Project 3: Inventory Tracking

In completing this project you will use the *BlueJ* IDE to develop an application that tracks products in the warehouse of a mail-order music business that stores and ships DVDs, CDs, and cassette tapes.

This project begins with instructions on how to prepare the *BlueJ* IDE for a new project. The main body of the project then falls into three parts. In Part A you will develop a small hierarchy to represent the products that are to be stored in the warehouse. In Part B, you will develop classes to represent the warehouse and its contents. Part C is both challenging and optional. It involves designing a system for tracking changes in inventory as some products are received and others are shipped out.

This project and those that follows it assume that, on the basis of your work on earlier projects, you know how to create classes in the *BlueJ* IDE, including how to create the class containing the `main` method.

Preparing BlueJ

1) Start the *BlueJ* IDE. Choose **New Project...** from the **Project** menu. Your teacher may have set up a folder into which all students' Java source code files are to be saved. If so and if this folder does not appear in the *New Project* dialog, then navigate to it in the dialog's tree control in the usual way so that its name appears in the "Look In" selector (see Figure 1).

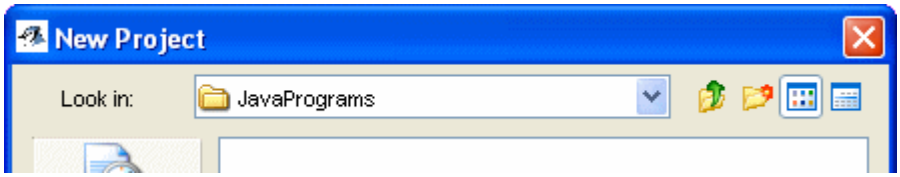

Figure 1: The New Project Dialog.

2) If this is your first project with *BlueJ*, you should create your own personal source code folder that will contain your various project files. Your teacher may assign you your own personal folder name. Click the "Create New Folder" icon — the second icon to the right of the "Look In" selector. A new folder icon appears in the body of the dialog (see Figure 2).

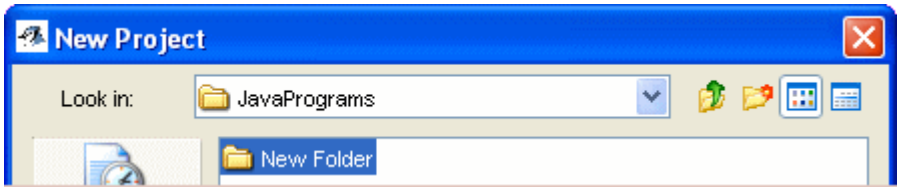

Figure 2: Creating Your Personal Source Code Folder.

3) Click the "New Folder" label once, change it to your personal folder name, and press the **Enter** key. Double-click the folder icon to the left of your new personal folder's name to transfer that name into the "Look In" selector. Enter

the new project's name, "Project3" (without the quote marks), into the *File name* box near the bottom of the *New Project* dialog, and click the **Create** button. The *New Project* dialog goes away and the new project's name appears in the main *BlueJ* application window's title bar (see Figure 3).

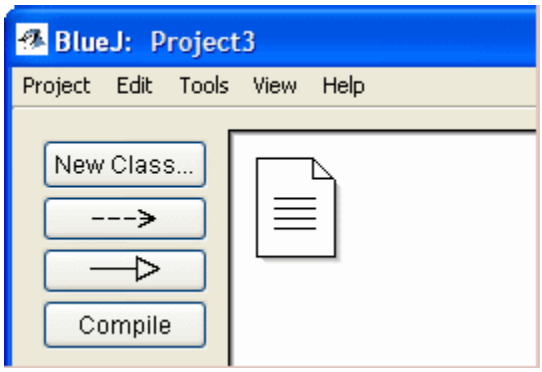

Figure 3: Project3 — Ready To Go!

BlueJ is now set up for you to begin programming.

Part A: DVDs, CDs and Cassettes

This is a required part of this project. In this part, you write class definitions for `MusicMedia`, `Disk`, `CassetteTape`, `CompactDisk`, and `DigitalVideoDisk`. The class `MusicMedia` represents physical media (such as compact disks, digital video disks and cassette tapes) on which music can be stored. Each instance contains information about the specific media type together with information about the artist and title. In addition, each instance contains a unique name called a *Stock Keeping Unit (SKU)* that identifies this product, distinguishing it from every other product. The classes `Disk` and `CassetteTape` will be derived from the `MusicMedia` class, and the classes `CompactDisk` and `DigitalVideoDisk` will be derived from `Disk`. Work through each of the numbered items in order.

1) Create a new `MusicMedia` class in *BlueJ* and enter the following definition:

```
public class MusicMedia
{
    private String myTitle,
                   myArtist,
                   mySKU;

    public MusicMedia( String title, String artist, String sku )
    {
        myTitle = title;
        myArtist = artist;
        mySKU = sku;
    }

    public String getTitle()
    {
        return myTitle;
    }

    public String getArtist()
    {
        return myArtist;
    }
```

```
    public String getMediaType()
    {
        return "Unknown";
    }

    public String getSKU()
    {
        return mySKU;
    }
}
```

2) Add a `toString` method to the `MusicMedia` class. The method should return a `String` containing the title followed by the name of the artist in parentheses. After adding a suitably-named main class to your project, test your overridden `toString` method using the following definition for `main`:

```
public static void main( String[] args )
{
    MusicMedia m = new MusicMedia( "The Eminem Show", "Eminem", "1323-6" );
    System.out.println( m );
}
```

When you compile and execute your program, the output should be the `String`:

```
The Eminem Show (Eminem)
```

3) Create a new class, `Disk`, derived from `MusicMedia`. Override the `getMediaType` method so that it returns the `String "Disk"`. If necessary, override the `toString` method so that the `String` it returns begins with `"Disk - "`. Test your new class using the following definition for `main`:

```
public static void main( String[] args )
{
    Disk d = new Disk( "The Eminem Show", "Eminem", "1323-7" );
    System.out.println( d );
}
```

When you compile and execute your program, the output should be the `String`:

```
Disk - The Eminem Show (Eminem)
```

4) Create a new class, `CompactDisk`, derived from `Disk`. Override the `getMediaType` method so that it returns the `String "CD"`. If necessary, override the `toString` method so that the `String` it returns begins with `"CD - "`. Test your new class using the following definition for `main`:

```
public static void main( String[] args )
{
    CompactDisk c =
        new CompactDisk( "The Eminem Show", "Eminem", "1323-8" );
    System.out.println( c );
}
```

When you compile and execute your program, the output should be the `String`:

```
CD - The Eminem Show (Eminem)
```

5) Create a new class, `DigitalVideoDisk`, derived from `Disk`. Override the `getMediaType` method so that it returns the `String "DVD"`. If necessary, override the `toString` method so that the String it returns begins with `"DVD - "`. Test your new class using the following definition for `main`:

```
public static void main( String[] args )
{
    DigitalVideoDisk v =
        new DigitalVideoDisk( "The Eminem Show", "Eminem", "1323-9" );
    System.out.println( v );
}
```

When you compile and execute your program, the output should be the `String`:

```
DVD - The Eminem Show (Eminem)
```

6) Create a new class, `CassetteTape`, derived from `MusicMedia`. Override the `getMediaType` method so that it returns the `String "Cassette"`. If necessary, override the `toString` method so that the String it returns begins with `"Cassette - "`. Test your new class using the following definition for `main`:

```
public static void main( String[] args )
{
    CassetteTape t =
        new CassetteTape( "The Eminem Show", "Eminem", "1323-10" );
    System.out.println( t );
}
```

When you compile and execute your program, the output should be the `String`:

```
Cassette - The Eminem Show (Eminem)
```

7) The following code generates an `ArrayList<MusicMedia>` of instances of `MusicMedia` with randomly chosen media types, titles, and artist names. Include this code as a `static` method in the class in which you have defined your `main` method:

```
public static ArrayList<MusicMedia> MakeMusicCatalog( int size )
{
    String[] titles =
        {
            "Greatest Hits Volume 1", "Greatest Hits Volume 2",
            "The Best Of", "Love Songs",
            "The Early Years", "The Later Years"
        };
    String[] artists =
        {
            "Michael Jackson", "Eminem",
            "The Beatles", "Shania Twain",
            "Limp Bizkit"
        };

    ArrayList<MusicMedia> a = new ArrayList<MusicMedia>();
    Random rn = new Random();
```

```
        for ( int i = 0 ; i < size ; i++ )
        {
            MusicMedia m;

            int mediatype = rn.nextInt( 3 );
            String title = titles[ rn.nextInt( titles.length ) ];
            String artist = artists[ rn.nextInt( artists.length ) ];
            String sku = "1234-" + i;
            if ( mediatype == 0 )
                m = new CompactDisk( title, artist, sku );
            else if ( mediatype == 1 )
                m = new DigitalVideoDisk( title, artist, sku );
            else
                m = new CassetteTape( title, artist, sku );

            a.add( m );
        }

        return a;
    }
```

Insert the following `import` statement before the header of the definition of the class in which you have defined your `main` method:

```
import java.util.*;
```

(This tells the Java compiler where to look for the definition of the `ArrayList` class.)

Modify your `main` method to include the following code, and execute your program to test that your code compiles and runs correctly:

```
ArrayList<MusicMedia> catalog = MakeMusicCatalog( 100 );
for ( MusicMedia m : catalog )
    System.out.println( m );
```

This completes Part A of this project.

Part B: The Warehouse

This too is a required part of this project. In this part you define a class that generates objects that simulate the physical storage locations in a typical warehouse and you write a static method for printing out a detailed list of all the items stored in the warehouse.

The storage locations are called *bins*, and a warehouse usually has a large number of them. Each bin has a name that allows the warehouse workers to identify its location within the warehouse. In each bin there are usually many items. The items in a bin may all have the same SKU, or the bin may contain items with a variety of SKUs. For example, all the items in Bin A below have the same SKU, whereas Bin B contains items with two different SKUs:

Bin A	*SKU 1234-5, Quantity 12*
Bin B	*SKU 1234-5, Quantity 19* *SKU 5432-1, Quantity 128*

Notice that Bins A and B both contain items with SKU 1234-5. It will quite often happen that the warehouse has so many units of in-demand items (such as an Eminem CD) in stock that they have to be distributed among several bins.

1) Let's call the collection of all the items in a bin that share the same SKU a "bin item". Complete this definition of a class that implements bin items, and then test your code by suitably modifying your `main` method:

```
public class BinItem
{
    private String mySKU;
    private int    myQuantity;

    public BinItem( String sku, int quantity )
    {
        mySKU = sku;
        myQuantity = quantity;
    }

    public String getSKU()
    {
        return mySKU;
    }

    public int getQuantity()
    {
        return myQuantity;
    }

    public String toString()
    {
        // Write code here to return a string using
        // the format: "SKU <sku>: <quantity>". For
        // example: "SKU 12345-15: 4320".
    }
}
```

2) Our next task is to introduce a `Bin` class. Each object of this class represents a bin in the warehouse. It has two instance variables: a `String` that contains the bin name and an `ArrayList<BinItem>` that contains a `BinItem` for each of the SKUs stored in the bin. It also has a method making it possible for us to add a new SKU to the bin. Implement the `Bin` class by completing the following definition:

```
import java.util.*;

public class Bin
{
    private String  myName;
    private ArrayList<BinItem>  myContents;

    public Bin( String name )
    {
        // write your code here
    }

    public String getName()
    {
        return myName;
    }

    public ArrayList<BinItem> getContents()
    {
        // write your code here
    }
```

```
    public void add( BinItem b )
    {
         // write your code here
    }

    public String toString()
    {
        String  s = "Bin " + myName + ":\n";
        for ( BinItem b : myContents )
            s += b + "\n";

        return s;
    }
}
```

Notice the appearance of `\n` in a couple of places in the above definition of the `toString` method. Recall that this inserts a newline character into the string; it causes the next part of the string to appear starting at the left margin of the next line.

3) Modify your `main` method to test your definition of `Bin`. For example, you might use the following:

```
ArrayList<Bin> warehouse = new ArrayList<Bin>();
Bin a = new Bin( "A" );
Bin b = new Bin( "B" );
warehouse.add( a );
warehouse.add( b );
a.add( new BinItem( "1234-0", 500 ) );
a.add( new BinItem( "1234-1", 25 ) );
a.add( new BinItem( "1234-2", 7720 ) );
b.add( new BinItem( "1234-3", 1000 ) );

for ( Bin bn : warehouse )
    System.out.println( bn );
```

4) The `String` representation of a `BinItem` object just gives its SKU and the quantity. While this is really all that is required for the purposes of inventory control, it would be more informative to human beings if there were a way to identify which CD, DVD, or cassette has that SKU. Recall that the `ArrayList` produced by the `MakeMusicCatalog` method combines information about various music media with their corresponding SKUs. We need a way to look up a SKU in such an `ArrayList`. Add the following method to the class that contains your `main` method, and replace the two comments by suitable code that will correctly complete the definition:

```
public static String lookupMedia( ArrayList<MusicMedia> catalog,
                                  String sku )
{
    for ( MusicMedia m : catalog )
    {
        if ( /* your code here */ )
            return /* your code here */;
    }

    return "SKU not in catalog";
}
```

5) Modify your `main` method to test your definition of `lookupMedia`. For example, you might use the following:

```
ArrayList<MusicMedia> catalog = MakeMusicCatalog( 10 );
BinItem a = new BinItem( "1234-3", 500 );
```

```
BinItem b = new BinItem( "9999-9", 2400 );
System.out.println( lookupMedia( catalog, a.getSKU() ) );
System.out.println( lookupMedia( catalog, b.getSKU() ) );
```

6) Now that we can look up a SKU in a catalog of music media items, we should be able to define a method for writing out a more human-friendly listing of the contents of a collection of warehouse bins. (The `toString` method of the `Bin` class uses the `toString` method of the `BinItem` class. It therefore generates lists that only involve SKUs and quantities.) Add the following method to the class that contains your `main` method, and replace the comment by suitable code that will correctly complete the definition:

```
public static String detailedInventory( ArrayList<MusicMedia> catalog,
                                        ArrayList<Bin> warehouse )
{
    String s = "";

    // your code goes here

    return s;
}
```

The strategy we recommend is as follows:

- Use a `for`-each loop to cycle through all the bins in the warehouse.
- For each bin, extend `s` by the String `"Bin "` followed by the name of the bin, followed by a colon and `\n` (to start a new line).
- Cycle through all the bin items in the current bin, for each one extending `s` by a new line of text that begins with the result of looking up the current bin item's SKU in the input `catalog`, and continues with a comma followed by the `String` representation of the bin item.

7) Modify your `main` method to test your definition of `detailedInventory`. For example, you might use the following:

```
ArrayList<MusicMedia> catalog = MakeMusicCatalog( 10 );
ArrayList<Bin> warehouse = new ArrayList<Bin>();
Bin a = new Bin( "A" );
Bin b = new Bin( "B" );
warehouse.add( a );
warehouse.add( b );
a.add( new BinItem( "1234-0", 500 ) );
a.add( new BinItem( "1234-1", 25 ) );
a.add( new BinItem( "1234-2", 7720 ) );
b.add( new BinItem( "1234-3", 1000 ) );

System.out.println( detailedInventory( catalog, warehouse ) );
```

The output should look something like this:

```
Bin A:
CD - The Later Years (Shania Twain), SKU 1234-0: 500
Cassette - Greatest Hits Volume 2 (The Beatles), SKU 1234-1: 25
Cassette - Greatest Hits Volume 1 (Shania Twain), SKU 1234-2: 7720
Bin B:
Cassette - Greatest Hits Volume 2 (Michael Jackson), SKU 1234-3: 1000
```

This completes Part B of the project.

Part C: Shipping and Receiving

This final part of the project is challenging; it is also optional. Typical warehouses are dynamic places with goods constantly coming and going. In this part you write code to simulate shipping (where goods leave the warehouse) and receiving (where goods enter the warehouse to be placed in bins).

After each of the following tasks, modify the `main` method and thoroughly test your code before proceeding to the next task.

1) In preparation for the greater demands that will be placed on the `Bin` class in the context of a warehouse, add the following two methods to the `Bin` class:

   - `remove`: This takes a single argument, namely, a non-negative integer `i` that is strictly less than the number of bin items in the bin. It returns no value, but it removes from the bin the bin item with index `i`.

   - `totalQuantity`: This takes no arguments. It returns the sum total of the quantities of all the bin items in the bin.

2) Modify the `add` method of the `Bin` class so that, when asked to add a bin item `b`, a bin first checks whether or not it already contains a bin item with the same SKU as `b`. If it does, then the bin item `a` already in the bin should be removed and a new bin item should be added whose quantity is the sum of the quantities of `a` and `b`. If it does not, then bin item `b` should just be added in the usual way.

3) Below we provide a skeleton of the first version of a definition for a `Warehouse` class. The argument, `binMax`, is a positive integer that specifies the maximum total quantity that any bin in the warehouse may contain. There are notes following the skeleton definition that provide information about the desired behavior of the methods. Complete the definition by writing code to replace each comment:

```
import java.util.*;

public class Warehouse
{
    // Declare instance variables here

    public Warehouse( int binMax )
    {
        myBinMax = binMax;
        myCatalog = new ArrayList<MusicMedia>();
        myBins = new ArrayList<Bin>( 5 );
        // Code that will start the warehouse
        // off with 5 empty bins
    }

    public String toString()
    {
        // See Note (1) concerning the form of the
        // string representation of the warehouse
    }
```

```
    public void addBin()
    {
        myBins.add( new Bin( "B" + myBins.size() ) );
    }

    public void receive( MusicMedia product, int quantity )
    {
        // See Note (2) concerning the required behavior
        // when the warehouse receives a quantity of a
        // certain music media product
    }
}
```

Note (1): The string representation of the warehouse should have the same form as the `String` produced by the `detailedInventory` method from Part B. In fact, you should be able to use almost exactly the same code here. Within the definition of this `toString` method, the value of the instance variable `myCatalog` plays the role of the `catalog` argument to `detailedInventory` and the value of the instance variable `myBins` plays the role of the `warehouse` argument to `detailedInventory`. Since this `toString` method is defined in the `Warehouse` class, you will need to reference the class in which the `lookupMedia` method is defined when you invoke it here.

Note (2): The `receive` method should first check (by comparing SKUs) whether or not the incoming `product` is already in `myCatalog`. If not, add the `product` to the catalog. In any case, and as long as there remains some of the incoming `product` that has not yet been placed in a bin, locate which bin contains the least total quantity. (If all the bins are full, then add a new, empty bin. Then the new bin will of course be the bin you are looking for.) Add to that bin as many of the incoming `product` as you can, taking care not to exceed `myBinMax` for the total quantity in the bin. (The most you can add, of course, is the `quantity` being received — provided there is room in the bin for such a number.) If after doing this, there still remains some `product` that has not yet been placed in a bin, look again for the bin that contains the least total quantity, place as much as possible of the remaining `product` in it. And so on, until no more incoming `product` is left.

Test your definition by using a `main` method that contains the following code:

```
Warehouse w = new Warehouse( 10 );
MusicMedia m =
    new CompactDisk( "The Early Years", "Michael Jackson", "1234-13" );
MusicMedia c =
    new CassetteTape( "Greatest Hits Volume 1", "The Beatles", "1234-5" );
MusicMedia d =
    new DigitalVideoDisk( "Love Songs", "Shania Twain", "1234-0" );
MusicMedia n = new CompactDisk( "The Best Of", "Limp Bizkit", "1234-3" );
w.receive( m, 4 );
w.receive( c, 25 );
w.receive( d, 18 );
w.receive( n, 5 );
System.out.println( w );
```

The output should be:

```
Bin B0:
CD - The Early Years (Michael Jackson), SKU 1234-13: 4
DVD - Love Songs (Shania Twain), SKU 1234-0: 6
Bin B1:
Cassette - Greatest Hits Volume 1 (The Beatles), SKU 1234-5: 10
```

```
Bin B2:
Cassette - Greatest Hits Volume 1 (The Beatles), SKU 1234-5: 10
Bin B3:
Cassette - Greatest Hits Volume 1 (The Beatles), SKU 1234-5: 5
DVD - Love Songs (Shania Twain), SKU 1234-0: 2
CD - The Best Of (Limp Bizkit), SKU 1234-3: 3
Bin B4:
DVD - Love Songs (Shania Twain), SKU 1234-0: 10
Bin B5:
CD - The Best Of (Limp Bizkit), SKU 1234-3: 2
```

4) Add a `find` method to the `Warehouse` class. This inputs a SKU and returns an `ArrayList<Bin>` of all the warehouse bins that contain a bin item with that SKU.

5) Add a `ship` method to the `Warehouse` class. This inputs a SKU and a quantity and removes items with that SKU from one or more bins so that, in total, the specified quantity is removed. It outputs a *Pick List*, that is, a list of the bins containing the SKU and the quantities that should be picked from each such bin. (To help you generate this pick list, the `ArrayList<Bin>` produced by the `find` method should prove useful.)

   For example, suppose that there are 250 of SKU 1234-1 in Bin A and 700 in Bin B. Then `ship( "1234-1", 900 )` might output this pick list:

```
Bin A: 1234-1, 250
Bin B: 1234-1, 650
```

   After the call, Bin A would no longer contain any SKU 1234-1 items, whereas Bin B would contain a quantity of just 50.

   To test your definition, you could extend the code used to test the `receive` method, adding these two lines at the end of the body of your `main` method:

```
System.out.println( w.ship( "1234-5", 12 ) );
System.out.println( w );
```

   The output should be as follows (the first two lines form the pick list):

```
Bin B1: 1234-5, 10
Bin B2: 1234-5, 2

Bin B0:
CD - The Early Years (Michael Jackson), SKU 1234-13: 4
DVD - Love Songs (Shania Twain), SKU 1234-0: 6
Bin B1:
Bin B2:
Cassette - Greatest Hits Volume 1 (The Beatles), SKU 1234-5: 8
Bin B3:
Cassette - Greatest Hits Volume 1 (The Beatles), SKU 1234-5: 5
DVD - Love Songs (Shania Twain), SKU 1234-0: 2
CD - The Best Of (Limp Bizkit), SKU 1234-3: 3
Bin B4:
DVD - Love Songs (Shania Twain), SKU 1234-0: 10
Bin B5:
CD - The Best Of (Limp Bizkit), SKU 1234-3: 2
```

This completes part C of this project.