LABORATORY 3

Object-Oriented Java

In this lab we want to explore Java's features as an object-oriented language. Our initial focus will be on graphical objects. Java provides a large collection of classes that support graphics, and it is relatively easy to use them.

The pre-laboratory reading will discuss the main features of object-oriented programming.

Pre-Laboratory Reading

Concepts

Object-oriented programming requires the programmer to view programs differently from the older procedural programming paradigm. It also requires a language that supports classes and objects. Both Java and C++ can be used for object-oriented programming. Object-oriented programming has been promoted as a more productive and natural way to view solutions to problems. Object-oriented programmers are more productive because the classes they create for one project may be reusable on another project. It is a natural way of viewing software development, because many problems are stated in terms of objects which then interact. The interaction is what causes the computation to progress.

The object-oriented paradigm requires that we examine the following concepts:

- classes, objects, and the creation process
- packages
- public
- methods
- data members or "fields"
- static methods and static data members
- extending a class to get a new class
- a class that implements an interface
- container classes
- a class hierarchy

This will be a whirlwind look at a very sophisticated topic. Don't worry if it doesn't all make sense. The entire course will be spent learning more about each part.

Classes, Objects, and the Creation Process

- An object is said to be an "instance" of a class.
- A class is a model for objects.

This is not very informative, so let's consider what an object is.

An object is a programming entity that has data components representing the current state of the object and methods[1] that allow the other programming entities to interact with it. For example, in a program that performed a traffic simulation at a busy intersection, one type of object would be a car. The car *class* would describe all car objects by indicating what data components each would have and what methods each would have. The data components maybe would include current position, current velocity, and type of car. Each object would have an area of memory set aside for its values for these data components. Typical methods would maybe set a new velocity for the car, allow other objects to obtain the current position or the type of the car, and destroy the car object to remove it from the simulation.

Every class provides a special method that is called when an object is initially created. This is called a *constructor method*, and usually provides special initialization of data components. There can be more than one constructor

1. Informally, a method is defined as a named collection of instructions. See the next page for more information.

method for a class, each distinguishable from the others by the data it requires in order to construct the object. This data is supplied in the form of a *parameter list*, described below.

Packages

A package is a collection of classes. All of the files in the same folder as the current program have their classes as part of the "default" package. The "import" directive in Java is used to tell the current program of other packages that are available and are not found in the current folder.

public

The adjective "public" is often associated with classes and their methods. It means that other classes and methods can use them without being part of the same package.

Java requires that a file contain exactly one public class. Other classes may be defined in the file but may not be designated as public.

Methods

Methods are named collections of instructions. Much of the work of a program is performed by making calls to methods, either ones that are provided with the language (e.g., `Math.sqrt()`) or ones that you or other programmers have written. Methods have the following basic syntax:

```
adjectives return_type NAME( parameter_list )
{
   instructions are here
}
```

The adjectives can be "public/private/protected" and "static," all of which can be omitted.

The `return_type` is either `void` or a primitive type (int, double, char, and so on) or a reference to an object (String, ViewFrame, Canvas, and so on). **It cannot be omitted.** If the return type is void, we call the method a *procedure,* and otherwise call it a *function*. Functions must have the special instruction

```
return expression;
```

among their instructions, where the value of *`expression`* has the type of what is to be returned.

If a car object named `x` had a method `setSpeed`, then it would be "called" in the following manner:

```
x.setSpeed(35);
```

The effect would be to change the speed data component of `x` to 35. This is a *procedure*. We think of `x.setSpeed(35)` as a short name for the instructions in `setSpeed`.

An example of a *function* associated with a car object might be a method getSpeed which would provide the current value of the speed data component for the object. It could be used as follows:

```
int s;
s = x.getSpeed();
if (s == 0) // x is stopped
...
```

Methods are recognized by a name followed by parentheses. Values may be put in the parentheses or not, depending upon the method's definition.

`parameter_list` is optional. If it appears, it specifies data that is provided for use by the method. For example, the method `setSpeed` of the car class could be defined as follows:

```
public void setSpeed(int s)
{
   ...
}
```

The parameter list would consist of only the single parameter s, which would be the new speed. All listed parameters must have their type declared. If more than one parameter is listed, each is separated from the others by commas.

Data Members or "Fields"

Data members are variables associated with a particular object. They are declared just as normal variables are, except they can be designated as public, private, or protected. If `x` were an object and `p` were a public data member, then `x.p` could be used to refer to the object. If the data member `p` was private or protected, then `x.p` would not be legal, and the normal access to `p`'s value would be via a method of `x`, if access were required.

Static

If a method or data member is `static`, then it can be referred to by

```
classname.datamembername
```

This allows you to refer to a data member or use a method without creating an object. A good example is the `Math` class. The method `sqrt` is static, so it can be used as `Math.sqrt(5.6)`. The `Math` class also has a public static data member, `PI`, a `double` value that estimates pi, the ratio of the circumference of a circle to its diameter. Any Java program that needs to use the value pi can do so by referring to `Math.PI`.

A special method is always static—`main`. Recall that execution of a program starts with the first instruction in `main`. If `main` were not static, then it could not be called before the creation of an object; but that would require an instruction prior to calling `main`.

We now can see all of the key parts of main's definition.

```
public static void main(String[] args)
{
   ... instructions
}
```

`main` must be `public` and `static` so it can be called to start a program. It performs actions and has no return statement, so its return type is `void`. It has one parameter, `args`, of type `String[]`. In most applications this parameter is not used, but it is still required in the definition.

Extending a Class

It is quite common for a programmer to have defined a nice class and then later, when writing some other program, realize that a specialized version of that class with some additional capabilities is needed. Rather than redefining the entire class to get the new version, the programmer can "extend" the class and add the new features. All the methods and data components of the original class are automatically available. The programmer adds the new data components or the new methods. If an existing method needs changing, it is just rewritten. The methods that are not rewritten are unaffected.

Consider a class `MotorVehicle` that has general data components and methods for motor vehicles used in a traffic simulation. It would be natural to have specialized versions like `Car` or `Truck` or `Motorcycle`. Java allows this by adding "extends *classname*" in the declaration of the specialized class. For example,

```
public class Truck extends MotorVehicle
```

automatically says that `Truck` can have access to all `MotorVehicle` data that is not private. Protected and public data components are automatically available to the extensions of a class, but private data components are not. Think of protected data as sharable within the family, public data as sharable with anyone, and private data as not sharable at all.

The big advantage of this concept is that we can extend a class provided by Java in one of its many packages. Our extension can be simple, and we can use all of the methods and data that come with the original class. We will see this done in this laboratory and study it extensively in later labs.

Implementing

An *interface* in Java is a listing of methods. A class is said to *implement* the interface if it contains all of the methods of the interface. Of course, it can have additional methods. For example, the Runnable interface lists only one method,

```
public void run()
```

so any class that contains the method `run` is said to implement Runnable.

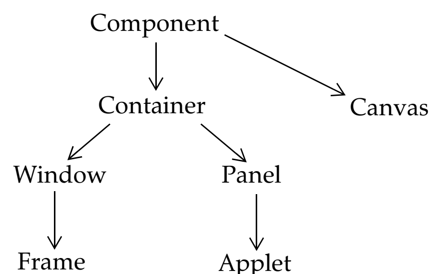
The purpose of an interface is to specify the behavior of an object by specifying what methods it must support. We shall see how useful this can be in a later laboratory.

Containers

Some classes are designed primarily to hold many data values that are objects. These are often called *container* classes. Java provides a Vector class, which maintains a list of objects in linear order. The class Frame is part of the java.awt package and appears as a window with a border. It can contain other visible objects like Panels, Buttons, Canvases, and so on. Container classes always organize the objects they contain in some designated way. Graphical container classes like Frame maintain a layout of their visible components so that when the window is resized, the components are properly repositioned.

Class Hierarchy

The ability to create specialized versions of classes using the "extends" concept and keyword automatically creates a hierarchy of classes. At the top of every class hierarchy is the Object class. The diagram below gives a listing of some classes in the hierarchy used for graphical components. The Component class is



the top of the hierarchy and is a special case of an `Object`. Its objects are visible graphical entities. A `Canvas` is a special extension of a `Component`. It allows drawing and image displays but cannot contain or display other visual components. `Container` extends `Component`; both `Window` and `Panel` extend `Container`. At each level of the hierarchy, each class extends the class above it and inherits all of the methods of the class that it extends. So `Frame` inherits all the methods from `Window`, which inherits all the methods from `Container`, which inherits all the methods from `Component`.

If we were to write a class that extends `Frame`, we could automatically have access to all the methods of `Frame` and its ancestors in the hierarchy!

Form of a Class Definition

Classes have four parts:

- data components—variables
- constructor methods
- other methods
- (optional) inner classes

The inner classes are classes that can be used only within this class. They have access to the data components and methods of the class.

A Person Class

We could define a class with objects that represented individuals and with data components that consisted of the individuals' names and dates of birth. The class might be defined as follows:

```
public class Person
{
   private String name;
   private String dob;   // date of birth

   // constructor - performs initialization
   public Person(String n, String d)
   {
       name = n;
       dob = d;
   }

   // method to change the name
   public void setName(String n)
   {
       name = n;
   }

   // method to access the name
   public String getName()
   {
       return name;
   }
}
```

There are no inner classes defined in this example. An object would be declared and created using the "new" operation.

```
Person p = new Person("Jane Austen", "December 16, 1775");
```

The `new` operation calls the constructor and creates a reference to an object with the given name and date of birth. The statement

```
System.out.println(p.getName());
```

would display the name on the screen.

How to Find Out about Existing Classes and Their Methods

Java comes with a large collection of prewritten classes gathered into various packages. Information on all of these is provided on the Web. A link to the Sun Microsystems documentation on Java classes may be found on the Web page

`file:///A:/index.html`

located on the diskette that accompanies this text.

The documentation is organized by packages. `Java.lang` contains the basic class information concerning the core classes like `Math` and `String`. `Java.awt` and `javax.swing` contain most of the classes used in graphics.

The swing classes were added in version 1.2 to provide a richer collection of visual components.

Before the lab starts, be sure you can answer questions like the ones below using the online documentation.

1. Find the `Frame` class documentation in the package `java.awt`. What methods are defined by `Frame` and which ones are inherited from `Window`?
2. Find the `Math` class in `java.lang`. All the methods are static. What does the documentation say about the `random` method?
3. Find the `Random` class in `java.util`. What are the key methods?

The Laboratory

Objectives

- To become familiar with classes that represent entities that can interact with the user.
- To construct a class that represents a simple ATM (automatic teller machine).

A Guide to the Laboratory

- Your first task is to create a project containing the files `Converters.java` and `CurrencyConverter.java` and compile and run the program. Take a few minutes to try to identify the key parts. There is nothing to turn in at this point.
- Task 2 requires that you modify `CurrencyConverter.java` and test the modifications. Print out and turn in the modified version of CurrencyConverter.java.
- After you close the previous project, task 3 has you create a new project and complete the file `ATM.java` according to step-by-step instructions. This takes some time, so read the instructions carefully. When finished, print out a copy of ATM.java to turn in.

Preliminaries

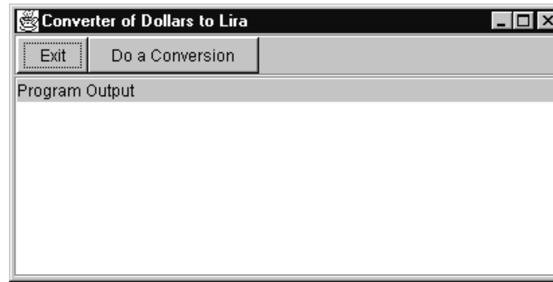
As usual, open the Web page

`file:///A:/index.html`

in your browser after inserting the diskette that accompanies this textbook. Follow the link to the files for this lab. Download the files to your folder `Z:\lab_files\` or a subfolder that you create for this laboratory.

Task 1: An Example—Currency Converter Objects

It is often useful to be able to convert between two currencies. We will create a class with objects that are specific currency converters. When created, these objects will be given the names of two currencies and the conversion factor that is used. The actual arithmetic is simple. Of course, we would like to have a visual form for our converter, so each instance will have its own ViewFrame object like the one shown below. This particular converter was created to convert between U.S. dollars and Italian lira.



The ViewFrame has a button that initiates a conversion. When it is pressed (i.e., clicked on with the mouse), the user is asked for the number of dollars to convert to lira. After the number of dollars is entered, the result of the conversion appears in the message window. Many conversions can be performed. The Exit button terminates the program.

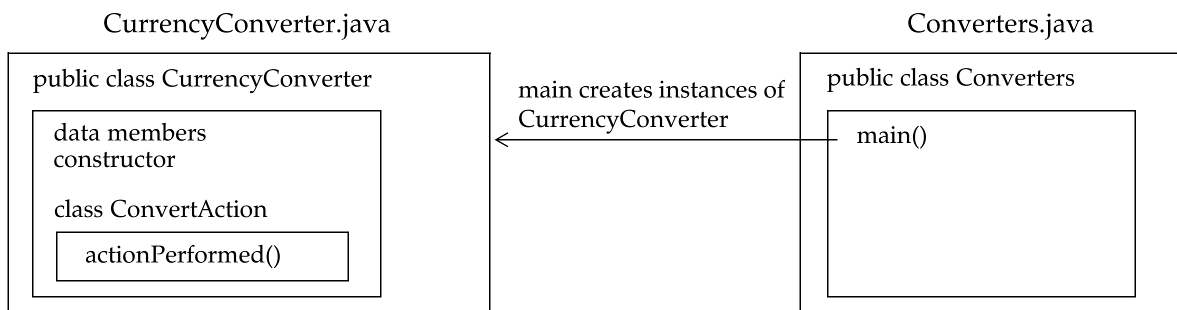
Notice that the title bar indicates which currencies are involved.

Program Structure

We have two files for this program, so a project should be set up. (Refer to Appendix B for how to create a project if you are using an IDE other than Kawa.)

- Open the IDE and close any open project.
- On the `Project` menu, select `New` (`Project>New`) and create the project in the folder for this lab using the name `CurrencyConversion`. (The Kawa IDE requires that files for a project be in the same folder as the project file itself to run properly.)
- Now on the `Project` menu use the `Add Files` option to add the two files `Converters.java` and `CurrencyConverter.java` to the project.

In the Project frame on the left side of the Kawa window you should see the project listed with its two files. The program is structured as two classes in two separate files. Only one has the main method, while the other defines the CurrencyConverter class and uses an inner class.



The core of `Converters.java` is simple.

```
public class Converters
{

  public static void main(String[] args)
  {
     new CurrencyConverter("Dollars", "Lira", 1856.8);
     new CurrencyConverter("Lira", "Dollars", 1.0 / 1856.8);
```

```
    //new CurrencyConverter("Canadian Dollars", "U.S. Dollars", 1 / 1.467);
    //new CurrencyConverter("Dollars", "French Francs", 6.31);
    //new CurrencyConverter("French Francs", "Dollars", 0.158);
  }
}
```

The current main method creates two objects:

- a converter from dollars to lira
- a converter of lira to dollars

The three commented-out lines show how other converters would be created.

Where is the work?

All the work in the program is done by the objects. The program creates objects and lets them interact with each other or the user. This is the essence of object-oriented programming.

Compiling and Running the Program

Compile all the project files. (In Kawa, select `Build>Rebuild All` or press **Shift** + **F7**.)

There should be no errors, so when compilation is completed, run the program. (In Kawa, select `Build>Run` or press **F4**.)

Warning: You will see only one window at the upper-left corner of the screen. Actually, there are two windows located at the same spot and one hides the other from view. Use the mouse to move the top window to the right and allow the other window to appear.

You should try a couple of conversions before exiting the application.

Now open the file `Converters.java` and uncomment the three lines in the main method. Rebuild all the files and re-run the program. You should see five windows—each an instance of the CurrencyConverter class—once you move the windows with the mouse.

Test, then exit the application. In the post-laboratory section you will see how to set the location of windows so they are not all at the same place.

CurrencyConverter Up Close

Open the file `CurrencyConverter.java` and print a copy to write notes on. The class `CurrencyConverter` has three parts:

- private data components

```
// data components
private double conversionConstant = 1.0;
private String sourceCurrencyName;
private String targetCurrencyName;
private ViewFrame vf;
```

- a *constructor* method that creates a ViewFrame object and assigns values to the data components

```
// constructor method
public CurrencyConverter(String source, // name of starting currency
                         String target, // name of converted currency
                         double c)      // multiplier to do conversion
{
    sourceCurrencyName = source;
    targetCurrencyName = target;
    conversionConstant = c;
    String msg = "Converter of " + sourceCurrencyName + " to "
                 + targetCurrencyName;
    vf = new ViewFrame(msg);
    // add an action button
    vf.addActionButton(new ConvertAction("Do a Conversion"));
    // this sets the size to be only half as big vertically
    // as the default
    vf.setSize(ViewFrame.WIDTH, ViewFrame.HEIGHT / 2);
```

```
        vf.setVisible(true);
    }
```

- an inner class that is an *action* class

```
    class ConvertAction extends AbstractAction
    {
        // simple constructor required
        public ConvertAction(String s)
        {
            super(s); // sets up button label in AbstractAction
        }

        // the method called when the associated JButton is clicked
        public void actionPerformed(ActionEvent e)
        {
                int amount = vf.readInt("Enter the number of "
                                        + sourceCurrencyName);
                double d = amount * conversionConstant;
                vf.println(amount + " " + sourceCurrencyName
                                  + " equals " + d + " "
                                  + targetCurrencyName);
        }
    }// end of ConvertAction class
```

The inner class is just like any other class except it is defined *inside* an enclosing class *but not inside a method*. By being defined there, it has full access to the data components of its enclosing class, but is not accessible to classes other than CurrencyConverter. It really needs access to only the ViewFrame object so it can interact with the user using this ViewFrame object.

Except for the layout of the ViewFrame object, which is handled in the constructor, all of the work is done in the method `actionPerformed()`, which is called automatically when the button is pressed. This is shown in bold. It requests an amount from the user, does the arithmetic, and then displays the answer neatly.

Task 2: Questions and Actions

Answer the following questions *and* modify CurrencyConverter.java to try out your answers:

1. How would you change the label on the conversion button to “CONVERT” instead of “Do a Conversion”?
2. How would you modify the code so that the user could enter a floating point value like 210.75 rather than a whole number when starting a conversion?
3. How would you modify the code so that after the user entered the amount to convert, the program would not do a conversion for a negative value? If a negative were entered, the program would execute the instruction

   `vf.showWarningMsg("Negative values not allowed.");`

   and take no further action. If a non-negative were entered, the conversion would be done and displayed as usual. (***Note:*** This requires use of an if-else statement. Look in the previous laboratory for a reminder of how this works.)

Once these questions have been answered and implemented, print out a copy of your modified CurrencyConverter.java and turn it in. Be sure to put your name and the name of your lab partner on it. Close the current project.

Task 3: ATMs

You are to create a new project that will have two files:

- `ATMCreater.java`, with a main method that creates ATM objects
- `ATM.java`, a class that simulates an automated teller machine (ATM)

It is similar in format to the currency conversion example.

Getting Started

First create a project called `ATM_Project` and add the file `ATMCreator.java` to it. `ATMCreator.java` is complete and need not be modified.

Now add the file `ATM.java` to the project. It is partially complete and requires some additions. The file is shown below, with the places for additions marked with boxes.

```
// ATM.java
// Programmers: [                              ]
//
// Description: Simulates an automated teller machine, but just the
// aspect. It is initiated with a certain number of $20 and $5 bills.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import LabPkg.*;

public class ATM
{

    ViewFrame vf;

   [                                        ]

    public ATM(int t, int f) // # of $20 and $5 bills to start with
    {
        vf = new ViewFrame("ATM Machine");
        vf.addActionButton(new GeneralButton("Withdrawal"));
        //vf.setIOEcho(true);
        vf.setVisible(true);

            [                                        ]

    }

    //////////////////////////////////////////////////
    ////////////// inner class for a button /////////
    //////////////////////////////////////////////////
    public class GeneralButton extends AbstractAction
    {
        public GeneralButton(String s)
        {
            super(s);
        }

        public void actionPerformed(ActionEvent e)
        {
            // action when button is pressed goes here

               [                                     ]

        }
```

```
    }
}
```

The inner class, which represents the button, is called `GeneralButton`. This shows that the name is not important provided it is used consistently within the program. In the `CurrencyConverter.java` file, the button class is called `ConvertButton`.

We do need to provide some specifications.

Specifications

First, the constructor will be given two non-negative integers representing the number of $20 bills and the number of $5 bills available in the machine when it starts up. The creation in ATMCreator.java looks like

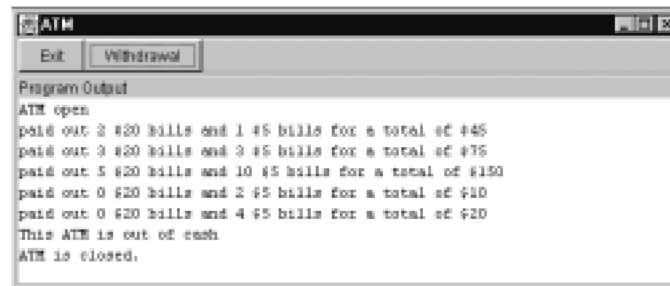
```
new ATM(30, 40); // 30 $20 bills,  40 $5 bills
```

The constructor indicates the values by the two parameters `t` and `f`. (Look at the file `ATM.java` and find the constructor.) In the example above, `t` would get 30 and `f` would get 40. If the constructor were to get a negative value for a parameter, it should use 0 as the value.

Please note that the two values `t` and `f` in the constructor will not be remembered after the creation process is over unless they are assigned to data members for the object. We suggest declaring two data members, `twenties` and `fives`, which will be used to record the current number of $20 and $5 bills available, respectively.

This ATM is simple because we don't ask for account number and PIN (personal identification number) and only do withdrawals. We simulate the withdrawal of funds, decreasing the number of $20 and $5 bills as the simulation progresses.

The ATM will have a ViewFrame object to display itself. Below is an example of the window after several transactions.



This ATM started with 10 twenties and 20 fives. When the money finally ran out it displayed the last two lines, indicating it was closing.

When the "Withdrawal" button is pressed, the machine will request the user to enter the amount as a multiple of $5 (without the dollar sign). If the user enters an amount like 34 or a negative value, the program displays a warning message using a pop-up dialog (recall that ViewFrame has a `showWarningMsg` method) and waits for the button to be pushed again.

Once a proper amount has been entered, the ATM will pay out as many $20 bills as possible, then use $5 bills for the rest. For example, if $65 is requested, the ATM will try to pay out three $20 bills and a single $5 bill. It will display the message

```
paid out 3 $20 bills and 1 $5 bill for a total of $65
```

in the output area of the window.

It is possible that the ATM will have insufficient funds to meet the request. If so, a pop-up warning message will display, saying the ATM has insufficient cash. (Again, use the `showWarningMsg` method of ViewFrame.)

Hints on Computations

If `amt` stands for the amount requested, then you can use `amt / 20` to get the maximum number of twenties you should give out. Recall that in integer division this gives the quotient.

It is possible that there may be enough money but a limited number of $20 bills. For example, if the ATM had one $20 bill and 30 $5 bills, then a $65 request would get the single $20 bill and nine $5 bills. You can handle this with the expression

```
int t;
t = Math.min(twenties, amt / 20);
```

where `twenties` is the name of the variable containing the current number of $20 bills in the ATM. `t` is now the *actual* number of $20 bills that the ATM will dispense. This avoids the use of some if-else statements.

Once you know how many $20 bills will be dispensed (`t`), you must see how many $5 bills are required to complete the withdrawal request. That is given by

```
int f;
f = (amt - t * 20) / 5;
```

Now all we have to do is see if we have that many $5 bills available. If so, we give out the money, reducing our stock of $20 and $5 bills. If not, we display "insufficient cash" in a warning window. This requires a Java if statement.

The code described above will go in the `actionPerformed` method associated with the Withdrawal button and will involve some if statements. If you wish to leave the actionPerformed method early, i.e., without executing additional statements, give the statement

```
return;
```

Notice it is not "return *expression*;" as `actionPerformed` is a procedure, not a function. For example, if a negative value was entered, you could have the statement

```
if (amt < 0)
{
   vf.showWarningMsg("Invalid amount entered.");
   return;
}
```

The return statement prevents the execution of any more statements in `actionPerformed`.

Step by Step

- Compile the project and run it. Currently, nothing happens when the button is pressed.
- The message "ATM is open" is displayed when the ATM becomes visible. At the end of the constructor after the call to setVisible you should add the statement

  ```
  vf.println("ATM is open");
  ```

- Have your action associated with the pressing of the Withdrawal button (this action is in the method actionPerformed) to initially just get the amount and display it in the output window. You may just want to

uncomment the `vf.setIOEcho(true)` statement in the constructor. Compile and run this.

- Add the data declarations for `twenties` and `fives` as discussed above. If you don't know where they should go, ask! They both should be integers. Initialize them in the constructor using t and f. Just to be certain you have the data members set up correctly, use the `vf.println()` method to print out the values of `twenties` and `fives` just before the constructor method finishes.
- Now try to get the logic correct that will dispense the bills or indicate insufficient cash on hand for the request. This requires some thought and is not easy. Don't expect to just dash something off and have it work the first time.

When you give out the money don't forget to decrease the values of `twenties` and `fives`. You will also have to figure out how to know when there is no money at all and the ATM should be closed.

When you and your partner complete the project, be sure your names are in a comment in the `ATM.java` file. Turn in a printout.

Post-Laboratory Exercises and Readings

Exercises

The file `XXXXXXXX.java` can be used as a template for classes like ATM. For simplicity, it contains a main method so you can create a single file program. Let's create a "cloning" class. It will have a window with a button that when pressed will create another version of itself. Here are the steps:

- Close any projects that might be open.
- In the IDE's editing window, open XXXXXXXX.java.
- With your cursor at the very beginning of the file, select `Edit>Replace`. When the box pops up, replace all occurrences of `XXXXXXXX` (eight upper-case Xs) with `CLONE`.
- Perform a "Save As" and name it `CLONE.java`.
- Remove the comment about the replacement and put in your name as the programmer.
- In the constructor use "Clone Machine" for the ViewFrame's title.
- In the constructor use "Clone Me!" for the label that will be on the button.

We won't have any data members except the ViewFrame object `vf`, so there is no more to do except fill in the `actionPerformed` method.

- In the `actionPerformed` method add the single line

```
new CLONE();
```

Save the file, compile it, and run it. When the window appears, move it from the upper-left corner to another location on the screen. Then press the "Clone Me!" button.

You can press any of the clone's buttons and a new clone will be made. Unfortunately all clones will initially appear at the upper-left corner and may hide each other. We will fix that problem with the modification below.

Killing one window kills them all.

Modification

We want to make the vf window appear at a random location on the screen. The statement

```
vf.setLocation(50,10);
```

will locate the upper-left corner of the ViewFrame object `vf` 50 pixels from the left border and 10 pixels from the top border of the screen. Unfortunately, if we used this all windows would appear at this spot!

To get a random location we will select a random integer from 0 to 99 for the first coordinate of the `setLocation` method. The expression

```
(int)(Math.random() * 100)
```

will give us the desired random integer. We will select a random value from 0 to 49 for the second coordinate. This will be done by a similar expression using 50 instead of 100. This will work because `Math.random()` gives a random number from 0.0 up to but less than 1.0.

In the constructor just before the vf.setVisible(true); statement, add the lines

```
vf.setLocation((int)(Math.random() * 100), // horizontal offset
               (int)(Math.random() * 50)); // vertical offset
```

Recompile and run the program again. Do the cloned windows appear at differing locations?

Extension

This example has no data members except for the ViewFrame `vf`. Think about an identifying name associated with each `CLONE` object. If the main method created several `CLONE` objects, each object would create clones with their own names. The main method might look like

```
public static void main(String[] args)
{
   new CLONE("Fred");
   new CLONE("Sarah");
   new CLONE("Joe");
}
```

This would require that the constructor have a single String parameter, as in

```
public CLONE(String n) // n is the name
{
   ...
}
```

Remembering the name requires another data member like

```
String name;
```

Add its declaration to the class, and in the constructor have

```
name = n;
```

Of course during cloning you would use an instruction like

```
new CLONE(name);
```

Make the indicated changes. Recompile and run the program. Close the project.