font	Arial, Plain, Size 16
gridx	0
gridy	1
anchor	WEST
insets	10, 10, 0, 0

interestTextField:

size	100, 25
font	Arial, Plain, Size 16
gridx	1
gridy	1
insets	10, 10, 0, 10

monthsLabel:

text	Number of Payments
font	Arial, Plain, Size 16
gridx	0
gridy	2
anchor	WEST
insets	10, 10, 0, 0

monthsTextField:

size	100, 25
font	Arial, Plain, Size 16
gridx	1
gridy	2
insets	10, 10, 0, 10

paymentLabel:

text	Monthly Payent
font	Arial, Plain, Size 16
gridx	0
gridy	3
anchor	WEST
insets	10, 10, 0, 0

paymentTextField:

size	100, 25
font	Arial, Plain, Size 16
gridx	1
gridy	3
insets	10, 10, 0, 10

Declare these controls using:

```
JLabel balanceLabel = new JLabel();
JTextField balanceTextField = new JTextField();
JLabel interestLabel = new JLabel();
JTextField interestTextField = new JTextField();
JLabel monthsLabel = new JLabel();
JTextField monthsTextField = new JTextField();
```

```
JLabel paymentLabel = new JLabel();
JTextField paymentTextField = new JTextField();
```

Note the labels and text fields all use the same font. Let’s create a Font object to use in each:

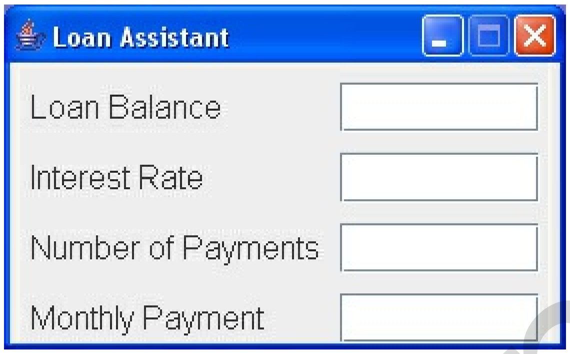
```
Font myFont = new Font("Arial", Font.PLAIN, 16);
```

Now, the controls are added to the frame using (recall code goes in frame constructor):

```
balanceLabel.setText("Loan Balance");
balanceLabel.setFont(myFont);
gridConstraints = new GridBagConstraints();
gridConstraints.gridx = 0;
gridConstraints.gridy = 0;
gridConstraints.anchor = GridBagConstraints.WEST;
gridConstraints.insets = new Insets(10, 10, 0, 0);
getContentPane().add(balanceLabel, gridConstraints);

balanceTextField.setPreferredSize(new Dimension(100, 25));
balanceTextField.setHorizontalAlignment(SwingConstants.RIGHT);
balanceTextField.setFont(myFont);
gridConstraints = new GridBagConstraints();
gridConstraints.gridx = 1;
gridConstraints.gridy = 0;
gridConstraints.insets = new Insets(10, 10, 0, 10);
getContentPane().add(balanceTextField, gridConstraints);

interestLabel.setText("Interest Rate");
interestLabel.setFont(myFont);
gridConstraints = new GridBagConstraints();
gridConstraints.gridx = 0;
gridConstraints.gridy = 1;
gridConstraints.anchor = GridBagConstraints.WEST;
gridConstraints.insets = new Insets(10, 10, 0, 0);
getContentPane().add(interestLabel, gridConstraints);

interestTextField.setPreferredSize(new Dimension(100, 25));
interestTextField.setHorizontalAlignment(SwingConstants.RIGHT);
interestTextField.setFont(myFont);
gridConstraints = new GridBagConstraints();
gridConstraints.gridx = 1;
gridConstraints.gridy = 1;
gridConstraints.insets = new Insets(10, 10, 0, 10);
getContentPane().add(interestTextField, gridConstraints);

monthsLabel.setText("Number of Payments");
monthsLabel.setFont(myFont);
gridConstraints = new GridBagConstraints();
gridConstraints.gridx = 0;
gridConstraints.gridy = 2;
gridConstraints.anchor = GridBagConstraints.WEST;
gridConstraints.insets = new Insets(10, 10, 0, 0);
getContentPane().add(monthsLabel, gridConstraints);

monthsTextField.setPreferredSize(new Dimension(100, 25));
monthsTextField.setHorizontalAlignment(SwingConstants.RIGHT);
monthsTextField.setFont(myFont);
gridConstraints = new GridBagConstraints();
gridConstraints.gridx = 1;
gridConstraints.gridy = 2;
gridConstraints.insets = new Insets(10, 10, 0, 10);
getContentPane().add(monthsTextField, gridConstraints);

paymentLabel.setText("Monthly Payment");
paymentLabel.setFont(myFont);
gridConstraints = new GridBagConstraints();
gridConstraints.gridx = 0;
gridConstraints.gridy = 3;
gridConstraints.anchor = GridBagConstraints.WEST;
gridConstraints.insets = new Insets(10, 10, 0, 0);
getContentPane().add(paymentLabel, gridConstraints);

paymentTextField.setPreferredSize(new Dimension(100, 25));
paymentTextField.setHorizontalAlignment(SwingConstants.RIGHT);
paymentTextField.setFont(myFont);
```

```
gridConstraints = new GridBagConstraints();
gridConstraints.gridx = 1;
gridConstraints.gridy = 3;
gridConstraints.insets = new Insets(10, 10, 0, 10);
getContentPane().add(paymentTextField, gridConstraints);
```

Save, run the project. You will see the added controls:



Let's add the two button controls that go under these controls. The properties are:

computeButton:

text	Compute Monthly Payments
gridx	0
gridy	4
gridwidth	2
insets	10, 0, 0, 0

newLoanButton:

text	New Loan Analysis
enabled	false
gridx	0
gridy	5
gridwidth	2
insets	10, 0, 10, 0

Declare the controls using:

```
JButton computeButton = new JButton();
JButton newLoanButton = new JButton();
```

Add the buttons to the frame using:

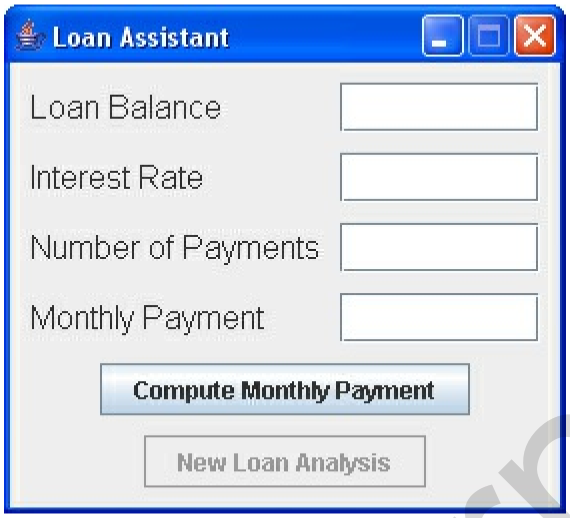
```
computeButton.setText("Compute Monthly Payment");
gridConstraints = new GridBagConstraints();
gridConstraints.gridx = 0;
gridConstraints.gridy = 4;
gridConstraints.gridwidth = 2;
gridConstraints.insets = new Insets(10, 0, 0, 0);
getContentPane().add(computeButton, gridConstraints);
computeButton.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        computeButtonActionPerformed(e);
    }
});

newLoanButton.setText("New Loan Analysis");
newLoanButton.setEnabled(false);
gridConstraints = new GridBagConstraints();
gridConstraints.gridx = 0;
gridConstraints.gridy = 5;
gridConstraints.gridwidth = 2;
gridConstraints.insets = new Insets(10, 0, 10, 0);
getContentPane().add(newLoanButton, gridConstraints);
newLoanButton.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        newLoanButtonActionPerformed(e);
    }
});
```

This code also adds listeners for each button. Add these empty methods:

```
private void computeButtonActionPerformed(ActionEvent e)
{
}
```

```
private void newLoanButtonActionPerformed(ActionEvent e)
{
}
```

Run to see the buttons (the **New Loan Analysis** button is disabled):



Now we add the two small button controls that go next to two of the text fields. The properties are:

monthsButton:

text	X
gridx	2
gridy	2
insets	10, 0, 0, 0

paymentButton:

text	X
enabled	false
gridx	0
gridy	5
gridwidth	2
insets	10, 0, 10, 0

Declare the controls using:

```
JButton monthsButton = new JButton();
JButton paymentButton = new JButton();
```

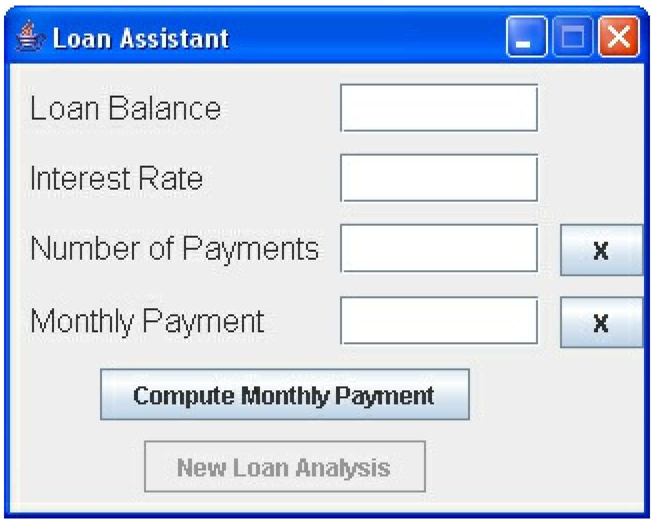
Add the buttons to the frame using:

```
monthsButton.setText("X");
gridConstraints = new GridBagConstraints();
gridConstraints.gridx = 2;
gridConstraints.gridy = 2;
gridConstraints.insets = new Insets(10, 0, 0, 0);
getContentPane().add(monthsButton, gridConstraints);
monthsButton.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        monthsButtonActionPerformed(e);
    }
});

paymentButton.setText("X");
gridConstraints = new GridBagConstraints();
gridConstraints.gridx = 2;
gridConstraints.gridy = 3;
gridConstraints.insets = new Insets(10, 0, 0, 0);
getContentPane().add(paymentButton, gridConstraints);
paymentButton.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        paymentButtonActionPerformed(e);
    }
});
```

This code also adds listeners for each button. Add these empty methods:

```
private void monthsButtonActionPerformed(ActionEvent e)
{
}

private void paymentButtonActionPerformed(ActionEvent e)
{
}
```

Run to see the buttons:



Both X buttons appear now. When we write code, only one of the buttons will display at a time.

Let's finish the frame by adding the three remaining controls (label, text field and button). The properties are:

analysisLabel:

text	Loan Analysis:
font	Arial, Plain, Size 16
gridx	3
gridy	0
anchor	WEST
insets	0, 10, 0, 0

analysisTextArea:

size	250, 150
border	Black line
font	Courier New, Plain, Size 14
editable	false
background	White
gridx	3
gridy	1
gridheight	4
insets	0, 10, 0, 10

exitButton:

text	Exit
gridx	3

```
gridy                    5
```

Declare the controls using:

```
JLabel analysisLabel = new JLabel();
JTextArea analysisTextArea = new JTextArea();
JButton exitButton = new JButton();
```

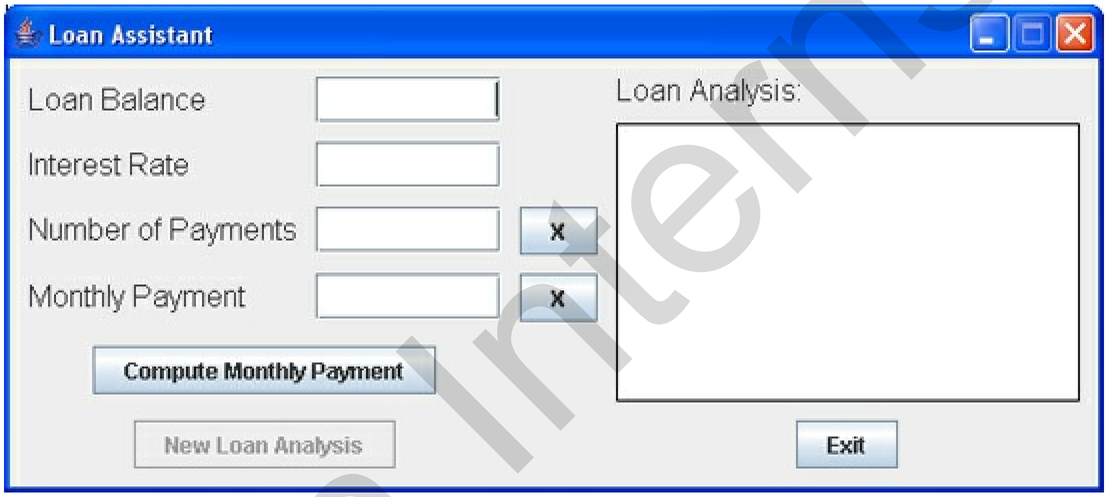
Add the controls to the frame using:

```
analysisLabel.setText("Loan Analysis:");
analysisLabel.setFont(myFont);
gridConstraints = new GridBagConstraints();
gridConstraints.gridx = 3;
gridConstraints.gridy = 0;
gridConstraints.anchor = GridBagConstraints.WEST;
gridConstraints.insets = new Insets(0, 10, 0, 0);
getContentPane().add(analysisLabel, gridConstraints);

analysisTextArea.setPreferredSize(new Dimension(250, 150));
analysisTextArea.setBorder(BorderFactory.createLineBorder(Color.BLACK));
analysisTextArea.setFont(new Font("Courier New", Font.PLAIN, 14));
analysisTextArea.setEditable(false);
analysisTextArea.setBackground(Color.WHITE);
gridConstraints = new GridBagConstraints();
gridConstraints.gridx = 3;
gridConstraints.gridy = 1;
gridConstraints.gridheight = 4;
gridConstraints.insets = new Insets(0, 10, 0, 10);
getContentPane().add(analysisTextArea, gridConstraints);

exitButton.setText("Exit");
gridConstraints = new GridBagConstraints();
gridConstraints.gridx = 3;
gridConstraints.gridy = 5;
getContentPane().add(exitButton, gridConstraints);
exitButton.addActionListener(new ActionListener()
{
        public void actionPerformed(ActionEvent e)
        {
            exitButtonActionPerformed(e);
        }
    });
```

This code also adds listeners for each button. Add these empty methods:

```
    private void exitButtonActionPerformed(ActionEvent e)
    {
    }
```

Run to see the final frame layout:



This completes the initial form. We will begin writing code for the application. We will write the code in several steps. As a first step, we write the code that switches the application between its two possible modes of operation: (1) compute monthly payment, or (2) compute number of payments.

Code Design – Switching Modes

There are two modes the loan assistant can operate in. In the first mode, you enter a loan balance, an interest rate and a number of payments. The assistant then computes the monthly payment. In the second mode, you enter a loan balance, an interest rate and a monthly payment. The assistant computes the number of payments. The buttons with **X** control which mode the assistant operates in. Click the **X** (**paymentButton**) next to the payment text field and you switch to the first mode (compute monthly payment). Click the **X** (**monthsButton**) next to the number of payments text field and you switch to the second mode (compute number of payments). Let's look at the steps for each operation.

When the user clicks the **X** next to the monthly payment text field (**paymentButton** button), we want to make **paymentTextField** available for user input and **monthsTextField** available for output. The steps are taken:

- Make **paymentButton** disappear.
- Make **monthsButton** appear.
- Set **enabled** property of **monthsTextField** to **false**.
- Set **monthsTextField** background to **White**.
- Blank out the **paymentTextField** text field.
- Set **enabled** property of **paymentTextField** to **true**.
- Set **paymentTextField** background to **Light Yellow**.
- Set **text** property of **computeButton** to **Compute Monthly Payment**.

When you click the **X** next to the number of payments text field (**monthsButton** button), we essentially 'reverse' the steps just listed:

- Make **paymentButton** appear.
- Make **monthsButton** disappear.
- Set **enabled** property of **monthsTextField** to **true**.
- Set **monthsTextField** background to **Light Yellow**.
- Blank out the **monthsTextField** text field.
- Set **enabled** property of **paymentTextField** to **false**.
- Set **paymentTextField** background to **White**.
- Set **text** property of **computeButton** to **Compute Number of Payments**.

Define a class level object to define a 'Light Yellow' color and a variable to keep track of what mode we are working in:

```
Color lightYellow = new Color(255, 255, 128);
boolean computePayment;
```

If **computePayment** is **true**, we are computing the payment, otherwise we are computing the number of payments.

The code for the **paymentButtonActionPerformed** method that implements the listed steps is then:

```
private void paymentButtonActionPerformed(ActionEvent e)
{
    // will compute payment
    computePayment = true;
    paymentButton.setVisible(false);
    monthsButton.setVisible(true);
    monthsTextField.setEditable(true);
    monthsTextField.setBackground(Color.WHITE);
    paymentTextField.setText("");
    paymentTextField.setEditable(false);
    paymentTextField.setBackground(lightYellow);
    computeButton.setText("Compute Monthly Payment");
}
```

The code for the **monthsButton Click** is:

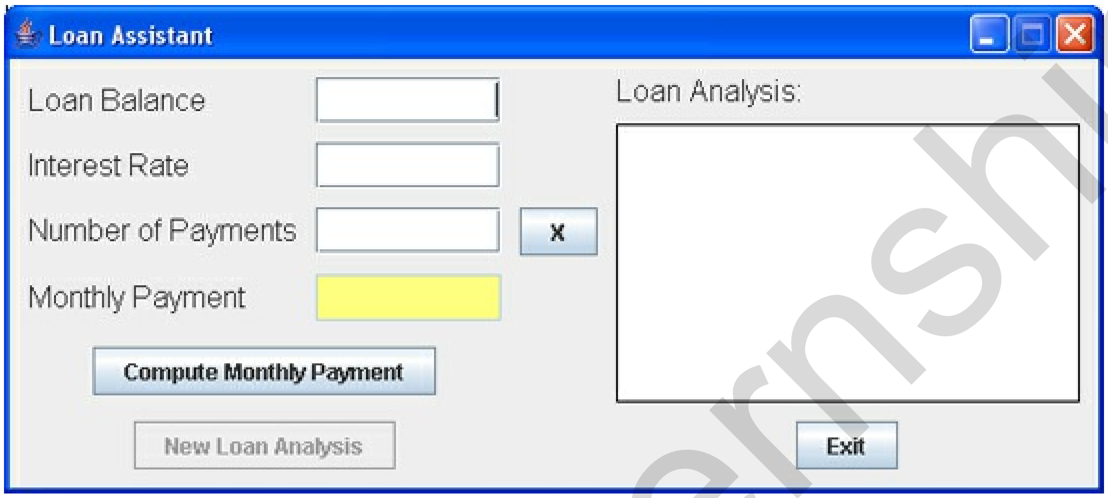
```
private void monthsButtonActionPerformed(ActionEvent e)
{
    // will compute months
    computePayment = false;
    paymentButton.setVisible(true);
    monthsButton.setVisible(false);
    monthsTextField.setText("");
    monthsTextField.setEditable(false);
    monthsTextField.setBackground(lightYellow);
    paymentTextField.setEditable(true);
    paymentTextField.setBackground(Color.WHITE);
    computeButton.setText("Compute Number of Payments");
}
```

We would like the application to begin in the mode where the monthly payment is computed. One way we could do this is by setting properties in design mode that correspond to the properties listed in the

paymentButtonActionPerformed method. But an easier approach is to have the application 'simulate' clicking on the **paymentButton** button when the application begins. This is done at the end of the frame constructor with

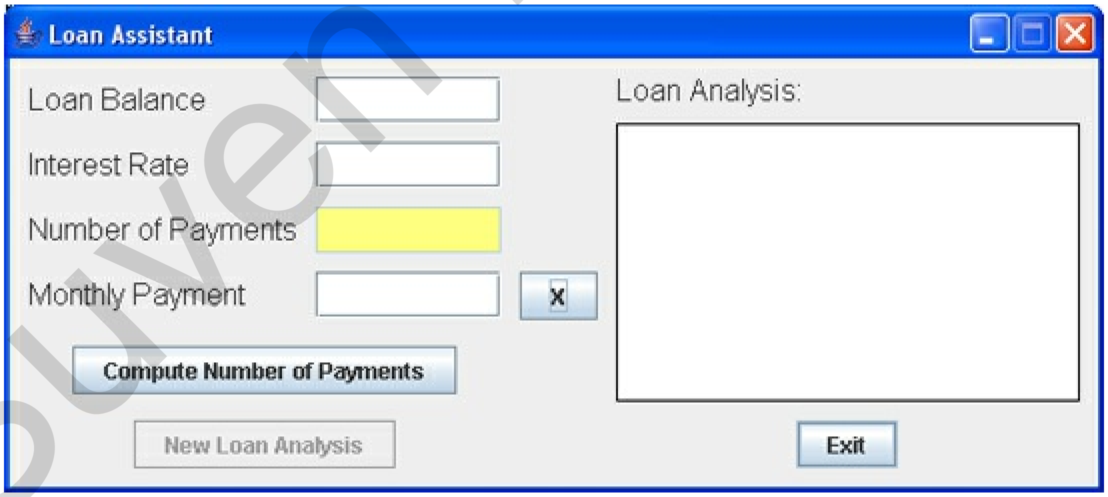
paymentButton.doClick();

Save and run the project. If the code is entered correctly, the form should appear in the 'compute payment' mode:



Note the **Monthly Payment** box is yellow, as desired. The **computeButton** caption is **Compute Monthly Payment**.

Click the **X** next to **Number of Payments** and you switch to the 'compute number of payments' mode:



Now, the **Number of Payments** box is yellow and the **computeButton** caption is **Compute Number of Payments**.

The mode switching should be working correctly. Before writing the code behind the actual computations, let's address interface issues of focus traversal and control focus.

Frame Design – Focus Traversal

When you run the loan assistant application, if you try to move from text field to text field using the <**Tab**> key, there may be no predictable order in how the cursor moves. Or, you may move to controls you don't want to move to (for example, a read-only text field). To enter values, you have to make sure you first click in the text field. To make this process more orderly, we need to look at something called **focus traversal**.

When interacting with a Java GUI application, we can work with a single control at a time. That is, we can click on a single button or type in a single text field. We can't be doing two things at once. The control we are working with is known as the **active** control or we say the control has **focus**. In our loan assistant example, when the cursor is in a particular text field, we say that text field has focus. In a properly designed application, focus is shifted from one control to another (in a predictable, orderly fashion) using the <**Tab**> key. Focus can only be given to controls that allow user interaction – buttons and text fields in our example, but not labels.

Java does a good job of defining an orderly tab sequence using something called the **FocusTransversalPolicy**. Essentially, the tab sequence starts in the upper left corner of the **GridBagLayout** and works its way across a row. It then moves down to the next row and continues until it reaches the last column of the last row. At that point, the sequence begins again. The process can be reversed using <**Tab**>in combination with the <**Shift**> key.

There are times you would like to remove a control from the tab sequence (transversal policy). To remove a control (named **myControl**) from the sequence, use:

```
myControl.setFocusable(false);
```

It is also possible to reorder the tab sequence, but that is beyond the scope of this course.

There are several places we'd like to remove focus in the loan assistant project. When computing payment, there is no need for the focus to go to the **paymentTextField** control, since it is not editable. Similarly, when computing number of payments, focus should not go to the **monthsTextField**. We also want to keep focus from three button controls: **monthsButton**, **paymentButton** and **exitButton** to avoid inadvertent option selections or exiting of the program. And, focus never needs to go to **analysisTextArea**, since no editing is possible.

One other modification is needed. Once a user changes the computation mode, it would be nice if focus would be moved to the **Loan Balance** text field so the user can type an entry there. To programmatically assign focus to a control, apply the **requestFocus** method to the control using this dot-notation:

```
myControl.requestFocus();
```

Let’s implement the specified changes. Three button controls (**monthsButton**, **paymentButton** and **exitButton**) and the text area (**analysisTextArea**) are permanently removed from the tab sequence, hence modify the code that creates these controls (in the frame constructor) with the three shaded lines:

```
monthsButton.setText("X");
monthsButton.setFocusable(false);
gridConstraints = new GridBagConstraints();
      .
      .

paymentButton.setText("X");
paymentButton.setFocusable(false);
gridConstraints = new GridBagConstraints();
      .
      .

analysisTextArea.setPreferredSize(new Dimension(250, 150));
analysisTextArea.setFocusable(false);
analysisTextArea.setBorder(BorderFactory.createLineBorder(Color.BLACK));
      .
      .

exitButton.setText("Exit");
exitButton.setFocusable(false);
gridConstraints = new GridBagConstraints();
      .
      .
```

Modifications to the **paymentButton** and **monthsButton ActionPerformed** methods to modify text field traversal and to give focus to the balance text field are shown as shaded lines:

```
private void paymentButtonActionPerformed(ActionEvent e)
{
    // will compute payment
    computePayment = true;
    paymentButton.setVisible(false);
    monthsButton.setVisible(true);
    monthsTextField.setEditable(true);
    monthsTextField.setBackground(Color.WHITE);
    monthsTextField.setFocusable(true);
    paymentTextField.setText("");
    paymentTextField.setEditable(false);
    paymentTextField.setBackground(lightYellow);
    paymentTextField.setFocusable(false);
    computeButton.setText("Compute Monthly Payment");
    balanceTextField.requestFocus();
}

private void monthsButtonActionPerformed(ActionEvent e)
{
    // will compute months
    computePayment = false;
    paymentButton.setVisible(true);
    monthsButton.setVisible(false);
    monthsTextField.setText("");
    monthsTextField.setEditable(false);
    monthsTextField.setBackground(lightYellow);
    monthsTextField.setFocusable(false);
    paymentTextField.setEditable(true);
    paymentTextField.setBackground(Color.WHITE);
    paymentTextField.setFocusable(true);
    computeButton.setText("Compute Number of Payments");
    balanceTextField.requestFocus();
}
```

Save and run the project. Switch from mode to mode. Notice how in each mode, the tab ordering is now predictable and as desired. Notice how the focus is always on the desired control. Notice how focus ends up on the compute button, where a tap of the space bar can ‘click’ on that button..

Code Design – Computing Monthly Payment

Let's develop the code to run the loan assistant in its initial 'compute payment' mode. We need an equation that computes the payment, knowing the loan balance, the interest rate and the number of payments. Computer programming is many times mathematical in nature. I recognize different people have different comfort levels with math. For those "math-phobes" out there, I'll just give you the code. For those interested, I'll show you the math behind the code.

Here's the code that does the necessary computations. In these lines, **balance** (**double** type) is the entered loan balance, **interest** (**double** type) is the entered interest rate and **months** (**int** type) is the entered number of payments (each of these values will come from the text field controls):

```
multiplier = Math.pow(1 + monthlyInterest, months);
payment = balance * monthlyInterest * multiplier / (multiplier - 1);
```

In this code, the input interest (a yearly percentage) is converted to a monthly interest (**monthlyInterest**). This conversion is done by dividing by 12 (the number of months in a year) times 100 (to convert percentage to a decimal number). A **multiplier** term is formed using the mathematical power (exponentiation) method (**pow**). These values are then used to compute **payment** (**double** type).

If you don't want to see mathematics, **stop now**!! Skip ahead to the code steps for the **computeButtonActionPerformed** method. If you're still with me, I'll go over the steps that derive the code above. Let B represent the initial loan balance, i the monthly interest and P the monthly payment (we'll be solving an equation for this value). With this notation, the product of B times i (Bi) represents one month's interest on the existing balance. We add this interest to the balance then subtract the payment to obtain the balance after one payment, B_1:

$$B_1 = B + Bi – P = B(1 + i) – P$$

Using the same approach, the balance after two payments (B_2) would be:

$$B_2 = B_1 + B_1i – P = B_1(1 + i) – P$$

Substituting the previous equation for B_1 into this equation gets things in terms of the original balance:

$$B_2 = [B(1 + i) – P](1 + i) – P = B(1 + i)^2 – P(1 + i) - P$$

Doing the same for B_3, we can show:

$$B_3 = B(1 + i)^3 – P(1 + i)^2 – P(1 + i) - P$$

Noting the trend in this relation, we can obtain an expression for B_N (the balance after N payments, when the loan is finally paid off):

$$B_N = B(1 + i)^N - P \sum_{k=0}^{N-1} (1 + i)^k$$

The Greek sigma in the above equation simply indicates that you add up all the corresponding elements next to the sigma.

After N payments, we want the balance of the loan to be zero. If we set B_N to zero in the above equation, we obtain a value for P, the payment:

$$P = B(1 + i)^N / \sum_{k=0}^{N-1} (1 + i)^k$$

This is the desired result and we could easily code it using a for loop to evaluate the summation in the denominator. We can avoid this step by consulting a handbook on "finite series." The denominator term actually has a "closed-form" (one not requiring the summation). It is (trust me on this):

$$\sum_{k=0}^{N-1} (1 + i)^k = [(1 + i)^N - 1]/i$$

Try a few values of i and N to convince yourself this works (if you need convincing). Substituting this into the equation for P and flipping a few terms around gives us the final equation for computing P:

$$P = Bi(1 + i)^N / [(1 + i)^N - 1]$$

Compare this equation to the code we gave you. You should see the code matches this equation (**B** is **balance**, **i** is **monthlyInterest**, **N** is **months** and **P** is **payment**).

When the user clicks **Compute Monthly Payment** (**computeButton**), the following steps are taken:

- ➢ Obtain the **balance** value from user input.
- ➢ Obtain the **interest** value from user input.
- ➢ Determine monthly interest.
- ➢ Obtain the **months** value from user input.
- ➢ Compute **payment** using given code.
- ➢ Display **payment** in **paymentTextField**.

The **computeButtonActionPerformed** method that implements these steps are (note we have declared all variables to have method level scope):

```
private void computeButtonActionPerformed(ActionEvent e)
{
    double balance, interest, payment;
    int months;
    double monthlyInterest, multiplier;
    balance =
Double.valueOf(balanceTextField.getText()).doubleValue();
    interest =
Double.valueOf(interestTextField.getText()).doubleValue();
    monthlyInterest = interest / 1200;
    // Compute loan payment
    months =
Integer.valueOf(monthsTextField.getText()).intValue();
    multiplier = Math.pow(1 + monthlyInterest, months);
    payment = balance * monthlyInterest * multiplier / (multiplier - 1);
    paymentTextField.setText(new DecimalFormat("0.00").format(payment));
}
```

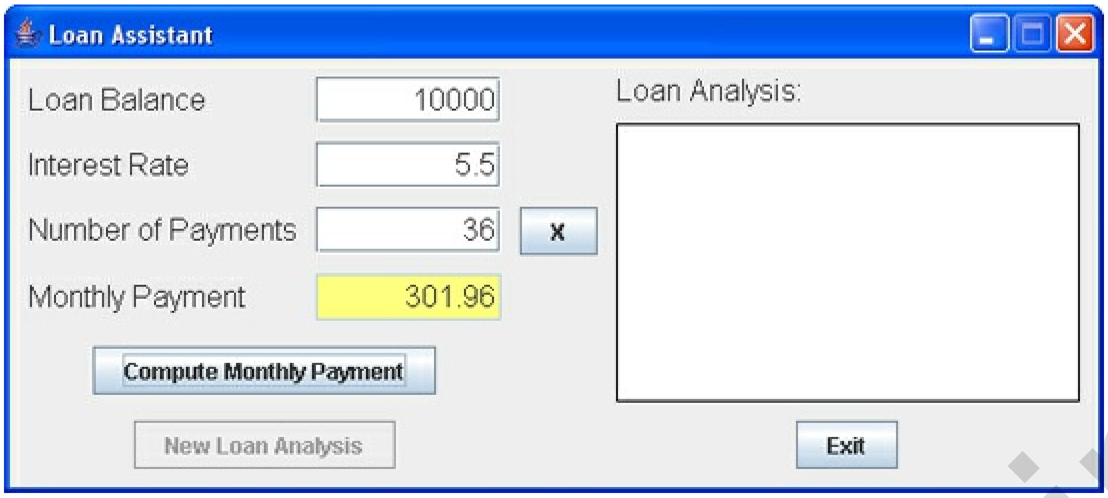
This method uses the Java **DecimalFormat** method (to assign to decimal points to the result). To use this, we need to add this import statement:

```
import java.text.*;
```

While we're at it, let's take care of the **Exit** button. Its **ActionPerformed** method is simply:

```
private void exitButtonActionPerformed(ActionEvent e)
{
    System.exit(0);
}
```

Save and run the project. Enter some numbers for balance, interest and number of payments, then click **Compute Monthly Payment**. Here's a run I made:



Loan Balance 10000
Interest Rate 5.5
Number of Payments 36
Monthly Payment 301.96
Loan Analysis:

A $10,000 loan at 5.5% yearly interest has a monthly payment of $301.96. Try as many possibilities as you'd like. Make sure **Exit** works.

Code Design – Computing Number of Payments

The second mode of operation for the loan assistant is 'compute number of payments' mode. We need an equation the computes the number of payments, knowing the loan balance, the interest rate and the monthly payment. Again, a bit of math is involved. And, again, for those interested, I'll show you the math behind the code.

Here's the code that does the necessary computations. In these lines, **balance** (**double** type) is the entered loan balance, **interest** (**double** type) is the entered interest rate and **payment** (**double** type) is the entered monthly payment (each of these values will come from the text field controls):

```
months = (int)((Math.log(payment) - Math.log(payment - balance * monthlyInterest)) /
Math.log(1 + monthlyInterest));
```

In this code, we again use the **monthlyInterest** value. The number of payments (**months**, an **int** type) is computed using the **Math.log** function. This is a mathematical logarithm.

All "math-phobes," skip ahead to the code to modify the **computeButtonActionPerformed** method. For those interested, let's see where that logarithm comes from. The equation we derived for the **Payment** (**P**) was:

$P = Bi(1 + i)^N / [(1 + i)^N – 1]$

where **B** is **Balance**, **i** is **MonthlyInterest**, and **N** is **Months**. In the current mode, we want to solve for N, given B, i, and P. Multiply both sides of the equation by the denominator on the right side to get:

$[(1 + i)^N – 1]P = Bi(1 + i)^N$

Multiply out the left side:

$(1 + i)^N P – P = Bi(1 + i)^N$

Then collect terms:

$(P – Bi)(1 + i)^N = P$

or

$(1 + i)^N = P / (P – Bi)$

Now, take the logarithm (hopefully you remember how these work) of both sides to yield:

Nlog(1 + i) = log(P) – log(P – Bi)

Or, solving for N, our desired result:

N = [log(P) – log(P – Bi)] / log(1 + i)

Look back at the code and you should see this equation. In the code, we cast the result to an **int** type (we can't make a fractional payment).

So when the user clicks **Compute Number of Payments** (**computeButton** when **ComputePayment** is **false**), the following steps are taken:

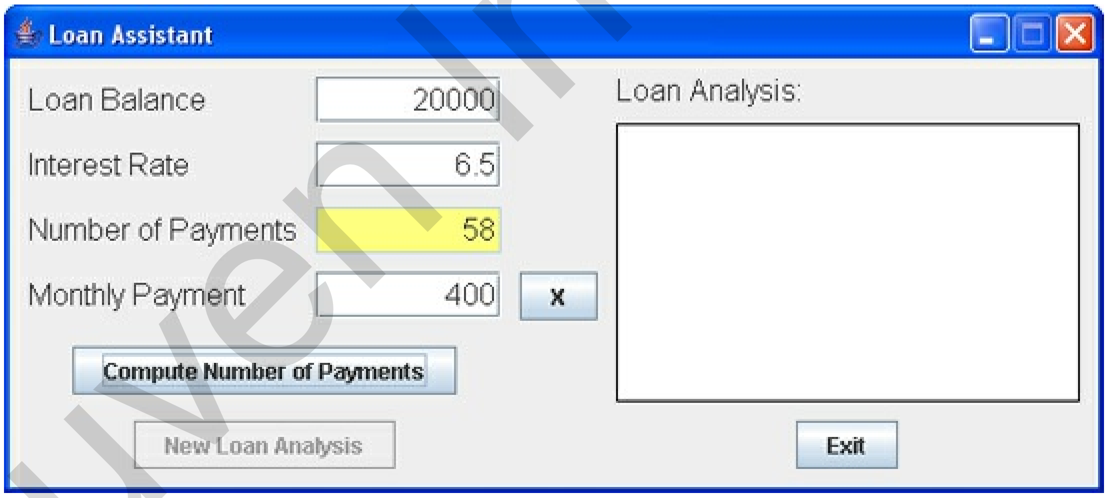
- ➢ Obtain the **balance** value from user input.
- ➢ Obtain the **interest** value from user input.
- ➢ Determine monthly interest.
- ➢ Obtain the **payment** value from user input.
- ➢ Compute **months** using given code.
- ➢ Display **months** in **monthsTextField**.

The modifed **computeButtonActionPerformed** method that implements these steps are (new code is shaded, notice we now look **ComputePayment** to see what 'mode' we are in):

```
private void computeButtonActionPerformed(ActionEvent e)
{
    double balance, interest, payment;
    int months;
    double monthlyInterest, multiplier;
    balance =
Double.valueOf(balanceTextField.getText()).doubleValue();
interest = Double.valueOf(interestTextField.getText()).doubleValue();

    monthlyInterest = interest / 1200;
    if (computePayment)
    {
        // Compute loan payment
        months =
Integer.valueOf(monthsTextField.getText()).intValue();
        multiplier = Math.pow(1 + monthlyInterest, months);
        payment = balance * monthlyInterest * multiplier / (multiplier - 1);
        paymentTextField.setText(new DecimalFormat("0.00").format(payment));
    }
    else
    {
        // Compute number of payments
        payment =
Double.valueOf(paymentTextField.getText()).doubleValue();
        months = (int)((Math.log(payment) - Math.log(payment - balance * monthlyInterest))
/ Math.log(1 + monthlyInterest));
        monthsTextField.setText(String.valueOf(months));
    }
}
```

Save and run the application. Make sure it still works in the initial mode for computing the monthly payment. When you're sure this is working okay, click the **X** next to the number of payment text field to switch to 'compute number of payments' mode.

Type in some values for balance, interest and payment. Click **Compute Number of Payments**. Here's a run I made:



Loan Balance: 20000
Interest Rate: 6.5
Number of Payments: 58
Monthly Payment: 400

This tells me if I borrow $20, 000 at 6.5% interest, I would need to make 58 monthly payments of $400 to pay the loan back. If you have a good memory (or look back earlier in this project), you'll remember we tried this when demonstrating the loan assistant project. In that earlier run, we obtained a value of 59 monthly payments. Is this a mistake? No – you'll see why next.

Code Design – Loan Analysis

Another desired feature of the consumer loan assistant project is to provide an analysis of the loan, once computations are done. The information this analysis should include is:

- ➢ Loan Balance
- ➢ Interest Rate
- ➢ Number of Payments
- ➢ Amount of Each Payment
- ➢ Total of Payments Made
- ➢ Total Interest Paid

Such information is very useful in analyzing how effective and economical a loan payoff plan is. Our frame has a text area control (**analysisTextArea**) available to provide these results. The analysis is generated after **computeButton** is clicked and the number of payments or payment amount have been computed.

At first, generating a loan analysis seems like a simple task. The balance (**balance**) and interest rate (**interest**) are input numbers. The number of payments (**months**) and monthly payment (**payment**) are either input or computed. So, it seems the total of payments would be given by:

totalPayments = months * payment;

while the interest paid would be:

interestPaid = totalPayments – balance;

The second equation is correct (assuming **totalPayments** is correct). But, the first equation (for **totalPayments**) doesn't quite apply. It's not that simple.

The code used for computing the payment amount (**computePayment** is **true**) and the number of payments (**computePayment** is **false**) is not exact. Truncation errors (making sure payments only have two decimal places) can affect the final payment amount. And, forcing the number of payments to be an integer value can result in significant errors in the final payment, perhaps even necessitating a final payment (remember the example we just ran?). We need to develop an analysis that recognizes the possibility of such errors and make necessary adjustments.

Here's the approach we will take. If the loan has N payments of P dollars, we will process all but the last payment and see what the remaining balance is at that point. If that balance is less than P, that will become the final payment. If that balance is greater than P, a payment of P will be applied and an additional payment of the final balance will be created. The displayed loan analysis will then show the final payment and the associated total of payments and interest paid.

For those of you who have avoided all the mathematical derivations up to this point, you need to know how to process a single payment to reduce the loan balance. If B is the current loan balance, i the monthly interest rate and P the payment. The balance (B_{after}) after the payment is:

$B_{after} = B + Bi – P$

This equation simply says the new balance is the old balance incremented by interest owed (Bi), then decreased by the payment amount (P). We use this equation to compute the final payment in the loan analysis.

The steps behind generating the loan analysis are:

➢ Display **balance**.
➢ Display **interest**.
➢ Compute **finalPayment** (adding a payment, if necessary).
➢ Compute and display total of payments.
➢ Compute and display interest paid.
➢ Disable **computeButton**.
➢ Enable **newLoanButton**.
➢ Set focus on **newLoanButton**.

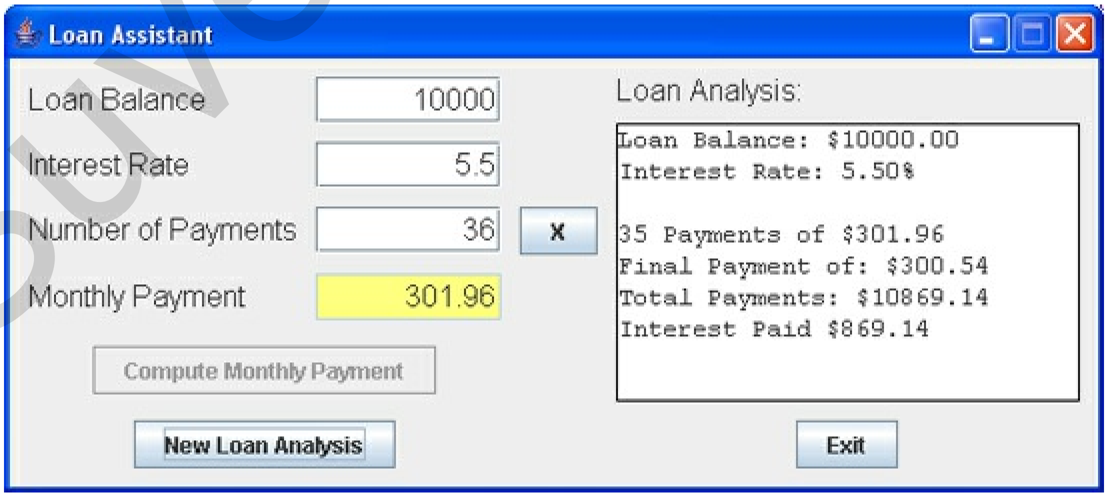
Each of these steps is performed in the **computeButtonActionPerformed** method. The modified method is (changes are shaded):

```
private void computeButtonActionPerformed(ActionEvent e)
{
    double balance, interest, payment;
    int months;
    double monthlyInterest, multiplier;
    double loanBalance, finalPayment;
    balance =
Double.valueOf(balanceTextField.getText()).doubleValue();
    interest =
Double.valueOf(interestTextField.getText()).doubleValue();
    monthlyInterest = interest / 1200;
    if (computePayment)
    {
        // Compute loan payment
        months =
Integer.valueOf(monthsTextField.getText()).intValue();
            multiplier = Math.pow(1 + monthlyInterest, months);
            payment = balance * monthlyInterest * multiplier / (multiplier - 1);
            paymentTextField.setText(new DecimalFormat("0.00").format(payment));
      }
      else
      {
            // Compute number of payments
            payment =
Double.valueOf(paymentTextField.getText()).doubleValue();
            months = (int)((Math.log(payment) - Math.log(payment - balance * monthlyInterest)) /
Math.log(1 + monthlyInterest));
            monthsTextField.setText(String.valueOf(months));
      }
      // reset payment prior to analysis to fix at two decimals
      payment =
Double.valueOf(paymentTextField.getText()).doubleValue();
      // show analysis
      analysisTextArea.setText("Loan Balance: $" + new
DecimalFormat("0.00").format(balance));
      analysisTextArea.append("\n" + "Interest Rate: " + new
DecimalFormat("0.00").format(interest) + "%");
      // process all but last payment
      loanBalance = balance;
      for (int paymentNumber = 1; paymentNumber <= months - 1; paymentNumber++)
      {
            loanBalance += loanBalance * monthlyInterest - payment;
      }
      // find final payment
      finalPayment = loanBalance;
      if (finalPayment > payment)
      {
            // apply one more payment
            loanBalance += loanBalance * monthlyInterest - payment;
            finalPayment = loanBalance;
            months++;
            monthsTextField.setText(String.valueOf(months));
        }
        analysisTextArea.append("\n\n" + String.valueOf(months - 1) + " Payments of $" + new
DecimalFormat("0.00").format(payment));
        analysisTextArea.append("\n" + "Final Payment of: $" + new
DecimalFormat("0.00").format(finalPayment));
        analysisTextArea.append("\n" + "Total Payments: $" + new
DecimalFormat("0.00").format((months - 1) * payment + finalPayment));
        analysisTextArea.append("\n" + "Interest Paid $" + new
DecimalFormat("0.00").format((months - 1) * payment + finalPayment - balance));
        computeButton.setEnabled(false);
        newLoanButton.setEnabled(true);
        newLoanButton.requestFocus();
    }
```

You should be able to identify all the steps of the loan analysis, especially the final payment adjustment.

A couple of comments. In the first line of the analysis code, we reassign the **payment** value to the displayed value in the **paymentTextField** text field. The displayed value is formatted to two decimal places. Through this reassignment, we make sure **payment** is just two decimal places. Second, note the analysis in the text field is essentially just one long **text** property. To start a new line, we use the control string **\n**.

Save and run the project. Enter values for balance, interest and number of payments. Click **Compute Monthly Payment** . Here are the results for the example I've been using:



Loan Balance: 10000

Interest Rate: 5.5

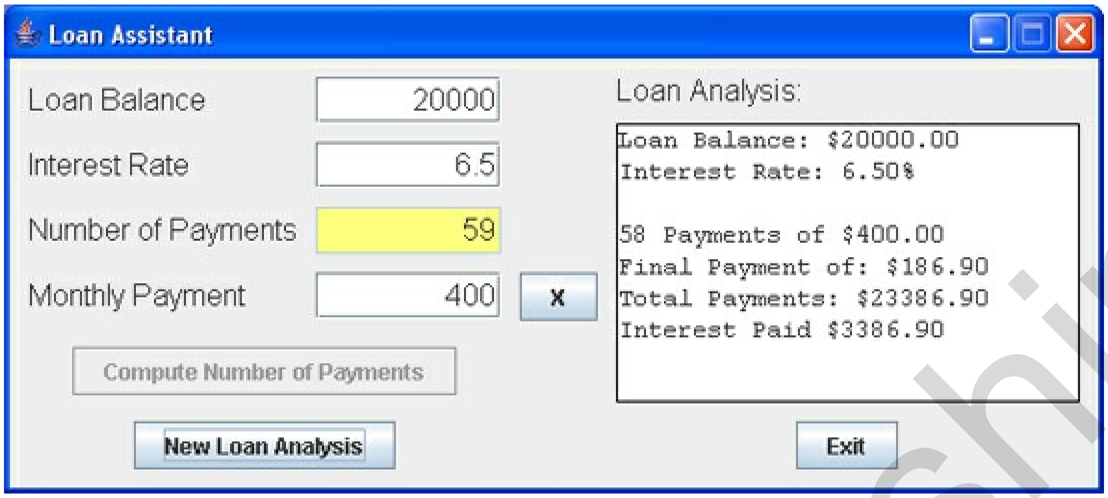
Number of Payments: 36

Monthly Payment: 301.96

```
Loan Balance: $10000.00
Interest Rate: 5.50%

35 Payments of $301.96
Final Payment of: $300.54
Total Payments: $10869.14
Interest Paid $869.14
```

Note the slight adjustment to the final payment amount. Note the focus on **New Loan Analysis**. You can't do another analysis at this point since **computeButton** is disabled – we'll fix that in the next section. Click **Exit**.

Run the project again, this time clicking the **X** next to the **Number of Payments** text field. Enter values for balance, interest and payment, then click **Compute Number of Payments**. Continuing with the example I've been using (remember we got 58 payments before?) shows:



```
Loan Balance: $20000.00
Interest Rate: 6.50%

58 Payments of $400.00
Final Payment of: $186.90
Total Payments: $23386.90
Interest Paid $3386.90
```

We now get 59 rather than 58 payments, the same result we saw earlier in the project specs. It was determined that once 58 payments of $400.00 per month were applied, there was still a balance over $400, necessitating a 59th payment of $400.00 plus an additional payment of $186.90. Click **Exit**, since you can't do anything else at this point.

Code Design – New Loan Analysis

Following an analysis, we would like the capability of performing a new analysis. When a user clicks the **New Loan Analysis** button (**newLoanButton**), the following things should happen:

- If computing payment, clear **paymentTextField**, else clear **monthsTextField**.
- Clear **analysisTextArea**.
- Enable **computeButton**.
- Disable **newLoanButton**.
- Set focus on **balanceTextField**.

We do not clear the **balanceTextField** or **interestTextField** boxes. If computing the payment, we do not clear the **monthsTextField** box. If computing the number of months, we do not clear the **paymentTextField** box. This allows a user to try different things with a specific loan. Individual boxes can be cleared by the user, if desired.

The **newLoanButtonActionPerformed** method is:

```
private void newLoanButtonActionPerformed(ActionEvent e)
{
    // clear computed value and analysis
    if (computePayment)
    {
        paymentTextField.setText("");
    }
    else
    {
        monthsTextField.setText("");
    }
    analysisTextArea.setText("");
    computeButton.setEnabled(true);
    newLoanButton.setEnabled(false);
    balanceTextField.requestFocus();
}
```

Add this new code. Save and run the project. The project should now have total ability to compute monthly payments or number of payments, providing complete loan analysis results. Play with the project as much as you'd like.

Improving a Java Project

The consumer loan assistant project works fine in its current configuration, but there are some hidden problems. You may have uncovered some of them already. Earlier, we saw the possibility of unpredictable tab ordering and fixed the problem, improving the performance of our project. This is something you, as a programmer, will do a lot. You will build a project and, while running it and testing it, will uncover weaknesses that need to be eliminated. These weaknesses could be actual errors in the application or just things that, if eliminated, make your application easier to use. Some weaknesses are easy to find, some more subtle.

You will find, as you progress as a programmer, that you will spend much of your time improving your projects. You will always find ways to add features to a project and to make it more appealing to your user base. You should never be satisfied with your first solution to a problem. There will always be room for improvement.

If you run the loan assistant project a few more times, you can identify some weaknesses:

- ➢ For example, what happens if you input a zero interest? The program will result in error messages and not compute a payment because the formulas implemented in code will not work with zero interest.
- ➢ As a convenience, it would be nice that when you hit the <**Enter**> key after typing a number, the focus would move to the next control in the tab sequence.
- ➢ Notice you can type any characters you want in the text fields when you should just be limited to numbers and a single decimal point – any other characters will cause the program to work incorrectly.
- ➢ What happens if you forget to input a value (leaving a text field empty)? You could get unpredictable results.
- ➢ A subtle problem arises when using the 'compute number of months' mode. In this configuration, the minimum desired payment must exceed the loan balance times the monthly interest. If it doesn't, you will never get the loan paid off – your balance will just keep growing (!), something called negative amortization.

We can (and will) address each of these points as we improve the loan assistant project. As we do, we'll look at some other Java features.

Code Design – Zero Interest

If you are lucky enough to find a bank or someone to give you a loan at zero percent interest, congratulations!! However, you can't use the current code to compute payment information. Try it if you like – you'll receive error messages in the NetBeans output window and see no computed payment.

The formulas used in the code assume a non-zero interest rate. If **interest** is zero, we can use much simpler formulas. For the 'compute payment' mode, the code is simply:

```
payment = balance / months;
```

While for the 'compute number of payments' mode, the code is:

```
months = (int)(balance / payment);
```

The modified **computeButtonActionPerformed** method (changes are shaded, some unmodified code is not shown for brevity):

```
private void computeButtonActionPerformed(ActionEvent e)
{
    .
    .
    if (computePayment)
    {
        // Compute loan payment
        months =
Integer.valueOf(monthsTextField.getText()).intValue();
        if (interest == 0)
        {
            payment = balance / months;
        }
            else
        {
            multiplier = Math.pow(1 + monthlyInterest, months);
            payment = balance * monthlyInterest * multiplier / (multiplier - 1);
        }
        paymentTextField.setText(new DecimalFormat("0.00").format(payment));
    }
    else
    {
        // Compute number of payments
        payment =
Double.valueOf(paymentTextField.getText()).doubleValue();
        if (interest == 0)
        {
            months = (int)(balance / payment);
        }
        else
        {
            months = (int)((Math.log(payment) - Math.log(payment - balance *
monthlyInterest)) / Math.log(1 + monthlyInterest));
        }
        monthsTextField.setText(String.valueOf(months));
    }
    .
    .
}
```

Save and run the application, making sure zero interest works under each mode. Notice adjustments to the final payment are only made when the balance is not an exact multiple of the payment. Make sure the non-zero interest options still work, too. Always make sure when you make changes to your code that you haven't disturbed portions that are working satisfactorily.

Code Design – Focus Transfer

We saw that the **<Tab>** key could be used to move from control to control, shifting the focus. Many times, you might like to move focus from one control to another in code, or programmatically. For example, in our savings example, once the user types in a **Deposit Amount**, it would be nice if focus would be moved to the **Interest** text field if the user presses **<Enter>.**

To move from the current control to the next control in the tab sequence, use **transferFocus**:

```
myControl.transferFocus();
```

To move from the current control to the previous control in the tab sequence, use **transerFocusBackward**:

```
myControl.transferFocusBackward();
```

So, where does this code go in our project? When a text field has focus and <**Enter**> is pressed, the **ActionPerformed** method is invoked. Hence, for each text field where we want to move focus based on keyboard input, we add an event method and place the needed code there. Adding event methods for a text field is identical to adding methods for other Swing components. For a text field named **myTextField**, use:

```
myTextField.addActionListener(new ActionListener ()
{
    public void actionPerformed(ActionEvent e)
    {
        myTextFieldActionPerformed(e);
    }
});
```

and the corresponding event method code to move focus would be:

```
private void myTextFieldActionPerformed(ActionEvent e)
{
    myTextField.transferFocus();
}
```

Let's make the modifications to the loan assistant. Each text field will need to have a method to transfer focus to the next control in sequence. In the frame constructor, after each text field is established, add code in the corresponding location to add a listener:

```
balanceTextField.addActionListener(new ActionListener ()
{
    public void actionPerformed(ActionEvent e)
    {
        balanceTextFieldActionPerformed(e);
    }
});

interestTextField.addActionListener(new ActionListener ()
{
    public void actionPerformed(ActionEvent e)
    {
        interestTextFieldActionPerformed(e);
    }
});

monthsTextField.addActionListener(new ActionListener ()
{
    public void actionPerformed(ActionEvent e)
    {
        monthsTextFieldActionPerformed(e);
    }
});

paymentTextField.addActionListener(new ActionListener ()
{
    public void actionPerformed(ActionEvent e)
    {
        paymentTextFieldActionPerformed(e);
    }
});
```

Next, add the four **ActionPerformed** methods that transfer focus:

```
private void balanceTextFieldActionPerformed(ActionEvent e)
{
    balanceTextField.transferFocus();
}

private void interestTextFieldActionPerformed(ActionEvent e)
{
     interestTextField.transferFocus();
}

private void monthsTextFieldActionPerformed(ActionEvent e)
{
     monthsTextField.transferFocus();
}

private void paymentTextFieldActionPerformed(ActionEvent e)
{
     paymentTextField.transferFocus();
}
```

Save, run the project. Try using the **<Enter>** key to move from field to field. Try both the compute payment and compute number of payments modes.

Code Design - Input Validation

In the loan assistant project, there is nothing to prevent the user from typing in meaningless characters (for example, letters) into the text fields expecting numerical data. We want to keep this from happening – if the input is not a valid number, it cannot be converted from a string to a number, resulting in errors. Whenever getting input from a user using a text field control, we need to **validate** the typed information before using it. Validation rules differ depending on what information you want from the user.

In this project, we will perform **input validation** in a general method (named **validateDecimalNumber**) we write. The method will examine the text property of a text field, trimming off leading and trailing spaces and checking that the field contains only numbers and a single decimal point. It will return a **boolean** value indicating if a valid number is found. If the number is valid, the method will return a **true** value. If not valid, the method will return a **false** value. It will give that control focus, indicating the user needs to modify his/her input.

Here's the method that accomplishes that task:

```
public boolean validateDecimalNumber(JTextField tf)
{
    // checks to see if text field contains
    // valid decimal number with only digits and a single decimal point
    String s = tf.getText().trim();
    boolean hasDecimal = false;
    boolean valid = true;
    if (s.length() == 0)
    {
        valid = false;
    }
    else
    {
        for (int i = 0; i < s.length(); i++)
        {
            char c = s.charAt(i);
            if (c >= '0' && c <= '9')
            {
                continue;
            }
            else if (c == '.' && !hasDecimal)
            {
                hasDecimal = true;
            }
            else
            {
                // invalid character found
                valid = false;
            }
        }
    }
    tf.setText(s);
    if (!valid)
    {
        tf.requestFocus();
    }
    return (valid);
}
```

You should be able to see how this works. The text field text property is stored in the string **s** (after trimming off leading and trailing spaces). Each character in this string is evaluated to see if it contains only allows number and a single decimal point. If only numbers and a decimal are found, **valid** is **true** and things proceed. If **valid** is **false**, indicating invalid characters or an empty string, the text field is given focus to allow changing the input value. Add this validation method to your project.

Make the shaded changes to the **computeButtonActionPerformed** method to insure we have valid entries before doing a computation:

```
private void computeButtonActionPerformed(ActionEvent e)
{
    double balance, interest, payment;
    int months;
    double monthlyInterest, multiplier;
    double loanBalance, finalPayment;
    if (validateDecimalNumber(balanceTextField))
    {
        balance =
Double.valueOf(balanceTextField.getText()).doubleValue();
    }
    else
    {
        return;
    }
    if (validateDecimalNumber(interestTextField))
    {
        interest =
Double.valueOf(interestTextField.getText()).doubleValue();
    }
    else
    {
        return;
    }
    monthlyInterest = interest / 1200;
    if (computePayment)
    {
        // Compute loan payment
        if (validateDecimalNumber(monthsTextField))
        {
        months =
Integer.valueOf(monthsTextField.getText()).intValue();
        }
        else
        {
            return;
        }
        if (interest == 0)
        {
            payment = balance / months;
        }
        else
        {
            multiplier = Math.pow(1 + monthlyInterest, months);
```

```
                payment = balance * monthlyInterest * multiplier / (multiplier - 1);
            }
            paymentTextField.setText(new DecimalFormat("0.00").format(payment));
        }
        else
        {
            // Compute number of payments
            if (validateDecimalNumber(paymentTextField))
            {
                payment =
    Double.valueOf(paymentTextField.getText()).doubleValue();
            }
            else
            {
                return;
            }
            if (interest == 0)
            {
                months = (int)(balance / payment);
            }
            else
            {
                months = (int)((Math.log(payment) - Math.log(payment - balance *
    monthlyInterest)) / Math.log(1 + monthlyInterest));
            }
            monthsTextField.setText(String.valueOf(months));
        }
        .
        .
    }
```

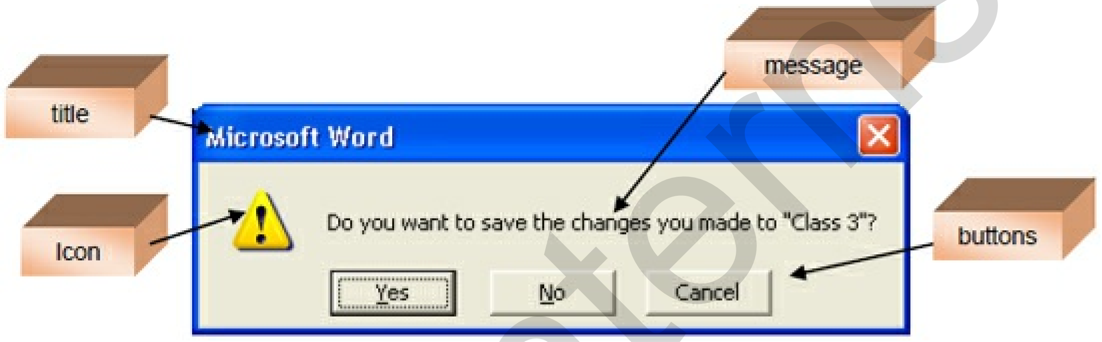
In each case where we need a value from a text field, we check to see if it is valid, using the **validateDecimalNumber** method. If valid, computations are as usual. If not valid, the method is exited and the focus is in the text field with the invalid entry.

Run the project again. Try invalid entries. Click the compute button. There should be no error messages, only a blanking out of the offending text field. A user of your program would like some indication of why the program isn't working in these cases (in addition to simply blanking out an invalid entry). Let's see how to add such messages using the **confirm dialog** control.

Confirm Dialog

An often used dialog box in Java GUI applications is a **confirm dialog** (also known as a **message box**). This dialog lets you display messages to your user and receive feedback for further information. It can be used to display error messages, describe potential problems or just to show the result of some computation. A confirm dialog is implemented with the Java Swing **JOptionPane** class. The confirm dialog is versatile, with the ability to display any message, an optional icon, and a selected set of buttons. The user responds by clicking a button in the confirm dialog box.

You've seen confirm dialog boxes if you've ever used a Windows (or other OS) application. Think of all the examples you've seen. For example, confirm dialogs are used to ask you if you wish to save a file before exiting and to warn you if a disk drive is not ready. For example, if while writing these notes in Microsoft Word, I attempt to exit, I see this confirm dialog:



Do you want to save the changes you made to "Class 3"?

In this confirm dialog box, the different parts that you control have been labeled. You will see how you can format a confirm dialog box any way you desire.

To use the **confirm dialog** method, you decide what the **message** should be, what **title** you desire, and what **icon** and **buttons** are appropriate. To display the confirm dialog box in code, you use the **showConfirmDialog** method.

The **showConfirmDialog** method is **overloaded** with several ways to implement the dialog box. Some of the more common ways are:

JOptionPane.showConfirmDialog(null, message);
JOptionPane.showConfirmDialog(null, message, title, buttons);
JOptionPane.showConfirmDialog(null, message, title, buttons, icon);

In these implementations, if **icon** is omitted, a question mark is displayed. If **buttons** is omitted, **Yes**, **No**, **Cancel** buttons are displayed. And, if **title** is omitted, a title of "**Select an Option**" is displayed. The first argument (**null**) must be there – it indicates the confirm dialog box is associated with the current frame.

As mentioned, you decide what you want for the confirm dialog **message** and **title** information (string data types). Be aware there is no limit to how long the message can be. If you have a long message, use the new line character (**n**) to break the message into multiple lines.

The other arguments are defined by Java **JOptionPane** predefined constants. The **buttons** constants are defined by:

Member	**Description**
DEFAULT_OPTION	Displays an OK button
OK_CANCEL_OPTION	Displays OK and Cancel buttons
YES_NO_CANCEL_OPTION	Displays Yes, No and Cancel buttons
YES_NO_OPTION	Displays Yes and No buttons

The syntax for specifying a choice of buttons is the usual dot-notation:

JOptionPane.Member

So, to display an OK and Cancel button, the constant is:

JOptionPane.OK_CANCEL_OPTION

The displayed icon is established by another set of constants:

Member	**Description**
PLAIN_MESSAGE	Display no icon
INFORMATION_MESSAGE	Displays an information icon
ERROR_MESSAGE	Displays an error icon
WARNING_MESSAGE	Displays an exclamation point icon
QUESTION_MESSAGE	Displays a question mark icon

To specify an icon, the syntax is:

JOptionPane.Member

To display an error icon, use:

JOptionPane.ERROR_MESSAGE

When you invoke the **showOptionDialog** method, the method returns a JOptionPane constant (an **int** type) indicating the user response. The available members are:

Member	**Description**
CLOSED_OPTION	Window closed without pressing button

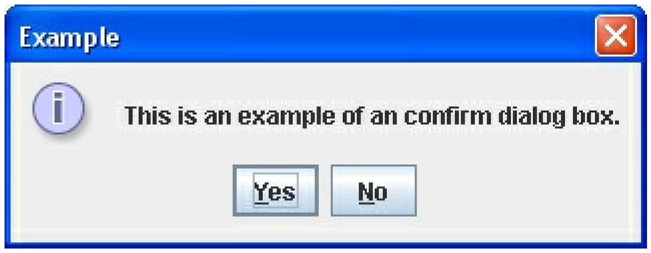
OK_OPTION	The OK button was selected
YES_OPTION	The Yes button was selected
NO_OPTION	The No button was selected
CANCEL_OPTION	The Cancel button was selected

Confirm Dialog **Example**:

This little code snippet (the second line is very long):

```
int response;
response = JOptionPane.showConfirmDialog(null, "This is an example of an confirm dialog
box.", "Example", JOptionPane.YES_NO_OPTION,
JOptionPane.INFORMATION_MESSAGE);
if (response == JOptionPane.YES_OPTION)
{
    // Pressed Yes
}
else if (response == JOptionPane.NO_OPTION)
{
    // Pressed No
}
else
{
    // Closed window without pressing button
}
```

displays this message box:



This is an example of an confirm dialog box.

Of course, you would need to add code for the different tasks depending on whether **Yes** or **No** is clicked by the user (or the window is simply closed).

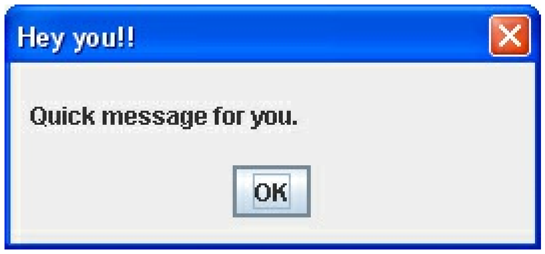
Another Confirm Dialog **Example**:

Many times, you just want to display a quick message to the user with no need for feedback (just an

OK button). This code does the job:

JOptionPane.showConfirmDialog(null, "Quick message for you.", "Hey you!!", JOptionPane.DEFAULT_OPTION, JOptionPane.PLAIN_MESSAGE);

The resulting message box:



Quick message for you.

Notice there is no icon and the OK button is shown. Also, notice in the code, there is no need to read the returned value – we know what it is! You will find a lot of uses for this simple form of the message box (with perhaps some kind of icon) as you progress in Java.

Let's use the confirm dialog control to provide our user some feedback when there are invalid entries in the loan assistant project.

Code Design – User Messages

When an invalid entry is encountered (either blank or containing invalid characters), we want to inform our users of the problem. We will use a simple form of the confirm dialog to provide this feedback. It will simply present the message with an **OK** button.

We need four confirm dialogs, one for each text field. The dialogs are added in the **computeButtonActionPerformed** method (changes are shaded):

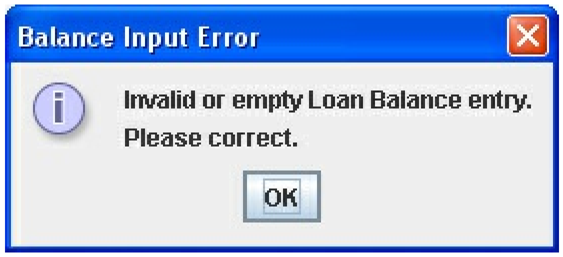
```
private void computeButtonActionPerformed(ActionEvent e)
{
    double balance, interest, payment;
    int months;
    double monthlyInterest, multiplier;
    double loanBalance, finalPayment;
    if (validateDecimalNumber(balanceTextField))
    {
        balance =
Double.valueOf(balanceTextField.getText()).doubleValue();
    }
    else
    {
        JOptionPane.showConfirmDialog(null, "Invalid or empty Loan Balance
entry.\nPlease correct.", "Balance Input Error", JOptionPane.DEFAULT_OPTION,
JOptionPane.INFORMATION_MESSAGE);
        return;
    }
    if (validateDecimalNumber(interestTextField))
    {
        interest =
Double.valueOf(interestTextField.getText()).doubleValue();
    }
    else
    {
        JOptionPane.showConfirmDialog(null, "Invalid or empty Interest Rate entry.\nPlease
correct.", "Interest Input Error", JOptionPane.DEFAULT_OPTION,
JOptionPane.INFORMATION_MESSAGE);
```

```
            return;
        }
        monthlyInterest = interest / 1200;
        if (computePayment)
        {
            // Compute loan payment
            if (validateDecimalNumber(monthsTextField))
            {
                months =
Integer.valueOf(monthsTextField.getText()).intValue();
            }
            else
            {
                JOptionPane.showConfirmDialog(null, "Invalid or empty Number of Payments
entry.\nPlease correct.", "Number of Payments Input Error",
JOptionPane.DEFAULT_OPTION, JOptionPane.INFORMATION_MESSAGE);
                return;
            }
            if (interest == 0)
            {
                payment = balance / months;
            }
            else
            {
                multiplier = Math.pow(1 + monthlyInterest, months);
                payment = balance * monthlyInterest * multiplier / (multiplier - 1);
            }
            paymentTextField.setText(new DecimalFormat("0.00").format(payment));
        }
        else
        {
            // Compute number of payments
            if (validateDecimalNumber(paymentTextField))
            {
                payment =
Double.valueOf(paymentTextField.getText()).doubleValue();
```

```
        }
        else
        {
            JOptionPane.showConfirmDialog(null, "Invalid or empty Monthly Payment
entry.\nPlease correct.", "Payment Input Error", JOptionPane.DEFAULT_OPTION,
JOptionPane.INFORMATION_MESSAGE);
            return;
        }
        if (interest == 0)
        {
            months = (int)(balance / payment);
        }
        else
        {
            months = (int)((Math.log(payment) - Math.log(payment - balance *
monthlyInterest)) / Math.log(1 + monthlyInterest));
        }
        monthsTextField.setText(String.valueOf(months));
    }
    .
    .
}
```

After making these modifications, save and run the project. Make sure each input validation works correctly. Make sure it works in both computation modes. And, make sure your changes have not affected previously correct calculations.

Each validation is similar. If the text field is not blank and all characters are valid, things proceed as usual. If blank or containing invalid characters, a message box like this appears (this one appears when **balanceTextField** is left blank, other message boxes are similar):


Balance Input Error

Invalid or empty Loan Balance entry.
Please correct.

The user clicks **OK**, the focus is returned to the blank control for another chance at inputting a non-blank value.

We have one last input validation to implement and then the consumer loan assistant project is complete (unless you can think of other improvements). Recall, when computing the number of months, we must enter a minimum payment or the balance will continue to grow. The minimum payment is the loan balance times the monthly interest:

minimumPayment = balance * monthlyInterest;

If this payment is made each month, it is called an "interest only" loan. This means, we just pay the interest owed each month, never decreasing the balance. Since our goal is to decrease the balance, we will suggest to the user a minimum payment at least $1 greater than the interest only option, or we will use:

minimumPayment = balance * monthlyInterest + 1;

The steps for minimum payment validation are (only needed when **computePayment** is **false**):

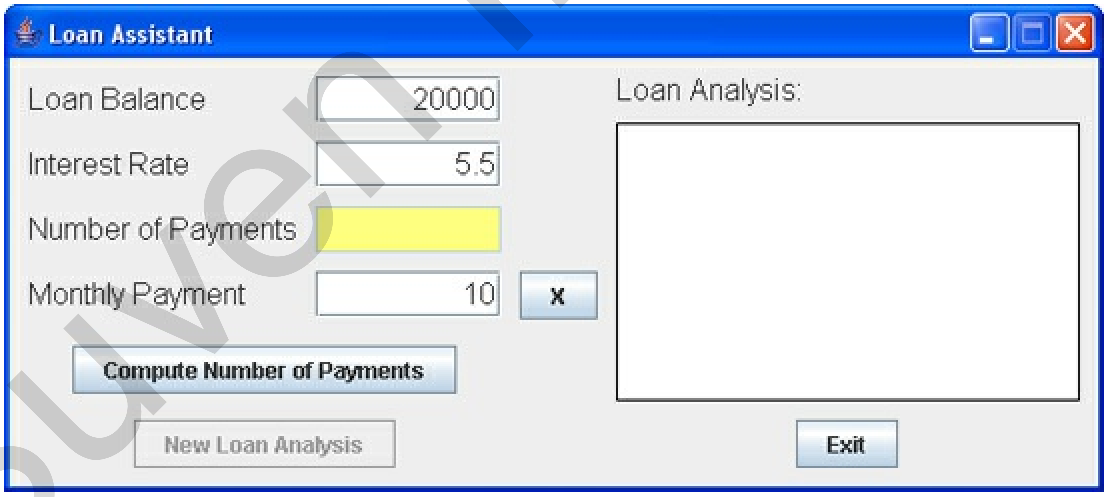
- ➢ If entered **payment** is less than minimum value, display message box informing user of minimum needed and ask if they would like to use that value.
  - o If user responds **Yes**, set **payment**, display in **paymentTextField** and continue.
  - o If user responds **No**, set focus on **paymentTextField** to allow new entry.
- ➢ If entered **payment** is above minimum value, continue as usual.

These modifications go in the **computeButtonActionPerformed** method. While implementing improvements to the loan assistant project, we have made many modifications to this event. For reference purposes, here is the final version of the **computeButton Click** method (with new additions shaded):

```
private void computeButtonActionPerformed(ActionEvent e)
{
    double balance, interest, payment;
    int months;
    double monthlyInterest, multiplier;
    double loanBalance, finalPayment;
    if (validateDecimalNumber(balanceTextField))
    {
        balance =
Double.valueOf(balanceTextField.getText()).doubleValue();
    }
    else
    {
          JOptionPane.showConfirmDialog(null, "Invalid or empty Loan Balance entry.\nPlease
correct.", "Balance Input Error", JOptionPane.DEFAULT_OPTION,
JOptionPane.INFORMATION_MESSAGE);
          return;
     }
     if (validateDecimalNumber(interestTextField))
     {
          interest =
Double.valueOf(interestTextField.getText()).doubleValue();
     }
     else
     {
          JOptionPane.showConfirmDialog(null, "Invalid or empty Interest Rate entry.\nPlease
correct.", "Interest Input Error", JOptionPane.DEFAULT_OPTION,
JOptionPane.INFORMATION_MESSAGE);
          return;
     }
     monthlyInterest = interest / 1200;
     if (computePayment)
     {
          // Compute loan payment
          if (validateDecimalNumber(monthsTextField))
          {
               months =
Integer.valueOf(monthsTextField.getText()).intValue();
          }
          else
          {
               JOptionPane.showConfirmDialog(null, "Invalid or empty Number of Payments
entry.\nPlease correct.", "Number of Payments Input Error",
JOptionPane.DEFAULT_OPTION, JOptionPane.INFORMATION_MESSAGE);
               return;
          }
          if (interest == 0)
          {
               payment = balance / months;
          }
          else
          {
               multiplier = Math.pow(1 + monthlyInterest, months);
               payment = balance * monthlyInterest * multiplier / (multiplier - 1);
          }
          paymentTextField.setText(new DecimalFormat("0.00").format(payment));
     }
     else
     {
          // Compute number of payments
          if (validateDecimalNumber(paymentTextField))
          {
               payment =
Double.valueOf(paymentTextField.getText()).doubleValue();
               if (payment <= (balance * monthlyInterest + 1.0))
               {
                    if (JOptionPane.showConfirmDialog(null, "Minimum payment must be $" +
new DecimalFormat("0.00").format((int)(balance * monthlyInterest + 1.0)) + "\n" + "Do you
want to use the minimum payment?", "Input Error", JOptionPane.YES_NO_OPTION,
JOptionPane.QUESTION_MESSAGE) == JOptionPane.YES_OPTION)
                    {
                         paymentTextField.setText(new DecimalFormat("0.00").format((int)
(balance * monthlyInterest + 1.0)));
                              payment =
Double.valueOf(paymentTextField.getText()).doubleValue();
                    }
                    else
                    {
                              paymentTextField.requestFocus();
                              return;
                    }
               }
          }
          else
          {
               JOptionPane.showConfirmDialog(null, "Invalid or empty Monthly Payment
entry.\nPlease correct.", "Payment Input Error", JOptionPane.DEFAULT_OPTION,
JOptionPane.INFORMATION_MESSAGE);
                return;
            }
            if (interest == 0)
            {
                months = (int)(balance / payment);
            }
            else
            {
                months = (int)((Math.log(payment) - Math.log(payment - balance *
monthlyInterest)) / Math.log(1 + monthlyInterest));
            }
            monthsTextField.setText(String.valueOf(months));
        }
        // reset payment prior to analysis to fix at two decimals
        payment =
Double.valueOf(paymentTextField.getText()).doubleValue();
        // show analysis
        analysisTextArea.setText("Loan Balance: $" + new
DecimalFormat("0.00").format(balance));
        analysisTextArea.append("\n" + "Interest Rate: " + new
DecimalFormat("0.00").format(interest) + "%");
        // process all but last payment
        loanBalance = balance;
        for (int paymentNumber = 1; paymentNumber <= months - 1; paymentNumber++)
        {
            loanBalance += loanBalance * monthlyInterest - payment;
        }
        // find final payment
        finalPayment = loanBalance;
        if (finalPayment > payment)
        {
            // apply one more payment
            loanBalance += loanBalance * monthlyInterest - payment;
            finalPayment = loanBalance;
            months++;
            monthsTextField.setText(String.valueOf(months));
        }
        analysisTextArea.append("\n\n" + String.valueOf(months - 1) + " Payments of $" + new
DecimalFormat("0.00").format(payment));
        analysisTextArea.append("\n" + "Final Payment of: $" + new
DecimalFormat("0.00").format(finalPayment));
        analysisTextArea.append("\n" + "Total Payments: $" + new
DecimalFormat("0.00").format((months - 1) * payment + finalPayment));
        analysisTextArea.append("\n" + "Interest Paid $" + new
DecimalFormat("0.00").format((months - 1) * payment + finalPayment - balance));
        computeButton.setEnabled(false);
        newLoanButton.setEnabled(true);
        newLoanButton.requestFocus();
}
```

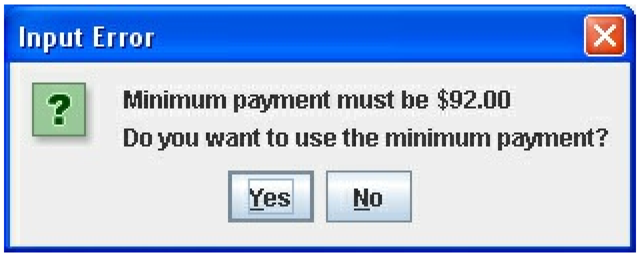
Make the noted changes.

Save and run the loan assistant project. Switch to 'compute number of payments' mode. Enter a **Loan Balance** and an **Interest Rate**. Enter a "too low" **Monthly Payment** amount. Here's some numbers I used:



Loan Balance: 20000
Interest Rate: 5.5
Monthly Payment: 10

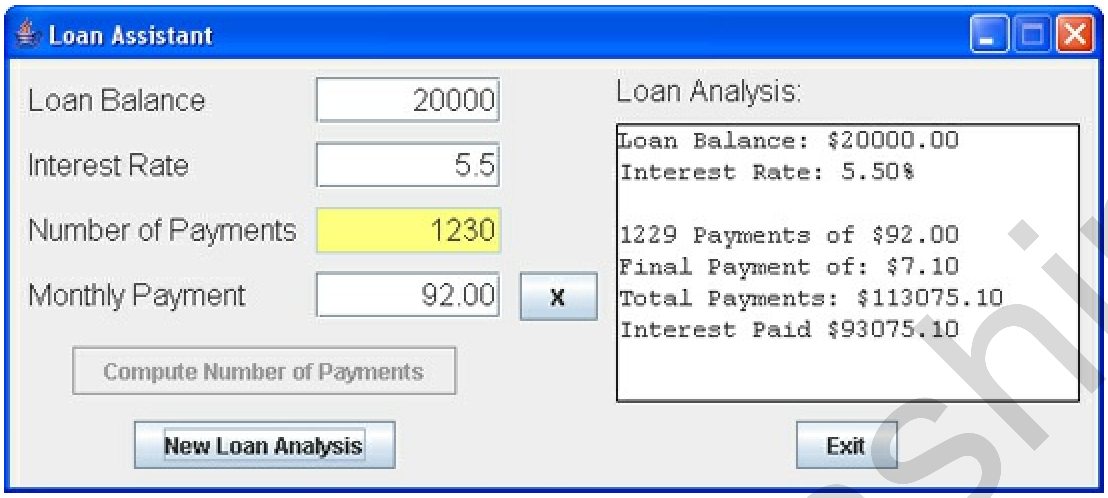
Now, click **Compute Number of Payments**. A message box like this should appear:



Minimum payment must be $92.00
Do you want to use the minimum payment?

At this point, if you click **No**, you will be returned to the **Monthly Payment** text field for another chance.

Click **Yes** and analysis will proceed using the suggested minimum payment ($92.00 in my example):



Loan Balance: 20000
Interest Rate: 5.5
Number of Payments: 1230
Monthly Payment: 92.00

```
Loan Balance: $20000.00
Interest Rate: 5.50%

1229 Payments of $92.00
Final Payment of: $7.10
Total Payments: $113075.10
Interest Paid $93075.10
```

With this low minimum payment, it would take over 102 years to pay off the loan!! And, unfortunately, many credit card companies don't let you know how many years it takes to pay off a balance if you just make the minimum payment each month. This new project arms you with the tool you need to make such computations.

Consumer Loan Assistant Project Review

The **Consumer Loan Assistant** project is now complete. Save and run the project and make sure it works as designed. Check that you can move back and forth between computation modes. Use the project to make informed payment decisions regarding any loans or credit cards you may have.

If there are errors in your implementation, go back over the steps of frame and code design. Go over the developed code – make sure you understand how different parts of the project were coded. As mentioned in the beginning of this project, the completed project is saved as **LoanAssistant** in the **\HomeJava\HomeJava Projects** folder.

While completing this project, new concepts and skills you should have gained include:

- Proper use of the text field control.
- How to use tab order and control focus.
- Different ways to improve a Java project.
- How to use message boxes in conjunction with input validation.

This project also showed that once you have built a working project, there is often still a lot of work to do. Much of the code in the loan assistant project was added to improve the application – making it more user friendly and less susceptible to erroneous entries. As mentioned previously in these notes, it is relatively easy to write a project that works properly when the user does everything correctly. It's difficult and takes time to write a project that can handle all the possible wrong things a user can do and still not bomb out. Added improvements separate the good projects from the adequate projects.

Consumer Loan Assistant Project Enhancements

Possible enhancements to the consumer loan assistant project include:

- ➢ Many times, you know how much you can afford monthly and want to know how much you can borrow. Add a capability to compute balance, given interest, months and payment. Follow similar steps for computing the other parameters.
- ➢ Add single payment processing capability so you can see how much the balance decreases each month and how much interest you are paying. Show results in the current text field or add other controls.
- ➢ Add the capability to stop after a certain number of payments have been processed.
- ➢ Add printing capability to see a complete repayment schedule for any loan you design. Printing is discussed in another project in these notes – **Home Inventory**.
- ➢ Add an output to the loan analysis that tells you what date your loan will be paid off based on the number of monthly payments.

Consumer Loan Assistant Project Review

/ Complete Project Code */*

```
/ *
* LoanAssistant.java
*/
package loanassistant;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.text.*;

public class LoanAssistant extends JFrame
{

   JLabel balanceLabel = new JLabel();
   JTextField balanceTextField = new JTextField();
   JLabel interestLabel = new JLabel();
   JTextField interestTextField = new JTextField();
   JLabel monthsLabel = new JLabel();
   JTextField monthsTextField = new JTextField();
   JLabel paymentLabel = new JLabel();
   JTextField paymentTextField = new JTextField();
   JButton computeButton = new JButton();
   JButton newLoanButton = new JButton();
   JButton monthsButton = new JButton();
   JButton paymentButton = new JButton();
   JLabel analysisLabel = new JLabel();
   JTextArea analysisTextArea = new JTextArea();
   JButton exitButton = new JButton();

   Font myFont = new Font("Arial", Font.PLAIN, 16);

   Color lightYellow = new Color(255, 255, 128);

   boolean computePayment;

   public static void main(String args[])
```

```
  {
       // create frame
       new LoanAssistant().show();
  }

  public LoanAssistant()
  {
       // frame constructor
       setTitle("Loan Assistant");
       setResizable(false);
       addWindowListener(new WindowAdapter()
       {
            public void windowClosing(WindowEvent evt)
            {
                 exitForm(evt);
            }
       });
       getContentPane().setLayout(new GridBagLayout());
       GridBagConstraints gridConstraints;

       balanceLabel.setText("Loan Balance");
       balanceLabel.setFont(myFont);
       gridConstraints = new GridBagConstraints();
       gridConstraints.gridx = 0;
       gridConstraints.gridy = 0;
       gridConstraints.anchor = GridBagConstraints.WEST;
       gridConstraints.insets = new Insets(10, 10, 0, 0);
       getContentPane().add(balanceLabel, gridConstraints);

       balanceTextField.setPreferredSize(new Dimension(100, 25));

balanceTextField.setHorizontalAlignment(SwingConstants.RIGHT);

       balanceTextField.setFont(myFont);
       gridConstraints = new GridBagConstraints();
       gridConstraints.gridx = 1;
       gridConstraints.gridy = 0;
```

```
        gridConstraints.insets = new Insets(10, 10, 0, 10);
        getContentPane().add(balanceTextField, gridConstraints);
        balanceTextField.addActionListener(new ActionListener ()
        {
            public void actionPerformed(ActionEvent e)
            {
                balanceTextFieldActionPerformed(e);
            }
        });

        interestLabel.setText("Interest Rate");
        interestLabel.setFont(myFont);
        gridConstraints = new GridBagConstraints();
        gridConstraints.gridx = 0;
        gridConstraints.gridy = 1;
        gridConstraints.anchor = GridBagConstraints.WEST;
        gridConstraints.insets = new Insets(10, 10, 0, 0);
        getContentPane().add(interestLabel, gridConstraints);
        interestTextField.setPreferredSize(new Dimension(100, 25));

interestTextField.setHorizontalAlignment(SwingConstants.RIGHT);
        interestTextField.setFont(myFont);
        gridConstraints = new GridBagConstraints();
        gridConstraints.gridx = 1;
        gridConstraints.gridy = 1;
        gridConstraints.insets = new Insets(10, 10, 0, 10);
        getContentPane().add(interestTextField, gridConstraints);
        interestTextField.addActionListener(new ActionListener ()
        {
            public void actionPerformed(ActionEvent e)
            {
                interestTextFieldActionPerformed(e);
            }
        });

        monthsLabel.setText("Number of Payments");
        monthsLabel.setFont(myFont);
        gridConstraints = new GridBagConstraints();
        gridConstraints.gridx = 0;
        gridConstraints.gridy = 2;
        gridConstraints.anchor = GridBagConstraints.WEST;
        gridConstraints.insets = new Insets(10, 10, 0, 0);
        getContentPane().add(monthsLabel, gridConstraints);

        monthsTextField.setPreferredSize(new Dimension(100, 25));

monthsTextField.setHorizontalAlignment(SwingConstants.RIGHT);
        monthsTextField.setFont(myFont);
        gridConstraints = new GridBagConstraints();
        gridConstraints.gridx = 1;
        gridConstraints.gridy = 2;
        gridConstraints.insets = new Insets(10, 10, 0, 10);
        getContentPane().add(monthsTextField, gridConstraints);
        monthsTextField.addActionListener(new ActionListener ()
        {
            public void actionPerformed(ActionEvent e)
            {
                monthsTextFieldActionPerformed(e);
            }
        });

        paymentLabel.setText("Monthly Payment");
        paymentLabel.setFont(myFont);
        gridConstraints = new GridBagConstraints();
        gridConstraints.gridx = 0;
        gridConstraints.gridy = 3;
        gridConstraints.anchor = GridBagConstraints.WEST;
        gridConstraints.insets = new Insets(10, 10, 0, 0);
        getContentPane().add(paymentLabel, gridConstraints);

        paymentTextField.setPreferredSize(new Dimension(100, 25));

paymentTextField.setHorizontalAlignment(SwingConstants.RIGHT);
        paymentTextField.setFont(myFont);
```

```
gridConstraints = new GridBagConstraints();
gridConstraints.gridx = 1;
gridConstraints.gridy = 3;
gridConstraints.insets = new Insets(10, 10, 0, 10);
getContentPane().add(paymentTextField, gridConstraints);
paymentTextField.addActionListener(new ActionListener ()
{
    public void actionPerformed(ActionEvent e)
    {
        paymentTextFieldActionPerformed(e);
    }
});

computeButton.setText("Compute Monthly Payment");
gridConstraints = new GridBagConstraints();
gridConstraints.gridx = 0;
gridConstraints.gridy = 4;
gridConstraints.gridwidth = 2;
gridConstraints.insets = new Insets(10, 0, 0, 0);
getContentPane().add(computeButton, gridConstraints);
computeButton.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        computeButtonActionPerformed(e);
    }
});

newLoanButton.setText("New Loan Analysis");
newLoanButton.setEnabled(false);
gridConstraints = new GridBagConstraints();
gridConstraints.gridx = 0;
gridConstraints.gridy = 5;
gridConstraints.gridwidth = 2;
gridConstraints.insets = new Insets(10, 0, 10, 0);
getContentPane().add(newLoanButton, gridConstraints);
```

```
newLoanButton.addActionListener(new ActionListener()
{
     public void actionPerformed(ActionEvent e)
     {
          newLoanButtonActionPerformed(e);
     }
});

monthsButton.setText("X");
monthsButton.setFocusable(false);
gridConstraints = new GridBagConstraints();
gridConstraints.gridx = 2;
gridConstraints.gridy = 2;
gridConstraints.insets = new Insets(10, 0, 0, 0);
getContentPane().add(monthsButton, gridConstraints);
monthsButton.addActionListener(new ActionListener()
{
     public void actionPerformed(ActionEvent e)
     {
          monthsButtonActionPerformed(e);
     }
});

paymentButton.setText("X");
paymentButton.setFocusable(false);
gridConstraints = new GridBagConstraints();
gridConstraints.gridx = 2;
gridConstraints.gridy = 3;
gridConstraints.insets = new Insets(10, 0, 0, 0);
getContentPane().add(paymentButton, gridConstraints);
paymentButton.addActionListener(new ActionListener()
{
     public void actionPerformed(ActionEvent e)
     {
          paymentButtonActionPerformed(e);
     }
```

```
        });

        analysisLabel.setText("Loan Analysis:");
        analysisLabel.setFont(myFont);
        gridConstraints = new GridBagConstraints();
        gridConstraints.gridx = 3;
        gridConstraints.gridy = 0;
        gridConstraints.anchor = GridBagConstraints.WEST;
        gridConstraints.insets = new Insets(0, 10, 0, 0);
        getContentPane().add(analysisLabel, gridConstraints);

        analysisTextArea.setPreferredSize(new Dimension(250, 150));
        analysisTextArea.setFocusable(false);

analysisTextArea.setBorder(BorderFactory.createLineBorder(Color.BLACK));
        analysisTextArea.setFont(new Font("Courier New", Font.PLAIN, 14));
        analysisTextArea.setEditable(false);
        analysisTextArea.setBackground(Color.WHITE);
        gridConstraints = new GridBagConstraints();
        gridConstraints.gridx = 3;
        gridConstraints.gridy = 1;
        gridConstraints.gridheight = 4;
        gridConstraints.insets = new Insets(0, 10, 0, 10);
        getContentPane().add(analysisTextArea, gridConstraints);

        exitButton.setText("Exit");
        exitButton.setFocusable(false);
        gridConstraints = new GridBagConstraints();
        gridConstraints.gridx = 3;
        gridConstraints.gridy = 5;
        getContentPane().add(exitButton, gridConstraints);
        exitButton.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent e)
            {
                exitButtonActionPerformed(e);
            }
        });

        pack();
        Dimension screenSize =
Toolkit.getDefaultToolkit().getScreenSize();
        setBounds((int) (0.5 * (screenSize.width - getWidth())), (int) (0.5 * (screenSize.height -
getHeight())), getWidth(), getHeight());
        paymentButton.doClick();
    }

    private void exitForm(WindowEvent evt)
    {
        System.exit(0);
    }

    private void computeButtonActionPerformed(ActionEvent e)
    {
        double balance, interest, payment;
        int months;
        double monthlyInterest, multiplier;
        double loanBalance, finalPayment;
        if (validateDecimalNumber(balanceTextField))
        {
            balance =
Double.valueOf(balanceTextField.getText()).doubleValue();
        }
        else
        {
            JOptionPane.showConfirmDialog(null, "Invalid or empty Loan Balance entry.\nPlease
correct.", "Balance Input Error", JOptionPane.DEFAULT_OPTION,
JOptionPane.INFORMATION_MESSAGE);
            return;
        }
        if (validateDecimalNumber(interestTextField))
        {
            interest =
Double.valueOf(interestTextField.getText()).doubleValue();
```

```
        }
        else
        {
            JOptionPane.showConfirmDialog(null, "Invalid or empty Interest Rate entry.\nPlease
correct.", "Interest Input Error", JOptionPane.DEFAULT_OPTION,
JOptionPane.INFORMATION_MESSAGE);
            return;
        }
        monthlyInterest = interest / 1200;
        if (computePayment)
        {
            // Compute loan payment
            if (validateDecimalNumber(monthsTextField))
            {
                months =
Integer.valueOf(monthsTextField.getText()).intValue();
        }
        else
        {
            JOptionPane.showConfirmDialog(null, "Invalid or empty Number of Payments
entry.\nPlease correct.", "Number of Payments Input Error",
JOptionPane.DEFAULT_OPTION, JOptionPane.INFORMATION_MESSAGE);
            return;
        }
        if (interest == 0)
        {
            payment = balance / months;
        }
        else
        {
            multiplier = Math.pow(1 + monthlyInterest, months);
            payment = balance * monthlyInterest * multiplier / (multiplier - 1);
        }
        paymentTextField.setText(new DecimalFormat("0.00").format(payment));
    }
    else
```

```
  {
       // Compute number of payments
       if (validateDecimalNumber(paymentTextField))
       {
               payment =
Double.valueOf(paymentTextField.getText()).doubleValue();
               if (payment <= (balance * monthlyInterest + 1.0))
               {
                   if (JOptionPane.showConfirmDialog(null, "Minimum payment must be $" +
new DecimalFormat("0.00").format((int)(balance * monthlyInterest + 1.0)) + "\n" + "Do you
want to use the minimum payment?", "Input Error", JOptionPane.YES_NO_OPTION,
JOptionPane.QUESTION_MESSAGE) == JOptionPane.YES_OPTION)
                   {
           paymentTextField.setText(new DecimalFormat("0.00").format((int)(balance *
monthlyInterest + 1.0)));
                          payment =
Double.valueOf(paymentTextField.getText()).doubleValue();
                   }
                   else
                   {
                          paymentTextField.requestFocus();
                          return;
               }
           }
       }
       else
       {
           JOptionPane.showConfirmDialog(null, "Invalid or empty Monthly Payment
entry.\nPlease correct.", "Payment Input Error", JOptionPane.DEFAULT_OPTION,
JOptionPane.INFORMATION_MESSAGE);
           return;
       }
       if (interest == 0)
       {
           months = (int)(balance / payment);
       }
       else
```

```
        {
            months = (int)((Math.log(payment) - Math.log(payment - balance * monthlyInterest)) /
Math.log(1 + monthlyInterest));
        }
        monthsTextField.setText(String.valueOf(months));
    }
    // reset payment prior to analysis to fix at two decimals
    payment =
Double.valueOf(paymentTextField.getText()).doubleValue();
    // show analysis
    analysisTextArea.setText("Loan Balance: $" + new
DecimalFormat("0.00").format(balance));
    analysisTextArea.append("\n" + "Interest Rate: " + new
DecimalFormat("0.00").format(interest) + "%");
    // process all but last payment
    loanBalance = balance;
    for (int paymentNumber = 1; paymentNumber <= months - 1; paymentNumber++)
    {
        loanBalance += loanBalance * monthlyInterest - payment;
    }
    // find final payment
    finalPayment = loanBalance;
    if (finalPayment > payment)
    {
        // apply one more payment
        loanBalance += loanBalance * monthlyInterest - payment;
        finalPayment = loanBalance;
        months++;
        monthsTextField.setText(String.valueOf(months));
    }
    analysisTextArea.append("\n\n" + String.valueOf(months - 1) + " Payments of $" + new
DecimalFormat("0.00").format(payment));
    analysisTextArea.append("\n" + "Final Payment of: $" + new
DecimalFormat("0.00").format(finalPayment));
    analysisTextArea.append("\n" + "Total Payments: $" + new
DecimalFormat("0.00").format((months - 1) * payment + finalPayment));
    analysisTextArea.append("\n" + "Interest Paid $" + new
DecimalFormat("0.00").format((months - 1) * payment + finalPayment - balance));
   computeButton.setEnabled(false);
   newLoanButton.setEnabled(true);
   newLoanButton.requestFocus();
}

   private void newLoanButtonActionPerformed(ActionEvent e)
   {
        // clear computed value and analysis
        if (computePayment)
        {
             paymentTextField.setText("");
        }
        else
        {
             monthsTextField.setText("");
        }
        analysisTextArea.setText("");
        computeButton.setEnabled(true);
        newLoanButton.setEnabled(false);
        balanceTextField.requestFocus();
   }

   private void monthsButtonActionPerformed(ActionEvent e)
   {
        // will compute months
        computePayment = false;
        paymentButton.setVisible(true);
        monthsButton.setVisible(false);
        monthsTextField.setText("");
        monthsTextField.setEditable(false);
        monthsTextField.setBackground(lightYellow);
        monthsTextField.setFocusable(false);
        paymentTextField.setEditable(true);
        paymentTextField.setBackground(Color.WHITE);
        paymentTextField.setFocusable(true);
```

```
    computeButton.setText("Compute Number of Payments");
    balanceTextField.requestFocus();
}

private void paymentButtonActionPerformed(ActionEvent e)
{
    // will compute payment
    computePayment = true;
    paymentButton.setVisible(false);
    monthsButton.setVisible(true);
    monthsTextField.setEditable(true);
    monthsTextField.setBackground(Color.WHITE);
    monthsTextField.setFocusable(true);
    paymentTextField.setText("");
    paymentTextField.setEditable(false);
    paymentTextField.setBackground(lightYellow);
    paymentTextField.setFocusable(false);
    computeButton.setText("Compute Monthly Payment");
    balanceTextField.requestFocus();
}

private void exitButtonActionPerformed(ActionEvent e)
{
    System.exit(0);
}

private void balanceTextFieldActionPerformed(ActionEvent e)
{
    balanceTextField.transferFocus();
}

private void interestTextFieldActionPerformed(ActionEvent e)
{
    interestTextField.transferFocus();
}

private void monthsTextFieldActionPerformed(ActionEvent e)
```

```
{
    monthsTextField.transferFocus();
}

private void paymentTextFieldActionPerformed(ActionEvent e)
{
    paymentTextField.transferFocus();
}

private boolean validateDecimalNumber(JTextField tf)
{
    // checks to see if text field contains
    // valid decimal number with only digits and a single decimal point
    String s = tf.getText().trim();
    boolean hasDecimal = false;
    boolean valid = true;
    if (s.length() == 0)
    {
        valid = false;
    }
    else
    {
        for (int i = 0; i < s.length(); i++)
        {
            char c = s.charAt(i);
            if (c >= '0' && c <= '9')
            {
                continue;
            }
            else if (c == '.' && !hasDecimal)
            {
                hasDecimal = true;
            }
            else
            {
                // invalid character found
```

```
                        valid = false;
                    }
                }
            }
            tf.setText(s);
            if (!valid)
            {
                tf.requestFocus();
            }
            return (valid);
    }
}
```