Chapter 6 Software Reuse

Introduction
Software is a new invention. It is nearly pure information, like a story, or DNA. Fortunes have been made, and will be made writing and selling software. Creating and distributing software as a commodity is very different from growing and selling soybeans, or building and shipping refrigerators. Once it is written, software can be distributed at almost no cost over the Web; this makes enormous profits possible.

A peculiarity of software is that new releases, revisions and updates are common. The extent to which the old software can be reused determines how much work these revisions entail. If even minor changes necessitate reworking large bodies of code, then they are difficult and expensive. On the other hand, if a new version of a product can be produced without extensive rewriting, it is simpler and cheaper. Thus, the possibility of reusing software without reworking it or even looking at it would be a tremendous advantage.

Object oriented programming makes possible software reuse by inheritance and composition. These techniques were introduced in Chapter 5, and will be revisited in more detail here.

Inheritance
Inheritance gives the programmer tremendous power. Once a class is written and debugged, another class can modify or enhance it without the programmer having to worry about destroying its functionality and often without even knowing what its code looks like.

For example, every user_applet class extends java.applet.Applet. The first_applet you wrote (Listing 2.5) had just an empty block following the heading in Listing 2.5, and it worked just fine (although it didn’t do much); see Listing 6.1.

Listing 6.1 Reprise of Listing 2.5

```java
public class RobotGreeter extends java.applet.Applet {
    ...
}
```

This is because it inherited all the functionality of java.applet.Applet. Here, RobotGreeter is said to be a subclass of java.applet.Applet, and java.applet.Applet is the super class of RobotGreeter.

The Object Class
At the top of the Java class hierarchy sits the Object class. Every Java class is a descendant of Object. All of the methods and variables in Object are defined in every object of any type. There are not very many methods defined in Object (you can check the documentation if you are curious -- [http://download.oracle.com/javase/6/docs/api/](http://download.oracle.com/javase/6/docs/api/)); the only one we will address here is toString().
Every user-defined class extends some class. If there is no `extends` keyword after the class name, the compiler inserts `extends Object` automatically.

The Mechanics of Message Sending

By now you are familiar with sending messages to objects. Perhaps it has begun to seem straightforward and natural; if a class has a method defined, you can send the associated message to objects of that type. If you try to send a message to an object whose class does not have the associated method defined, then a compiler error is generated and your code will not run. But, consider the `FilledCircle` class in Listing 6.2 and the code in Listing 6.3 where `iris`, a `FilledCircle`, is being sent `setX()`.

Listing 6.2 The Complete `FilledCircle` Class, From Listing 4.11

```java
1    import java.awt.*;
2    public class FilledCircle extends Circle {
3        protected Color myColor = new Color(100,0,100);
4
5        /** Creates a new instance of FilledCircle */
6        public FilledCircle() {
7            
8        public void setColor(Color c) {
9            myColor = c;
10        }
11
12        public void paint(Graphics g) {
13            g.setColor(myColor);
14            g.fillOval(x-radius, y-radius, radius*2, radius*2);
15        }
16    }
```

Listing 6.3 From Listing 4.13

```java
1    public void moveLeft() {
2        iris.setX(iris.getX()-2);
3        pupil.setX(iris.getX());
4    }
```

There is no `setX()` method declared in the `FilledCircle` class, so why does this code compile? For that matter, do you see an `int x` variable in `FilledCircle`? How can you set what doesn’t exist? You probably already know the answer, if not, look at line 2 in Listing 6.2.

At runtime, when an object is sent a message, if its class declares the associated method, the Java Virtual Machine (VM) executes that method. If its class does not declare that method, the VM checks its super class; if the VM finds it there, it executes it, if not, the search continues up the class hierarchy. When it reaches the `Object` class; if the method is still not found there, a `NoSuchMethodException` is generated (you can read about Exceptions in Appendix B: “Exceptions”, which may be found from the readings page; the last link... Simply Java Whole text (slightly outdated).
The `FilledCircle` class extends the user-defined `Circle` class, therefore `FilledCircle` inherits the `x`, `y`, and `radius` variables plus their accessors from the `Circle` class. The compiler thus allows `setX()` to be sent to `iris` because it was defined in `iris`’s super class.

It is often useful to have a picture of the class hierarchy. It aids memory and facilitates communication. Figure 6.1 shows two common styles of class diagram for `FilledCircle`. (Booch on the left, UML on the right). Both show that `FilledCircle` extends `Circle` which extends the `Object` class.

You can send any object `toString()` and be confident it will work, why?

This is super!
Sometimes, in a constructor, you want to invoke the super class’s constructor. The most common time is in an initializing constructor with a number of parameters, some of which are for
variables in the superclass. Just as this() invokes the default constructor for this class (see the paragraph titled, this(), in Chapter 5), super() invokes the superclass constructor.

For example, if you wanted to write an initializing constructor for the four variables in FilledCircle and there was already one for x, y, and radius in Circle, you could write the constructor in Listing 6.4. Line 2 invokes Circle’s initializing constructor to set x, y, and radius. Line 3 sets the color (!).

Listing 6.4 Initializing Constructor for FilledCircle Using super()

```java
1    public FilledCircle(int x, int y, int r, Color c) {
2        super(x,y,r);
3        setColor(c);
4    }
```

The superclass initializes its three variables, then the color variable is set locally. Invoking super() must be the first thing in the constructor.

**Composition**

Having one thing composed of several others is common and familiar in everyday life. A student’s daily schedule is composed of classes, meetings and meals. A face is composed of various features. Generally, we experience objects in the world as sets of features that travel around together. Containment is also familiar in the world. Dressers contain clothes of various kinds. Toolboxes are used to hold various tools. A backpack may contain books, pens, a water bottle and a computer.

Composition in object programming shares features from both composite objects and containers in the world. It refers to a class being made up of several objects of other classes. So a composite object may be thought of as a container for other objects or it may be thought of as being the combination of those objects -- either way can be useful in different contexts.

The Eye class was composed of an iris and a pupil, both FilledCircles. Figure 6.2 shows the class diagram for Eye, again in two different styles. Both show that an Eye extends Object and is composed of two FilledCircles which extend Circle which extends Object.
In the Booch diagram, each cloud represents a class. There are two kinds of arcs; arrows, representing inheritance, and lines with a dot and an h representing composition. The class at the dotted end has or contains instances of the other. The number at the other end indicates how many instances; by default it is one. In the UML diagram, inheritance is also represented by an arrow, but composition is represented by a horizontal arrow with a “has” label.

To write a Face class, you might include two Eyes, a Nose, and a Mouth (once you wrote Nose and Mouth). To display it you would display each component in a particular spatial relationship to each other. You might write a Person class that included a Face, as well as various other body parts. Then you could create a Scene involving several Persons.

Composition allows you to assemble multiple objects of various types into a container. This is efficient and useful, as it allows you to group things and treat the ensemble as a single thing, thus simplifying your thinking.
Composition Programming Example: Snowpeople

A Description of the Task
Here is the programming task for this chapter. Write a Java Applet that will display several snowpeople of various sizes at various locations around the screen. To simulate warm weather, make the snowfolk melt some each time the user pushes a button labeled “warm sunny day”. When a snowperson melts, increase the size of a gray puddle of melted snow under it.

Overall Design
Perhaps, given your experience with the Eye class, you feel that you know enough to immediately start writing code. It would be fine to write a prototype SnowPerson class immediately, that’s a reasonable approach. But, before writing very much of the detailed code, it is important to think through the entire problem. What classes will you need? How will they be related? Which will do what? Carefully designing code before writing any can save many hours of frustrating debugging and redesign.

As always, the first questions in GUI object design are, “What will the GUI look like?”, and “What classes will be needed?”. There is never one right answer to either question, and after working on the implementation some you may realize the class structure or the GUI you’ve chosen needs revision. Making considered choices could save a tremendous amount of time. Aim first, then shoot.

The GUI
The interface for this problem is very simple. There is a single button that the user can push to simulate one day of melting. The only output is to display the snowpeople. It might be nice to draw some background for the snowpeople as well.

Classes
In addition to the Applet class (which, as usual, will handle the GUI), we will definitely need a SnowPerson class. We might also need a SnowBall and Puddle class, each of which would reuse FilledCircle. Or perhaps the SnowPerson could just contain four FilledCircles, named head, middle, base, and puddle.

How Many Classes Should You Have?
The right number of classes for a particular problem is somewhat ambiguous. It is perhaps a matter of taste, and as the expression goes, “there’s no accounting for taste”! For a large problem, having only one class would be too few; for a small problem, having more than a handful would likely be too many. Let’s keep open the choice of whether to have Puddle and SnowBall classes for the present, until after considering some of the details.

SnowPerson Design
The description of the task left a number of details unspecified, including: How many snowballs is a SnowPerson composed of? What are their relative sizes? What color are they? How much
does a `SnowPerson` melt in one day? When it melts, how quickly does the puddle grow? This lack of specificity forces the programmer to either make these decisions arbitrarily, or request additional information. If someone had hired you to produce a `SnowPerson` Applet, you would ask them if they preferred to specify those things, or if you should make your own decisions. Imagine how upset everyone would be if you made the decisions yourself, wrote all the code, delivered it, and your customer had totally different expectations. Here, these decisions have been made arbitrarily. Feel free to implement them differently.

**How Many? How Big?**

Let’s assume that a `SnowPerson` is composed of three white snowballs getting smaller as they go up (as usual). Call them the base, the middle and the head. The exact ratio of sizes is not important; let’s say the radius of the middle is 2/3 the radius of the bottom and the head radius, 2/3 the middle. If you don’t like how this looks you can adjust it later.

**Where Do the Three Snowballs Go?**

Perhaps the most difficult decision is where to locate each of the snowballs. Recall that `FilledCircles` keep track of the position of their centers. Assume the Applet will specify where each `SnowPerson` goes using x and y-coordinates. Should that location be used for the top of the head? Or the bottom of the base? Remember that the snowfolk are going to melt. If the location of a `SnowPerson` were the top of the head, as it melted, the base would rise up into the air! That could be amusing, but is hardly how real snowpeople behave. Thus, the location of a `SnowPerson` will be where the base touches the ground.

Given that decision, the positions of all three snowballs are fixed. The base is centered its radius above its location. The middle is centered above that by the radius of the base plus its own radius (see Figure 6.4). The head’s position is similar. The details of computing these positions are properly part of implementation.

**Displaying the SnowPerson**

To display a composite object, simply display each component (remember, the order of display can be important). Assuming the three snowballs and the puddle each store their color and position, this should be trivial.

**Melting**

This is another totally arbitrary decision. Let’s say, for simplicity, that the radius of the base decreases by 10% each day and the ratios of the radii remains 2/3. For now, let’s say the size of the puddle increases by 10 pixels each day.

**Implementation**

Once you understand the problem, how you will address it and what classes you will write, it is time to start on implementation. Of course, depending on your level of expertise and the difficulty of the problem, the design phase may be shorter or longer (beginner’s tend to jump into implementation too soon).
** Keeping Things Simple**
Perhaps the most important skill a programmer learns is to keep every method simple. It is possible to make a program work with huge, sprawling methods, just as it is possible to build a vehicle out of spare parts, tape and baling wire -- but, it is almost never a good idea, especially if you have far to go.

**Strike Out on Your Own?**
It won’t be long before you start programming independently; coming up with your own tasks to program, or, at least writing programs to complete programming assignments without being provided with the answers. The sooner you can start implementing yourself the better. If you are ready to jump into implementation by yourself already, do it! Try it out and then come back here for hints. On the other hand, if it is still feeling a bit new and strange and you’re not sure how to proceed, follow along here; but, try to do the things the text recommends doing before looking at the answers. Dependence is good, but not for too long.

**SnowBall? Puddle?**
The decision about whether to create SnowBall and Puddle classes, or just go with four FilledCircles must now be faced. If you decide to write SnowBall and Puddle, should they extend FilledCircle, or contain a FilledCircle (those are the two techniques to reuse software)? The answers to both those questions hinge on two things: what actions those classes must implement in addition to FilledCircle, how complicated those actions are, and how many modifications you anticipate making in the future. If this code will never be used again, and there is only one additional method, and it is simple, then the answer is use FilledCircle and get it done with. On the other hand, if there are many complex methods needed and extensive modifications may be required, then the answer is write subclasses.

The actions of SnowBall and Puddle are shrink, grow, and paint(). If they extend FilledCircle they will automatically inherit paint() from FilledCircle, as well as the accessors for y and radius (which are needed for computing their new sizes and positions during melting). If they wrap up a FilledCircle (i.e. if composition is used) they would have to implement those accessors all over again. So, that choice is simple, inheritance is more appropriate here.

Although it would be possible to implement a SnowPerson using four FilledCircles, composing it of three SnowBalls and a Puddle will be more elegant and allow inheritance to be illustrated.

Given that decision, the class structure for this program is illustrated in Figure 6.3.
Notice that a SnowPerson extends Object and has three SnowBalls and one Puddle. The classes SnowBall and Puddle extend FilledCircle which extends Circle.

Implementation Plan
Here’s a series of tasks that will lead to a working program. The rest of this section will detail how to implement them.

1. Create a new project in a new directory
2. Create a GUI Applet with a Button (for melting); hook up the Button
3. Create (or better, copy!) the Circle, and FilledCircle classes.
4. Create SnowBall, Puddle, and SnowPerson classes.
5. Add the chain of `paint()` methods so that repainting the Applet will repaint all the `FilledCircles` of all the `SnowPersons`.

6. Add the code to calculate the locations of the `SnowBalls`.

7. Add additional `SnowPersons`.

8. Add the melting code.

Naturally, after every step, test your code if you can.

You already know how to do the first three steps; come back here after you’ve done that.

Creating the Classes
There are five classes in this program: `Circle`, `FilledCircle`, `SnowBall`, `Puddle`, and `SnowPerson`.

Circle and `FilledCircle`
Use the Classmaker for `Circle`, that way you will get an initializing constructor. The `FilledCircle` class from your `Eye` project does almost everything it should, except it does not have an initializing constructor; add the one in Listing 6.4.

SnowBall
The `SnowBall` class extends `FilledCircle`. Since all `SnowBalls` are the same color, the constructor does not need a color parameter. The only method the `SnowBall` class needs is `melt()`, which reduces the size by 10%; so the body of that method would be one line:

```
setRadius(getRadius() * 9/10);
```

It also needs an initializing constructor, with a single line:

```
super(x, y, r, java.awt.Color.WHITE);
```

Puddle
Like `SnowBall`, `Puddle` extends `FilledCircle`. It needs the same initializing constructor (but that sets the color to `java.awt.Color.GRAY`) and a `grow()` method to increase its size by 10 pixels (`setRadius(getRadius() + 10);`).

SnowPerson
The `SnowPerson` class has three `SnowBall` variables, called `base`, `middle` and `head`. It will need an initializing constructor that is passed the location and size. That constructor will then calculate where and how big the three `SnowBalls` should be. For a first prototype let all three be the same size, as in Listing 6.5.

Listing 6.5 Prototype Initializing Constructor for `SnowPerson`
```
1    public SnowPerson(int x, int y, int size) {
2        base = new SnowBall(x, y, size);
3        middle = new SnowBall(x, y-size, size);
4        head = new SnowBall(x, y-size*2, size);
5    }
```
**Line 2**: Creates the base SnowBall.
**Line 3**: Makes the middle SnowBall the same size, but higher.
**Line 4**: Makes the head SnowBall the same size, and higher still.

This code centers the base at \((x, y)\) instead of putting the bottom of the base at \((x, y)\). But, that’s good enough for now, you can adjust the locations and sizes later (after the SnowPerson shows up on the screen). First build the structure of the program, then refine it.

**Adding the paint() Chain**

You will recall from Chapter 4, that to update the way an Applet looks we send it repaint(), which causes paint(Graphics) to be sent to it. When the Applet gets the paint() message, it should paint() all the things it displays. Add a public void paint(Graphics) method to your Applet that simply sends paint(g) to the SnowPerson (see Listing 4.6). Before that will compile, you must have a SnowPerson to send the message to; so, declare and initialize a SnowPerson instance variable in your Applet. That last instruction is completely explicit. If you don’t know how to accomplish it you might look back at the paragraph titled, “Variables I (state)” in Chapter 5. A big part of introductory programming expertise is familiarity with terms and concepts; the sooner you become familiar with them, the sooner programming will be easy. Perhaps you would consider rereading the section titled, “Recapitulation” in Chapter 5?

Test that code (note that this is step 6 of the section titled, “Implementation plan”, from earlier in this chapter), then on to making it look like a snow person. If there are any problems, and the solutions don’t jump right out at you, you might check “What could go wrong?” at the end of this chapter.

**Letting the Computer Do the Arithmetic**

The SnowPerson constructor, when passed its size and position, must calculate how big the middle and head are and where the three SnowBalls will go. Calculating the sizes is very easy, see Listing 6.6.

**Listing 6.6 Setting the Sizes of the middle and head from the base**

```java
1    private void adjustSnowBallSizes() {
2        middle.setRadius(base.getRadius()*2/3);
3        head.setRadius(middle.getRadius()*2/3);
4    }
```

**Line 2**: Sets the radius of middle to \(2/3\) the radius of the base.
**Line 3**: Sets the radius of head to \(2/3\) the radius of the middle. Calculating the locations is more complicated.

Consider, by way of example, a SnowPerson of size 50, located at \((200, 300)\) (see Figure 6.4). All three SnowBalls have the same \(x\)-coordinate, but the \(y\)-coordinates must be calculated. The center of the base will be 50 pixels above the bottom (since the radius is 50) at \((200, 250)\), the top will be 50 pixels above that. The center of the middle SnowBall will be 33 pixels above that (since 33 is \(2/3\) of 50) at \((200, 167)\). Similar reasoning puts the center of the head at \((200, 112)\).
If, more abstractly, the SnowPerson is at \((x, y)\), and we call the radii of the three SnowBalls \(baseR\), \(middleR\) and \(headR\), then we could write the \(y\)-coordinate of the base as, \(y - baseR\).

The \(y\)-coordinate of the middle is \(y-baseR*2 - middleR\), or \(baseY - baseR - middleR\). The \(y\)-coordinate of the head is \(y-baseR*2-middleR*2-headR\), or \(middleY-middleR-headR\).

In Java, this looks like:

```java
base.setY(y-base.getRadius());
middle.setY(y-base.getRadius()*2-middle.getRadius());
head.setY(y-base.getRadius()*2-middle.getRadius()*2-head.getRadius());
```

Notice that the \(y\) variables in the various SnowBalls store the \(y\)-coordinates as they are computed. An alternative computation is:

```java
base.setY(y-base.getRadius());
middle.setY(base.getY()-base.getRadius()-middle.getRadius());
head.setY(middle.getY()-middle.getRadius()-head.getRadius());
```

Add whichever version you like better to your constructor, and check that it works. Note that the \(y\)-coordinate should be stored in the SnowPerson (since when the base melts its center must be moved down to its radius above where it touches the ground). You could either type the three
lines directly in the constructor, or put them in a method, called something like adjustSnowBallLocatons() and invoke that method from the constructor. Which is better? It depends on two things. Grouping code together in a method with a name helps to make obvious what it does. Also, if it turns out you will need to use that code somewhere else in your program, then you can reuse a method instead of copying and pasting the code. A completed constructor appears in Listing 6.7.

Listing 6.7 Initializing Constructor for SnowPerson

```java
public SnowPerson(int x, int y, int size) {
    this.y = y;
    base = new SnowBall(x, y-size, size);
    middle = new SnowBall(x, y-size, size);
    head = new SnowBall(x, y-size*2, size);
    thePuddle = new Puddle(x,y,0);
    adjustSnowBallSizes();
    adjustSnowBallLocations();
}
```

Note that the computation of SnowBall sizes and locations is done in methods; these will be reused in melt().

An alternative design

Isn't the preceding rather too complicated? Might this mean we should think about what we are doing? Although Circle and FilledCircle are based on (x,y) as the center, everything we do with a Snowball has to do with the bottom of the FilledCircle. So perhaps Snowball should store (x,y) of the bottom of the snowball instead of the middle? Like this:

```java
public class SnowBall extends FilledCircle {
    public SnowBall(int x, int y, int r) {   //initializing constructor
        super(x, y, r, Color.yellow);
    }

    void paint(Graphics g) {
        g.setColor(color);
        g.fillOval(x - r, y - 2 * r, 2 * r, r + r); // <== y-2*r!!
    }

    int getTop() {
        return y - 2 * r;
    }
    ...}
```

Snowball overrides paint, so that it draws itself with the bottom of the snowball at (x,y).

Plus, the next snowball up needs to know where the top of this one is, so there is a getTop() method.

Now SnowPerson is a bit simpler (which is good!):
public class SnowPerson {
    
    SnowBall head, middle, base;
    FilledCircle puddle;

    public SnowPerson(int x, int y, int r) {
        base = new SnowBall(x, y, r);
        middle = new SnowBall(x, base.getTop(), r * 7/9);
        head = new SnowBall(x, middle.getTop(), r * 5/9);
        puddle = new FilledCircle(x, y, 0, Color.gray);
    }

    Which way is better? You decide!

    **Adding Additional SnowPersons**
    
    Add at least two more SnowPersons to your scene. Make them different sizes. Test to make sure they display correctly.

    If you know how to that, do it. If not, read your Applet code (so it is in your mind). Look carefully at each line that you have written. Think about what each does. Now decide how to add another SnowPerson.

    Stop. Don’t read on until you have puzzled over what to do. Okay... There are only two changes needed. First, declare and instantiate another SnowPerson. Second, modify `paint()` so that it also draws the new one. That’s it.

    **Making the SnowPerson Melt**
    
    When the user pushes the melt button it must send the `melt()` message to however many SnowPersons there are and then send `repaint()` (so you can see the changes). When a SnowPerson gets a `melt()` message, it must both decrease the size of its SnowBalls (and lower them) and increase the size of its Puddle. You can make these changes either starting with the Applet and working down, or starting with SnowPerson and working up. The text will take the latter approach.

    There are a number of details that must be attended to in order to accomplish melting, but the `melt()` method can be written without paying any attention to them as can be seen in Listing 6.8.

    **Listing 6.8 melt() Method for SnowPerson**
    
    1    public void melt() {
    2        base.melt();
    3        adjustSnowBallSizes();
    4        adjustSnowBallLocations();
    5        thePuddle.grow();
    6    }

    There are four actions that must take place: reduce the size of the base, calculate the new sizes of the middle and head, calculate the new locations of their centers, and finally growing the Puddle. The four lines of this method do that, but the details are in the methods.
Because the code to calculate the sizes of the middle and head, and the code to calculate the locations of all three SnowBalls were written as methods, this code is simple and easy to write.

Now add the code to send the melt() message to all the SnowPersons in the ActionPerformed() method of the melt Button and test the melt() method.

**Displaying the Puddle**
If you are simply following these instructions, then the Puddle is not being displayed yet. Add the code to do that. All you need is to add puddle.paint(g) in paint() in SnowPerson. Does it matter where in that method it goes? Test the completed (!) code.

Congratulations! You have just completed a programming assignment more complex and sophisticated than any introductory programming text could have imagined presenting in a procedural language. You have used composition and inheritance to leverage already written classes. You have implemented a paint() chain with a GUI interface and started down the road to understanding object programming. Not everyone makes it this far.

**Conclusion**
In Java, all classes are organized into a hierarchy, called a tree, with the class Object at the root. Every class has Object as an ancestor, i.e. Object is a superclass (although possible at several removes) to every other class.

Inheritance and composition allow software to be reused. This is a tremendous advantage over languages where software cannot be reused without extensive reworking. Given a working FilledCircle class, writing SnowBall and Puddle only took a few lines of code. Once you have working SnowBall and Puddle classes, the SnowPerson class becomes simple.

In object languages, algorithmic complexity can be reduced by building an appropriate class hierarchy. The more complex the code, the harder it is to understand. Complex code is difficult to write correctly and more difficult to debug when there are errors. Simple code is easy to write correctly and easier to debug when there are errors. And there are always errors. Only neophyte programmers imagine their code will not have bugs. Experienced programmers know better. Good programmers learn techniques to make it easier to find the bugs that inevitably creep in. They develop good habits; these habits allow them to succeed, even in difficult situations. Well thought out, coherent class hierarchy and incremental implementation are techniques that allow programmers to succeed.

**What Could Go Wrong?**

**Problem 1: The SnowPerson doesn’t appear on the screen.**
**Possible Causes:** 1) paint() is never sent, 2) White FilledCircles on a white background are invisible,
**Possible Solutions:** 1) Send the paint message, 2) Change the color of the SnowBalls or the background
Problem 2: The SnowPerson never changes size
Possible Causes: 1) The melt Button actionPerformed() method is not written, 2) It does not
tell the SnowPerson to melt, 3) repaint() was not sent to the Applet, 4) The Snowperson
melt() method is written incorrectly.
Possible Solutions: 1) Hook up the Button, 2) Send the melt() message from the body of
ActionPerformed(), 3) Send repaint() after melt().

Problem 3: The SnowBalls overlap in the initial SnowPerson.
Possible Causes: Bad arithmetic in the SnowPerson constructor.
Possible Solutions: Fix the arithmetic.

Problem 4: The SnowPerson is upside-down!
Possible Causes: The programmer forgot that 0 in the y direction is the top of the screen. The
author made this mistake.
Possible Solutions: Redo the code for calculating where the middle and head go, remembering
which way is up!

Problem 5: After melting, the middle and/or head are floating.
Possible Causes: 1) The programmer forgot to add the code to adjust the position of the middle
and/or head after melting, 2) the programmer remembered to adjust the position, but forgot to
send the message, 3) the programmer wrote the code to adjust the position after melting
incorrectly.
Possible Solutions: Add that code, invoke it.

Problem 6: The original SnowPerson appears on the screen, but the second doesn’t.
Possible Causes: 1) It’s on top of the first SnowPerson. 2) paint() does not send it paint()
Possible Solutions: 1) Give it different coordinates. 2) If you called it person2, add
person2.paint(g) to paint() in the Applet.

Problem 7: The original SnowPerson melts, but the second doesn’t.
Possible Causes: The melt actionPerformed() method doesn’t send melt() to the second one.
Possible Solutions: Add the melt() message.

Review Questions
6.1 What are the two techniques to reuse classes?
6.2 Write a SnowBall class that uses each technique to reuse FilledCircle. Which seems
better to you? Why?
6.3 Does SnowBall use composition or inheritance?
6.4 How do you know if you need more classes?
6.5 How do you know if you have too many classes?
6.6 Write down, one per line, every method invoked (including the values of the parameters)
when new SnowPerson(50, 200, 300) is executed.
6.7 Write the melt() method for a SnowBall. Make it reduce the size of the SnowBall by
17%. Do you have to worry about which order the operators are applied?
6.8 Describe in detail what happens when the SnowPerson sends melt() to the base. What object is this? What methods are invoked in what classes in what order?

6.9 Describe in detail what happens when a user pushes the Melt for a day Button. Include every method invoked and what class it resides in, in the correct order. Pretty scary, eh?

Programming Exercises

6.10 Send toString() to your Applet and System.out.println() what it returns. What does it print?

6.11 Put this line in initComponents() in your Applet: System.out.println(this); How do you explain why it works?

6.12 Send toString() to an object that you know does not have it defined. What does that print?

6.13 Modify your Applet to display 50 SnowPersons of random sizes at random locations. Look back at the previous chapter for how to do this.

6.14 Modify your code so that all of the SnowPersons move a few pixels toward the center of the screen each time you push the button (in addition to melting). Hint: you can calculate the change you should make to the x-coordinate to move right or left (depending on if it is left or right of center) arithmetically. There is a method, Math.abs(), that will return the absolute value of an int.

6.15 You may notice that the Puddle of one SnowPerson covers up others. Modify the display methods so that the Puddles are all in the background. Hint: draw all the Puddles first -- i.e. add a paintPuddle() method and in the Applet first paint all the Puddles, then the SnowBalls.