A Guide to Programming Asteroids as a Java Applet

By Brandon Carroll

Table of Contents

List of Figures, Tables, and Code Sections

Chapter 1: Introduction

Purpose of the manual

This manual was written to provide beginning programmers with an introduction to game programming that is otherwise not readily available. Lower level programming courses typically do not deal with game programming, although that is often what the students would like to do. Furthermore, books and other resources on game programming are usually too in-depth for beginners to use easily. This manual will help you develop an understanding of the basic concepts involved in programming games so that you can then better use more thorough sources.

Overview

This manual covers the complete, chronological process of developing a basic Asteroids game as a Java Applet. Asteroids is a simple, two dimensional game where the player controls a triangular spaceship in an asteroid field. The object is to destroy all the asteroids without crashing into any of them. Asteroids was chosen because it involves relatively simple programming and because it is one of the greatest classics of the gaming industry (Wikipedia contributors 2007).

Who this manual is for

This manual is designed for beginning Java programmers who are interested in learning to make simple computer games. You will need to know the very basics of Java, but not much more than that. If you ever get stuck, there are several helpful resources listed under the "Where to go for help" section in this chapter. This manual assumes that you already know at least the following:

1. How to create and use a class with **methods***.
2. How to **compile** java programs.
3. How to use variables (**primitive types**, **Strings**, and **arrays**).
4. How to use basic **control flow statements** (if, while, for, etc.).

*Note: Bolded terms can be found in the glossary at the end of the manual.

Some basic trigonometry will also be involved. An understanding of the following concepts will be helpful, though not necessarily required (you can just copy the code given):

1. Rotating a point around the origin (used to turn the ship).
2. Finding x and y components of a vector from its magnitude and direction (used for movement).
3. Calculating distances with the distance formula (used to detect collisions).

Needed materials

To be able to use this manual, you will need to have the **Java Software Development Kit** (JDK) installed on your computer so that you can compile Java programs. I used the current version, Java Standard Edition 6, to write this manual. Either a **text editor** designed for Java code or a Java **Integrated Development Environment** (IDE) will also be extremely helpful. It is possible to write code in a basic text editor like Notepad, but much more difficult because there is no **syntax highlighting**. There are several free text editors and IDEs made for Java that are available to download online. Refer to the "Where to go for help" section for some of their download sites.

How to use this manual

Because of the way this manual has been designed, it is best to go through it from beginning to end, writing the code as you go. Each section builds on those sections that have gone before it. Chapter 2 explains the overall process of creating the game and how the classes interact. Chapter 3 covers setting up the basic framework for a game. Then, the following chapters build on that framework to make the Asteroids game. If you already understand the concepts covered in Chapter 3, you may use the completed framework code given in Appendix E and skip to Chapter 4. If you do this, be sure to change the name of the class given in Appendix E from GameFramework to AsteroidsGame.

It will likely take several hours to get through the entire manual. If you do it in multiple sessions, I recommend stopping between chapters because they represent the major sections of the game. It may be more difficult to pick back up where you were if you leave off in the middle of a chapter.

The following conventions are used throughout this manual:

Bolded Terms

Bolded words are terms that are defined in the glossary at the end of the manual.

Code sections

The names of the files that code sections describe are given in bolded text following the code section number. Code sections only contain the code pertinent to that section of the manual, assuming that all other code that has been written up to that point (as described by all the previous sections) remains the same. This may make it difficult to successfully program Asteroids if you skip sections or if your code written for a previous section does not work. If you are confused about where the code fits into the whole, the completed code for the entire game is provided in appendices A through D. For Chapter 3, the completed framework code is located in Appendix E.

The syntax highlighting in the code sections is the default syntax highlighting used by the Eclipse IDE. The different colors and styles represent the following:

`keywords` – Used for all Java keywords (i.e. int, void, public, for, if, etc.).

`variableNames` – Used for all variable or object names (identifiers).

`comments` – Used for **comments** in the code. Comments include any text following the "`//`" symbol on the same line, or between the "`/*`" and "`*/`" symbols for multi-line comments.

`other` – Used for everything else, including numbers, symbols, class names, and method names.

In-text references to variable, method, and class names

In the text of this manual, references to variable, method, and class names will follow the following conventions:

variableName – Variable names will be italicized. According to common Java practice, the first letter of variable names will always be lower-case and the first letters of subsequent words (if any) within the variable name will be capitalized.

methodName() – Method names will have an empty set of parentheses attached at the end, though parameters will not be included. They follow the same capitalization convention as variable names.

ClassName – Class names will always begin with a capital letter according to the standard Java convention.

Where to go for help

This section contains references to websites that may be helpful in resolving a variety of problems, questions, or concerns you may have as you use this manual. If the code you have written for any of Chapters 3 through 6 does not work, I recommend first verifying that it exactly matches the code given, then turning to online sources. I have verified that all the code in this manual works by copying and pasting it directly into source code files, then testing them.

Site for downloading and installing Java
The JDK download page – http://java.sun.com/javase/downloads/index.jsp

Sites to download free editors and IDEs
Eclipse IDE – http://www.eclipse.org/
NetBeans IDE – http://www.netbeans.info/downloads/index.php
gVim text editor – http://www.vim.org/download.php

Site for learning Java basics
The Java Tutorials – http://java.sun.com/docs/books/tutorial/index.html
The New to Java Programming Center – http://java.sun.com/developer/onlineTraining/new2java/

Site with descriptions of all classes and methods in the Java standard library
The Java 6 API – http://java.sun.com/javase/6/docs/api/index.html

*Note: The Java API site is an invaluable resource for any Java programmer. You should become familiar with it if you are not already.

About the author

Brandon Carroll is a student at Brigham Young University studying Computer Science and Electrical Engineering. He started programming early in high school and has been writing

many of his own small games since then. He works primarily in Java, but also uses C++ and Visual C#.

Chapter 2: Planning the Game

What will the game be like?

Asteroids is a simple, two-dimensional game with the objective of destroying all the asteroids on the screen. The player controls a small, triangular ship with keyboard controls for rotating left, rotating right, accelerating, and shooting. When the player shoots an asteroid, it breaks up into smaller asteroids. This continues until a limit is reached where the asteroids will be totally destroyed instead of breaking up. The playing field is rectangular and wraps around, meaning that if an object goes off one side of the screen, it will reappear at the same location on the opposite side.

How long will it take to make the game?

I would expect it to take several, if not many, hours to program Asteroids using this manual. If you do not want to do it all at once, it is best to break it up between chapters in the manual. Your current level of expertise could greatly affect the amount of time needed to complete the program.

What classes are needed?

For this project, we will need a **class** that runs the game as a whole as well as classes that take care of each individual part within the game. I have divided the game into the following four classes. Each will be in its own Java source code file (.java).

1. AsteroidsGame – The main class that represents the game as a whole.
2. Ship – The class that represents the player's ship.
3. Shot – The class that represents the shots fired by the player.
4. Asteroid – The class that represents the asteroids.

How will the classes interact?

The relationships between the classes are displayed graphically in Figure 2.1. As indicated by the solid arrows, a web browser will contain the AsteroidsGame applet. In turn, AsteroidsGame will contain an **instance** of the Ship class and arrays of the Shot and Asteroid classes. The dotted lines represent interactions between classes.

The AsteroidsGame class will contain the code that runs the game as a whole. It will be the first class we create. AsteroidsGame will create and manage all the instances of the other classes, take care of **event handling** (responding to user input), and connect the game into internet browsers as an **applet**.

The Ship class will draw and move the player's ship. Because the setting of the game is in space, the ship's movement will include drifting. Ship will also create new instances of Shot when the player presses the fire key.

The Shot class will draw and move the shots fired by the player. These shots will disappear if they go too long without hitting an asteroid.

The Asteroid class will draw and move the asteroids. Additionally, it will contain methods to detect collisions with the player's ship and the player's shots. When the ship and an asteroid collide, the player will have to start the level over again. When the asteroid is hit by a shot, it will either split itself up into smaller asteroids or disappear completely.

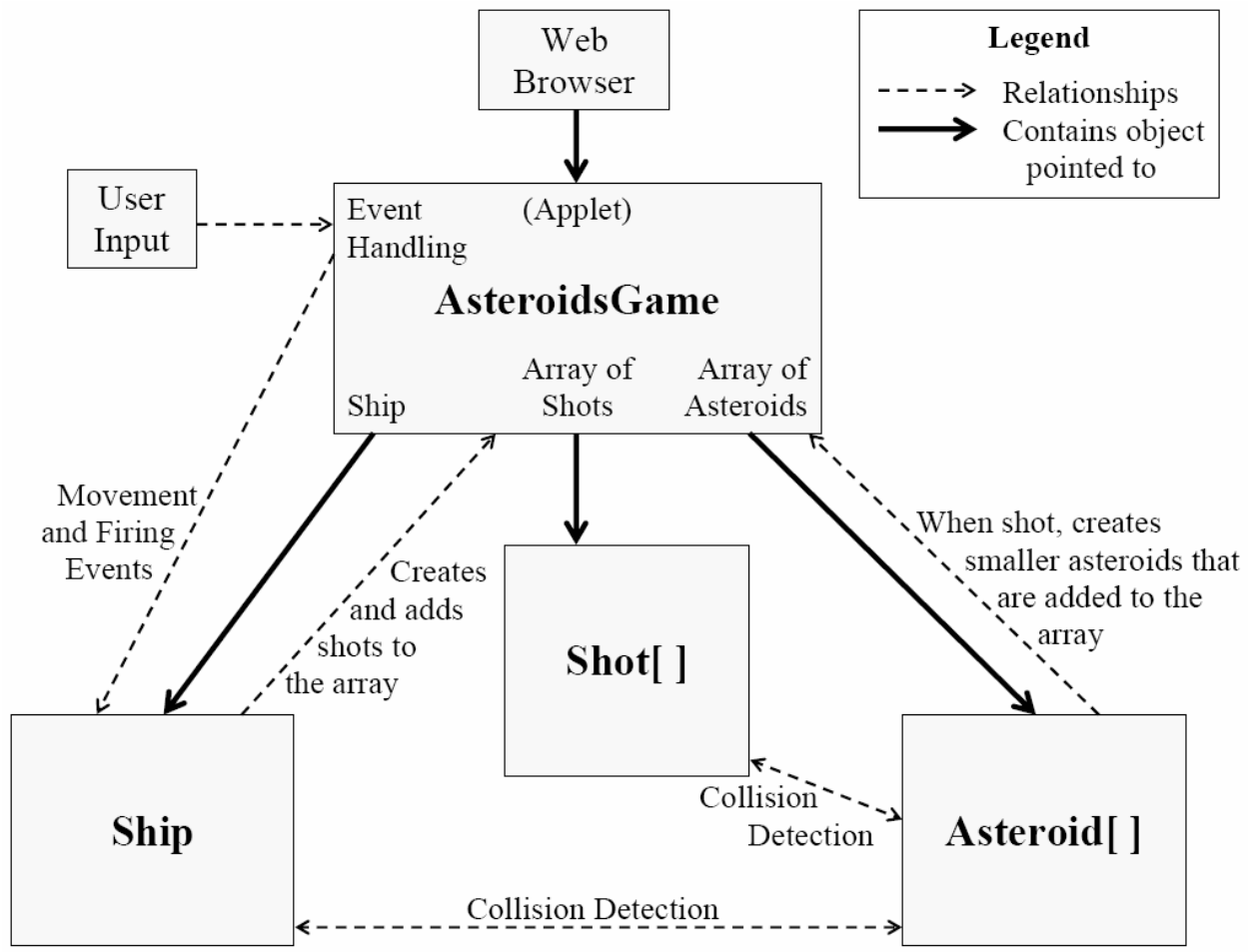
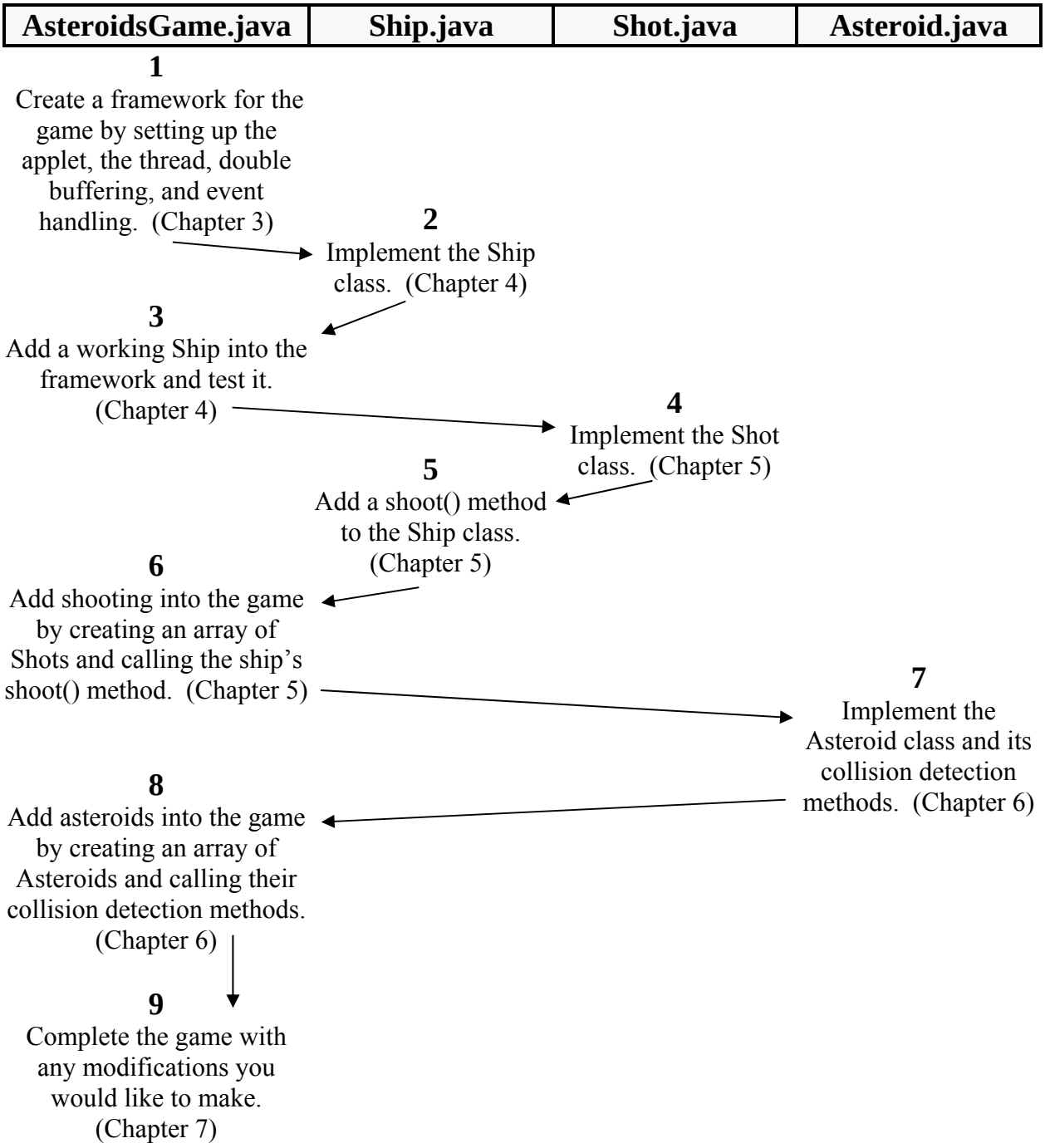


Figure 2.1: Boxes with large, bolded text in the center represent classes. The AsteroidsGame class will inherit from the Applet class so that it can be embedded in and run by a web browser. AsteroidsGame will contain an instance of the Ship class and arrays of the Shot and Asteroid classes to represent the different objects in the game. A thread inside AsteroidsGame will continually move the objects on the screen and check for collisions between either the shots and the asteroids or the asteroids and the ship. User input will be handled by the AsteroidsGame class, which will tell the ship when to move and when to fire. When firing, the ship will create an instance of the Shot class and return it to AsteroidsGame to be added to the array of shots. When an asteroid is hit by a shot, it will break itself up by creating smaller asteroids and returning them to AsteroidsGame to be added to the array. The original, larger asteroid will then be deleted.

What is the overall process?

The order in which each part of the game will be implemented is described in Flow Chart 2.2 below. Each step is numbered in chronological order and describes the modifications that will be made in the file named at the top of the column in which the step is located. The chapter containing each step is listed in parentheses.

Flow Chart 2.2: Creating the Asteroids game



Chapter 3: Building a Framework

Overview

In this chapter, we will create a general framework with some functionality that is needed for almost any game. Chapters 4 through 7 explain how to build an Asteroids game within this framework. We will start by creating an applet and embedding it in a webpage. Next, we will create a thread that will continuously run the game. This is needed because applets do not actively run code. Then, we will implement double buffering so that we can display graphics smoothly. Finally, we will add event handling so that the program can respond to user input.

Setting up the skeleton of the applet

Importing packages and creating the AsteroidsGame class

First of all, create a file named "AsteroidsGame.java" in the text editor or IDE that you use for Java programming. Once you have done this, we need to import some **packages** from the Java **standard library**. We need the java.applet package so we can make the AsteroidsGame class an applet. The java.awt package will give us the classes needed to draw graphics. The Java.awt.event package provides the resources needed to implement **event handling** (responding to user input). After writing the import lines, create an AsteroidsGame class that **extends**, or inherits from, the Applet class (see Code 3.1).

> *Note: Extensive information on all the standard Java packages and classes can be found in the **Java Application Programming Interface (API)** at http://java.sun.com/javase/6/docs/api/. This website is extremely useful for Java programmers.

Code 3.1 – AsteroidsGame.java: Needed imports and AsteroidsGame class extending Applet.

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class AsteroidsGame extends Applet {
}
```

Overriding methods inherited from Applet

There are several methods in the Applet class that web browsers make **calls** to at certain points in time. However, because the Applet class was created to define the functionality of an applet rather than to be used as an applet, several of these methods don't do anything. They are included in the Applet class to ensure that every class that extends Applet will have those methods so that errors are not produced when a web browser tries to call them.

To make our AsteroidsGame applet work the way we want it to, we must **override** some of these methods. Descriptions of the ones we will override are as follows:

public void init() – This method is called when the applet is first created.
public void paint(Graphics gfx) – This method paints the graphics of the applet.
public void update(Graphics gfx) – This method must be overridden to prevent flickering.

We will override init() so that we can use it to get our game started. It will set everything up when the game first loads and then get the thread running. Right now, we will simply make it call the resize() method to make sure the applet is the size we want. The resize() method is also inherited from Applet, but does not need to be overridden.

We will override paint() to make it draw the game's graphics. For now, we will just draw a black circle so that we can make sure the applet is working (see Code 3.2).

We will override update() to prevent flickering. The update() method inherited from Applet first clears the screen, then makes a call to paint(). Clearing the screen is visible to the user and causes flickering. We will override it so that it makes a call to paint() without clearing the screen.

The paint() and update() methods are not meant to be called directly. Rather, we must notify the system when the applet needs to be redrawn by calling the repaint() method (also inherited from Applet). Repaint() buffers calls to update() so that proper timing can be used and so that redundant calls can be eliminated (repaint() can also be called by things other than our program, such as the web browser).

Copy the code from Code 3.2 into your class so that we can test the applet in the next section. The Graphics class and its methods (setColor(), fillOval(), etc) will be covered later in the "Using the Graphics class" section.

Code 3.2 – AsteroidsGame.java: Overridden Applet methods (inside the AsteroidsGame class).

```
public void init(){
	resize(500,500); // make sure the applet is the right size
}

public void paint(Graphics gfx){
	gfx.setColor(Color.red);
	gfx.fillOval(0,0,50,50); // draw a red circle
}

public void update(Graphics gfx){
	paint(gfx); // call paint without clearing the screen
}
```

Inserting the applet into an HTML webpage

If you have not yet compiled AsteroidsGame.java, compile it now so that the AsteroidsGame.class file will be available for testing. Then, open a text editor (like Notepad) and type in the text exactly as show in Code 3.3.

Code 3.3 – Asteroids.html: Type this into a text editor and save it as "Asteroids.html" in the same folder as your AsteroidsGame.class file.

```
<center>
<h1>Asteroids</h1>
<br>
<applet code=”AsteroidsGame.class” height=500 width=500></applet>
</center>
```

The code in Code 3.3 is written in **Hypertext Markup Language** (HTML). In HTML, opening tags are signified by <tagName> (with the name of the tag in place of “tagName”) and closing tags are signified by </tagName>. The first tag, , tells the web browser to center everything between it and its corresponding ending tag, . The <h1> tag tells the browser to display the text between it and its ending tag using heading size 1 (the largest). The
 tag stands for “break” and just tells the browser to move to a new line. Notice that it does not have or need a closing tag.

Parameters can be passed in HTML by adding the parameter name followed by an equals sign and the parameter’s value. All of this is placed inside the opening tag between the tag name and the closing brace. In Code 3.3, the applet tag is sent parameters telling the browser where to find the file containing the compiled applet code and how much space the applet will take up (height and width in pixels).

*Note: For more information on HTML tags, visit www.html-reference.com.

Save the text file as “Asteroids.html” in the same folder that the AsteroidsGame.class file is in (make sure to save it with the extension .html and not .txt). Browse to that folder and open the new Asteroids.html file. Your web browser should open and display the text “Asteroids” centered in the top of the screen. The browser may initially block the applet from running, in which case it will pop up a notification that it was blocked. If this happens, you will need to click on the notification and tell the browser to allow the blocked content. When the applet is running properly, a red circle will appear slightly below and to the left of the “Asteroids” text.

*Note: From now on, any references to running the program mean compiling it, then opening the web page you have created (Asteroids.html). The AppletViewer program provided with the Java SDK may also be used.

Using a thread

Up to this point, everything our program does happens in response to external events. The applet does not run any of its own code by itself. Things outside the applet, like the web browser, call its init() and repaint() methods. However, the game will need continuously running code that will keep the objects moving around and check for collisions. Since applets do not provide this functionality by themselves, we will add in a thread to take care of it.

A **thread** is simply a task being performed by the computer. Computers run many threads concurrently by breaking each task up into small pieces, or time slices (see Figure 3.4). Because these time slices are so small, it looks like many things are happening at once (Brakeen 2003, chapter 1).

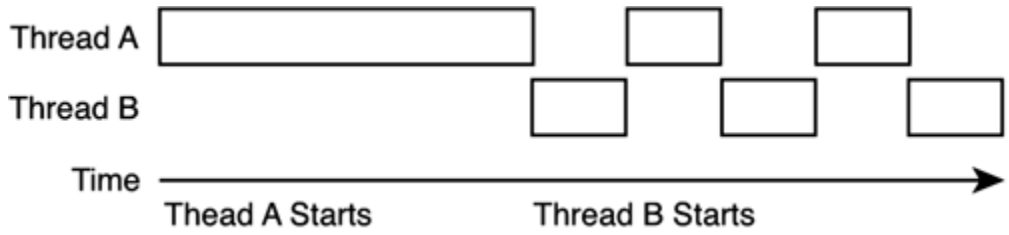


Figure 3.4: Computers appear to perform multiple tasks at once because they break each task into small segments, then rapidly cycle through each task executing a segment at a time. (Graphic source: Brakeen 2003, chapter 1)

Java provides the Thread class and the Runnable **interface** for creating threads. Since we can only inherit from one class (and we already have from Applet), we will make the AsteroidsGame class **implement** the Runnable interface. This interface only requires that we have one additional method, run(), in our class. This method will contain a loop with the code necessary to execute a **frame** of the game per **iteration**.

After implementing the Runnable interface and creating a run() method, we need to actually create and start the thread in the init() method. The code for all of this is given in Code 3.5. The variables *x*, *y*, *thread*, *currentTime*, *previousTime*, and *cycleTime* are new variables in the AsteroidsGame class. This code will create a red circle that moves across the screen repeatedly against a black background. When you run it, you may notice some black flickering in the circle as it moves. We will remedy this with double buffering in the next section.

> *Note: If you really want to see some flickering, comment out the update() method that we overrode previously. This should cause white flickering across the entire screen as the program runs. The flickering is caused by the default update() method, which paints the screen white before calling paint().

The Thread.sleep() method pauses the execution of the thread for the number of milliseconds passed in as a parameter. During this time, the computer is free to run parts of other threads that it is executing. Thread.sleep() must be placed in a try/catch block because the sleep method can throw an InterruptedException (which we will ignore if it does). Exceptions and exception handling are beyond the scope of this manual, but information on them can be found in the Java Tutorials at java.sun.com.

For our game to run smoothly, we want each frame to last for the same amount of time. We will call this amount of time the frame period (*framePeriod* in the code). Because the code in the loop can take different amounts of time to execute, we want our thread to sleep for whatever time is left in the frame period after the code has been executed. To do this, we first mark the beginning and ending times for the code execution using the *startTime* and *endTime* variables and the System.currentTimeMillis() method. This method returns a value of type **long** that represents the current system time in milliseconds. Once the starting and ending times have been obtained, the amount of time the code took to execute can be determined by subtracting *startTime* from *endTime*. This value is then subtracted from *framePeriod* to obtain the amount of time that the thread needs to sleep for (see Code 3.5).

Code 3.5 – AsteroidsGame.java: Thread test – moving a red circle across the screen.

```
public class AsteroidsGame extends Applet implements Runnable{
```

```
	int x,y; // these are all new variables
	Thread thread;
	long startTime, endTime, framePeriod; //long is just a very big integer

	public void init(){
		this.resize(500,500);
		x=0; // the top left x and y coordinates of the rectangle
		y=0; // containing the circle.
		startTime=0;
		endTime=0;
		framePeriod=25; // 25 milliseconds is a good frame period
		thread=new Thread(this); // create the thread
		thread.start(); // start the thread running
	}

	public void paint(Graphics gfx){
		gfx.setColor(Color.black); // clear the screen with black
		gfx.fillRect(0,0,500,500);
		gfx.setColor(Color.red);
		gfx.fillOval(x,y,50,50);
	}

	... // update() the same as before

	public void run(){
		for(y=0; y<450; y+=50){ // x and y are the location of the circle
			for(x=0; x<450; x+=2){
				// mark start time
				startTime=System.currentTimeMillis();
				repaint();
				// the next 6 lines pause execution for 20
				// milliseconds minus the time it took to move the
				// circle and repaint.
				try{
					// mark end time
					endTime=System.currentTimeMillis();
					// don't sleep for a negative amount of time
					if(framePeriod-(endTime-startTime)>0)
						Thread.sleep(framePeriod -
							(endTime-startTime));
				}catch(InterruptedException e){
				}
			}
		}
	}
}
```

Understanding double buffering

In the previous section, we saw that a small amount of flickering was visible in our animation even though we have overridden the update method. The flickering occurs because everything we draw in paint() is immediately displayed on the screen. We first paint the entire screen black and then paint the red circle on top of the black background. Because there is a tiny

delay between painting the screen black and painting the red circle, we see flickers in the circle on the screen.

Double buffering involves creating a virtual "screen" (a back buffer stored in memory) to draw on. Once drawing on the back buffer is complete, it is copied as a whole to the actual screen. This prevents the user from seeing the brief amounts of time between when the individual elements are drawn.

In Code 3.6, we have added three new variables to AsteroidsGame used to create the back buffer. The *dim* variable stores the dimensions of the back buffer, which need to match the dimensions of the applet. *Img* is the actual back buffer and stores the color of each pixel to be drawn to the applet's area on the screen. The variable *g* is the Graphics object that we will use to draw on *img*.

The actual double buffering occurs in the paint() method. All of the *gfx*'s have been changed to *g*'s except for in the last line. We first use *g* to draw everything on *img* (the back buffer), then we copy *img* to the actual screen using *gfx* (the last line).

After making the changes in Code 3.6 and running the program, the red circle should move across the screen without any flickering.

Code 3.6 – AsteroidsGame.java: Double Buffering test – moving a red circle across the screen without flickering.

```
Dimension dim; // stores the size of the back buffer
Image img; // the back buffer object
Graphics g; // used to draw on the back buffer

public void init(){
      this.resize(500,500);
      x=0;
      y=0;
      startTime=0;
      endTime=0;
      framePeriod=25;
      dim=getSize(); //set dim equal to the size of the applet
      img=createImage(dim.width, dim.height);//create the back buffer
      g=img.getGraphics(); //retrieve Graphics object for back buffer
      thread=new Thread(this);
      thread.start();
}

public void paint(Graphics gfx){
      g.setColor(Color.black); //Notice these first four lines all
      g.fillRect(0,0,500,500); //use g instead of gfx now.  g draws
      g.setColor(Color.red);   //to the back buffer, while gfx draws
      g.fillOval(x,y,50,50);   //directly to the screen.
      gfx.drawImage(img,0,0,this); //copies back buffer to the screen
}
```

Using the Graphics class

The Graphics class is what we use to draw all of our graphics to either the screen or the back buffer. This section explains only the methods within the Graphics class pertinent to our

program. For an exhaustive listing of methods in the Graphics class, visit the **Java API** at http://java.sun.com/javase/6/docs/api/.

The coordinate system used by Graphics has its origin at the top left pixel on the screen. The x coordinate increases by one for each pixel as you move to the right and the y coordinate increases by one for each pixel as you move down.

Table 3.7 describes the methods in the Graphics class that we will need to create our asteroids game. All of the methods are void (they do not return any value). Feel free to experiment and familiarize yourself with them by making calls to *g.*methodName() in the paint() method. Make sure you don't put them in after the last line that copies the back buffer to the screen, or you will not be able to see what they do.

Method	Description
drawImage(Image img, int x, int y, ImageObserver observer)	Draws the image contained in img with the top left corner at (x,y). We use this for double buffering. Observer is the object that will be notified whether or not the operation was successful. We simply pass the "this" pointer in as observer.
drawString(String str, int x, int y)	Draws the text contained in str to the screen with the baseline for the first character at (x, y). In other words, the text is drawn immediately above the y coordinate.
fillOval(int x, int y, int width, int height)	Draws a solid oval contained in the rectangle described by x, y, width, and height. The point (x,y) is above and to the left of the oval that is drawn.
fillPolygon(int[] xPoints, int[] yPoints, int nPoints)	Draws a solid polygon enclosed by the ordered points: (xPoints[0], yPoints[0]), (xPoints[1], yPoints[1]), … , (xPoints[nPoints-1], yPoints[nPoints-1]). nPoints is the number of points the polygon has. xPoints and yPoints should be arrays of integers that each contain at least nPoints integers.
fillRect(int x, int y, int width, int height)	Draws a solid rectangle with the top left corner at (x,y). Width and height give the respective horizontal and vertical size of the rectangle in pixels.
setColor(Color c)	Sets the current drawing color to the color described by c. All shapes drawn after calling this method will be color c until the next time setColor is called.

Table 3.7: Methods we will use from the Graphics class (source for data: The Java API 2006)

Handling events

Now that we can create basic graphics, it would be nice if the user could interact with our program. This is done through event handling. An **event** is something that happens outside of the program that the program might want to respond to. Examples of events include keys being pressed, the mouse moving, and the window losing focus. **Event handling** is the part of a program that detects and responds to these events.

For our Asteroids game, the only events we need to worry about will come from the keyboard. To detect these events, we will make AsteroidsGame implement the KeyListener interface. This interface requires the addition of the following three methods to our AsteroidsGame class:

public void keyPressed(KeyEvent e) – This method is called when a key is pushed down.
public void keyReleased(KeyEvent e) – This method is called when a key is released.
public void keyTyped(KeyEvent e) – This method is called only for keys that represent characters. It is called between the times when the two above methods are called.

Of these three methods, we only need keyPressed() and keyReleased() for our game. These two methods allow us to respond immediately to keys being pressed or released. The keyTyped() method is not called immediately when a key is pressed and would make the controls for our program less responsive. However, the interface requires that we have this method in our code even though it will be empty. KeyTyped() might be used in situations where the user needs to type text in to the program.

The KeyEvent objects passed into the above three methods contain information describing which key on the keyboard caused the event. This information can be retrieved by using KeyEvent's getKeyCode() method, which returns an integer code. This integer can be compared with the constants in the KeyEvent class to determine which key was pressed or released. These constants are in the form KeyEvent.VK_SHIFT, KeyEvent.VK_A, KeyEvent.VK_ESCAPE, KeyEvent.VK_UP, etc. The "VK" in the constant names stands for "virtual key." Refer to the Java API for a complete listing of the constants available.

After we have implemented the KeyListener interface, we need to start our event handling up in the init() method. We do this with the addKeyListener() method that AsteroidsGame inherits from Applet. This method takes one parameter of KeyListener type and causes it to listen to the keyboard events for the applet. Since our AsteroidsGame both the applet and the KeyListener we are going to use, we pass *this* in as the parameter. In Java, *this* is simply a built-in reference to whatever class it is used in. Since init() is inside AsteroidsGame, we can call addKeyListener(*this*) in the init() method to tell our AsteroidsGame class to listen for and respond to its own KeyEvents.

To test our event handling, we will make the arrow keys move the red circle around the screen. To accomplish this, we add the *xVelocity* and *yVelocity* variables which affect the coordinates of the circle each frame. When the user presses a key, we set the velocity to move the circle in the direction indicated by that key. When the key is released, we set the velocity back to zero. Code 3.8 contains the necessary changes in the code. Though infinite loops are normally bad programming practice, the infinite loop in the run() method will be terminated when the webpage containing the applet is closed.

Code 3.8 – AsteroidsGame.java: Event handling test – moving a red circle with the arrow keys.

```
public class AsteroidsGame extends Applet implements Runnable, KeyListener{
      int x, y, xVelocity, yVelocity; //velocity variables store the
                                      //direction the circle is currently moving.
      Thread thread;
      long endTime, startTime, framePeriod;
      ... //ellipses indicates that other (previous) code remains the same

      public void init(){
            resize(500,500);
            x=225; //start with the circle in the center of the screen
            y=225;
            xVelocity=0; //the circle is not moving when the applet starts
```

```
            yVelocity=0;
            addKeyListener(this);
            ... //other code the same
      }

      public void run(){
            for(;;){//this infinite loop ends when the webpage is exited
                  startTime=System.currentTimeMillis();
                  x+=xVelocity; //Moves the circle according to the x and y
                  y+=yVelocity; //velocities, which are set by event handling
                  repaint();
                  try{
                        endTime=System.currentTimeMillis();
                        if(framePeriod-(endTime-startTime)>0)
                              Thread.sleep(framePeriod-(endTime-startTime));
                  }catch(InterruptedException e){
                  }
            }
      }

      public void keyPressed(KeyEvent e){ //starts moving the circle when an
                                          //arrow key is pressed
            if(e.getKeyCode()==KeyEvent.VK_UP) //VK_UP is the up arrow key
                  yVelocity=-1; //subtracting from y moves it upward
            else if(e.getKeyCode()==KeyEvent.VK_DOWN)
                  yVelocity=1; //adding to y moves it down
            else if(e.getKeyCode()==KeyEvent.VK_LEFT)
                  xVelocity=-1; //subtracting from x moves it left
            else if(e.getKeyCode()==KeyEvent.VK_RIGHT)
                  xVelocity=1; //adding to x moves it right
      }

      public void keyReleased(KeyEvent e){ //stops moving the circle when the
                                           //arrow key is released
            if(e.getKeyCode()==KeyEvent.VK_UP ||
                        e.getKeyCode()==KeyEvent.VK_DOWN)
                  yVelocity=0; //stops vertical motion when either up or down
                               //is released
            else if(e.getKeyCode()==KeyEvent.VK_LEFT ||
                        e.getKeyCode()==KeyEvent.VK_RIGHT)
                  xVelocity=0; //stops horizontal motion when either right or
                               //left is released
      }

      public void keyTyped(KeyEvent e){ //empty method, but still needed to
      }                                 //implement the KeyListener interface
}
```

Conclusion

We have now explored all the tools we need to program Asteroids. Delete all the lines of code that involve *x*, *y*, *xVelocity*, or *yVelocity*. Your code should now look the same as the GameFramework code in Appendix E, except for the name of the class. The following chapters build on this framework to create the Asteroids game.

Chapter 4: Implementing the Ship Class

Overview

In this chapter, we add a fully functional ship into our game. The ship will be able to fly around the screen as the user presses the arrow keys and will have a red booster flame that displays only when it is accelerating. We begin by creating the Ship class, along with its variables and constructor, in a new file. Then we will implement the methods that draw and move the ship. Several other small utility methods will also be added. Finally, we will return to the AsteroidsGame class to insert the ship into the game.

Creating the variables and the constructor

Create a new file named "Ship.java" in the same folder as your "AsteroidsGame.java" file. In the file, import java.awt.* (once again needed for graphics) and create a Ship class.

To draw the ship, we will use arrays of x and y coordinates that can be passed into the Graphics.drawPolygon() method. To ensure that the shape of the ship remains constant, we will use two arrays of constant values (*origXPts* and *origYPts*) that we will translate to the ship's location. To simplify rotation calculations, these constant values will be coordinates relative to the center of rotation of the ship. The polygon to draw the booster flame will be implemented in the same way (using *origFlameXPts* and *origFlameYPts*). The values I have used in code 4.1 create the classic arrowhead shaped ship as seen to the right. →

*Note: The coordinates for the booster flame create a triangle, but it when drawn it meets up flush with the indented back of the ship because the ship is drawn over a small part of the flame. This happens because the ship is drawn *after* the flame and overwrites any pixels that overlap.

For descriptions of each variable (and method) in the Ship class, refer to the table in Appendix C. In Code 4.1, the variables of the Ship class are given along with the **constructor** that **initializes** them.

Code 4.1 – Ship.java: Class Variables and the constructor (refer to Appendix C for descriptions of variables of the Ship class)

```
import java.awt.*;

public class Ship {
	// define the shape of the ship and its flame
	final double[] origXPts={14,-10,-6,-10},origYPts={0,-8,0,8},
		origFlameXPts={-6,-23,-6},origFlameYPts={-3,0,3};
	final int radius=6; // radius of circle used to approximate the ship

	double x, y, angle, xVelocity, yVelocity, acceleration,
		velocityDecay, rotationalSpeed; //variables used in movement
	boolean turningLeft, turningRight, accelerating, active;
	int[] xPts, yPts, flameXPts, flameYPts; //store the current locations
					//of the points used to draw the ship and its flame
	int shotDelay, shotDelayLeft; //used to determine the rate of firing
```

```
	public Ship(double x, double y, double angle, double acceleration,
			double velocityDecay, double rotationalSpeed,
			int shotDelay){
		// this.x refers to the Ship's x, x refers to the x parameter
		this.x=x;
		this.y=y;
		this.angle=angle;
		this.acceleration=acceleration;
		this.velocityDecay=velocityDecay;
		this.rotationalSpeed=rotationalSpeed;
		xVelocity=0; // not moving
		yVelocity=0;
		turningLeft=false; // not turning
		turningRight=false;
		accelerating=false; // not accelerating
		active=false; // start off paused
		xPts=new int[4]; // allocate space for the arrays
		yPts=new int[4];
		flameXPts=new int[3];
		flameYPts=new int[3];
		this.shotDelay=shotDelay; // # of frames between shots
		shotDelayLeft=0; // ready to shoot
	}
}
```

Drawing the ship

Before drawing the ship, we need to calculate where the points for the polygons of the ship and its booster flame should be on the screen. Since *origFlameXPts* and *origFlameYPts* are relative to the rotation point of the ship, we will first rotate their values to the correct direction. The formulas to rotate a point (X, Y) around the origin by the angle A are:

$$newX = X*\cos(A) - Y*\sin(A)$$
$$newY = X*\sin(A) + Y*\cos(A)$$

Note that the angle is in radians, not degrees. After rotating the points, we must move them to the location of the ship by adding *x* and *y* (the coordinates for the center of the ship). Finally, we round the values to the nearest integer by adding .5 and casting them as integers, which truncates the decimal (i.e. .5 + .5 = 1.0 truncates to 1; .4 + .5 = 0.9 truncates to 0). We cast the values as integers because the drawPolygon() method we will use only accepts integers as parameters.

Once we have created arrays of integers representing the polygons, we simply set the color we want them drawn and then make a call to drawPolygon(). The first two parameters of drawPolygon() are an array of x coordinates and an array of y coordinates for the points of the polygon. The last parameter is the number of points in the polygon.

Code 4.2 – Ship.java: The draw method calculates where the polygons should be, then draws them.

```
public void draw(Graphics g){
	//rotate the points, translate them to the ship's location (by
```

```
        //adding x and y), then round them by adding .5 and casting them
        //as integers (which truncates any decimal place)
        if(accelerating && active){ // draw flame if accelerating
                for(int i=0;i<3;i++){
                        flameXPts[i]=(int)(origFlameXPts[i]*Math.cos(angle)-
                                        origFlameYPts[i]*Math.sin(angle)+
                                        x+.5);
                        flameYPts[i]=(int)(origFlameXPts[i]*Math.sin(angle)+
                                        origFlameYPts[i]*Math.cos(angle)+
                                        y+.5);
                }
                g.setColor(Color.red); //set color of flame
                g.fillPolygon(flameXPts,flameYPts,3); // 3 is # of points
        }
        //calculate the polygon for the ship, then draw it
        for(int i=0;i<4;i++){
                xPts[i]=(int)(origXPts[i]*Math.cos(angle)- //rotate
                                origYPts[i]*Math.sin(angle)+
                                x+.5); //translate and round
                yPts[i]=(int)(origXPts[i]*Math.sin(angle)+ //rotate
                                origYPts[i]*Math.cos(angle)+
                                y+.5); //translate and round
        }
        if(active) // active means game is running (not paused)
                g.setColor(Color.white);
        else // draw the ship dark gray if the game is paused
                g.setColor(Color.darkGray);
        g.fillPolygon(xPts,yPts,4); // 4 is the number of points
}
```

Moving the ship

The move method will be called once for each frame of the game. This makes it a convenient place to decrement the *shotDelayLeft* variable, which must be zero for the ship to be able to fire. When the ship does fire, *shotDelayLeft* is set to the value in *shotDelay*. Thus, the ship can fire one shot for every *shotDelay* frames (letting it fire a shot every frame would be giving the player a super machine gun).

After decrementing *shotDelay*, add the negative or positive *rotationalSpeed* to *angle* according to whether the ship is turning left or right. Although it may not be practically necessary, I consider it good programming practice to make sure *angle* stays within the bounds of one full circle from 0 to 2π. Otherwise the user could theoretically spin the ship until *angle* overloaded the capacity of the double type, at which point it would exhibit erratic behavior.

Next, add the respective components of acceleration to the x and y velocities if the ship is accelerating. Then move the ship by adding the velocity to its position. After moving the ship, the velocity needs to be decayed by multiplying it by *velocityDecay*, which is a percentage value. This creates a neat drift effect and also effectively limits the maximum speed of the ship. For instance, if *velocityDecay* is .8 and *acceleration* is 1, then the maximum speed of the ship would be 5 pixels per frame. This is where the decay and acceleration cancel each other out. The formula would be (max velocity) * (velocity decay) + (acceleration) = (max velocity). In this case, 5*.8 + 1 = 5, resulting in zero net change in velocity.

Finally, when the ship goes out of the bounds of the screen, we want to make it wrap around into the opposite side of the screen. The code for the move() method is given in code 4.3.

Code 4.3 – Ship.java: The move method (1) decrements *shotDelayLeft*, (2) adds the acceleration value to the speed of the ship if it is accelerating, (3) moves the ship according to its velocity, (4) decays the ships velocity, and (5) wraps the location of the ship around to the opposite side of the screen if it goes out of bounds.

```
public void move(int scrnWidth, int scrnHeight){
	if(shotDelayLeft>0) //move() is called every frame that the game
		shotDelayLeft--; //is run, so this ticks down the shot delay
	if(turningLeft) //this is backwards from typical polar coordinates
		angle-=rotationalSpeed; //because positive y is downward.
	if(turningRight) //Because of that, adding to the angle is
		angle+=rotationalSpeed; //rotating clockwise (to the right)
	if(angle>(2*Math.PI)) //Keep angle within bounds of 0 to 2*PI
		angle-=(2*Math.PI);
	else if(angle<0)
		angle+=(2*Math.PI);
	if(accelerating){ //adds accel to velocity in direction pointed
		//calculates components of accel and adds them to velocity
		xVelocity+=acceleration*Math.cos(angle);
		yVelocity+=acceleration*Math.sin(angle);
	}
	x+=xVelocity; //move the ship by adding velocity to position
	y+=yVelocity;
	xVelocity*=velocityDecay; //slows ship down by percentages (velDecay
	yVelocity*=velocityDecay; //should be a decimal between 0 and 1
	if(x<0) //wrap the ship around to the opposite side of the screen
		x+=scrnWidth; //when it goes out of the screen's bounds
	else if(x>scrnWidth)
		x-=scrnWidth;
	if(y<0)
		y+=scrnHeight;
	else if(y>scrnHeight)
		y-=scrnHeight;
}
```

Implementing other needed methods

All that is left to write in our Ship class are methods that allow other classes to get and set different values and a shoot() method. The getter and setter methods are all fairly straightforward and are given in code 4.4. Shoot() will be added in chapter 5 when we create the Shot class.

Code 4.4 – Ship.java: Getter and setter methods

```
public void setAccelerating(boolean accelerating){
	this.accelerating=accelerating; //start or stop accelerating the ship
}
```

```
public void setTurningLeft(boolean turningLeft){
	this.turningLeft=turningLeft; //start or stop turning the ship
}

public void setTurningRight(boolean turningRight){
	this.turningRight=turningRight;
}

public double getX(){
	return x; // returns the ship's x location
}

public double getY(){
	return y;
}

public double getRadius(){
	return radius; // returns radius of circle that approximates the ship
}

public void setActive(boolean active){
	this.active=active; //used when the game is paused or unpaused
}

public boolean isActive(){
	return active;
}

public boolean canShoot(){
	if(shotDelayLeft>0) //checks to see if the ship is ready to
		return false;	//shoot again yet or if it needs to wait longer
	else
		return true;
}
```

Using a Ship in AsteroidsGame

To put the ship into the game, make the following changes in AsteroidsGame (see code 4.5 for the actual code):

1. Declare a Ship object and a boolean *paused* variable.
2. Instantiate the Ship object and *paused* in the init function.
3. Call the Ship's draw() method (sending *g* as the parameter) in paint().
4. Insert a conditional call to the Ship's move() method in run(). It should only get called if the game is not paused.
5. Insert code for event handling for the arrow keys and for enter (the pause key).

After making these changes, your game should start out with a grayed out ship in the middle of the screen when you run the program. Nothing should happen until you push enter to start the game. When the game is unpaused, the ship should be white and should move around as you push the arrow keys.

Code 4.5 – AsteroidsGame.java: Ship test – moving a ship with the arrow keys.

```
public class AsteroidsGame extends Applet implements Runnable, KeyListener{

	... //Same variables as in framework
	Ship ship;
	boolean paused; // True if the game is paused. Enter is the pause key

	public void init(){
		resize(500,500);

		// I like .35 for acceleration, .98 for velocityDecay, and
		// .1 for rotationalSpeed.  They give the controls a nice feel.
		ship=new Ship(250,250,0,.35,.98,.1,12); //creates the ship
		paused=false;

		... //same as from framework
	}

	public void paint(Graphics gfx){
		g.setColor(Color.black);
		g.fillRect(0,0,500,500);
		ship.draw(g); //draw the ship
		gfx.drawImage(img,0,0,this);
	}

	public void run(){
		for(;;){
			startTime=System.currentTimeMillis();
			if(!paused)
				ship.move(dim.width,dim.height); //move the ship
			repaint();
			try{
				endTime=System.currentTimeMillis();
				if(framePeriod-(endTime-startTime)>0)
					Thread.sleep(framePeriod-(endTime-startTime));
			}catch(InterruptedException e){
			}
		}
	}

	public void keyPressed(KeyEvent e){
		if(e.getKeyCode()==KeyEvent.VK_ENTER){
			//These first two lines allow the asteroids to move
			//while the player chooses when to enter the game.
			//This happens when the player is starting a new life.
			if(!ship.isActive() && !paused)
				ship.setActive(true);
			else{
				paused=!paused; //enter is the pause button
				if(paused) // grays out the ship if paused
					ship.setActive(false);
				else
					ship.setActive(true);
			}
```

```
            }else if(paused || !ship.isActive()) //if the game is
                  return;     //paused or ship is inactive, do not respond
                        //to the controls except for enter to unpause
            else if(e.getKeyCode()==KeyEvent.VK_UP)
                  ship.setAccelerating(true);
            else if(e.getKeyCode()==KeyEvent.VK_LEFT)
                  ship.setTurningLeft(true);
            else if(e.getKeyCode()==KeyEvent.VK_RIGHT)
                  ship.setTurningRight(true);
      }

      public void keyReleased(KeyEvent e){
            if(e.getKeyCode()==KeyEvent.VK_UP)
                  ship.setAccelerating(false);
            else if(e.getKeyCode()==KeyEvent.VK_LEFT)
                  ship.setTurningLeft(false);
            else if(e.getKeyCode()==KeyEvent.VK_RIGHT)
                  ship.setTurningRight(false);
      }

      ... //Other methods the same as in framework
}
```

Chapter 5: Implementing the Shot Class

Overview

In this chapter, we create the Shot class and give our ship the ability to shoot. As with the Ship class, we make a new file and create the Shot class in it. Then we create the variables, the constructor, the draw() and move() methods, and other utility methods. After that, we return to the Ship class to add a new shoot() method to it. Finally, we go to the AsteroidsGame class to finish adding shots into the game.

Creating the variables and the constructor

Create a Shot.java file in the same folder as your other files and then create the Shot class in the file. The Shot class will have variables to represent its location, velocity components, and how much longer it will last for before disappearing if it doesn't hit anything. It will also have a constant representing the speed at which the shots travel.

The best way to set up the constructor is to take the initial coordinates of the shot, the angle it was fired at, the velocity of the ship it was fired from, and its life as parameters. Then the corresponding variables in Ship (coordinates and angle) can just be passed into the constructor when a shot is fired. The x and y velocity is calculated from the angle of the ship, the velocity of the ship, and the constant *shotSpeed* in the constructor as shown in code 5.1. Shot speed can be thought of as the speed the shots would be fired at if the ship were not moving. If the ship is moving, then its movement is added to that speed.

Like the Ship class, Shot will have a move() method that is called once per frame. This makes it a convenient place to decrement the amount of time the Shot has left before disappearing (*lifeLeft*). Then the move() method will add the velocity to the position and wrap it around to the other side of the screen if needed.

The draw() method simply draws a yellow circle of radius 3 (to make it visible) at the nearest point with integer coordinates.

The complete code for Shot is shown in code 5.1.

Code 5.1 – Shot.java: The completed Shot class.

```
import java.awt.*;

public class Shot {
	final double shotSpeed=12; //the speed at which the shots move, in
                                   //pixels per frame
	double x,y,xVelocity,yVelocity; //variables for movement
	int lifeLeft; //causes the shot to eventually disappear if it doesn't
                      //hit anything

	public Shot(double x, double y, double angle, double shipXVel,
                    double shipYVel, int lifeLeft){
		this.x=x;
		this.y=y;
		// add the velocity of the ship to the shot velocity
		// (so the shot's velocity is relative to the ship's velocity)
```

```
            xVelocity=shotSpeed*Math.cos(angle)+shipXVel;
            yVelocity=shotSpeed*Math.sin(angle)+shipYVel;
            // the number of frames the shot will last for before
            // disappearing if it doesn't hit anything
            this.lifeLeft=lifeLeft;
      }

      public void move(int scrnWidth, int scrnHeight){
            lifeLeft--; // used to make shot disappear if it goes too long
                        // without hitting anything
            x+=xVelocity; // move the shot
            y+=yVelocity;
            if(x<0) // wrap the shot around to the opposite side of the
                  x+=scrnWidth; // screen if needed
            else if(x>scrnWidth)
                  x-=scrnWidth;
            if(y<0)
                  y+=scrnHeight;
            else if(y>scrnHeight)
                  y-=scrnHeight;
      }

      public void draw(Graphics g){
            g.setColor(Color.yellow); //set shot color
            //draw circle of radius 3 centered at the closest point
            //with integer coordinates (.5 added to x-1 and y-1 for rounding)
            g.fillOval((int)(x-.5), (int)(y-.5), 3, 3);
      }

      public double getX(){
            return x;
      }

      public double getY(){
            return y;
      }

      public int getLifeLeft(){
            return lifeLeft;
      }
}
```

Implementing Ship's shoot() method

After creating the shot class, the shoot() method can be added to the Ship class. This method is very simple. It only involves creating a new Shot and returning it. The ship's position, angle, and velocity are sent in to the Shot constructor to be used in calculating the shot's position and velocity. See code 5.2 for the completed method.

Code 5.2 – Ship.java: The completed shoot() method.

```
public Shot shoot(){
      shotDelayLeft=shotDelay; //set delay till next shot can be fired
```

```
        //a life of 40 makes the shot travel about the width of the
        //screen before disappearing
        return new Shot(x,y,angle,xVelocity,yVelocity,40);
}
```

Using Shot in AsteroidsGame

To make it to where the ship can fire, do following things in the AsteroidsGame class (see Code 5.3):

1. Add an array of Shot objects, the *numShots* integer to store how many shots are on the screen, and a boolean *shooting* value for event handling.
2. Allocate space for the Shot array, set *numShots* to 0, and set *shooting* to false in the init() method.
3. Add a loop to call the draw() method for each shot in paint().
4. Add code in run() to move each shot, delete the ones that die off, and add new ones to the array when the ship is shooting.
5. Add event handling to set the *shooting* variable when the ctrl key is pressed or released.

After making these changes, run the program. The ship should now be able to shoot little yellow dots when "ctrl" is pressed and the game is not paused.

Code 5.3 – AsteroidsGame.java: Shooting test – Use ctrl key to shoot.

```
... // Same variables from framework

Ship ship;
boolean paused; // True if the game is paused. Enter is the pause key
Shot[] shots; //Variable that stores the new array of Shots
int numShots; //Stores the number of shots in the array
boolean shooting; //true if the ship is currently shooting

public void init(){
        resize(500,500);

        // I like .35 for acceleration, .98 for velocityDecay, and
        // .1 for rotationalSpeed.  They give the controls a nice feel.
        ship=new Ship(250,250,0,.35,.98,.1,12); //creates the ship
        paused=true;
        shots=new Shot[41]; //Allocate the space for the array.
                //We allocate enough space to store the maximum number of
                //shots that can possibly be on the screen at one time.
                //41 is the max because no more than one shot can be fired per
                //frame and shots only last for 40 frames (40 is the value passed
                //in for lifeLeft when shots are created)
        numShots=0; //no shots on the screen to start with.
        shooting=false; //the ship is not shooting

        ... //Same code from framework here
}
```

```
public void paint(Graphics gfx){
	g.setColor(Color.black);
	g.fillRect(0,0,500,500);

	ship.draw(g); //draw the ship
	for(int i=0;i<numShots;i++) //loop that calls draw() for each shot
		shots[i].draw(g);

	gfx.drawImage(img,0,0,this);
}

public void run(){
	for(;;){
		startTime=System.currentTimeMillis();

		if(!paused){
			ship.move(dim.width,dim.height); // move the ship
			//this loop moves each shot and deletes dead shots
			for(int i=0;i<numShots;i++){
				shots[i].move(dim.width,dim.height);
				//removes shot if it has gone for too long
				//without hitting anything
				if(shots[i].getLifeLeft()<=0){
					//shifts all the next shots up one
					//space in the array
					deleteShot(i); //SEE NEW METHOD BELOW
					i--; // move the outer loop back one so
						// the shot shifted up is not skipped
				}
			}
			if(shooting && ship.canShoot()){
				//add a shot on to the array if the ship is shooting
				shots[numShots]=ship.shoot();
				numShots++;
			}
		}

		repaint();
		try{
			endTime=System.currentTimeMillis();
			if(framePeriod-(endTime-startTime)>0)
			Thread.sleep(framePeriod-(endTime-startTime));
		}catch(InterruptedException e){
		}
	}
}

private void deleteShot(int index){
	//delete shot and move all shots after it up in the array
	numShots--;
	for(int i=index;i<numShots;i++)
		shots[i]=shots[i+1];
	shots[numShots]=null;
}

public void keyPressed(KeyEvent e){
```

```
        ... //Same as before
        else if(e.getKeyCode()==KeyEvent.VK_CONTROL)
                shooting=true; //Start shooting when ctrl is pushed
}

public void keyReleased(KeyEvent e){
        ... // Same as before
        else if(e.getKeyCode()==KeyEvent.VK_CONTROL)
                shooting=false; //Stop shooting when ctrl is released
}
```

Chapter 6: Implementing the Asteroid Class

Overview

In this chapter, we complete our basic Asteroids game. First, we create the Asteroid class complete with its variables, constructor, and basic methods (similar to the Ship and Shot class). Next, we add methods that detect when collisions occur between the asteroid and the ship or any of the shots. After implementing collision detection, we create a method used to split the asteroid up when it is hit by a shot. Finally, we add asteroids into the AsteroidsGame class. For simplicity's sake, each level will have the number of asteroids equal to the level number. When all the asteroids are destroyed, the game will advance to the next level. If the ship collides with an asteroid, the game will start that level over.

Creating the variables, constructor, and basic methods

In the same folder with your other files, create an Asteroid.java file with a class named Asteroid in it. Just like the Shot class, Asteroid will need variables to store its location (*x* and *y*) and velocity (*xVelocity* and *yVelocity*). It will also need variables for its size (*radius*), the number of smaller asteroids it will break up into (*numSplit*), and the number of hits it has left (*hitsLeft*). Each time the asteroid is shot, it will break up into *numSplit* smaller asteroids that each have one fewer *hitsLeft* than the asteroid that was shot. When an asteroid only has one hit left, it will completely disappear when it is shot instead of breaking up any further.

The constructor naturally initializes each variable according to the parameters passed in. The main difference between this constructor and the constructor for Shot is that this one calculates a random velocity and direction based on the *maxVelocity* and *minVelocity* parameters. Math.random() is a function that returns a random decimal value between zero and one. Using it, we can generate a random value between zero and any number simply by multiplying it by that number. In the constructor, it is used to give the asteroid a random velocity between *minVelocity* and *maxVelocity* as well as a random angle (or direction) between 0 and 2π (a full circle in radians).

The move() and draw() methods are also very similar to those for Shot. The major difference in move() is that we allow the asteroids to go a distance equal to their radius beyond the edges of the screen before being wrapped around to the other side. This allows the asteroids to go completely off the screen before reemerging on the other side. Otherwise, the asteroid would disappear when it is half off one side and reappear half on the other side of the screen. If this happened while the ship were close to the other edge, the asteroid could appear on top of the ship, giving the player no time to react. The only differences in the draw() method are the color and the radius of the circle being drawn.

The getHitsLeft() method simply returns hitsLeft so that the AsteroidsGame class can determine whether the asteroid should be split up further or not. GetNumSplit() returns the number of smaller asteroids that this asteroid should split into when hit by a shot.

Code 6.1 – Asteroid.java: The basic methods of the Asteroid class.

```
import java.awt.*;
```

```
public class Asteroid {
	double x, y, xVelocity, yVelocity, radius;
	int hitsLeft, numSplit;

	public Asteroid(double x,double y,double radius,double minVelocity,
			double maxVelocity,int hitsLeft,int numSplit){
		this.x=x;
		this.y=y;
		this.radius=radius;
		this.hitsLeft=hitsLeft; //number of shots left to destroy it
		this.numSplit=numSplit; //number of smaller asteroids it
					//breaks up into when shot
		//calculates a random direction and a random
		//velocity between minVelocity and maxVelocity
		double vel=minVelocity + Math.random()*(maxVelocity-minVelocity),
			dir=2*Math.PI*Math.random(); // random direction
		xVelocity=vel*Math.cos(dir);
		yVelocity=vel*Math.sin(dir);
	}

	public void move(int scrnWidth, int scrnHeight){
		x+=xVelocity; //move the asteroid
		y+=yVelocity;
		//wrap around code allowing the asteroid to go off the screen
		//to a distance equal to its radius before entering on the
		//other side.  Otherwise, it would go halfway off the sceen,
		//then disappear and reappear halfway on the other side
		//of the screen.
		if(x<0-radius)
			x+=scrnWidth+2*radius;
		else if(x>scrnWidth+radius)
			x-=scrnWidth+2*radius;
		if(y<0-radius)
			y+=scrnHeight+2*radius;
		else if(y>scrnHeight+radius)
			y-=scrnHeight+2*radius;
	}

	public void draw(Graphics g){
		g.setColor(Color.gray); // set color for the asteroid
		// draw the asteroid centered at (x,y)
		g.fillOval((int)(x-radius+.5),(int)(y-radius+.5),
				(int)(2*radius),(int)(2*radius));
	}

	public int getHitsLeft(){
		//used by AsteroidsGame to determine whether the asteroid should
		//be split up into smaller asteroids or destroyed completely.
		return hitsLeft;
	}

	public int getNumSplit(){
		return numSplit;
	}
```

```
}
```

Detecting collisions

Collision detection is the programming necessary to determine when one object in a game collides with another. In asteroids, the only collisions we need to detect and respond to are collisions between a shot and an asteroid and collisions between the ship and an asteroid. Because these both involve an asteroid, we will implement the methods that detect collisions in the Asteroid class.

To greatly simplify the calculations needed to detect collisions, we will approximate the shape of the ship with a circle. If the sum of the radii of two circles is greater than the distance between the centers of those circles, then the circles are touching (or overlapping) each other. Using the distance formula, the circles have collided if the following statement is true,

$$r1+r2 > \text{sqrt}((x1-x2)^2 + (y1-y2)^2)$$

Where (x1,y1), (x2,y2) are the centers of the circles and r1, r2 are their radii. Because calculating square roots requires more processing time than multiplying, we can make our program more efficient by squaring both sides of the inequality. Then, we can determine if the circles have collided by using this inequality:

$$(r1+r2)^2 > (x1-x2)^2 + (y1-y2)^2$$

When determining if a shot has collided with an asteroid, we will treat the shots as points with no radius. This leaves only the radius of the asteroid to be squared on the left side of the inequality.

The two methods we need for collision detection will accept either a Ship or a Shot as a parameter and will return true if the asteroid has collided with that Ship or Shot. Otherwise, they will return false. Their encoding is given in Code 6.2.

Code 6.2 – Asteroid.java: Collision detection methods in the Asteroid class.

```
public boolean shipCollision(Ship ship){
      // Use the distance formula to check if the ship is touching this
      // asteroid: Distance^2 = (x1-x2)^2 + (y1-y2)^2    ("^" denotes
      // exponents).  If the sum of the radii is greater than the
      // distance between the center of the ship and asteroid, they are
      // touching.
      // if (shipRadius + asteroidRadius)^2 > (x1-x2)^2 + (y1-y2)^2,
      // then they have collided.
      // It does not check for collisions if the ship is not active
      // (the player is waiting to start a new life or the game is paused).
      if(Math.pow(radius+ship.getRadius(),2) > Math.pow(ship.getX()-x,2)+
                  Math.pow(ship.getY()-y,2) && ship.isActive())
            return true;
      return false;
}

public boolean shotCollision(Shot shot){
      // Same idea as shipCollision, but using shotRadius = 0
      if(Math.pow(radius,2) > Math.pow(shot.getX()-x,2)+
                  Math.pow(shot.getY()-y,2))
            return true;
      return false;
```

```
}
```

Splitting the asteroid

We need one last method that will create the smaller asteroids when the asteroid is shot. A good way to determine the size of the smaller asteroids is to make the sum of their areas equal to the area of the original asteroid. This can be done simply by setting their radii equal to the radius of the original asteroid divided by the square root of *numSplit*. The smaller asteroids are given one less hit left than the original asteroid. They will split up into the same number of asteroids as the original if they still have a hit left. The code for the method is given in Code 6.3.

Code 6.3 – Asteroid.java: Method that creates smaller asteroids for when this asteroid is split up.

```
public Asteroid createSplitAsteroid(double minVelocity,
            double maxVelocity){
      //when this asteroid gets hit by a shot, this method is called
      //numSplit times by AsteroidsGame to create numSplit smaller
      //asteroids.  Dividing the radius by sqrt(numSplit) makes the
      //sum of the areas taken up by the smaller asteroids equal to
      //the area of this asteroid.  Each smaller asteroid has one
      //less hit left before being completely destroyed.
      return new Asteroid(x,y,radius/Math.sqrt(numSplit),
                  minVelocity,maxVelocity,hitsLeft-1,numSplit);
}
```

Adding asteroids into AsteroidsGame

Now that our Ship, Shot, and Asteroid classes are complete, we are ready to finish off the game with some final changes to the AsteroidsGame class. We need to do the following:

1. Create an array of Asteroids large enough to hold the maximum number of asteroids possible and a variable to store the number of asteroids in it.
2. Call the draw() function of each asteroid and display the level number in paint().
3. Move each asteroid and check for collisions with the ship or any of the shots in the run() method. Restart the level if the ship collides with an asteroid. Split or delete the asteroid if a shot collides with it.
4. Start a new level when all the asteroids are destroyed.

To accomplish these tasks, we will need to add more variables and methods to AsteroidsGame. The new variables are explained at the beginning of Code 6.4. All of the methods in Code 6.4 have either been modified or added to the AsteroidsGame class. Descriptions are available in Appendix A.

Code 6.4 – AsteroidsGame.java: Changes to existing methods and new methods to complete AsteroidsGame.

```
... // same code as before
Asteroid[] asteroids; //the array of asteroids
int numAsteroids; //the number of asteroids currently in the array
double astRadius,minAstVel,maxAstVel; //values used to create asteroids
int astNumHits,astNumSplit;

int level; //the current level number

public void init(){
      resize(500,500);

      // NOTE THAT SOME OF THE INITIALIZATIONS HERE HAVE BEEN MOVED TO THE
      // setUpNextLevel() FUNCTION.
      shots=new Shot[41]; //41 is a shot's life period plus one.
            //since at most one shot can be fired per frame,
            //there will never be more than 41 shots if each one only
            //lives for 40 frames.

      numAsteroids=0;
      level=0; //will be incremented to 1 when first level is set up
      astRadius=60; //values used to create the asteroids
      minAstVel=.5;
      maxAstVel=5;
      astNumHits=3;
      astNumSplit=2;

      endTime=0;
      startTime=0;
      framePeriod=25;
      addKeyListener(this); //tell it to listen for KeyEvents
      dim=getSize();
      img=createImage(dim.width, dim.height);
      g=img.getGraphics();
      thread=new Thread(this);
      thread.start();
}

public void setUpNextLevel(){ //starts a new level with one more asteroid
      level++;
      // create a new, inactive ship centered on the screen
      // I like .35 for acceleration, .98 for velocityDecay, and
      // .1 for rotationalSpeed.  They give the controls a nice feel.
      ship=new Ship(250,250,0,.35,.98,.1,12);
      numShots=0; //no shots on the screen at beginning of level
      paused=false;
      shooting=false;
      //create an array large enough to hold the biggest number
      //of asteroids possible on this level (plus one because
      //the split asteroids are created first, then the original
      //one is deleted).  The level number is equal to the
      //number of asteroids at it's start.
      asteroids=new Asteroid[level *
              (int)Math.pow(astNumSplit,astNumHits-1)+1];
```

```
		numAsteroids=level;
		//create asteroids in random spots on the screen
		for(int i=0;i<numAsteroids;i++)
			asteroids[i]=new Asteroid(Math.random()*dim.width,
					Math.random()*dim.height,astRadius,minAstVel,
					maxAstVel,astNumHits,astNumSplit);
	}

	public void paint(Graphics gfx){
		g.setColor(Color.black);
		g.fillRect(0,0,500,500);
		for(int i=0;i<numShots;i++) //draw all the shots on the screen
			shots[i].draw(g);
		for(int i=0;i<numAsteroids;i++)
			asteroids[i].draw(g);
			ship.draw(g); //draw the ship
		g.setColor(Color.cyan); //Display the level number in top left corner
		g.drawString("Level " + level,20,20);

		gfx.drawImage(img,0,0,this);
	}

	public void run(){
		for(;;){
			startTime=System.currentTimeMillis();

			//start next level when all asteroids are destroyed
			if(numAsteroids<=0)
				setUpNextLevel();

			if(!paused){
				ship.move(dim.width,dim.height); // move the ship
				//move shots and remove dead shots
				for(int i=0;i<numShots;i++){
					shots[i].move(dim.width,dim.height);
					//removes shot if it has gone for too long
					//without hitting anything
					if(shots[i].getLifeLeft()<=0){
						//shifts all the next shots up one
						//space in the array
						deleteShot(i);
						i--; // move the outer loop back one so
							// the shot shifted up is not skipped
					}
				}
				//move asteroids and check for collisions
				updateAsteroids();//SEE NEW METHOD BELOW

				if(shooting && ship.canShoot()){
					//add a shot on to the array
					shots[numShots]=ship.shoot();
					numShots++;
				}
			}

			repaint();
			try{
```

```
                endTime=System.currentTimeMillis();
                if(framePeriod-(endTime-startTime)>0)
                      Thread.sleep(framePeriod-(endTime-startTime));
            }catch(InterruptedException e){
            }
      }
}

private void deleteShot(int index){
      //delete shot and move all shots after it up in the array
      numShots--;
      for(int i=index;i<numShots;i++)
            shots[i]=shots[i+1];
      shots[numShots]=null;
}

private void deleteAsteroid(int index){
      //delete asteroid and shift ones after it up in the array
      numAsteroids--;
      for(int i=index;i<numAsteroids;i++)
            asteroids[i]=asteroids[i+1];
      asteroids[numAsteroids]=null;
}

private void addAsteroid(Asteroid ast){
      //adds the asteroid passed in to the end of the array
      asteroids[numAsteroids]=ast;
      numAsteroids++;
}

private void updateAsteroids(){
      for(int i=0;i<numAsteroids;i++){
            // move each asteroid
            asteroids[i].move(dim.width,dim.height);
            //check for collisions with the ship, restart the
            //level if the ship gets hit
            if(asteroids[i].shipCollision(ship)){
                  level--; //restart this level
                  numAsteroids=0;
                  return;
            }
            //check for collisions with any of the shots
            for(int j=0;j<numShots;j++){
                  if(asteroids[i].shotCollision(shots[j])){
                        //if the shot hit an asteroid, delete the shot
                        deleteShot(j);
                        //split the asteroid up if needed
                        if(asteroids[i].getHitsLeft()>1){
                              for(int k=0;k<asteroids[i].getNumSplit();k++)
                                    addAsteroid(
                                          asteroids[i].createSplitAsteroid(
                                          minAstVel,maxAstVel));
                        }
                        //delete the original asteroid
                        deleteAsteroid(i);
                        j=numShots; //break out of inner loop - it has
                              //already been hit, so don't need to check
```

```
                        //for collision with other shots
                    i--; //don't skip asteroid shifted back into
                        //the deleted asteroid's position
                }
            }
        }
}

... // other methods the same
```

With these changes, you should now have a working, basic Asteroids game. The complete code needed for each class is located in Appendices A through D.

Chapter 7: Customizing and Modifying the Game

Feel free to customize or modify your game in any way you desire. I have listed below some of my ideas for ways to expand and enhance the game. They are listed roughly in order of increasing difficulty.

1. Play with some of the different values to find the ones you like best for the game.
2. Display the players hit percentage (total number of shots that have hit an asteroid divided by the total number of shots fired, then multiplied by 100). You will need to add variables to keep track of the cumulative number of shots fired and of shots that hit asteroids.
3. Add in a limited number of lives. When they have been used up, start the game over at level 1. If you do this, do not restart the level when the ship crashes.
4. Add in a scoring system. This could include points for destroying asteroids, beating levels, etc. Bonus lives could be awarded for certain amounts of points
5. Create a customized set of levels using different values for things like the number of asteroids, their number of hits, their size, the number of asteroids they break into, the rate at which the ship can fire, the velocities of the ship and asteroids, etc. These values could be read in from text files to initialize each level. I personally enjoy having a level where the ship has a machinegun. You can do this by giving the ship a very small delay between when shots can be fired.
6. Create a Graphical User Interface (GUI) for the game. This could include a starting menu with buttons, options, etc. (Note that if you use Swing components, your AsteroidsGame class will need to extend JApplet instead of Applet. Swing components do not display properly in a regular Applet).
7. Make the game two player (cooperative and/or player vs. player). Perhaps the letters W, S, A, and D could be used as the controls for player two.
8. Add in UFOs that either shoot randomly or at the player (like in the original asteroids game).
9. Add in different weapons or bonuses. A spread shot would not be hard to add (instead of shooting one bullet at a time, the ship fires multiple bullets that fan out).

Appendix A: The AsteroidsGame Class

In this appendix, Table A.1 describes each variable and method in the AsteroidsGame class and Code A.2 provides a complete listing of the finished code for the AsteroidsGame class. The variables and methods in the table are in the same order as they appear in the code and are grouped together when they are closely related.

Variables	Description
Thread thread	The thread that runs the game.
Dimension dim	Stores the width and height of the applet.
Image img	Acts as the back buffer. Used to implement double buffering.
Graphics g	Used to draw on *img*, the back buffer.
long endTime, startTime, framePeriod	Used to mark the beginning and ending time for the calculations performed in run() so that the rate at which the program is run can be regulated to *framePeriod* milliseconds per frame.
Ship ship	Represents the player's ship.
boolean paused	True if the game is paused. The enter key toggles it.
Shot[] shots int numShots	Array that holds all the Shot objects currently on the screen and integer that tracks the number of shots on the screen.
boolean shooting	True if the user is holding down the fire key, ctrl.
Asteroid[] asteroids int numAsteroids	Array that holds all the Asteroid objects currenly on the screen and integer that tracks the number of asteroids on the screen.
double astRadius, minAstVel, maxAstVel	Variables that hold the radius, minimum velocity, and maximum velocity used for each new asteroid created.
int astNumHits, astNumSplit	Variables that hold the number of hits and the number of smaller asteroids split into used for each new asteroid created.
int level	Holds the current level number, which is equal to the number of asteroids on the screen when the level starts.
Methods	**Description**
void init()	First method called when the Applet starts. Used to initialize all necessary variables and objects.
void setUpNextLevel()	Sets everything up to begin the next level.
void paint()	Draws all the graphics to a back buffer, then copies it to the screen.
void update()	Overridden to only prevent flickering (the inherited version clears the screen then calls paint())
void run()	The method that is continuously being run by the thread. It moves the objects in game and checks for collisions.
void deleteShot()	Removes the shot at the index passed in and shifts all the shots after it up one spot in the array.
void deleteAsteroid()	Removes the asteroid at the index passed in and shifts all the ones after it up one spot in the array.

void addAsteroid()	Adds the asteroid passed in as a parameter to the end of the asteroids array.
void updateAsteroids()	Moves each asteroid, then checks for collisions with both the ship and any of the shots. Responds as needed to collisions.
void keyPressed()	Determines what happens when the user presses a key.
void keyReleased()	Determines what happens when the user releases a key.
void keyTyped()	Empty method that is needed to implement KeyListener.

Table A.1: Variables and methods of the AsteroidsGame class

Code A.2 – AsteroidsGame.java: The final code for the AsteroidsGame class.

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class AsteroidsGame extends Applet implements Runnable, KeyListener{

	Thread thread;
	Dimension dim;
	Image img;
	Graphics g;

	long endTime, startTime, framePeriod;

	Ship ship;
	boolean paused; // True if the game is paused. Enter is the pause key
	Shot[] shots;
	int numShots;
	boolean shooting;

	Asteroid[] asteroids; //the array of asteroids
	int numAsteroids; //the number of asteroids currently in the array
	double astRadius, minAstVel,maxAstVel; //values used to create
	                                       //asteroids
	int astNumHits,astNumSplit;

	int level; //the current level number

	public void init(){
		resize(500,500);

		shots=new Shot[41]; //41 is a shot's life period plus one.
			//since at most one shot can be fired per frame,
			//there will never be more than 41 shots if each one only
			//lives for 40 frames.

		numAsteroids=0;
		level=0; //will be incremented to 1 when first level is set up
		astRadius=60; //values used to create the asteroids
		minAstVel=.5;
		maxAstVel=5;
		astNumHits=3;
		astNumSplit=2;
```

```
        endTime=0;
        startTime=0;
        framePeriod=25;
        addKeyListener(this); //tell it to listen for KeyEvents
        dim=getSize();
        img=createImage(dim.width, dim.height);
        g=img.getGraphics();
        thread=new Thread(this);
        thread.start();
}

public void setUpNextLevel(){ //start new level with one more asteroid
        level++;
        // create a new, inactive ship centered on the screen
        // I like .35 for acceleration, .98 for velocityDecay, and
        // .1 for rotationalSpeed.  They give the controls a nice feel.
        ship=new Ship(250,250,0,.35,.98,.1,12);
        numShots=0; //no shots on the screen at beginning of level
        paused=false;
        shooting=false;
        //create an array large enough to hold the biggest number
        //of asteroids possible on this level (plus one because
        //the split asteroids are created first, then the original
        //one is deleted).  The level number is equal to the
        //number of asteroids at it's start.
        asteroids=new Asteroid[level *
           (int)Math.pow(astNumSplit,astNumHits-1)+1];
        numAsteroids=level;
        //create asteroids in random spots on the screen
        for(int i=0;i<numAsteroids;i++)
              asteroids[i]=new Asteroid(Math.random()*dim.width,
                          Math.random()*dim.height,astRadius,minAstVel,
                          maxAstVel,astNumHits,astNumSplit);
}

public void paint(Graphics gfx){
        g.setColor(Color.black);
        g.fillRect(0,0,500,500);

        for(int i=0;i<numShots;i++) //draw all the shots on the screen
              shots[i].draw(g);

        for(int i=0;i<numAsteroids;i++)
              asteroids[i].draw(g);

        ship.draw(g); //draw the ship

        g.setColor(Color.cyan); //Display level number in top left corner
        g.drawString("Level " + level,20,20);

        gfx.drawImage(img,0,0,this);
}

public void update(Graphics gfx){
        paint(gfx);
}
```

```
public void run(){
	for(;;){
		startTime=System.currentTimeMillis();

		//start next level when all asteroids are destroyed
		if(numAsteroids<=0)
			setUpNextLevel();

		if(!paused){
			ship.move(dim.width,dim.height); // move the ship
			//move shots and remove dead shots
			for(int i=0;i<numShots;i++){
				shots[i].move(dim.width,dim.height);
				//removes shot if it has gone for too long
				//without hitting anything
				if(shots[i].getLifeLeft()<=0){
					//shifts all the next shots up one
					//space in the array
					deleteShot(i);
					i--; // move the outer loop back one so
					    // the shot shifted up is not skipped
				}
			}
			//move asteroids and check for collisions
			updateAsteroids();

			if(shooting && ship.canShoot()){
				//add a shot on to the array
				shots[numShots]=ship.shoot();
				numShots++;
			}
		}

		repaint();
		try{
			endTime=System.currentTimeMillis();
			if(framePeriod-(endTime-startTime)>0)
				Thread.sleep(framePeriod-(endTime-startTime));
		}catch(InterruptedException e){
		}
	}
}

private void deleteShot(int index){
	//delete shot and move all shots after it up in the array
	numShots--;
	for(int i=index;i<numShots;i++)
		shots[i]=shots[i+1];
	shots[numShots]=null;
}

private void deleteAsteroid(int index){
	//delete asteroid and shift ones after it up in the array
	numAsteroids--;
	for(int i=index;i<numAsteroids;i++)
		asteroids[i]=asteroids[i+1];
```

```
        asteroids[numAsteroids]=null;
    }

    private void addAsteroid(Asteroid ast){
        //adds the asteroid passed in to the end of the array
        asteroids[numAsteroids]=ast;
        numAsteroids++;
    }

    private void updateAsteroids(){
        for(int i=0;i<numAsteroids;i++){
            // move each asteroid
            asteroids[i].move(dim.width,dim.height);
            //check for collisions with the ship, restart the
            //level if the ship gets hit
            if(asteroids[i].shipCollision(ship)){
                level--; //restart this level
                numAsteroids=0;
                return;
            }
            //check for collisions with any of the shots
            for(int j=0;j<numShots;j++){
                if(asteroids[i].shotCollision(shots[j])){
                    //if the shot hit an asteroid, delete the shot
                    deleteShot(j);
                    //split the asteroid up if needed
                    if(asteroids[i].getHitsLeft()>1){
                        for(int k=0;k<asteroids[i].getNumSplit();
                                                        k++)
                            addAsteroid(
                                asteroids[i].createSplitAsteroid(
                                minAstVel,maxAstVel));
                    }
                    //delete the original asteroid
                    deleteAsteroid(i);
                    j=numShots; //break out of inner loop - it has
                        //already been hit, don't need to check
                        //for collision with other shots
                    i--; //don't skip asteroid shifted back into
                        //the deleted asteroid's position
                }
            }
        }
    }

    public void keyPressed(KeyEvent e){
        if(e.getKeyCode()==KeyEvent.VK_ENTER){
            //These first two lines allow the asteroids to move
            //while the player chooses when to enter the game.
            //This happens when the player is starting a new life.
            if(!ship.isActive() && !paused)
                ship.setActive(true);
            else{
                paused=!paused; //enter is the pause button
                if(paused) // grays out the ship if paused
                    ship.setActive(false);
                else
```

```
                        ship.setActive(true);
                }
            }else if(paused || !ship.isActive()) //if the game is
                return;     //paused or ship is inactive, do not respond
                    //to the controls except for enter to unpause
            else if(e.getKeyCode()==KeyEvent.VK_UP)
                ship.setAccelerating(true);
            else if(e.getKeyCode()==KeyEvent.VK_LEFT)
                ship.setTurningLeft(true);
            else if(e.getKeyCode()==KeyEvent.VK_RIGHT)
                ship.setTurningRight(true);
            else if(e.getKeyCode()==KeyEvent.VK_CONTROL)
                shooting=true; //Start shooting if ctrl is pushed
        }

        public void keyReleased(KeyEvent e){
            if(e.getKeyCode()==KeyEvent.VK_UP)
                ship.setAccelerating(false);
            else if(e.getKeyCode()==KeyEvent.VK_LEFT)
                ship.setTurningLeft(false);
            else if(e.getKeyCode()==KeyEvent.VK_RIGHT)
                ship.setTurningRight(false);
            else if(e.getKeyCode()==KeyEvent.VK_CONTROL)
                shooting=false;
        }

        public void keyTyped(KeyEvent e){
        }
}
```

Appendix B: The Completed Ship Class

In this appendix, Table B.1 describes each variable and method in the Ship class and Code B.2 provides a complete listing of the finished code for the Ship class. The variables and methods in the table are in the same order as they appear in the code and are grouped together when they are closely related.

Variables	Description
final double[] origXPoints, origYPoints,	Constants that define the shape of the ship. They store the original x and y values for the points used to draw the ship. To draw the ship, we rotate these values in the direction the ship is facing, then translate them to the ship's location.
final double[] origFlameXPoints, origFlameYPoints	Constants that define the shape of the flame that comes out of the back of the ship when it is accelerating. They are used to draw the flame in the same manner as the ship is drawn.
final int radius	Constant that stores the radius of a circle that approximates the ships shape. It is used in collision detection (see chapter 6).
double x, y	Store the current coordinates of the center of the ship.
double angle	Stores the angle (in radians) that the ship is facing. Zero represents facing right and the angle increases as you rotate clockwise.
double xVelocity, yVelocity	Store the velocity the ship is currently moving in the x and y directions.
double acceleration	When the ship is accelerating, this value is added to the ship's velocity (by adding x/y components). It is added once per frame.
double velocityDecay	A value that represents drag, which causes the ship to slow down as it coasts. The ship's velocity is multiplied by this value once each frame. It should be a decimal value between 0 and 1.
double rotationalSpeed	The speed at which the ship turns when the user presses the left or right arrow keys.
boolean turningLeft, turningRight	These are true when the user is holding down the left or right arrow key, respectively.
boolean accelerating	True if the user is holding down the up arrow key
boolean active	False if the game is paused or a level just started. The ship is grayed out, cannot move, and is not hurt by asteroids when it is false. This allows the user to choose when to "enter" a level.
int[] xPts, yPts, flameXPts, flameYPts	Store the current points used to draw the ship (the rotated and translated versions of the constants).
int shotDelay, shotDelayLeft	Store the minimum number of frames between shots and the number of frames left until the next shot can be fired, respectively.

Methods	Description
Ship()	The constructor. Initializes all needed variables when a ship is created.
void draw()	Draws the ship using the Graphics object passed in.
void move()	Moves the ship based on its x and y velocity. Decrements *shotDelayLeft* once per frame.
void setAccelerating() void setTurningLeft() void setTurningRight()	Sets *accelerating*, *turningLeft*, or *turningRight* to true or false. Used for event handling (the value is set to true when the corresponding key is pressed and false when it is released).
double getX() double getY() double getRadius()	Getter methods used in by the Asteroid class for collision detection.
void setActive() boolean isActive()	Sets *active* to true/false and returns the value of *active*. Used when the game is paused and at the beginnings of levels.
boolean canShoot()	Returns true if the ship is ready to shoot (*shotDelayLeft* is zero).
Shot shoot()	Creates a new shot based on the ship's position and velocity, then returns it to be added to the *shots* array.

Table B.1: Variables and methods of the Ship class.

Code B.2 – Ship.java: The final code for the Ship class.

```
import java.awt.*;

public class Ship {
	final double[] origXPts={14,-10,-6,-10},origYPts={0,-8,0,8},
		origFlameXPts={-6,-23,-6},origFlameYPts={-3,0,3};
	final int radius=6;

	double x, y, angle, xVelocity, yVelocity, acceleration,
			velocityDecay, rotationalSpeed;
	boolean turningLeft, turningRight, accelerating, active;
	int[] xPts, yPts, flameXPts, flameYPts;
	int shotDelay, shotDelayLeft;

	public Ship(double x, double y, double angle, double acceleration,
			double velocityDecay, double rotationalSpeed,
			int shotDelay){
		//this.x refers to the Ship's x, x refers to the x parameter
		this.x=x;
		this.y=y;
		this.angle=angle;
		this.acceleration=acceleration;
		this.velocityDecay=velocityDecay;
		this.rotationalSpeed=rotationalSpeed;
		xVelocity=0; // not moving
		yVelocity=0;
		turningLeft=false; // not turning
		turningRight=false;
		accelerating=false; // not accelerating
		active=false; // start off paused
```

```
        xPts=new int[4]; // allocate space for the arrays
        yPts=new int[4];
        flameXPts=new int[3];
        flameYPts=new int[3];
        this.shotDelay=shotDelay; // # of frames between shots
        shotDelayLeft=0; // ready to shoot
    }

    public void draw(Graphics g){
        //rotate the points, translate them to the ship's location (by
        //adding x and y), then round them by adding .5 and casting them
        //as integers (which truncates any decimal place)
        if(accelerating && active){ // draw flame if accelerating
            for(int i=0;i<3;i++){
                flameXPts[i]=(int)(origFlameXPts[i]*Math.cos(angle)-
                        origFlameYPts[i]*Math.sin(angle)+
                        x+.5);
                flameYPts[i]=(int)(origFlameXPts[i]*Math.sin(angle)+
                        origFlameYPts[i]*Math.cos(angle)+
                        y+.5);
            }
            g.setColor(Color.red); //set color of flame
            g.fillPolygon(flameXPts,flameYPts,3); // 3 is # of points
        }
        //calculate the polgyon for the ship, then draw it
        for(int i=0;i<4;i++){
            xPts[i]=(int)(origXPts[i]*Math.cos(angle)- //rotate
                    origYPts[i]*Math.sin(angle)+
                    x+.5); //translate and round
            yPts[i]=(int)(origXPts[i]*Math.sin(angle)+ //rotate
                    origYPts[i]*Math.cos(angle)+
                    y+.5); //translate and round
        }
        if(active) // active means game is running (not paused)
            g.setColor(Color.white);
        else // draw the ship dark gray if the game is paused
            g.setColor(Color.darkGray);
        g.fillPolygon(xPts,yPts,4); // 4 is number of points
    }

    public void move(int scrnWidth, int scrnHeight){
        if(shotDelayLeft>0) //move() is called every frame that the game
            shotDelayLeft--; //is run; this ticks down the shot delay
        if(turningLeft) //this is backwards from typical polar coodinates
            angle-=rotationalSpeed; //because positive y is downward.
        if(turningRight) //Because of that, adding to the angle is
            angle+=rotationalSpeed; //rotating clockwise (to the right)
        if(angle>(2*Math.PI)) //Keep angle within bounds of 0 to 2*PI
            angle-=(2*Math.PI);
        else if(angle<0)
            angle+=(2*Math.PI);
        if(accelerating){ //adds accel to velocity in direction pointed
            //calculates components of accel and adds them to velocity
            xVelocity+=acceleration*Math.cos(angle);
            yVelocity+=acceleration*Math.sin(angle);
        }
        x+=xVelocity; //move the ship by adding velocity to position
```

```
		y+=yVelocity;
		xVelocity*=velocityDecay; //slows ship down by percentages
		yVelocity*=velocityDecay; //(velDecay should be between 0 and 1)
		if(x<0) //wrap the ship around to the opposite side of the screen
			x+=scrnWidth; //when it goes out of the screen's bounds
		else if(x>scrnWidth)
			x-=scrnWidth;
		if(y<0)
			y+=scrnHeight;
		else if(y>scrnHeight)
			y-=scrnHeight;
	}

	public void setAccelerating(boolean accelerating){
		this.accelerating=accelerating;
	}

	public void setTurningLeft(boolean turningLeft){
		this.turningLeft=turningLeft;
	}

	public void setTurningRight(boolean turningRight){
		this.turningRight=turningRight;
	}

	public double getX(){
		return x;
	}

	public double getY(){
		return y;
	}

	public double getRadius(){
		return radius;
	}

	public void setActive(boolean active){
		this.active=active;
	}

	public boolean isActive(){
		return active;
	}

	public boolean canShoot(){
		if(shotDelayLeft>0) //checks to see if the ship is ready to
			return false; //shoot again yet or needs to wait longer
		else
			return true;
	}

	public Shot shoot(){
		shotDelayLeft=shotDelay; //set delay till next shot can be fired
		//a life of 40 makes the shot travel about the width of the
		//screen before disappearing
		return new Shot(x,y,angle,xVelocity,yVelocity,40);
```

```
        }
}
```

Appendix C: The Completed Shot Class

In this appendix, Table C.1 describes each variable and method in the Shot class and Code C.2 provides a complete listing of the finished code for the Shot class. The variables and methods in the table are in the same order as they appear in the code and are grouped together when they are closely related.

Variables	Description
final double shotSpeed	Constant storing the speed at which the shots are fired relative to the motion of the ship.
double x, y, xVelocity, yVelocity	Store the coordinates and x/y components of the velocity of the shot.
int lifeLeft	Stores the number of frames this shot has left before disappearing if it doesn't hit an asteroid first.
Methods	**Description**
Shot()	Constructor. Initializes all necessary variables based on parameters passed in.
void move()	Moves the shot according to its x/y velocity. Decrements *lifeLeft*.
void draw()	Draws the shot using the Graphics object passed in.
double getX() double getY()	Return the x and y coordinates of the shot, respectively. Used by the Asteroid class in collision detection.
int getLifeLeft()	Returns the number of frames this shot has left before disappearing. Used in AsteroidsGame to determine when to delete the shot.

Table C.1: Variables and methods of the Shot class.

Code C.2 – Shot.java: The finished code for the Shot class.

```
import java.awt.*;

public class Shot {
	final double shotSpeed=12;
	double x,y,xVelocity,yVelocity;
	int lifeLeft;

	public Shot(double x, double y, double angle, double shipXVel,
			double shipYVel, int lifeLeft){
		this.x=x;
		this.y=y;
		// add the velocity of the ship to the velocity the shot velocity
		// (so the shot's velocity is relative to the ship's)
		xVelocity=shotSpeed*Math.cos(angle)+shipXVel;
		yVelocity=shotSpeed*Math.sin(angle)+shipYVel;
		// the number of frames the shot will last for before
```

```
		// disappearing if it doesn't hit anything
		this.lifeLeft=lifeLeft;
	}

	public void move(int scrnWidth, int scrnHeight){
		lifeLeft--; // used to make shot disappear if it goes too long
				// without hitting anything
		x+=xVelocity; // move the shot
		y+=yVelocity;
		if(x<0) // wrap the shot around to the opposite side of the
			x+=scrnWidth; // screen if needed
		else if(x>scrnWidth)
			x-=scrnWidth;
		if(y<0)
			y+=scrnHeight;
		else if(y>scrnHeight)
			y-=scrnHeight;
	}

	public void draw(Graphics g){
		g.setColor(Color.yellow); //set shot color
		//draw circle of radius 3 centered at the closest point
		//with integer coordinates (.5 added to x-1 and y-1 for rounding)
		g.fillOval((int)(x-.5), (int)(y-.5), 3, 3);
	}

	public double getX(){
		return x;
	}

	public double getY(){
		return y;
	}

	public int getLifeLeft(){
		return lifeLeft;
	}
}
```

Appendix D: The Completed Asteroid Class

In this appendix, Table D.1 describes each variable and method in the Asteroid class and Code D.2 provides a complete listing of the finished code for the Asteroid class. The variables and methods in the table are in the same order as they appear in the code and are grouped together when they are closely related.

Variables	Description
double x, y, xVelocity, yVelocity, radius	Store the coordinates, x/y components of velocity, and radius of the asteroid.
int hitsLeft	Stores the number of times this asteroid can be broken up before the pieces disappear.
int numSplit	Stores the number of smaller asteroids this asteroid will break into when it is shot.
Methods	**Description**
Asteroid()	Constructor. Initializes variables according to parameters passed in.
void move()	Moves the asteroid according to x/y velocity.
void draw()	Draws the asteroid using the Graphics object passed in.
Asteroid createSplitAsteroid()	Creates and returns a smaller asteroid. Used when this asteroid gets shot and is being broken into smaller asteroids.
boolean shipCollision()	Returns true if this asteroid is colliding with the ship passed in.
boolean shotCollision()	Returns true if this asteroid is colliding with the shot passed in.
int getHitsLeft()	Returns *hitsLeft*. Used by AsteroidsGame to determine if this asteroid should be deleted or broken up into smaller ones.
int getNumSplit()	Returns *numSplit*, the number of smaller asteroids this one will break up into when shot.

Table D.1: Variables and methods of the Asteroid class.

Code D.2 – Asteroid.java: The finished code for the Asteroids class.

```
import java.awt.*;

public class Asteroid {
	double x, y, xVelocity, yVelocity, radius;
	int hitsLeft, numSplit;

	public Asteroid(double x,double y,double radius,double minVelocity,
			double maxVelocity,int hitsLeft,int numSplit){
		this.x=x;
		this.y=y;
		this.radius=radius;
		this.hitsLeft=hitsLeft; //number of shots left to destroy it
```

```
        this.numSplit=numSplit; //number of smaller asteroids it
                                              //breaks up into when shot
        //calculates a random direction and a random
        //velocity between minVelocity and maxVelocity
        double vel=minVelocity + Math.random()*(maxVelocity-minVelocity),
              dir=2*Math.PI*Math.random(); // random direction
        xVelocity=vel*Math.cos(dir);
        yVelocity=vel*Math.sin(dir);
}

public void move(int scrnWidth, int scrnHeight){
        x+=xVelocity; //move the asteroid
        y+=yVelocity;
        //wrap around code allowing the asteroid to go off the screen
        //to a distance equal to its radius before entering on the
        //other side.  Otherwise, it would go halfway off the sceen,
        //then disappear and reappear halfway on the other side
        //of the screen.
        if(x<0-radius)
              x+=scrnWidth+2*radius;
        else if(x>scrnWidth+radius)
              x-=scrnWidth+2*radius;
        if(y<0-radius)
              y+=scrnHeight+2*radius;
        else if(y>scrnHeight+radius)
              y-=scrnHeight+2*radius;
}

public void draw(Graphics g){
        g.setColor(Color.gray); // set color for the asteroid
        // draw the asteroid centered at (x,y)
        g.fillOval((int)(x-radius+.5),(int)(y-radius+.5),
                          (int)(2*radius),(int)(2*radius));
}

public Asteroid createSplitAsteroid(double minVelocity,
              double maxVelocity){
        //when this asteroid gets hit by a shot, this method is called
        //numSplit times by AsteroidsGame to create numSplit smaller
        //asteroids.  Dividing the radius by sqrt(numSplit) makes the
        //sum of the areas taken up by the smaller asteroids equal to
        //the area of this asteroid.  Each smaller asteroid has one
        //less hit left before being completely destroyed.
        return new Asteroid(x,y,radius/Math.sqrt(numSplit),
                    minVelocity,maxVelocity,hitsLeft-1,numSplit);
}

public boolean shipCollision(Ship ship){
        // Use the distance formula to check if the ship is touching this
        // asteroid: Distance^2 = (x1-x2)^2 + (y1-y2)^2    ("^" denotes
        // exponents).  If the sum of the radii is greater than the
        // distance between the center of the ship and asteroid, they are
        // touching.
        // if (shipRadius + asteroidRadius)^2 > (x1-x2)^2 + (y1-y2)^2,
        // then they have collided.
        // It does not check for collisions if the ship is not active
        // (player is waiting to start a new life or the game is paused).
```

```
            if(Math.pow(radius+ship.getRadius(),2) >
                        Math.pow(ship.getX()-x,2) + Math.pow(ship.getY()-y,2)
                        && ship.isActive())
                  return true;
            return false;
      }

      public boolean shotCollision(Shot shot){
            // Same idea as shipCollision, but using shotRadius = 0
            if(Math.pow(radius,2) > Math.pow(shot.getX()-x,2)+
                        Math.pow(shot.getY()-y,2))
                  return true;
            return false;
      }

      public int getHitsLeft(){
            //used by AsteroidsGame to determine whether the asteroid should
            //be split up into smaller asteroids or destroyed completely.
            return hitsLeft;
      }

      public int getNumSplit(){
            return numSplit;
      }
}
```

Appendix E: A Game Framework

Code E.1 in this appendix contains the completed code for the framework created in Chapter 3. The framework contains basic functionality that is needed for almost any game that is programmed as an applet. Places where code specific to the game being created would need to be added are indicated by comments in all caps.

Code E.1 – GameFramework.java: The completed framework from Chapter 3.

```
//A generic framework for a game implemented as a Java Applet

import java.applet.*; //needed to create an applet
import java.awt.*; //needed for graphics
import java.awt.event.*; //needed for event handling

public class GameFramework extends Applet implements Runnable, KeyListener{

	Thread thread; // used to run the game

	Dimension dim; // these are used for double buffering
	Image img; // the back buffer
	Graphics g; // used to draw on the back buffer

	// used to regulate the speed of the game
	long endTime, startTime, framePeriod;

	/**
	 * This method is called when the applet is first created.
	 */
	public void init(){
		resize(500,500); //can be set to any dimension desired
		// INITIALIZE ANY OF YOUR OWN VARIABLES HERE
		endTime=0;
		startTime=0;
		framePeriod=25; //may be adjusted to change the speed that
		                //the game runs.
		addKeyListener(this); //tell the class to listen for KeyEvents
		dim=getSize();
		img=createImage(dim.width, dim.height);//create back buffer
		g=img.getGraphics(); //create Graphics obj to draw on back buffer
		thread=new Thread(this); //create the thread that runs game
		thread.start(); //start the thread
	}

	/**
	 * This method paints the graphics of the applet.
	 * @param gfx - the applet's graphics object
	 */
	public void paint(Graphics gfx){
		g.setColor(Color.black); //set color to clear the screen with
		g.fillRect(0,0,500,500); //clear the screen
```

```
        // CODE TO DRAW GRAPHICS HERE
        gfx.drawImage(img,0,0,this); //copys back buffer onto the screen
    }

    /**
     * This is the method called by repaint() and makes a call to
     * paint() without clearing the screen.
     * The original method, Applet.update(), clears the screen
     * with a white rectangle and then calls paint.
     * Clearing the screen causes flickering in games, so we have
     * overriden update() to call paint without clearing.
     * We can do this because we will make paint() redraw the entire
     * screen, so clearing it first would be pointless.
     * @param gfx - the applet's graphics object
     */
    public void update(Graphics gfx){
        paint(gfx);
    }

    /**
     * This method contains the code to run the game.
     * It is executed by a thread.
     */
    public void run(){
        for(;;){
            startTime=System.currentTimeMillis();
            // CODE TO EXECUTE A FRAME OF THE GAME HERE
            repaint();
            try{ //regulate the speed of the game
                endTime=System.currentTimeMillis();
                if(framePeriod-(endTime-startTime)>0)
                    Thread.sleep(framePeriod-(endTime-startTime));
            }catch(InterruptedException e){
            }
        }
    }

    /**
     * Responds to any keys being pressed on the keyboard
     * @param e - contains information on the event (what key was
     * pressed, etc)
     */
    public void keyPressed(KeyEvent e){
        if(e.getKeyCode()==KeyEvent.VK_UP){
            // CODE TO RESPOND TO UP KEY BEING PRESSED
        }else if(e.getKeyCode()==KeyEvent.VK_DOWN){
            // CODE TO RESPOND TO DOWN KEY BEING PRESSED
        }else if( ...
        ... // DITTO FOR ALL CONTROLS IN THE GAME
    }

    /**
     * Responds to any keys being released on the keyboard
     * @param e - contains information on the event (what key was
     * released, etc)
     */
    public void keyReleased(KeyEvent e){
```

```
            if(e.getKeyCode()==KeyEvent.VK_UP){
                  // CODE TO RESPOND TO UP KEY BEING RELEASED
            }else if(e.getKeyCode()==KeyEvent.VK_DOWN){
                  // CODE TO RESPOND TO DOWN KEY BEING RELEASED
            }else if( ...
            ... // DITTO FOR ALL CONTROLS IN THE GAME
      }

      /**
       * This method doesn't usually need to do anything for simple games.
       * It might be used if the game involved the user typing in text.
       */
      public void keyTyped(KeyEvent e){
      }
}
```

Glossary

Applet: A small Java program that is designed to be embedded in a web page.

Application Programming Interface (API): The set of functions that give a programmer access to the operating system and other services (Application Program Interface).

Array: A series (or list) of variables accessed by the name of the array and an index value (i.e. arrayName[index]).

Call: To execute a function. For instance, making a call to the draw() function means telling the computer to execute the code in the draw() function.

Class: The basic structure in the Java programming language. A class may contain variables and methods and typically represents an object of some kind.

Collision Detection: The code (usually in a game) that determines when one object collides with another.

Comment: Text inside source code that is ignored when the file is compiled. Comments are supplemental explanations used to make code easier to read and understand. They do not in any way affect how the program runs.

Compile: To translate Java code into bytecode that can be read and run by the **Java Virtual Machine**. Compiling is done using the Java compiler, javac.exe. It generates .class file with the same name as the .java file containing the code.

Constant: A value that never changes.

Constructor: A Method that initializes an object immediately when a new instance of that object is first created.

Control Flow Statement: Statements that alter the path followed by the computer as it runs a program. These include if/else, while, for, switch, etc.

Event: Something that happens outside of a program that the program might want to respond to. Most commonly, events are forms of user input, such as a keystroke or mouse click.

Event Handling: The code that causes a program to respond appropriately to **events**.

Extends: The Java keyword used to indicate that one class inherits from another. This means that all the variables and methods in the class being inherited from will automatically be included in the new class.

Distance Formula: The mathematical formula used for determining the distance between two points in a Cartesian coordinate plane. The formula is as follows:
$d = \text{sqrt}((x1-x2)^2 + (y1-y2)^2)$ where d is the distance, (x1,y1) and (x2,y2) are the two points, and sqrt indicates a square root.

Double Buffering: Drawing on a back buffer, then copying it to the actual screen instead of just drawing directly to the screen. This makes graphics smoother by reducing flickering.

Frame: One of many consecutive still images used in animation. Movies are animated by displaying thousands of prerecorded frames in rapid succession. Games are animated in the same way, but without prerecorded frames. Instead, the computer draws each frame on the fly according to what is happening in the game.

Function: See **Method** (they are called functions in other languages, but in Java they are methods).

Hypertext Markup Language (HTML): The most commonly used scripting language for creating webpages.

Implements: A Java keyword used to specify that a class will have the functionality defined by

the **interface** being implemented. For instance, if a class implements the KeyListener interface, then that class must be able to act as a KeyListener. In other words, the class must have all the methods that a KeyListener would need to have.

Initialize: To set a newly created variable to its first value.

Instance: An occurrence of an object (or class). A class defines a type and an instance is an actual object of that type.

Integrated Development Environment (IDE): A text editor that is linked into a programming language. Typically they feature **syntax highlighting**, shortcuts (or buttons) for **compiling**, and debugging tools. Some Java IDEs are Eclipse, NetBeans, and JBuilder.

Interface: A set of related methods that define a certain functionality. In the actual interface definition, the method names alone are specified without any accompanying body of code. Any class that **implements** the interface is required to include definitions of all of these methods (The Java Tutorials).

Iteration: One repetition of a repeating process. One iteration of a loop in programming means running the code inside the loop once.

Java: A programming language developed by Sun Microsystems. It is designed specifically to be able to "create networking applications that will run on any computer," regardless of its operating system (Java).

Java Application Programming Interface (Java API): The standard packages and classes included in the Java programming language. Every class, interface, method, variable, etc. in the Java API is documented at http://java.sun.com/javase/6/docs/api/.

Java Software Development Kit (JDK): The programs/tools necessary to both compile and run Java code. Includes the JRE plus tools to develop your own Java programs.

Java Runtime Environment (JRE): The programs/tools necessary to run Java programs.

Java Virtual Machine (JVM): The program (called an "interpreter") that runs Java bytecode. Code is translated into by tecode by the Java compiler and the bytecode is then executed by the JVM.

Long: A data type exactly the same as an integer, but with twice the capacity.

Method: A reusable section of Java code. Also known as a function in other programming languages.

Override: To create a new method that replaces one that was inherited from another class.

Package: A folder containing a group of related Java classes. Whole packages can be imported to gain access to all the classes within them.

Primitive Type: A variable that is not a class. Primitive types include int, char, double, float, boolean, long, etc.

Software Development Kit (SDK): The programs/tools necessary to write, compile, and run your own code in a programming language.

(Java) Standard Library: All the packages and classes in the **Java API**.

String: A class in the Java standard library that is used to store and manipulate text.

Syntax Highlighting: Changing the color and/or formatting of text to aid the user in seeing and understanding the syntax of a programming language. The different colors and heaviness of font used in the code sections of this manual are an example of syntax highlighting.

Text Editor: Any program that allows a user to create and edit plain text files (.txt). Notepad is the standard Windows text editor. Some text editors have been specifically designed to be used in programming. These usually include **syntax highlighting** and other features

useful in programming. Emacs and gVim are common examples of this type of text editor.

Thread: A process that runs continuously and concurrently with all the other processes running on a computer. In Java, the Thread and the Runnable interface are used to create threads.

Bibliography

Application Program Interface. Dictionary.com. *The Free On-line Dictionary of Computing*. Denis Howe. http://dictionary.reference.com/browse/Application Program Interface (accessed: March 27, 2007).

Brackeen, David. 2003. *Developing Games in Java™*. New Riders. http://proquest.safaribooksonline.com.erl.lib.byu.edu/1592730051 (accessed: April 5, 2007).

Davison, Andrew. 2005. *Killer Game Programming in Java.* O'Reilly. http://proquest.safaribooksonline.com.erl.lib.byu.edu/0596007302 (accessed: March 27, 2007).

Horstmann, Cay S., and Gary Cornell. 2004. *Core Java™ 2 Volume I – Fundamentals, Seventh Edition.* Prentice Hall. http://proquest.safaribooksonline.com.erl.lib.byu.edu/0131482025 (accessed: April 5, 2007).

Java. Reference.com. *Crystal Reference Encyclopedia*. Crystal Reference Systems Limited. http://www.reference.com/browse/crystal/37341 (accessed: February 08, 2007).

The Java API. 2006. Java™ Platform, Standard Edition 6 API Specification. http://java.sun.com/javase/6/docs/api/ (accessed: April 5, 2007).

The Java Tutorials. 2006. What is an Interface? http://java.sun.com/docs/books/tutorial/java/concepts/interface.html (accessed: April 5, 2007).

Lannon, John M. 2006. *Technical Communication*. Longman.

Wikipedia contributors, "Asteroids (arcade game)," *Wikipedia, The Free Encyclopedia,* http://en.wikipedia.org/w/index.php?title=Asteroids_%28arcade_game%29&oldid=105145229 (accessed February 9, 2007).