Using the ACM Java Libraries in CS 1

Andrew Mertz, William Slough, and Nancy Van Cleave
aemertz@eiu.edu waslough@eiu.edu nkvancleave@eiu.edu
Department of Mathematics and Computer Science
Eastern Illinois University
Charleston, IL 61920

Abstract

The Java language is commonly used in CS 1 courses, though its complexity and evolving nature introduce difficulties to instructors and students alike. In recognition of these challenges, the ACM established the Java Task Force to study the problem. The work of this committee resulted in a collection of Java-based resources which simplifies the teaching and learning of computer science. In this paper, we describe some of these resources, discuss programming assignments which utilize their features, and relate our positive experiences.

1 Introduction

At our university, we teach a CS 1 course which serves four primary and rather distinct audiences. In addition to the expected computer science majors, the course is required for pre-engineering, mathematics, and mathematics education students. For many of these students, CS 1 is the only computer science course required for their major. This presents some unique challenges—we want to provide a rigorous course while at the same time reaching out to those with limited interest. One approach we have employed to engage students is the early and frequent use of graphics, along with a selection of problems with mathematical content. With an appropriate graphics library, it is possible to write programs with interesting graphical results relatively early in the semester.

Historically, we have used Pascal and C++ for this course, but recently chose to switch to Java to stay in step with the language used for the Advanced Placement Computer Science exam. Since we are interested in keeping the focus on fundamental concepts, the complexities of modern programming languages such as C++ and Java can be impediments. In fact, we were somewhat late adopters of Java, realizing its complexity and rapid pace of change would introduce difficulties. Recognizing these concerns, the ACM Education Board formed the *ACM Java Task Force* (JTF) in 2004 with the following charter [6]:

> To review the Java language, APIs, and tools from the perspective of introductory computing education and to develop a stable collection of pedagogical resources that will make it easier to teach Java to first-year computing students without having those students overwhelmed by its complexity.

In 2006, the JTF released the ACM Java libraries, a collection of Java packages intended to simplify the teaching and learning of Java in CS 1. The accompanying website [3] documents these packages and provides rationale and a wide variety of sample Java programs. In 2007, Eric Roberts' textbook[7] for CS 1 appeared, which integrated the ACM Java libraries. In the Fall 2007 semester, we adopted this textbook for our introductory course.

We have found the ACM Java libraries and the pedagogy described in the Roberts textbook to be a good match for the needs of our course; we expect others would find similar benefits from this approach. The ACM Java libraries include an object-oriented model for programs, simple methods for I/O, and accessible graphics capabilities. Additionally, programs developed with these libraries can run as stand-alone applications or as applets embedded within web pages. In the remainder of this paper, we provide a glimpse of how these libraries can be used in a CS 1 course.

2 Rudiments of the ACM Java Libraries

The ACM Java libraries consist of five packages:

`acm.graphics`	Supports simple, object-oriented graphics
`acm.gui`	Classes which simplify the creation of interactive programs
`acm.io`	Contains two classes to assist with input and output operations
`acm.program`	Classes providing an object-oriented model for programs
`acm.util`	Miscellaneous utility classes

While the ACM Java libraries are fairly comprehensive, students can achieve impressive results after being introduced to just a few classes from the `graphics` and `program` packages. In this section we focus on these packages.

Students implement their programs by creating classes that extend the appropriate program class. Each class provides a different way of handling input and output:

`Program`	Input and output using the system console
`ConsoleProgram`	Uses a program frame with an interactive console (`acm.io.Console`)
`DialogProgram`	Input and output is done via dialog boxes
`GraphicsProgram`	Fills the program frame with a `GCanvas` for drawing graphics objects

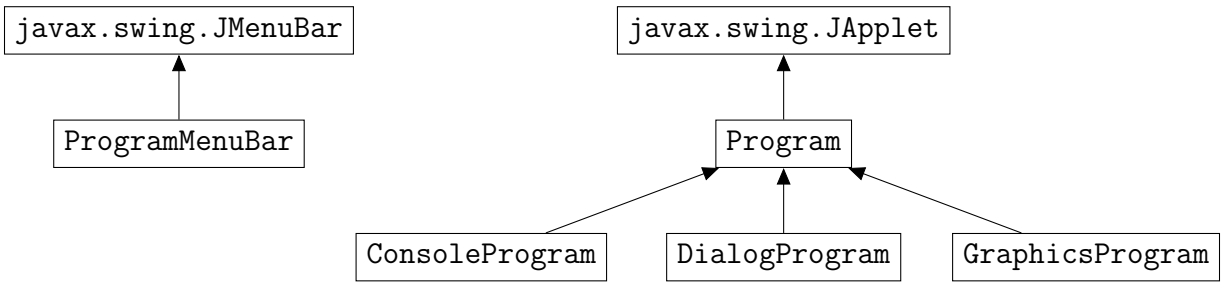
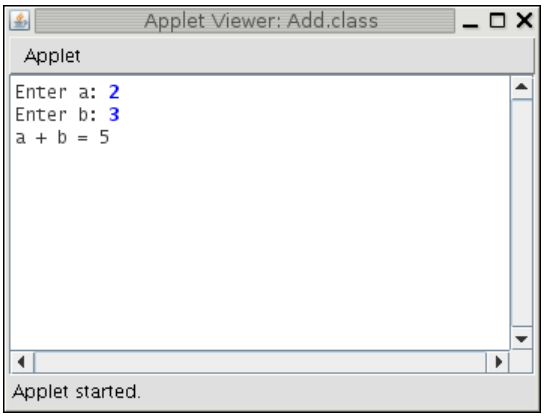


Figure 1: Class diagram for `acm.program`.

One advantage of this model is that students do not need to know the details of dialog boxes, `GCanvas` objects, output streams and the like, since the program classes take care

```
public class Add extends ConsoleProgram
{
  public void run() {
    int a = readInt("Enter a: ");
    int b = readInt("Enter b: ");
    println("a + b = " + (a + b));
  }
}
```

```
public class Add extends DialogProgram
{
  public void run() {
    int a = readInt("Enter a: ");
    int b = readInt("Enter b: ");
    println("a + b = " + (a + b));
  }
}
```



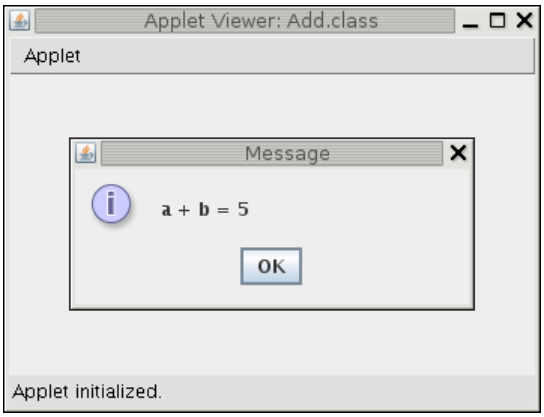


Figure 2: ACM console and dialog program examples for adding two integers.

of the details. As they learn more about these classes they can add new functionality by manipulating the underlying objects or adding to them—for example, to create a graphics program with multiple drawing surfaces.

Additionally, the `Program` class unifies applets and applications. As Figure 1 shows, all programs are applets and thus can be embedded in web pages. Because of this, the principal entry point for programs is the `run` method, not `main`. This has the pedagogical advantage of not being encumbered by the syntax required of `main`, such as the `static` modifier and `String` array parameter. The examples given in Figure 2 show the only modification required to switch from using a console to using dialog boxes is changing what class our program extends. The print and read methods of the various program classes provide a simple and consistent syntax for I/O.

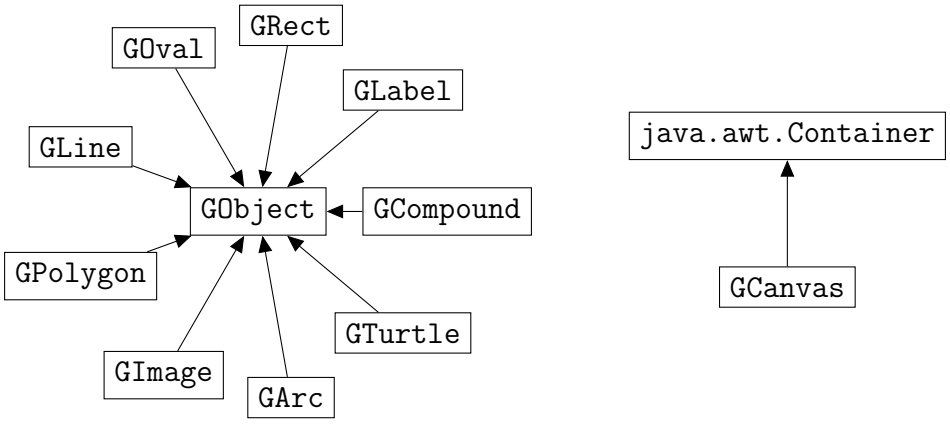


Figure 3: Partial class diagram for `acm.graphics`.

```
import acm.graphics.*;
import acm.program.*;

public class ShowImage extends GraphicsProgram {
  public void run() {
    add(new GImage("JuliaSet.jpg"));
    GLabel label = new GLabel("The Julia Set");
    double x = (getWidth() - label.getWidth())/2;
    double y = getHeight() - label.getDescent();
    add(label, x, y);
  }
}
```

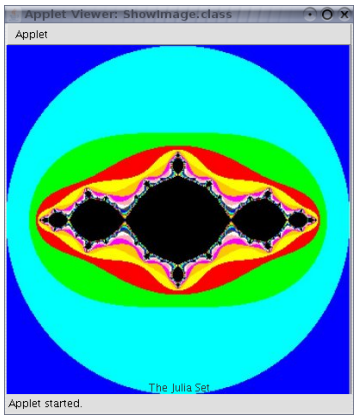


Figure 4: Example graphics program and its output.

The `acm.graphics` package uses a collage model in which an image is created by adding various objects to a `GCanvas`. This is similar to a felt board that serves as a backdrop for colored shapes that stick to the felt surface. Note that newer objects can obscure those added earlier. All distances and coordinates in the graphics library are measured in pixels and specified as floating point values. Figure 3 shows many of the classes available for drawing graphics and Figure 4 shows a small graphics program that displays an image and a caption.

3 Laboratory Exercises

In this section, a sampling of laboratory exercises is provided to underscore the role played by the ACM Java libraries in our course. The libraries provide significant simplifications to the Java experience and allow for a number of interesting programs early in the course.

One aspect of the libraries which is especially attractive is the ease with which a Java program can be turned into an applet. We chose to make this a central feature of the course. Everyone enrolled at our university is provided with a personal web site, and our students are required to provide links on their site to applications they implement in the course. This gives them a feeling of ownership, while at the same time providing a convenient mechanism to display their work (and extensions) to friends and family.

3.1 Introductory Lab

In the first week's laboratory session, students learn the mechanical skills that will serve them for the entire semester. They are exposed to the NetBeans IDE [4], set up and understand how to upload their web sites, modify a few sample Java programs and are introduced to three of the program classes provided by the libraries: `ConsoleProgram`, `DialogProgram`, and `GraphicsProgram` In the interest of "truth in advertising," we need to point out that we experienced stability issues with `ConsoleProgram` applets embedded in a web page, viewed under Linux. We have not been able to identify the source of the problem—Java plug-in, browser, and/or ACM libraries. As a workaround, we avoided assigning problems which would require running as a console program.

Although there are many diverse mechanical skills to digest in this first laboratory experience, students receive their first exposure to writing Java programs which produce distinctly

different types of behaviors, yet share a common structure. They also obtain an intuition for the coordinate system used in graphics programs using an experimental approach to placing text on the graphics "canvas." By adjusting x and y values, students are able to gain a sense of the coordinate system with very little instruction. This initial exposure to the graphics system provides a foundation for the laboratory experiences to follow.

For this first laboratory experience, we follow Roberts' lead, by asking students to experiment with the now-famous "hello, world" program, updated for the Java language and specialized to the ACM Java libraries. Figure 5 shows the code details and the results. Students provide HTML for their web sites and applets by following templates we provide. Every applet produced throughout the course has a similar structure, so a first example sets the pattern for everything to follow.

```
import acm.graphics.*;
import acm.program.*;

public class HelloProgram extends GraphicsProgram {
  public void run() {
    add(new GLabel("Hello, world!", 100, 75));
    add(new GLabel("My Name is Julie.", 100, 95));
    add(new GLabel("I'm from Chicago.", 100, 115));
  }
}
```

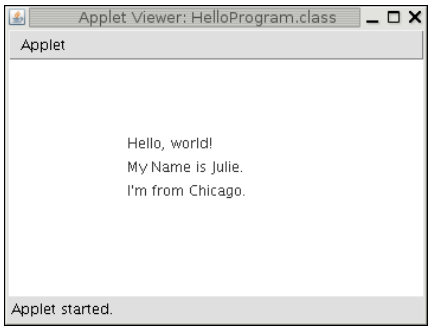


Figure 5: "Hello, world" Java program and output.

3.2 Curve Stitch

Simple curve stitch designs [2] (sometimes known as "string art"), a type of recreational mathematics attributed to Mary Everest Boole, can be used to reinforce looping constructs within a graphical context. The design illustrated in Figure 6 involves two perpendicular line segments subdivided into s pieces of equal length. Generating diagrams of this type with a Java program reinforces the ideas of parameterization (s and the size of the square, in this case) and provides visual clues for debugging programs with loops. For example, "off by one" errors common among beginners are revealed in obvious ways in the graphical output. Since these diagrams can be produced with `GLine` objects and a rudimentary knowledge of graphics and looping, it can be assigned early in the semester.

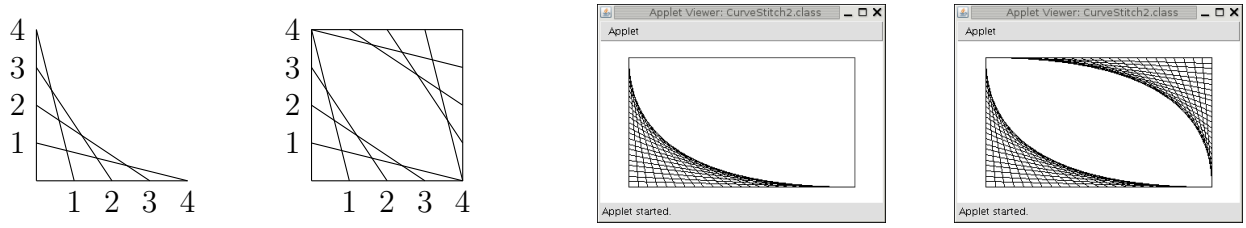


Figure 6: From left to right, a simple curve-stitch design, the same design combined with a mirror image, and two sample program executions.

3.3 Bouncing Ball

An exercise from the Roberts textbook involving the animation of a bouncing ball is nicely supported by the ACM Java libraries and is of interest to a number of students. Like

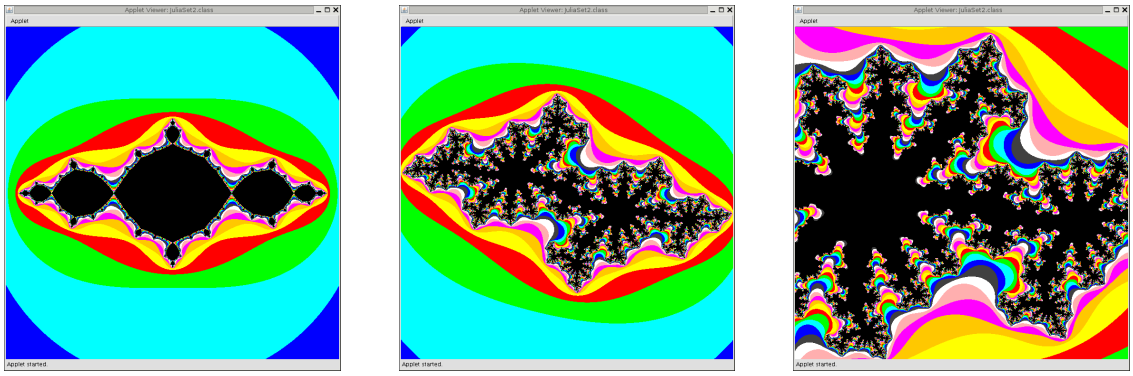


Figure 7: Three Julia set results.

the curve-stitch program, it can be completed early in the semester and has strong visual appeal. Exercises of this type allow students to make connections between the static nature of program code and its dynamic effect when executed. This exercise can be accomplished with knowledge of looping and conditional statements and the `GOval` class. The animation is particularly accessible since the ACM Java libraries include simple mechanisms to move graphical objects and to introduce program delays using the `move` and `pause` methods.

This problem provides many interesting possibilities for later assignments. One extension is to simulate the trajectory of a billiard ball on a pool table, complete with pockets. Variations of this exercise could introduce a differently shaped table or use multiple balls, for example. A more ambitious project involving a "Breakout" arcade-style game is described in the Roberts text.

3.4 Julia Sets

By the midpoint of the semester, students have written several programs dealing with a grid of blocks in the graphics window—a lower triangle of blocks, a pyramid, a checkerboard, and a checkerboard with playing pieces. It is a small step from these examples to view the window as a grid of blocks to be colored according to some formula: for example, to display a Julia set [5].

One popular visualization of the Julia set employs the quadratic function $f(z) = z^2 + c$, defined for complex values z and a fixed, complex parameter c. The function f is iterated for each point z in a region of the complex plane. Colors are assigned to each point according to the number of iterations needed to "escape" a circle of fixed radius. If the iterated function for a point does not leave this circle within a prescribed number of iterations, it is considered to be in the Julia set and is colored black.

The main idea in this problem is to map each block in the graphics window to a corresponding point in the world region that we are representing, then iteratively apply f in order to determine the color of that block. With the help of a skeleton program that provided several methods, students were able to generate graphic images of Julia sets. By changing a few parameters, students can discover new images and zoom in to ever decreasing portions of images. Figure 7 shows some results. The popularity of this assignment extended to students enrolled in more advanced courses, who decided to implement it just for fun.

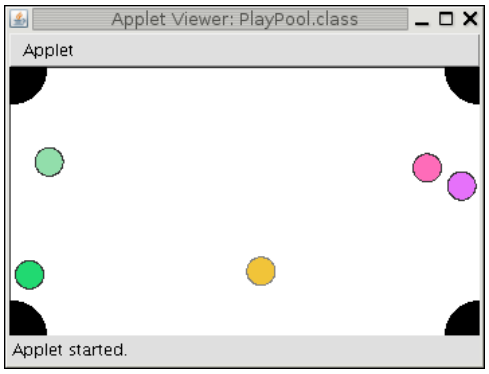


Figure 8: One snapshot of a pool table animation.

3.5 Pool Table

As mentioned previously, one extension of the bouncing ball exercise is to simulate the trajectory of a billiard ball on a pool table. This pool table project is a variant which formally introduces students to derived classes and Java packages. It also seeks to more firmly establish in students' minds the difference between static and non-static methods.

Although students begin extending classes from the very beginning of our course, much of the power of inheritance is not apparent to them. This project was designed to be created incrementally over several phases. The first phase is used to reinforce the *is-a* relationship and the specialization of derived classes. Students extend the `GRect` rectangle class to squares, squares to smart squares which can determine their relation to the window boundaries, and finally, smart squares to moving squares which can be animated and stay within the window. Skeletons for these classes and their associated testing programs are given, along with enough of the code to provide examples for implementing the rest.

Phase two is used to reinforce class syntax and structure. Students rework the same exercise from scratch, but create circles, smart circles, and moving circles instead of squares. Although the modifications needed to switch from squares to circles are minor, this provides a chance to review what was learned.

The last phase of this exercise utilizes the moving circles as pool balls in the window, with the added capability of knowing when their centers fall within circles representing the pockets of the pool table. This requires additional methods in both the smart and moving circles. To complete this phase, students implement a program which uses both circles (the pockets) and moving circles (the pool balls), plus several static methods—for example, to create a moving circle with random trajectory, speed, and color. Although this assignment is challenging, many students become excited and are proud when they see their final program work.

3.6 Random Walks on a Lattice

A problem with a rich mathematical history is the random walk on a lattice [8]. Like the other problems we have described here, this has strong visual appeal, yet is readily accessible by beginning students. In our version of the problem an "ant" begins at the origin and wanders the graphics window, choosing a random compass direction after each step. The simulation terminates when the ant returns to the origin or is about to step off any edge of the window.

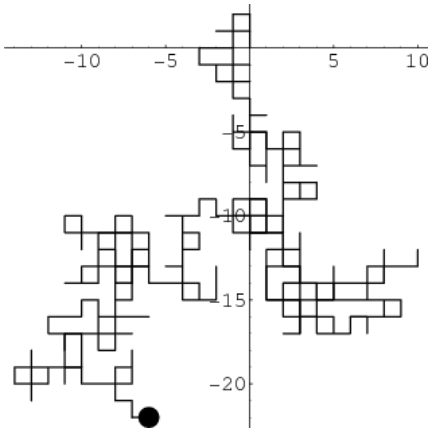


Figure 9: A random walk on a lattice, from [8].

A sample path is shown in Figure 9. Like the bouncing ball problem, random walks can be simulated with the introduction of a delay. This problem also introduces the idea of pseudo-random number generation, a feature supported by the `acm.util.RandomGenerator` class.

3.7 The Jailer's Problem

A famous problem [1], often used in programming contests, involves the warden of a jail making repeated passes over a sequence of cells, toggling the locks. Suppose there are N cells. On the first pass, the warden will toggle each of the locks, cells $1, 2, 3, \ldots, N$. Toggling a lock unlocks it if it is locked and locks it if it is unlocked. On the second pass, the warden toggles every other lock, starting with the second, cells $2, 4, 6, \ldots$ On the third pass, the warden toggles every third lock, starting with the third cell. The warden continues this pattern of toggling locks. On the final pass, the warden toggles every Nth lock, starting with cell N. Since there are only N cells, this last pass toggles just one lock, at cell N. The problem is to predict which cells will be unlocked at the end of this process.

In our version of the problem, we would like to obtain a graphical demonstration of the status of each lock, as a function of time. Figure 10 shows sample output for $N = 12$ cells, where the mth row is color-coded to show the status of the cells after the warden has performed m passes.

This exercise provides an opportunity to extend the `GCompound` class, creating a class that represents a numbered rectangle object. For our course, this was assigned in the latter part of the semester, after arrays had been discussed. In addition to its suitability as a programming exercise, this problem has interesting mathematical content involving factorization of integers.

4 Conclusions

The simplification of the Java language, made possible by the ACM Java libraries, has been of benefit to us and our students. Creating and maintaining web sites proved to be straightforward for students. The simplicity of graphics programming provides a good tool for teaching basic concepts in an interesting way, helping to keep students engaged. There is a wide variety of feasible assignments from which to choose, keeping the course fresh and interesting for the instructors, too.

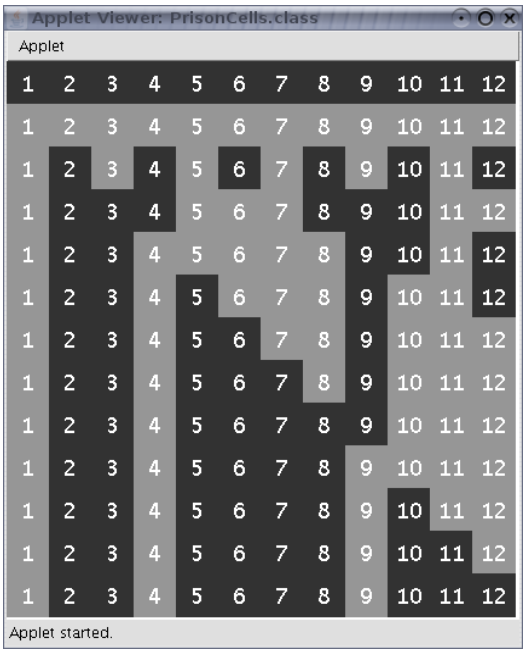


Figure 10: Sample output for the Jailer's problem, $N = 12$. Each cell is color-coded to indicate the status of the lock.

We concur with the motivations of the ACM Java Task Force and believe their work has resulted in valuable resources for those involved with a CS 1 course. The JTF website is extensive and provides a rationale document, many sample programs, tutorials and thorough documentation of the API in the form of JavaDoc.

References

[1] ACM Greater New York Programming Contest 2002. *The Drunk Jailer*. `http://acmicpc.cpsc.ucalgary.ca/contest060429/problems/Jailer.pdf`.

[2] Lionel Deimel. *Curve-stitch Designs website*. `http://deimel.org/rec_math/curve_stitch.htm`.

[3] Eric Roberts *et al. ACM Java Task Force website*, August 2006. `http://jtf.acm.org`.

[4] Sun Microsystems. *NetBeans IDE*. `http://www.netbeans.org`.

[5] Heinz-Otto Peitgen, Hartmut Jürgens, and Dietmar Saupe. *Chaos and Fractals*. Springer-Verlag, 2004.

[6] Eric Roberts. Resources to support the use of Java in introductory computer science. *SIGCSE Bulletin*, 36(1):233–234, 2004.

[7] Eric S. Roberts. *The Art & Science of Java: An Introduction to Computer Science*. Addison-Wesley, 2008.

[8] Eric W. Weisstein. *Random Walk–2 Dimensional. From MathWorld—A Wolfram Web Resource*. `http://mathworld.wolfram.com/RandomWalk2-Dimensional.html`.