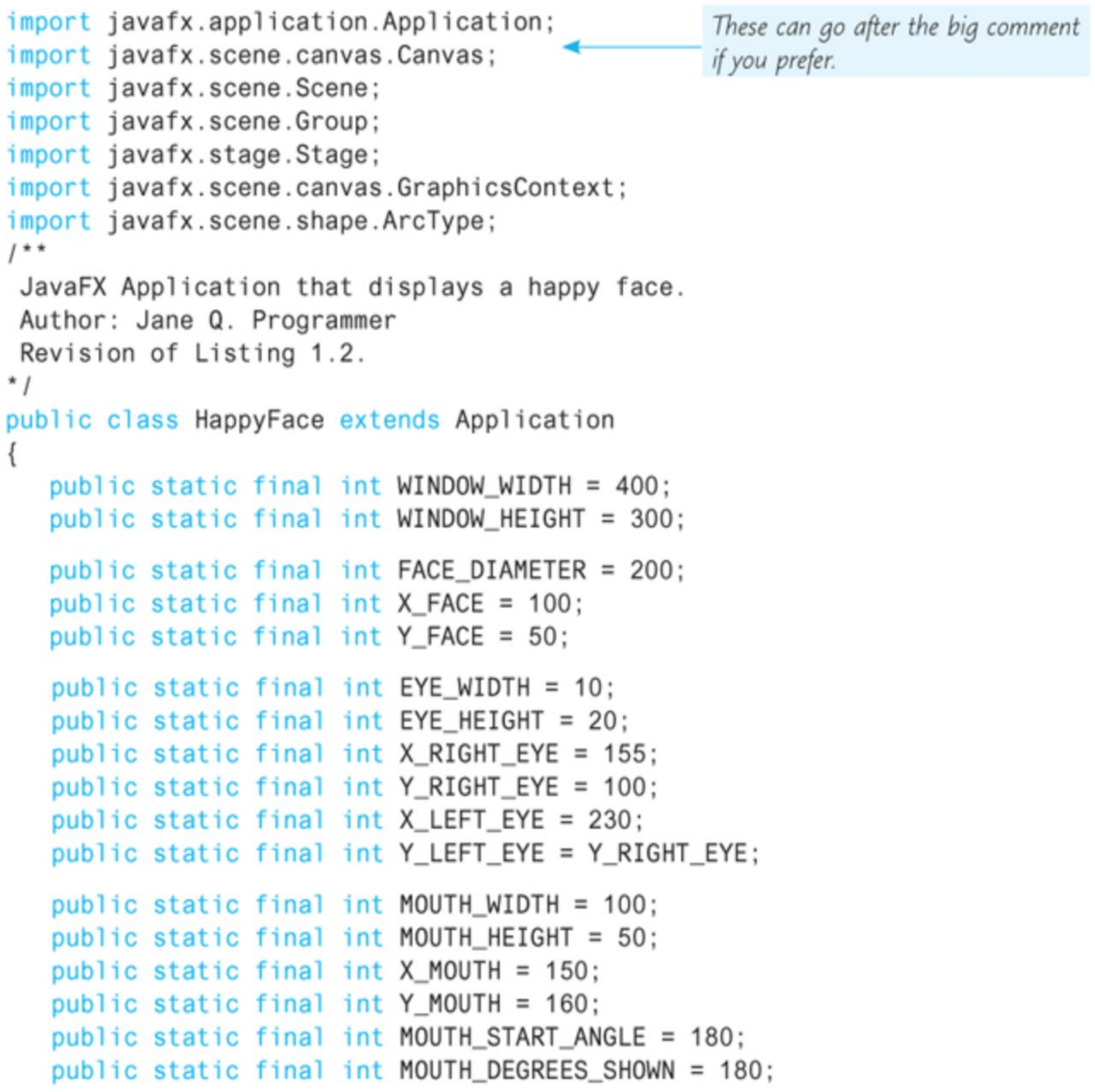
**Style Rules Applied to a JavaFx Application:**

Adding named constants make writing and changing the code much easier.

Revision of Example using Comments and Named Constants:



```
import javafx.application.Application;
import javafx.scene.canvas.Canvas;
import javafx.scene.Scene;
import javafx.scene.Group;
import javafx.stage.Stage;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.shape.ArcType;
/**
 JavaFX Application that displays a happy face.
 Author: Jane Q. Programmer
 Revision of Listing 1.2.
*/
public class HappyFace extends Application
{
   public static final int WINDOW_WIDTH = 400;
   public static final int WINDOW_HEIGHT = 300;

   public static final int FACE_DIAMETER = 200;
   public static final int X_FACE = 100;
   public static final int Y_FACE = 50;

   public static final int EYE_WIDTH = 10;
   public static final int EYE_HEIGHT = 20;
   public static final int X_RIGHT_EYE = 155;
   public static final int Y_RIGHT_EYE = 100;
   public static final int X_LEFT_EYE = 230;
   public static final int Y_LEFT_EYE = Y_RIGHT_EYE;

   public static final int MOUTH_WIDTH = 100;
   public static final int MOUTH_HEIGHT = 50;
   public static final int X_MOUTH = 150;
   public static final int Y_MOUTH = 160;
   public static final int MOUTH_START_ANGLE = 180;
   public static final int MOUTH_DEGREES_SHOWN = 180;
```

*These can go after the big comment if you prefer.*

```
        public static void main(String[] args)
        {
           launch(args);
        }
        @Override
        public void start(Stage primaryStage) throws Exception
        {
              Group root = new Group();
              Scene scene = new Scene(root);
              Canvas canvas = new Canvas(WINDOW_WIDTH, WINDOW_HEIGHT);
              GraphicsContext gc = canvas.getGraphicsContext2D();
              // Draw face outline
              gc.strokeOval(X_FACE, Y_FACE, FACE_DIAMETER, FACE_DIAMETER);
              // Draw eyes
              gc.fillOval(X_RIGHT_EYE, Y_RIGHT_EYE, EYE_WIDTH, EYE_HEIGHT);
              gc.fillOval(X_LEFT_EYE, Y_LEFT_EYE, EYE_WIDTH, EYE_HEIGHT);
              // Draw mouth
              gc.strokeArc(X_MOUTH, Y_MOUTH, MOUTH_WIDTH, MOUTH_HEIGHT,
                           MOUTH_START_ANGLE, MOUTH_DEGREES_SHOWN, ArcType.OPEN);
              root.getChildren().add(canvas);
              primaryStage.setTitle("HappyFace in JavaFX");
              primaryStage.setScene(scene);
              primaryStage.show();
        }
    }
```

Writing such constants helps you to plan and organize your drawing. The named constants enable you to clearly and explicitly specify constraints. For example, the statement

**public static final int Y_LEFT_EYE = Y_RIGHT_EYE;**

ensures that the two eyes appear at the same level.

A graphics program like this one often needs to be tuned by adjusting the various integer values. Finding the right value to change when you need to adjust, say, the mouth width is much easier when you use named constants.

**Programming Tip- Use Named Constants in a Graphics Application**

When designing a drawing, identify the components and their dimensions. Give names to these dimensions and define them as named constants. As much as possible and reasonable, make these constants interdependent. That is, since it is likely that certain dimensions are related to others, define one named constant in terms of another one.

Fine-tuning the values of these constants is easier if you can describe the relationships among the various dimensions symbolically by using named constants.

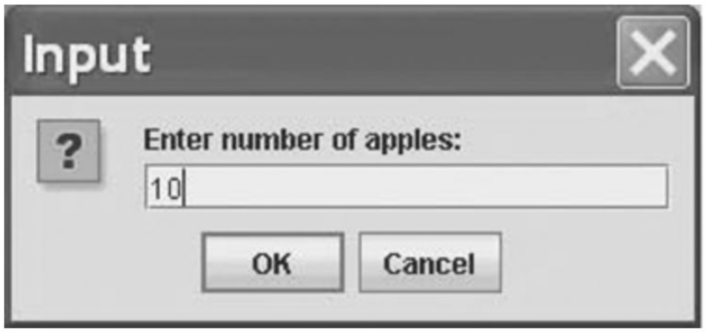
**Class JOptionPane:** Any application program can use windows for I/O

**Example below** contains a very simple Java application program that has a windowing interface. The program produces three windows, one at a time. The first window to appear is labeled Dialog 1. The user enters a number in the text field of this window and then clicks the OK button with the mouse. The first window then goes away and the second window appears. The user handles the second window in a similar way. When the user clicks the OK button in the second window, the second window goes away and the third window appears. Let's look at the details.

**Program Using JOptionPane for I/O**

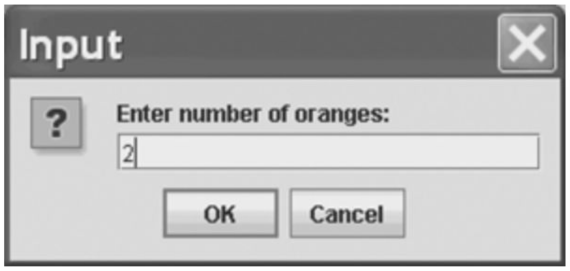
```
import javax.swing.JOptionPane;

public class JOptionPaneDemo
{
    public static void main(String[] args)
    {
        String appleString =
          JOptionPane.showInputDialog("Enter number of apples:");
          int appleCount = Integer.parseInt(appleString);

        String orangeString =
           JOptionPane.showInputDialog("Enter number of oranges:");
        int orangeCount = Integer.parseInt(orangeString);

        int totalFruitCount = appleCount + orangeCount;

        JOptionPane.showMessageDialog(null,
               "The total number of fruits = " + totalFruitCount);
        System.exit(0);
    }
}
```

Dialog 1

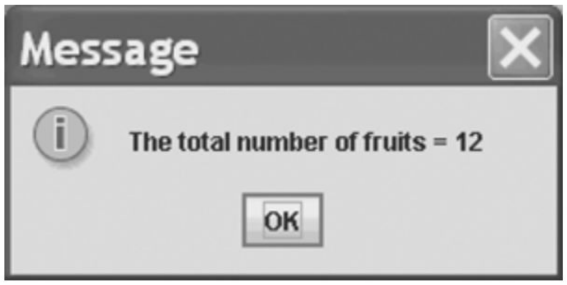


*When the user clicks OK, the window goes away and the next window (if any) is displayed.*

Dialog 2



Dialog 3



The total number of fruits = 12

**Running a JOptionPane Program**

You run a program that uses **JOptionPane**, in the same way you run any application program. You do not run it as if it were an applet.

This program uses the class **JOptionPane** to construct the windows that interact with the user. **JOptionPane** is a standard, predefined class that comes with every installation of Java. To make it available to your program, you write

**import javax.swing.JOptionPane;**

This statement tells the computer where to find the definition of the **JOptionPane** class. You may recall that we mentioned a library called **Swing**, which **is a library of classes that we can use for windowing interfaces.** These libraries are called packages, and in a Java program the **Swing** package is denoted **javax.swing**, with a lowercase ***s***. The class **JOptionPane** is in this package. You put this statement at the start of any program file that uses the class **JOptionPane**.

In **Graphics - Lesson 1** we indicated that **JavaFx** is replacing **Swing**. However, at this time there is not an equivalent version of **JOptionPane** in **JavaFx** with the same simplicity as the version discussed here. For this reason we continue to use **JOptionPane** until a better alternative is made available in **JavaFx**.

The first program instruction for the computer is

**String appleString = JOptionPane.showInputDialog("Enter number of apples:");**

It declares **appleString** to be a variable of type String and then starts the windowing action.

**JOptionPane** is a class used for producing special windows—called **dialog windows, dialog boxes,** or simply **dialogs**—that either obtain input or display output from your program. The method **showInputDialog** produces a dialog for obtaining input. The string argument, in this case "Enter number of apples:", is written in the window to tell the user what to enter. You the programmer choose this string, depending on what sort of input you want. This invocation of the method **showInputDialog** will produce the first dialog shown in **example.** The user clicks the mouse in the text field and then types in some input. The user can use the backspace key to back up and change the input if necessary. Once the user is happy with the input, the user clicks the OK button and the window goes away. As an alternative, the user can press the Enter (Return) key instead of clicking the OK button. The method invocation

**JOptionPane.showInputDialog("Enter number of apples:");**

returns—that is, produces—the input that the user typed into the text field. This invocation is within an assignment statement that stores this input. Specifically, the string input is stored in the variable appleString. When you use **JOptionPane** to read user input, only string values are read. If you want numbers, your program must convert the input string to a number.

The next statement begins by declaring the variable **appleCount** as an int. The int says that the data stored in the variable **appleCount** must be an *integer*. The programmer wants the program to store this integer in the variable **appleCount**. Because **JOptionPane** reads only strings, this means converting the string to a value of type int..

All program input from the user and all output to the user consist of strings of characters. When using the class **JOptionPane** to read numeric input, you must convert the string that is read to the desired numeric value. Even the class Scanner reads user input as a string. However, when you call a method like nextInt or nextDouble, for example, the conversion from string input to numeric input is done for you.

In **example**, the string typed in by the user is stored in the variable **appleString**. Since we expect the user to type the digits of an integer, our program needs to convert this **string** to an **int**. We store the resulting int value in the variable **appleCount**, as follows:

**int appleCount = Integer.parseInt(appleString);**

The **parseInt**() function parses a string and returns an integer.

Integer is a class provided by Java, and **parseInt** is a method of the class **Integer**. The method invocation **Integer.parseInt(appleString)** converts the string stored in the variable **appleString** into the corresponding integer number. For example, if the string stored in appleString is "10", this method invocation will return the integer 10.

**Why do you invoke some methods using a class name instead of an object name?**

Normally, a method invocation uses an object name. For example, if greeting is a variable of type **String**, we write **greeting.length()** to invoke the method **length**. However, when we call the methods of the class **JOptionPane**, we use the class name **JOptionPane** in place of an object name. The same is true of the method **parseInt** in the class **Integer**. **Some special methods do not require an object to be invoked and, instead, are called using the class name. These methods are called static**

**methods.** You can tell whether a standard method in the Java Class Library is static by looking at the documentation for its class on the Oracle Web site.

***Inappropriate Input:*** A crash is an abnormal end to a program’s execution

A program is said to crash when it ends abnormally, usually because something went wrong. When a program uses the method **JOptionPane.showInputDialog** to get input the user must enter the input in the correct format, or else the program is likely to crash. If your program expects an integer to be entered, and the user enters 2,000 instead of 2000, your program will crash, because integers in Java cannot contain a comma.

The next few statements in **example** are only a slight variation on what we have just discussed. A dialog box—the second one in **example**—is produced and gets an input string from the user. The string is converted to the corresponding integer and stored in the variable **orangeCount**.

The next line of the program contains nothing new to you:

```
int totalFruitCount = appleCount + orangeCount;
```

It declares **totalFruitCount** as a variable of type **int** and sets its value to the sum of the **intvalues** in the two variables **appleCount** and **orangeCount**.

The program now should display the number stored in the variable **totalFruitCount**. This output is accomplished by the following statement:

```
JOptionPane.showMessageDialog(null,
                                            "The total number of fruits = " + totalFruitCount);
```

This statement calls the method **showMessageDialog**, which is another method in the class **JOptionPane**. This method displays a dialog window that shows some output. The method has two arguments, which are separated by a comma. For now the first argument will always be written as null. The second argument is the string that is written in the output dialog. So the previous method invocation produces the third dialog shown in **example**. This dialog stays on the screen until the user clicks the **OK** button with the mouse or presses the Enter (Return) key, at which time the window disappears.

Note that you can give the method **showMessageDialog** its string argument using the plus symbol in the same way you do when you give a string as an argument to **System.out.println**. That is, you can append the **integer** value stored in **totalFruitCount** to a **string** literal. Moreover, Java will automatically convert the integer value stored in **totalFruitCount** to the corresponding **string**.

The last program statement is,

**System.exit(0);**

simply says that the program should end. **System is a predefined Java class that is automatically provided by Java, and exit is a method in the class System**. **The method exit ends the program as soon as it is invoked.** In the programs that we will write, the integer argument 0 can be any integer, but by tradition we use 0 because it is used to indicate the normal ending of a program.

***Forgetting System.exit(0);***

If you omit the last line

**System.exit(0);**

from the program in **example**, everything will work as we described. The user will enter input using the input windows, and the output window will show the output. However, when the user clicks the OK button in the output window, the output window will go away, but the program will not end. The "invisible" program will still be there, using up computer resources and possibly keeping you from doing other things. With windowing programs, *it ain't over till it's over*. System.exit(0) is what really ends the program, not running out of statements to execute. So do not forget to invoke System.exit(0) in all of your windowing programs.

What do you do if you forget to call System.exit(0) and the program does not end by itself? You can end a program that does not end by itself, but the way to do so depends on your particular operating system. On many systems (but not all), you can stop a program by typing control-C, which you type by holding down the control (Ctrl) key while pressing the C key.

When you write an application program that has a windowing interface you always need to end the program with the statement

**System.exit(0);**

If the program does not use a windowing interface, you do not need to invoke **System.exit(0)**.

## JOptionPane for Windowing Input/Output

You can use the methods **showInputDialog** and **showMessageDialog** to produce input and output windows—called dialogs—for your Java programs. When using these methods, you include the following at the start of the file that contains your program:

**import javax.swing.JOptionPane;**

The syntax for input and output statements using these methods is given below:

### Syntax For Input :

***String_Variable*** **= JOptionPane.showInputDialog(*String_ Expression*);**

The ***String_Expression*** is displayed in a dialog window that has both a text field in which the user can enter input and a button labeled OK. When the user types in a string and clicks the OK button in the window, the method returns the string. That string is stored in the ***String_Variable***. As an alternative, pressing the Enter (Return) key is equivalent to clicking the OK button. Note that when input is done in this way, it is read as a string. If you want the user to enter, for example, integers, your program must convert the input string to the equivalent number.

### Example

**String orangeString =JOptionPane.showInputDialog( "Enter number of oranges:");**

### Syntax For Output

**JOptionPane.showMessageDialog(null, *String_Expression*);**

The *String_Expression* is displayed in a dialog window that has a button labeled OK. When the user clicks the OK button with the mouse or presses the Enter (Return) key, the window disappears.

**Example**

**JOptionPane.showMessageDialog(null, "The total number of fruits = " + totalFruitCount);**

**Reading Input as Other Numeric Types**

Since the method **JOptionPane.showInputDialog** reads a string from the user, we had to convert it to an **int** in our previous program by using the method **parseInt** from the class **Integer**. You can convert a number represented as a string to any of the other numeric types by using other methods. For example, the following code asks the user to enter a value of type double and stores it in the variable **decimalNumber** of type **double**:

```
String numberString = JOptionPane.showInputDialog

("Enter a number with a     decimal point:");
double decimalNumber = Double.parseDouble(numberString);
```

**Table below** lists the correct conversion method for each numeric primitive type.

**Methods for Converting Strings to Numbers**

| Result Type | Method for Converting |
|---|---|
| `byte` | `Byte.parseByte(`*String_To_Convert*`)` |
| `short` | `Short.parseShort(`*String_To_Convert*`)` |
| `int` | `Integer.parseInt(`*String_To_Convert*`)` |
| `long` | `Long.parseLong(`*String_To_Convert*`)` |
| `float` | `Float.parseFloat(`*String_To_Convert*`)` |
| `double` | `Double.parseDouble(`*String_To_Convert*`)` |

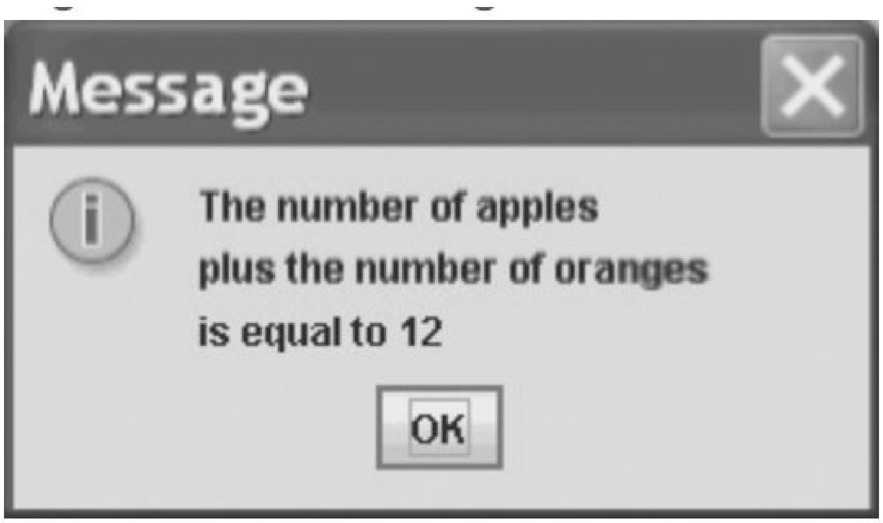
**Multiline Output in a Dialog Window-** If you want to display multiple lines by using **JOptionPane's** method **showMessageDialog**, you can insert the new-line character **'\n'** into the string used as the second argument. If the string becomes too long, which almost always happens with multiline output, you can write each line as a separate string ending with **'\n'** and connect them with plus symbols. If the lines are long or numerous, the window will expand as needed to hold all the output.

**For example, consider**

```
JOptionPane.showMessageDialog(null,
               "The number of apples\n"
               + "plus the number of oranges\n"
               + "is equal to " + totalFruit);
```

## A Dialog Window Containing Multiline Output

## Programming Example Change-Making Program with Windowing I/O



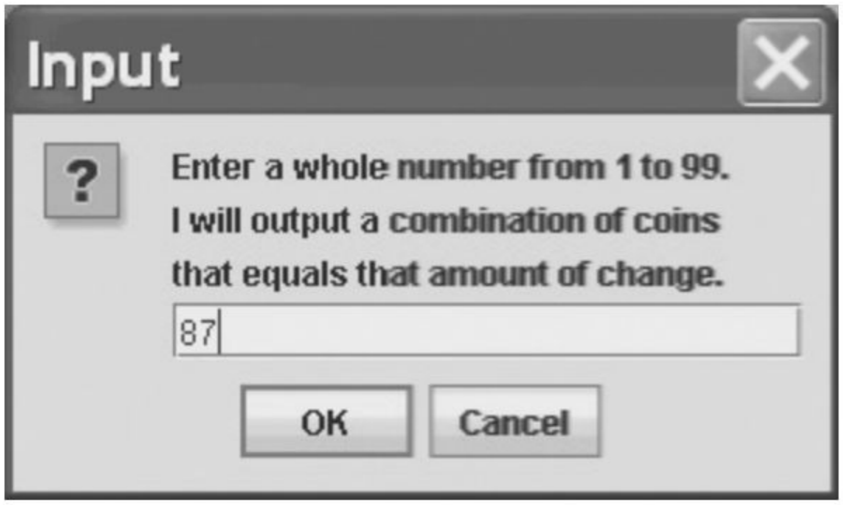
The number of apples
plus the number of oranges
is equal to 12

The program in **example below** has a windowing interface. Notice that both the input dialog and the output dialog display multiple lines of text.

```java
import javax.swing.JOptionPane;
public class ChangeMakerWindow

{
    public static void main(String[] args)
    {
        String amountString = JOptionPane.showInputDialog(
                    "Enter a whole number from 1 to 99.\n" +
                    "I will output a combination of coins\n" +
                    "that equals that amount of change.");
        int amount, originalAmount,
        quarters, dimes, nickels, pennies;
        amount = Integer.parseInt(amountString);
        originalAmount = amount;

        quarters = amount / 25;
        amount = amount % 25;
        dimes = amount / 10;
        amount = amount % 10;
        nickels = amount / 5;
        amount = amount % 5;
        pennies = amount;

        JOptionPane.showMessageDialog(null, originalAmount +
                      " cents in coins can be given as:\n" +
                      quarters + " quarters\n" +
                      dimes    + " dimes\n" +
                      nickels  + " nickels and\n" +
                      pennies  + " pennies");
        System.exit(0);
    }
}
```

*Do not forget that you need* System.exit *in a program with input or output windows.*

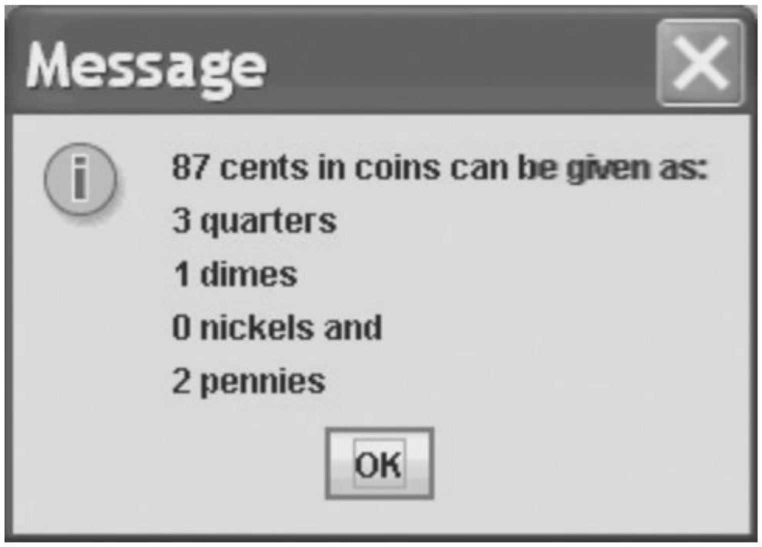
Input Dialog



Enter a whole number from 1 to 99.
I will output a combination of coins
that equals that amount of change.

87

Output Dialog



```
87 cents in coins can be given as:
3 quarters
1 dimes
0 nickels and
2 pennies
```