The ACM Java Task Force

Project Rationale

Eric Roberts (chair), Stanford University, **eroberts@cs.stanford.edu**
Kim Bruce, Pomona College, **kim@cs.pomona.edu**
Robb Cutler, The Harker School, **robbc@harker.org**
James Cross, Auburn University, **cross@eng.auburn.edu**
Scott Grissom, Grand Valley State University, **grissom@gvsu.edu**
Karl Klee, Alfred State College, **kleekj@alfredstate.edu**
Susan Rodger, Duke University, **rodger@cs.duke.edu**
Fran Trees, Drew University, **fran@ftrees.com**
Ian Utting, University of Kent, **i.a.utting@kent.ac.uk**
Frank Yellin, Google, Inc., **fyellin@gmail.com**

Second Public Draft
(February 23, 2006)

Table of Contents

Chapter 1
Introduction

Since its introduction in 1995, Java has created quite a buzz in the computer science education community. If you look, for example, at the names of programming languages appearing in the titles of papers accepted for the SIGCSE annual symposium over the past eight years, references to Java outnumber those of all other programming languages combined. The trend toward the use of Java as the language for introductory computer science courses was evident as early as 1998 [Stevenson98] and has since gathered additional momentum, bolstered in part by the decision of the College Board to move the Advanced Placement Computer Science program to Java in the 2003-04 academic year [Astrachan00]. Some authors have suggested that Java has achieved a position of prominence within the academic community similar to that of Pascal in the 1980s, arguing that "we should shift our attention from the *whether Java* question to the *if Java, then how* question." [Wallace97]

Despite the fact that an increasing number of institutions are moving to adopt Java in their introductory curriculum, those institutions do not by any means report universal satisfaction with Java as a teaching language. The problems that arise in using Java at the introductory level were analyzed in more detail in a paper by Eric Roberts at SIGCSE 2004 [Roberts04a], which concluded that the observed difficulties in teaching Java are representative of a more general challenge facing computer science education. At its essence, that challenge arises from two self-reinforcing characteristics of modern programming languages that have a profoundly negative effect on pedagogy:

- *Complexity*. The number of details that students must master, particularly in the Application Programmer Interfaces (APIs) supplied along with the language itself, has grown much faster than the corresponding number of high-level concepts.
- *Instability*. The languages, APIs, and tools on which introductory computer science education depends are changing more rapidly than they have in the past.

To address the problems involved in using Java at the introductory level, the ACM Education Board initiated the ACM Java Task Force in the fall of 2003 with the following general charter:

> 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. [Roberts04b]

This report is the second public draft of the "Project Rationale" document produced by the Java Task Force; the first draft appeared on February 1, 2005. The purpose of the rationale document is to describe the design decisions we chose to make and the reasons that underlie those decisions. Chapter 2 outlines the principles adopted by the Task Force along with a summary of how we have carried out our work. Chapter 3 goes on to present a detailed analysis of the problems that face teachers who use Java at the introductory level.

The general taxonomy of problems from Chapter 3 revealed four significant challenges that instructors face teaching Java, along with several smaller ones. Those four challenges, shown here with their identifying labels from the taxonomy, are as follows:

L1. Static methods, including **`main`**
A1. Lack of a simple input mechanism
A2. Conceptual difficulty of the graphics model
A3. GUI components inappropriate for beginners

To provide some help in reducing the many difficulties associated with problem L1 (Static methods, including **`main`**), the Task Force has developed the **`acm.program`** package. This package defines a hierarchy of program classes that simplify the creation of programs and encourage students to work in an object-based environment rather than a static environment. The package, moreover, makes it possible to unify the concepts of applets and applications and provides support to application writers that is ordinarily not available. Because the description of the **`acm.program`** package depends on several of the other proposed packages, the detailed discussion of **`acm.program`** appears in Chapter 6, following the descriptions of the packages on which it depends.

Our approach to addressing problem A1 (Lack of a simple input mechanism) is to produce an **`acm.io`** package that offers two principal classes: an **`IOConsole`** class that offers simple input/output operations in the context of a traditional line-oriented console and an **`IODialog`** class that implements these operations using a popup dialog. These two share a common interface, which makes it easy to substitute one model for the other. The **`acm.io`** package is described in detail in Chapter 4.

Problem A2 (Conceptual difficulty of the graphics model) represents the area in which we invested the greatest part of our effort. We examined several models suggested by members of the computer science education community [Bruce04a, Parlante04a, SandersK04a] and used those as models for an **`acm.graphics`** package that combines the best features of each. The **`acm.graphics`** package is covered in Chapter 5.

As of the first draft of this rationale document, the Task Force had chosen not to propose a new package to address problem A3 (GUI components inappropriate for beginners). The feedback we received over the next few months, however, made it clear that we needed to try harder to come up with a solution. The community clearly identified the difficulty of building graphical user interfaces (GUIs) to be a significant barrier to the effective use of Java in introductory courses. The Task Force continued its deliberation and reviewed in more detail several of the proposed solution strategies [Lambert04a, SandersD04b, Rasala04a]. The product of that review was a new **`acm.gui`** package described in Chapter 7 along with a set of useful extensions to the **`Program`** class described in section 6.5.

The feedback we received on our initial proposal led to yet another change in the design of the various ACM packages. In the first public draft of this rationale document, the Task Force had coded the library packages so as to maintain compatibility with Java versions extending all the way back to release 1.1 of the Java Development Kit (JDK), which appeared in April 1997. The purpose of that decision was to ensure that programs built using the Task Force libraries could run in old browsers, including most of those currently deployed at commercial Internet-access sites. Even though that decision in no way limited the features that adopters might themselves choose to use when building applications on top of our packages, the choice made those packages *appear* obsolete and was all too often criticized on that basis. In part to forestall that criticism—and in part because the **`acm.gui`** package made it more difficult to continue with our original strategy—the new versions of the Task Force packages use JDK 1.4 as their foundation. The reasons behind that decision and the strategies we now use for maintaining compatibility with older browsers are described in Chapter 9, which also identifies the subset of Java classes that the Task Force recommends for use with the ACM packages.

Chapter 2
Principles and Methodology

The ACM Java Task Force has held five face-to-face meetings since its inception:

1. January 2004 (Washington, DC)
2. May 2004 (Pittsburgh, PA)
3. September 2004 (Palo Alto, CA)
4. December 2004 (Cambridge, MA)
5. June 2005 (Chicago, IL)

These meetings were supported by grants from the National Science Foundation, the ACM Education Board, and the SIGCSE Special Projects Fund. As is typical of this style of committee, however, much of the work of the Task Force took place between the meetings, which were used primarily to allow for the high-bandwidth, focused discussion that is difficult to achieve in online forums. In particular, these meetings allowed the members of the Task Force to reach consensus on several basic principles of operation.

2.1 Principles adopted by the Task Force

At our first two meetings of 2004, the Task Force was able to reach consensus on the following general principles:

1. *Design for an object-oriented approach.* In our work, the Java Task Force has been guided by the prospect of an "objects-first" approach to the introductory curriculum, as defined by in the joint ACM/IEEE-CS *Computing Curricula 2001* report [ACM01]. The tools and packages that we have designed therefore emphasize object-oriented design and adopt an object-oriented usage paradigm. At the same time, we have made every effort to ensure that our library packages and tools are usable with other pedagogical approaches.
2. *Adopt a minimalist strategy.* To avoid the proliferation of complex tools that do little to address the scale problems of Java, the Task Force has sought to minimize both the number and conceptual complexity of our packages. There are, for example, only four packages in the ACM collection. These packages, moreover, are specifically chosen to address the high-level problems outlined in Chapter 3. Although the temptation was often strong to provide additional functionality beyond what we needed to solve the reported problems, we chose to limit our concern to those areas that currently constitute stumbling blocks to the effective use of Java.
3. *Promote flexibility for adopters.* Introductory courses in computer science vary widely in philosophy, topic coverage, and approach. For those of us on the Java Task Force, such diversity is a very good thing. We are not interested in defining any sort of rigid standard, but are instead seeking to empower teachers and students by providing tools to extend their reach. To some extent, our design philosophy is aligned with the maxim formulated by the designers of the X Windows System, who sought to provide "mechanism, not policy." The packages in the ACM collection are intentionally general enough to support a wide variety of programming styles.
4. *Maintain conformance with the Java standard.* Throughout our design, the Task Force sought to use the standard Application Programmer Interfaces (APIs) provided by Sun Microsystems. The only occasions in which we have proposed alternative

APIs are those for which there was clear evidence that the existing facilities were not working well at the introductory level. We have also made the ACM packages separable to make sure that no one is required to adopt parts of the nonstandard packages that they regard as unnecessary. For each of the domains covered by the ACM packages—simplified input/output operations, object-based graphics, and the common applet/application framework—there is always an alternative approach that remains within the official Java standard.

5. *Retain compatibility with earlier releases of Java.* In order to be forward-looking and to ensure that our resources remain current for as long as possible, our baseline version of Java will be Standard Edition 5.0 of the Java 2 Platform (J2SE 5.0), which was released under the code name "Tiger" in September 2004 [Sun04]. We recognize, however, that many institutions will not be in a position to adopt Java 5.0 by the time at which our materials are released, and possibly not for some time to come. Moreover, many browsers—and particularly those installed at commercial Internet-access outlets like Kinko's and EasyInternetCafe which still operate in the JDK 1.1 world—run considerably older versions of the Java environment. To ensure that applications and applets built with our tools run on as many platforms as possible, we have made available a version of the **`acm.jar`** library that allows programs compiled using it to run with any version of the Java Development Kit from 1.1 on, assuming that they use only the classes identified as part of the JTF subset in Chapter 9.
6. *Support multiple environments.* The Task Force has not designed its packages with any particular programming environment in mind but has instead sought to ensure that our packages work in all major development environments.

2.2 Methodology for soliciting feedback

The Java Task Force sought community feedback at several points during the course of the project. The formation of the Task Force was announced officially at SIGCSE 2004, when we issued a call for input from the community on the nature of the problems that arise in teaching Java at the introductory level and on any successful strategies developed to address those problems. We used this feedback to guide the development of the Java Task Force packages, which we then described in the initial draft of this rationale document in February 2005. We then solicited feedback on the initial design in the form of a web-based discussion board announced at SIGCSE 2005. The subsections that follow describe the feedback we received in each of these phases of the project.

Soliciting reports of problems and proposed solutions

The Java Task Force adopted two strategies for identifying the problems that arise in teaching Java at the introductory level and for determining what solution strategies had been attempted:

1. From January to March, we conducted an extensive review of the computer science education literature to see what problems had been reported. The results of that review are presented in Chapter 3 along with an analysis of how those problems interrelate.
2. After publishing a draft of the problem taxonomy that came out of the literature review, we solicited input from the community as to what problems they had encountered in teaching Java and what solutions they had found. That call was issued as part of the Task Force presentation at SIGCSE 2004 and then followed up by the posting to SIGCSE-MEMBERS shown in Figure 2-1.

Figure 2-1. Java Task Force call for proposals

```
To: SIGCSE-MEMBERS@ACM.ORG
From: Eric Roberts <eroberts@cs.stanford.edu>
Subject: ACM Java Task Force announcement
Date: Wed, 17 Mar 2004 11:43:10 -0800
--------
Everyone,

The growth in popularity of the object-oriented paradigm and the
decision by the College Board to move the Advanced Placement Computer
Science program to Java have led an increasing number of universities,
colleges, and high schools to adopt Java as the programming language for
their introductory computer science course. At the same time, many of
those institutions report finding Java difficult to teach, partly
because of its significant detail complexity and its tendency to evolve
rapidly over time.

To help address these concerns, the ACM Education Board has recently
appointed a new task force with the following charter:

      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.

The formation of the ACM Java Task Force was officially announced at the
recent SIGCSE symposium in Norfolk, Virginia.  The background materials
and symposium presentations are available on the task force web site at

      http://www.sigcse.org/topics/javataskforce/

The Java Task Force held its initial meeting on January 31st in
Washington D.C. and identified a set of challenges that arise in
teaching Java based on a review of the literature and our experiences.
These challenges are summarized in a draft report entitled "Taxonomy of
Problems in Teaching Java," which is available on the web site.

At this point in the process, we are seeking contributions from the
computing education community in the following two categories:

  1. A description of any additional challenges you have beyond
     those identified in our taxonomy.

  2. A description of any solutions you have developed in response
     to these challenges.

We have established a process for submitting materials to the Task Force.
Please refer to the submission requirements page at

      http://www.sigcse.org/topics/javataskforce/Submissions.shtml

To give the community more time to respond, we have extended the submission
deadline until Friday, April 30, 2004.

                          -- The ACM Java Task Force
                             Eric Roberts (chair), Stanford University
                             Kim Bruce, Williams College
                             James Cross, Auburn University
                             Scott Grissom, Grand Valley State University
                             Karl Klee, Alfred State College
                             Susan Rodger, Duke University
                             Fran Trees, Drew University
                             Ian Utting, University of Kent
                             Frank Yellin, Sun Microsystems
```

Interestingly, those who responded to the Task Force's call for input did not identify any major problems beyond the ones already uncovered in the literature review. Instead, the 33 submissions we received focused on solution strategies. The most common themes for these submissions were graphics, the creation of graphical user interfaces (GUIs), and simple I/O. Although the Task Force did not adopt any of these solution strategies in the precise form given in the submission, the submissions we received provided extremely useful models that guided the Task Force toward the development of an integrated set of packages that could address these problems in a consistent way. Those packages are described in Chapters 4 through 8.

Soliciting feedback on the first release

The Java Task Force posted the first version of this rationale document and the associated library packages on February 1, 2005. After a couple of days to make sure that we had the feedback site ready to go, we sent a message to the SIGCSE-MEMBERS list announcing the availability of the new draft on February 3. That release message appears in Figure 2-2. The web forum opened the following day.

Discussion on the web forum was never particularly intense, although we received a modest but steady stream of feedback, with a total of 62 messages posted to the forum. The forum was organized by topic, and it is interesting to consider which topics attracted the most attention. The statistics for the responses posted on the forum appear in Figure 2-3, which shows both the number of separate topic threads and the total number of postings. The following topic areas received ten or more postings:

- **Chapter 6—The Program Hierarchy.** This chapter covers the **`Program`** class, which was introduced to simplify the creation of programs that could run as both applets and applications. This chapter, however, also included the discussion of why the Task Force had decided against adding special GUI support, and a significant fraction of the postings address this concern.
- **Chapter 4—Simplified Input and Output.** The high level of interest in this topic presumably simply reflects the fact that the problem of teaching I/O was the most frequently cited problem to appear in the review of the literature described in Chapter 3. Few suggestions for changes were offered. The only modification that we made to the **`acm.io`** package in response to the feedback on the web forum was to allow programmers to specify optional range limits in the **`readInt`** and **`readDouble`** methods, as suggested by Alice Brady [Brady05].
- **General Info about Task Force.** The major item of discussion under this topic was the question of license restrictions for the package. The first release was issued under a more restrictive licence than the Task Force intends to use for the final release. Our concern at the time was to discourage anyone working with the initial release from publishing programs derived from that early version of the library packages. Allowing people to build too freely on top of the first release would create barriers to change in subsequent releases. The public license for the final release will allow anyone to develop software of any kind using the ACM packages and will protect only the ACM's interest in the package code itself. The danger of not having this minimal level of protection is that someone else could come along and seek to assert intellectual property rights over the ACM code.

Several of the topic areas on the web forum received no posts at all. It is unclear whether the lack of response in these areas indicated that the ideas were uncontroversial or merely uninteresting. The Task Force did take some satisfaction in the fact that there were no postings for the topic "Installing the Libraries," which suggests that readers of the forum were able to download the packages without too much difficulty.

Figure 2-2. Announcement of initial release and request for feedback

```
To: SIGCSE-MEMBERS@ACM.ORG
From: Eric Roberts <eroberts@cs.stanford.edu>
Subject: ACM Java Task Force draft release
Date: Thu, 3 Feb 2005 12:48:46 -0800
--------
Dear SIGCSE members and colleagues,

At the SIGCSE Symposium in Norfolk in 2004, the ACM Education Board
announced the formation of the ACM Java Task Force and assigned it
the following charge:

      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.

The Task Force issued a request last spring to the SIGCSE community
seeking feedback on the problems people have encountered in teaching
Java and the solutions they have developed in response.  We spent the
remainder of the year (and the first month of this one) reviewing
those suggestions and using them to guide the design of a small set
of packages intended to simplify the use of Java in introductory
courses.  Although the Task Force will not complete its work until
the summer, we are ready to solicit more extensive community feedback
and have released our first public draft, which is available on the
web at

      http://www.acm.org/education/jtf/

The web site includes both source and compiled versions of the proposed
new packages, a gallery of demo programs, documentation trees offering
both a complete description of the package and an abridged student view,
and an extensive rationale document outlining the reasons for our design
decisions.

We hope that you will look over the materials available on that site and
give us feedback.  There will be a report and feedback session at SIGCSE
2005 in St. Louis, which is scheduled for 10:30am on Thursday, February
24.

In the next few days, we will complete the installation of a new
threaded discussion group for the Task Force recommendations.  That
discussion group will be available at the following web site:

      http://www.cs.duke.edu/phpbbjtf/

We will send a second announcement to the SIGCSE lists when that site is
enabled.  In the meantime, you can send mail to the Java Task Force at
the following address:

      java-task-force@cs.stanford.edu

We plan to collect feedback through March 31 and then use that feedback
to guide further development of the materials prior to the final release
in June.

We look forward to your comments.

                                    -- Eric Roberts
                                       for the ACM Java Task Force
```

Figure 2-3. Response rate for each topic on the feedback site

Forum	Topics	Posts
Java Task Force		
General Info about Forum READ FIRST BEFORE POSTING Information about creating accounts and posting messages. READ THIS FIRST before posting messages.	4	6
General info about Task Force This forum describes general information about the Java Task Force	4	10
Chapter 1—Introduction Comment on Chapter 1 of the JTF Document	0	0
Chapter 2—Principles and Methodology Comment on Chapter 2 of the JTF Document	0	0
Chapter 3—Taxomomy of Problems in Teaching Java Comment on Chapter 3 of the JTF Document	0	0
Chapter 4—Simplified Input and Output Comment on Chapter 4 of the JTF Document	3	10
Chapter 5—Object-Oriented Graphics Comment on Chapter 5 of the JTF Document	2	5
Chapter 6—The Program Hierarchy Comment on Chapter 6 of the JTF Document	3	14
Chapter 7—Utilities Comment on Chapter 7 of the JTF Document	0	0
Chapter 8—Subsetting the Java API Comment on Chapter 8 of the JTF Document	2	5
JTF Installing the Libraries Ask questions here if you are having problems installing the acm libraries.	0	0
JTF Demo Programs Comment on Demo Programs	2	5
Other Topics Post here if your posting does not fit into any other category	5	7

The members of the Java Task Force presented the highlights of the Task Force report at the SIGCSE 2005 symposium on February 24. The response was overwhelmingly positive and provided considerable encouragement that we were on the right track. The changes that were necessary after the first release, unfortunately, took considerably longer than the Task Force had initially hoped. The original schedule, however, was unduly aggressive and did not include sufficient time to refine the design after the feedback phase.

Chapter 3
Taxonomy of Problems in Teaching Java

Before it is possible to search for solutions to the problems involved in teaching Java at the introductory level, it is essential to know what those problems are. To this end, one of the first tasks that the Java Task Force undertook was to survey the literature of computer science education in an attempt to identify the problems that have generated the greatest level of concern.

On the basis of that survey, we have divided the pedagogical problems associated with Java into three categories: high-level issues that are in some sense beyond the details of the language itself, language issues that arise from the design of Java itself, and API issues associated with the application programmer interfaces provided as part of Sun's standard Java releases. The issues in each of these categories are shown in Figure 3-1 along with a capsule assessment of their current status as problems for the community.

As the annotations in the right-hand column indicate, many of the problems with Java reported in the literature have been at least partially addressed, either by incremental improvements over time or by new releases of the Java environment. Version 1.4 of the Java Development Kit (JDK), for example, added an assertion mechanism to Java, thereby enabling the specification of preconditions [Zukowski02]. The many changes—parameterized types, enumeration types, boxing and unboxing of primitive data, an extension to the `for` statement syntax to support iteration, and new APIs to support concurrency and formatted I/O—introduced in Java 5.0 [Austin04, Heiss03] will have an even more dramatic effect on pedagogy.

Figure 3-1. Summary of issues identified by the Java Task Force

High-level issues:	
H1. Scale	(remains a concern)
H2. Instability	(remains a concern)
H3. Speed of execution	(improving over time)
H4. Lack of good textbooks and environments	(improving over time)
Language issues:	
L1. Static methods, including `main`	(remains a concern)
L2. Exceptions	(remains a concern)
L3. Poor separation of interface and implementation	(partly addressed by tools)
L4. Wrapper classes	(added in Java 5.0)
L5. Lack of parameterized classes (templates)	(added in Java 5.0)
L6. Lack of enumerated types	(added in Java 5.0)
L7. Inability to code preconditions	(added in JDK 1.4)
L8. Lack of an iterator syntax	(added in Java 5.0)
L9. Low-level concerns	(disposition varies)
API issues:	
A1. Lack of a simple input mechanism	(remains a concern)
A2. Conceptual difficulty of the graphics model	(remains a concern)
A3. GUI components inappropriate for beginners	(remains a concern)
A4. Inadequate support for event-driven code	(remains a concern)

While the release of Java 5.0 is likely to solve several problems, its adoption in the near term represents a significant challenge to the educational community. Sun released Standard Edition 5.0 of the Java 2 Platform (J2SE 5.0) only in September 2004, and it remains unclear how quickly educational institutions will be able to adopt it. Moreover, given that we have very little experience teaching using the new language features included in Java 5.0, there is no direct evidence to indicate how well the new facilities will work in the classroom. In light of this uncertainty, the Java Task Force has decided that its recommendations must be implementable, at least in part, by those who continue to use earlier versions of Java.

The sections that follow offer additional details on the set of problems we have identified from the literature.

3.1 High-level issues

This section outlines those challenges that in some sense transcend the details of the Java language and its libraries. These concerns therefore represent a set of meta-level issues that as often as not are intrinsic to the character of modern programming languages. They are nonetheless quite important in terms of their practical impact on teaching Java.

H1. Scale

At some level, the most serious problem facing instructors who try to teach Java—or any modern industrial-strength language for that matter—is the problem of scale. While such languages may themselves be reasonably simple, writing any useful programs requires the use of classes supplied as application programmer interfaces (APIs) along with the language. For modern languages, such API collections are vast. The existence of these huge libraries makes it difficult for students and teachers to learn the language without suffering from conceptual overload. This point was made elegantly in Niklaus Wirth's invited address at ITiCSE'02 [Wirth02] and has also been addressed by other writers in recent years [Hadjerrouit98, Roberts01, Roberts04a].

The problem of overwhelming scale is also unlikely to diminish as time goes on. Each release of Java is larger than its predecessor, which only adds to the scale problem. Java 5.0 [Austin04, Heiss03], in particular, adds considerable complexity to the language, even as it addresses many of the problems identified in earlier versions. Some critics of Java 5.0 have gone so far as to argue that the added complexity in the new release might spell the "end of Java" [Cooper03], although the jury is clearly still out on this question.

We believe that the best way to address the problem of scale is to define a subset of Java and its APIs that reduce the level of detail complexity while enabling a variety of approaches to introductory computer science education. This goal is reflected in the first two deliverables as described in the summary of the special session on the Java Task Force at SIGCSE'04 [Roberts04b]:

1. *A definition of a subset of the standard Java APIs appropriate for first-year computer science*. This subset would involve restricting both the number of classes used as well as the number of public methods made visible within those classes. Note that this subset must be sufficient to have students write significant applications using Java. To this end, it will presumably be a superset of the AP Java subset [Astrachan00] which seeks to define what aspects of the language will be tested on the AP exam.
2. *A public web site containing an updated JavaDoc reference manual for the approved Java subset*. This web site would make it possible for students to browse the standard classes and methods defined in the subset without being overwhelmed by classes, methods, and concepts they are unlikely to use. For the classes and methods that are

included, the web site will contain more examples and tutorial material than is currently supplied with the Java APIs.

It is important to understand that the Task Force is not seeking to define a minimal subset, but rather a subset that is rich enough to support different pedagogical strategies while still providing some relief from the overwhelming scale and complexity that adhere to the full-blown releases of Java. There is, moreover, nothing to prevent an individual instructor from venturing beyond the limits provided by the subset. In fact, we encourage instructors to experiment with Java classes—whether home-grown or part of the standard Java release—whenever such classes can be used to advance computer science pedagogy.

H2. Instability

The problem of scale is reinforced by the fact that Java continues to evolve rapidly. As a result, pedagogical materials must be redesigned on an ongoing basis, making it difficult for both those who create those materials and those who try to use them. One of the goals of the ACM Java Task Force is to provide a modicum of stability by defining the standard Java subset in a way that insulates adopters from at least some of the rapid evolution going on in the Java community. A more complete discussion of the nature of this instability along with some of the underlying reasons why it occurs can be found in the background paper presented at SIGCSE'04 [Roberts04a].

H3. Speed of execution

Many of the early papers describing classroom experience with Java [Bergin98, King97, Tyma98] cite the relatively slow execution time of Java programs as a problem. Because Java is compiled to a virtual machine rather than the instruction set of the underlying hardware, Java will never have the runtime efficiency of programs in C or C++. Even so, the situation has improved markedly since the early days of Java. If nothing else, the increase in processor speed means that most users now see much faster execution times than they did a few years ago. In addition, the development of just-in-time compiler technology (JIC) means that the Java runtime environment tends to run much faster on frequently executed portions of code.

H4. Lack of good textbooks and environments

Most of the early adopters of Java complained about the shortage of usable textbooks and teaching environments, but that problem has certainly been alleviated in recent years. At this point in time, there are far too many Java textbooks to list, many of which have been used effectively in introductory courses. There are, moreover, a number of Java-based environments that have been used successfully in the classroom, including BlueJ [Kölling00], DrJava [Allen02], and jGRASP [Cross04]. It does not seem appropriate for the ACM Java Task Force to endorse any specific textbooks or commercial development environments. At the same time, it is important to ensure that the strategies we recommend work in as many of the leading environments as possible.

3.2 Language issues

The following problems arise from the details of the Java language itself, as opposed to the associated APIs. Because it is generally harder to change the language definition than it is to provide an alternative for an API, these problems are likely to be more difficult to solve than those described in section 3.2 on "API issues."

L1. Static methods (including main)

One of the common complaints about Java to emerge in papers from recent years is the fact that beginning students must come to grips with static methods too early in the first

course [Biddle98, Martin98, Reges00, Reges02]. Part of the problem comes from the fact that the standard mathematical functions are implemented as static methods in the **Math** class, forcing students to understand that the call **Math.sqrt(x)**, for example, is different from most method invocations in that there is no object to which a message is sent. For better or worse, Java 5.0 allows one to import all static methods and constants from a class. Thus, if the programmer includes the line

```
import static Math.*;
```

one can subsequently drop the **Math** prefix and simply invoke **sqrt(x)**. While such a change may make the resulting code easier for students, the conceptual distinction has not gone away, and it is still necessary to explain the differences between static and nonstatic method invocation.

The heart of the problem, however, is not the existence of static methods in the public APIs as much as the fact that Java applications, as opposed to applets, are invoked statically through a call to the method

```
public static void main(String[] args)
```

The need to include a **main** method introduces—presumably on day one of the course—a whole host of confusing issues including the keywords **public**, **static**, and **void**, the class **String**, and the empty-bracket syntax for designating an array parameter. To some extent, these problems can be finessed simply by declaring the syntax to be a bit of arcane boilerplate whose meaning will become clear in the fullness of time. The more difficult problem is that the **main** method is invoked without instantiating an object, which means that any subsidiary methods must also be declared as static. Moreover, the entire idea of object-oriented behavior is undermined through this mechanism.

The situation is very different with applets. When you execute an applet, the browser creates a new object of the appropriate applet subclass and then sends messages like **init** and **start** to the newly instantiated object. The applet paradigm is thus considerably more object-oriented, although it has other problems of its own. Developing a mechanism to make application invocation more object-oriented and to reduce the asymmetry between the two models would be of considerable pedagogic value.

L2. Exceptions

Many of the early reports on using Java as an introductory language express concern about the fact that the definitions of various library APIs make it impossible to ignore the concept of exceptions, even in simple examples [Biddle98, Hosch96, Weiss98, Roberts01]. The classic illustration of this problem occurs when the programmer wants to specify a delay in execution. The static method **Thread.sleep** offers the necessary functionality, but it is not possible to write

```
Thread.sleep(time);
```

because **Thread.sleep** can throw **InterruptedException**, which must be caught by the client, as follows:

```
try {
   Thread.sleep(time);
} catch (InterruptedException ex) {
   This exception is typically ignored.
}
```

Similar problems with exceptions also occur during I/O operations, which exacerbates the problems that arise under topic A1 (Lack of a simple input mechanism).

The rules concerning exceptions and the list of exceptions raised by the existing Java APIs are beyond the boundaries of what this Task Force can change. The only thing that the Task Force can do—as many who have implemented Java libraries have done—is to provide additional mechanism that allow novices to accomplish the most common operations without having to deal with the exceptions. The graphics libraries developed at Williams [Bruce01] and Stanford [Roberts98], for example, both include a **`pause`** method whose implementation encapsulates the exception-handling phase of the **`Thread.sleep`** call.

L3. Poor separation of interface and implementation

Several of the critical reviews of Java as a teaching language point out that Java makes it harder to separate the high-level specification of what a class does from the low-level implementation that determines how it does it [Biddle98, Hosch96, Roberts01]. The crux of the problem is that Java classes, unlike those of C++, do not have a separate specification section that includes the prototypes of the methods but not their bodies. As a result, anyone who looks at the code for a class is forced to see all of the detail rather than a more abstract specification indicating what the client needs to know.

While it is impossible to fix this problem within the confines of the Java definition, sensitivity to the value of separating interface-level specification from the underlying implementation has an effect both on pedagogical strategy and the design of new classes for teaching. Abstract classes and interfaces each provide some level of relief from this concern. Another approach altogether is to rely on documentation tools (such as **`javadoc`**) and programming environments to provide the conceptual separation between specification and implementation that the language lacks.

L4. Wrapper classes

At the language level, one of the most often cited concerns in the past has been the distinction between primitive types such as **`int`**, **`boolean`**, and **`double`** from the corresponding full-fledged counterparts (typically called *wrapper classes*) **`Integer`**, **`Boolean`**, and **`Double`** [Biddle98, Reges02, Roberts01]. The problem with the distinction between these two categories is partly the conceptual overload students feel when they try to understand why there are two distinct flavors of what seem to be the same thing. In practical terms, working with primitive types introduces considerable extra complexity whenever those values are introduced into a container. It is, for example, impossible to add an **`int`** value **`n`** to an array list using the otherwise intuitive code

```
list.add(n);
```

because the value of **`n`** is a primitive type rather than an object. Historically, Java has insisted that the programmer explicitly allocate the corresponding **`Integer`** object before invoking **`add`**, as follows:

```
list.add(new Integer(n));
```

Similarly, the programmer must also specify the corresponding reverse conversion when removing a primitive value from a collection. To retrieve the first integer from list and assign it to the variable **`first`** requires the incantation

```
first = ((Integer) list.get(0)).intValue();
```

which is certain to confuse most novices.

Fortunately, this problem—particularly when one also makes use of the template mechanism described in the following section—goes away in Java 5.0. The new 5.0

release includes automatic "boxing" and "unboxing" of the primitive types so that the preceding insertion and selection operations can be reduced to

```
list.add(n);
```

and

```
first = list.get(0);
```

assuming that **`list`** is declared as an **`ArrayList<Integer>`**.

L5. Lack of parameterized classes

Another widespread complaint about Java is that the language does not support strongly typed collections because it lacks a counterpart to the C++ template facility [Biddle98, Hadjerrouit98, Hosch96, Martin98]. Java 5.0 introduces a structure for parameterized types in a way that eliminates many problems associated with the C++ implementation of this mechanism while retaining the convenience of the syntax. This mechanism, which is usually called *generics* in Java, has also been retrofitted into the Java collection framework so that it is possible, for example, to declare an array list of integers by writing

```
ArrayList<Integer> list;
```

or a hash table containing strings by

```
HashMap<String> dictionary;
```

Judging from the documentation and examples, parameterized types in Java unfortunately introduce a number of complex and potentially confusing issues, but it should be possible to avoid such complexity in classroom settings either by limiting their use to the standard collections framework or by imposing judicious restrictions on the ways in which type parameters can be used. It is, however, difficult to assess whether generics will succeed with an introductory audience until people have had a chance to experiment with it.

L6. Lack of enumerated types

Another weakness in the Java type system that has attracted wide attention is the lack of enumerated types of the sort available in Pascal, C, and C++ [Biddle98, Hosch96, Reges02]. Java 5.0 introduces enumerated types, and it should be possible for instructors to use these in much the same way that they have in those other languages.

L7. Inability to code preconditions

In one of the very early papers to discuss Java as a teaching language, Frederick Hosch expressed concern that Java included no mechanism for specifying runtime assertions to check the preconditions of a method, such as those provided in C and C++ by the **`assert`** facility [Hosch96]. Assertions have become part of Java as of release 1.4, so that this problem is now historical. Interestingly, however, the documentation of release 1.4 specifically discourages the use of **`assert`** to enforce preconditions, arguing that such failures are more properly handled by raising an illegal argument exception.

L8. Lack of an iterator syntax

In his comparison of Java and C#, Stuart Reges expresses a fondness for the **`foreach`** statement in C# [Reges02]. Other writers have expressed similar views about the pedagogic value of including syntactic support for iteration [Roberts01]. Java 5.0 includes a new syntax for iteration based on the **`for`** statement, making it possible, for

example, to iterate over the elements in a variable **`list`** of type **`ArrayList<Integer>`** by writing

```
for (Integer i : list) {
    Body of loop
}
```

L9. Low-level concerns

In the interest of completeness, it is useful to identify a few other features of Java that have been cited in the literature as problems:

- *Holdovers from C and C++*. Several writers complained about features of Java that are simply the result of its historical roots as a descendant of C and C++. These concerns include, for example, the fact that it is possible to fall through from one **`case`** clause to the next in a **`switch`** statement, that the distinction between **`=`** and **`==`** is confusing for novices, and that the language lacks call-by-reference [Biddle98, King97, Martin98, Reges02, Weiss98].
- *Ambiguities in equality testing*. In an early critique of Java in the instructional environment, Robert Biddle and Ewan Tempero express concern that the semantics of the **`==`** operator are confusing when the operands are objects [Biddle98]. The problem is particularly annoying in the case of the **`String`** class, where students can easily be misled by the fact that the Java interpretation (reference equality) often gives correct results, given Java's propensity for storing a single copy of each string value. Similarly, the default implementation of the **`equals`** method in the **`Object`** class can lead to surprising behavior if subclasses fail to override it when appropriate.
- *Default visibility specification*. A couple of early papers point out that the default visibility for methods is not the same as that imposed by any of the explicit modifiers **`public`**, **`private`**, or **`protected`** [Biddle98, Hosch96]. As a result, students who do not include an explicit visibility specifier will have difficulty understanding exactly what classes have access to that method. Since many of those students will also be writing classes as part of the unnamed "anonymous" package, the default visibility is effectively **`public`**, which undermines the philosophy of information hiding.

Each of these concerns represents an issue that is on the one hand deeply embedded in the design of Java and on the other hand of relatively minor consequence. The Java Task Force has no plans to address these concerns other than by encouraging the use of coding styles that minimize the associated problems.

3.3 API issues

This final category of documented problems in Java concerns the limitations of the existing APIs.

A1. Lack of a simple input mechanism

By far the most widely cited problem in papers about Java is the lack of any simple facility for accepting input from the user [Grissom00, Koffman01, Martin98, Reges02, Roberts01, Wallace99, Weiss98, Wolz99]. Many textbook authors have developed packages that offer a simple input mechanism, but these have met with resistance from the marketplace because they are not part of the Java standard. Others have suggested that a console-like input facility is a throwback to an out-of-date procedural programming style and that such techniques should be replaced by dialog input that fits the more modern, interactive style.

With the release of Java 5.0, console-style user input—for those who want it—is less of a problem. The new **`Scanner`** class in the **`java.util`** package makes it relatively easy to read input values from a stream using the following code:

```
Scanner sc = new Scanner(System.in);
int n = sc.nextInt();
```

Unlike the stream operations defined in the **`java.io`** package, the **`Scanner`** abstraction does not throw exceptions that the client code must catch, thereby simplifying the required code enormously.

A2. Conceptual difficulty of the graphics model

Ever since the technology has made it possible, many instructors have included graphical applications in their introductory programming courses because such applications generate so much excitement in students even as they illustrate important programming concepts. Although the Java APIs offer extensive graphical capabilities, it is often difficult for novices to master the Java graphics model [Bruce01, Martin98]. In a 1998 paper [Roberts98], Eric Roberts identified three problems in the Java graphics model that make it difficult to use for beginning students:

1. *Forgetful bitmaps*. Under the Java graphics model, each component has the responsibility to respond to update events, which generate calls to **`paint`** (or **`paintComponent`** under Swing). Implementing those repainting methods requires students to maintain enough state information to regenerate the image. Designing the data structure to maintain this state is much more difficult than calling a set of methods to create a static display. This problem can be solved through a technique called *double-buffering,* in which the user's program draws into an offscreen memory buffer that can be copied to the screen whenever an update event occurs.
2. *Statelessness of the graphics context*. In Java, the graphics context preserves relatively little state from call to call, which forces programmers to maintain more state in their code. In particular, Java does not maintain the notion of a current point, which forces students to specify the endpoints of each line segment explicitly rather than chaining together a set of vector displacements.
3. *An unfamiliar definition of the coordinate system*. Java's standard coordinate system differs from traditional mathematical coordinates in three ways: (1) it uses integers rather than real numbers to specify points, (2) coordinate values depend on the device resolution, and (3) the axes are not arranged in the familiar Cartesian form. Each of these differences makes it more difficult for the student to use the graphics model.

A3. GUI components inappropriate for beginners

The standard interactor classes provided by the Java Swing library are often difficult for novices to use. To overcome this problem, several institutions that have adopted Java have developed new interactive toolkits to provide similar functionality in a form more easily understood by introductory students. The toolkits include the Java Power Tools collection developed at Northeastern [Raab00, Rasala00] and the **`simpleIO`** package developed by Ursula Wolz and Elliot Koffman [Wolz99]. The advantages of such a approach are illustrated in the following list of goals for the Java Power Tools package, which seeks to provide

- An infrastructure for creating well designed programs that illustrates the concepts of computer science and its practical applications
- An environment for learning the basic ideas of interface design and for experimenting with a variety of designs

- A paradigm for building interfaces in Java that scales from individual data items to large structures, using recursively displayable container classes [Raab00]

A4. Inadequate support for event-driven code

Given that most computer science instructors are more familiar with such an approach, many institutions have continued to teach their introductory courses in a procedural style, even if they have adopted Java as the language of instruction. In fact, the Computing Curriculum 2001 report [ACM01] found that most courses we surveyed operated under a traditional procedural paradigm, even if they used an object-oriented language to do so. A few particularly forward-looking institutions were experimenting with alternative curriculum designs in which students are taught the object-oriented paradigm from the very beginning. Such strategies typically begin with simple objects that respond to messages generated either by asynchronous events or through interaction with other objects. Creating programs in this style gives students a much deeper perspective on the philosophy of object-oriented design.

Unfortunately, such approaches can be difficult to implement in standard Java because of the complexity of the event model and the difficulties involved in coding concurrent programs [Bergin98, Hartley98]. To get around these problems, some institutions—notably Williams [Bruce01] and MIT [Stein98]—have experimented with toolkits that support highly interactive, event-driven programs. To allow more institutions to use similar curricular strategies that emphasize an object-oriented approach, it will be useful to make this sort of toolkit more widely available.

The Java 5.0 release does include a new toolkit for concurrency based on earlier work by Doug Lea [Lea99]. While this new API will provide much better support for concurrency in Java, it is not clear to what extent this package will prove useful in the introductory course.

Chapter 4
The **acm.io** Package

The survey of problems presented in Chapter 3 notes that the problem most often cited by those attempting to teach Java to novices is the lack of a simple input mechanism, either through a traditional text-stream interface or a more modern dialog-based approach. Although this problem—identified as A1 in the taxonomy—is partly addressed by the introduction of the **Scanner** class in Java 5.0 and the **JOptionPane** class in Swing, the Task Force concluded that it would be worth introducing a package that offered an integrated approach to user input that would provide both traditional and dialog-based input options in a simple, consistent way. In the Java Task Force release, this capability is provided by the **acm.io** package, which contains two public classes:

1. An **IOConsole** class that supports traditional text-based interaction within the standard Java window-system hierarchy.
2. An **IODialog** class that offers similar functionality to the **JOptionPane** class in a significantly simplified form.

In addition to these classes, the **acm.io** package defines an **IOModel** interface that specifies the input and output operations common to **IOConsole** and **IODialog**. The sections that follow describe the two public classes in more detail.

4.1 Console-based I/O

One of the most common strategies used by Java textbook authors to address the problem of user input is to create an **IOConsole** class that provides interactive text-based I/O within the framework of the standard Java graphical APIs. As examples, the Java textbooks published by Holt Software Associates [Hume00] and the web-based textbook written by David Eck [Eck02] define their own classes that provide this functionality. Similar packages not linked to textbooks have also been proposed, including those developed by Ron Poet [Poet00] and Eric Roberts [Roberts04d].

While the large number of existing implementations clearly indicate that an **IOConsole** class enjoys a certain popularity in the community, its inclusion in the JTF packages has generated a certain amount of controversy. The principal objection to having such a class is that the underlying computational paradigm is not representative of the event-driven, interactive user interfaces for which the object-oriented paradigm is so well suited. In the papers surveyed for the problem taxonomy presented in Chapter 3, several authors argue that Java programs should avoid such traditional modes of interaction and instead make use of dialogs that fit more closely with modern paradigms of interactive programming. After extensive discussion, the Java Task Force decided that the **acm.io** package should support both console- and dialog-based I/O, but only if the two models could implement a single interface, which would allow easy transition from one paradigm to the other. Our reasons for retaining the console-based model include:

- *The console-based model has significant utility in its own right.* The **IOConsole** class has many uses beyond its obvious role as a framework for traditional procedural programs. Interpreters, for example, can use it to provide a *read-eval-print* mechanism, which serves as a powerful interactive tool.
- *Including both paradigms makes it easier to support automated testing.* It is often harder to design test programs for an interactive environment, particularly if such tests are automated. Consider, for example, traditional stream-based programs written for

the Unix/Linux domain (or, equivalently, Mac OS X). It is usually easy to test such programs by using redirection to supply an input script and an output log. That style of testing is typically unavailable under an interactive, dialog-based paradigm. By including compatible console- and dialog-based I/O facilities, however, the ACM packages support such testing strategies under either input model.

- *The existence of a console mechanism enables teachers without object-oriented programming experience to teach Java more effectively*. Many computer science teachers, particularly at the secondary school level, have little experience with Java and event-driven, interactive programs, but considerable familiarity with programs that use console-based interaction. The Task Force concluded that it would be valuable to offer teachers a mechanism that falls within their domain of expertise, at least as an interim measure until object-oriented programming skills are more widely distributed.

As a simple illustration, the code shown in Figure 4-1 shows the Java code necessary to create a Swing application that adds two numbers using the **`IOConsole`** class. The following screen snapshot shows how this program appears when running on a Macintosh under Mac OS X:

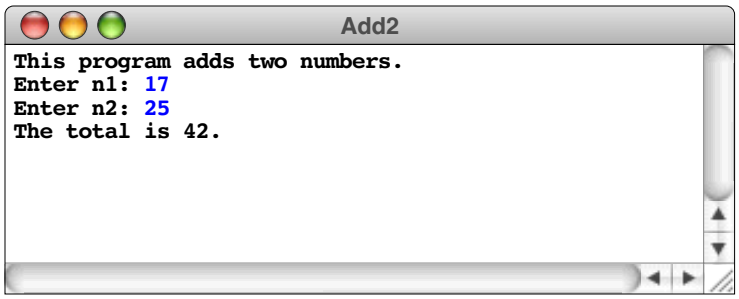


Figure 4-1. Swing application to add two numbers

```
/*
 * This file contains a Swing application that uses an IOConsole
 * object to request two numbers from the user and then print
 * their sum.
 */

import acm.io.*;
import java.awt.*;
import javax.swing.*;

/** Application to add two numbers */
public class Add2Application {
   public static void main(String[] argv) {
      JFrame frame = new JFrame("Add2Application");
      IOConsole console = new IOConsole();
      frame.getContentPane().add(BorderLayout.CENTER, console);
      frame.setSize(500, 300);
      frame.show();
      console.println("This program adds two numbers.");
      int n1 = console.readInt("Enter n1: ");
      int n2 = console.readInt("Enter n2: ");
      int total = n1 + n2;
      console.println("The total is " + total + ".");
   }
}
```

The code in Figure 4-1 can be simplified substantially by using the **ConsoleProgram** class described in Chapter 6. The intent of this example is to illustrate the use of **IOConsole** as a standalone tool.

The public methods in the **IOConsole** class are shown in Figure 4-2. The only new methods in the class added since the first release are the versions of **readInt** and **readDouble** that include a range specification. These methods were added in response to the following comment on the JTF web forum by Alyce Brady:

> I was wondering . . . whether the Task Force also considered (and possibly rejected) the possibility of straightforward range checking for **int** and **double** types? This is easy to support if both the lower- and upper-bounds are required—just provide an additional method for each that takes the two bounds as parameters. This is much cleaner than requiring the students to do range-checking within a loop. Of course, they should eventually be able to do this, but to provide range-checking would seem to be consistent with the type-checking that is already being done. [Brady05]

Given that **IOConsole** derives its inspiration from the familiar paradigm of text-based, synchronous interaction, the design of the class is relatively straightforward. Even though the underlying paradigm is familiar, there are nonetheless several important features of the **IOConsole** class that are worth highlighting:

- *The console mechanism supported by the package is part of the component hierarchy*. One of the common areas of confusion in discussing this proposal was that the **IOConsole** class is part of the window system hierarchy and not simply a layered structure on top of the standard I/O streams **System.in** and **System.out**. Making the console facility part of the standard windowing system increases the flexibility of the package and makes it usable in both the application and applet worlds. In many browsers, the standard I/O streams are not normally displayed, which makes them very difficult for students to use.
- *The console facility makes it possible to differentiate user input and error messages from program output*. The **IOConsole** class makes it possible for students to tell the difference between user input and program output. The methods **setInputColor** and **setInputStyle** set these properties for user input, while **setErrorColor** and **setErrorStyle** do the same for error messages. Output text is displayed in the style and color given by the font and foreground settings and can therefore be manipulated by calling **setFont** and **setForeground**. By default, user input is shown in blue, and error messages appear in red. One of the principal advantages of making these distinctions is that the pattern of user interaction is obvious when the program is displayed on a classroom projection screen.
- *The package includes support for menu bars that support printing and saving the contents of the console, along with standard editing operations*. When an **IOConsole** object acquires the keyboard focus, it looks to see whether a menu bar has been installed in the frame and, if not, creates a menu bar with standard **File** and **Edit** menus. In either case, the menu bar is set to direct its actions to that console. This feature is described in more detail in section 6.6.
- *The input and output streams for a console are available as readers and writers*. The **IOConsole** class defines the methods **getReader** and **getWriter**, which return a **BufferedReader** and a **PrintWriter**, respectively. These objects can therefore serve as character-stream versions of the **System.in** and **System.out** objects, which rely on the less portable byte-stream protocol. Moreover, the **IOConsole** class defines a special constant **SYSTEM_CONSOLE**, which reads and writes to the standard **System.in** and **System.out** streams, thereby providing access to these streams using the more modern discipline of readers and writers as opposed to byte streams.

Figure 4-2. Public methods in the `IOConsole` class

Constructor

`IOConsole()`
Creates a new console object, which is a lightweight component capable of text I/O.

Output methods

`void print(`*any* **`value)`**
Writes the value to the console with no terminating newline.

`void println(`*any* **`value)`**
Writes the value to the console followed by a newline.

`void println()`
Returns the cursor on the console to the beginning of the next line.

`void showErrorMessage(String msg)`
Displays an error message on the console.

Input methods

`String readLine()` *or* **`readLine(String prompt)`**
Reads and returns a line of text from the console without the terminating newline.

`int readInt()` *or* **`readInt(int min, int max)`** *or*
`readInt(String prompt)` *or* **`readInt(String prompt, int min, int max)`**
Reads and returns an **`int`** value, with optional specification of a prompt and limits.

`double readDouble()` *or* **`readDouble(double min, double max)`** *or*
`readDouble(String prompt)` *or*
`readDouble(String prompt, double min, double max)`
Reads and returns a **`double`** value, with optional specification of a prompt and limits.

`boolean readBoolean()` *or* **`readBoolean(String prompt)`**
Reads and returns a boolean value (**`true`** or **`false`**) from the console.

`boolean readBoolean(String prompt, String trueLabel, String falseLabel)`
Reads a boolean value by matching the user input against the specified labels.

Additional methods (most of which are unlikely to be used by novices)

`void setFont(Font font)` *or* **`void setFont(String str)`**
Sets the overall font for the console, which may also be specified as a string.

`void setInputStyle(int style)` *and* **`int getInputStyle()`**
Sets (or retrieves) the font style for user input, which can be different from the output style.

`void setInputColor(Color color)` *and* **`int getInputStyle()`**
Sets (or retrieves) the font color for user input, which can be different from the output color.

`void setErrorStyle(int style)` *and* **`int getErrorStyle()`**
Sets (or retrieves) the font style for error messages.

`void setErrorColor(Color color)` *and* **`int getErrorStyle()`**
Sets (or retrieves) the font color for error messages.

`void setExceptionOnError(boolean flag)`
Sets the error-handling mode: **`false`** means retry on error, **`true`** means raise an exception.

`boolean getExceptionOnError()`
Returns the error-handling mode, as defined in **`setExceptionOnError`**.

`PrintWriter getWriter()`
Returns a writer that can be used to write to the console (analogous to **`System.out`**).

`BufferedReader getReader()`
Returns a reader that can be used to read from the console (analogous to **`System.in`**).

`void setInputScript(BufferedReader rd)`
Sets the console so that it reads its input from the specified reader rather than the user.

`BufferedReader getInputScript()`
Returns the current input script from which the console is reading.

`void clear()`
Clears the console screen.

`void save(Writer wr)`
Saves the contents of the console to the specified writer.

`void print(PrintJob pj)`
Prints the contents of the console to the printer as specified by **`pj`**.

4.2 Dialog-based I/O

The **IOConsole** class—useful though it is—does not in any sense constitute a complete solution to the user-input problem. Most instructors are looking for some way to support dialog-based I/O that is simple enough for students to use. As noted in the discussion of this problem in Chapter 3, several mechanisms have been proposed to address this problem, including the Java Power Tools project developed at Northeastern [Raab00, Rasala00] and the **simpleIO** package developed by Ursula Wolz and Elliot Koffman [Wolz99].

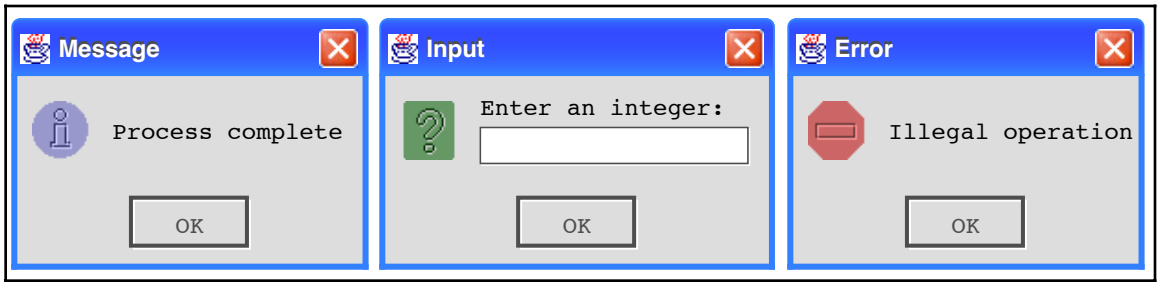
The standard Java APIs, of course, include many methods for constructing user dialogs. In fact, since Swing was introduced as part of JDK 1.2, the Java APIs have included the **JOptionPane** class that makes it possible to put up simple dialogs to request basic data types from the users. Many instructors have used **JOptionPane** successfully, and it appears to be the most common paradigm for this style of user interaction.

Unfortunately, the **JOptionPane** mechanism has several weaknesses that make it problematic for novice users. Of these deficiencies, the most serious are the following:

- *The API is complex*. The **JOptionPane** class includes seven different variants of the constructor, 37 public fields, and 61 public methods. When printed, the online documentation runs 50 pages. While students clearly don't need to understand the entire structure, the danger of information overload is significant.
- *The typical usage paradigm involves static methods*. The **JOptionPane** class adopts a curious mix of the object-based and procedural paradigms. While the displayed dialogs are objects of class **JOptionPane**, the methods that create those dialogs and request input from them are static methods in the **JOptionPane** class. As a result, students are easily confused about the paradigm. Instead of creating an object and then sending it a message requesting user input, the student must instead make static calls on the class.
- *Many of the more useful interaction styles require students to use arrays*. Any **JOptionPane** dialog that provides options outside the default set must supply those options as an array of objects. At the beginning of an introductory programming course, students have not yet learned about arrays, but it is precisely during this period that dialog-based input would be most useful.

The **IODialog** class in the **acm.io** package is designed to solve each of these problems. The customary paradigm is to create an **IODialog** object and then invoke methods on that object to display information, request input, or report errors. As with **JOptionPane** (which is used internally in the implementation as long as Swing is available), these three styles of use generate slightly different dialog formats, as shown in Figure 4-4.

Figure 4-4. The three styles of IODialog interactors



The code that produces this series of dialogs looks like this:

```
IODialog dialog = new IODialog();
dialog.showMessage("Process complete");
int n = dialog.readInt("Enter an integer: ");
dialog.showErrorMessage("Illegal operation");
```

The public methods in the **IODialog** class are shown in Figure 4-5. As the small number of entries in the table makes clear, the **IODialog** class is vastly simpler than the **JOptionPane** facility and should prove much easier for students to use.

One of the design goals for the **acm.io** package was to have the **IOConsole** and **IODialog** classes implement the same interface so that students could easily switch back and forth between the two modes. In the **acm.io** design, that common behavior is

Figure 4-5. Public methods in the IODialog class

Constructor

IODialog() Creates a new **IODialog** object that supports dialog-based interaction.

Output methods

void print(any value) Adds the string to a message that will be displayed when the line is complete.
void println(any value) Displays the value in a dialog box.
void println() Displays any text in the dialog box so far.
void showErrorMessage(String msg) Displays an error message in a dialog box.

Input methods

String readLine() *or* **readLine(String prompt)** Pops up a dialog asking the user to enter a line of text (returned with no terminating newline).
int readInt() *or* **readInt(int min, int max)** *or* **readInt(String prompt)** *or* **readInt(String prompt, int min, int max)** Pops up a dialog asking the user to enter an **int**, with optional prompt and limits.
double readDouble() *or* **readDouble(double min, double max)** *or* **readDouble(String prompt)** *or* **readDouble(String prompt, double min, double max)** Pops up a dialog asking the user to enter a **double**, with optional prompt and limits.
int readInt() *or* **readInt(String prompt)** Pops up a dialog asking the user to enter an integer value.
double readDouble() *or* **readDouble(String prompt)** Pops up a dialog asking the user to enter a double-precision value.
boolean readBoolean() *or* **readBoolean(String prompt)** Pops up a dialog asking the user to select a **true**/**false** value.
boolean readBoolean(String prompt, String trueLabel, String falseLabel) Pops up a dialog asking the user to choose one of the specified labels.

Additional methods (most of which are unlikely to be used by novices)

void setExceptionOnError(boolean flag) Sets the error-handling mode: **false** means retry on error, **true** means raise an exception.
boolean getExceptionOnError() Returns the error-handling mode, as defined in **setExceptionOnError**.
void setAllowCancel(boolean flag) Sets the cancel mode: **true** adds a "Cancel" button that throws an exception.
boolean getAllowCancel() Returns the error-handling mode, as defined in **setAllowCancel**.

specified by the **IOModel** interface, which includes the input and output methods common to the two classes. The existence of this interface makes it possible for code to remain insensitive to the interaction model. Consider, for example, the following method definition:

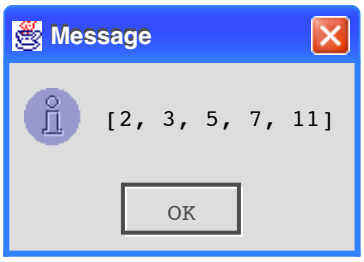
```
void showArray(IOModel io, int[] array) {
   io.print("[");
   for (int i = 0; i < array.length; i++) {
      if (i > 0) io.print(", ");
      io.print(array[i]);
   }
   io.println("]");
}
```

The important idea to notice here is that the first argument to **showArray** can be either an **IOConsole** or an **IODialog** object; since both implement the **IOModel** interface, the implementation doesn't care.

Designing **IODialog** so that it provides a reasonable implementation of **IOModel** turns out to be harder than it looks. The input side is easy, because both consoles and dialogs wait for the user to supply a single input value. The output side is more difficult, largely because a dialog is not a stream device. Having each call to **print** display a separate dialog box is not a good choice, particularly for methods like **showArray** that assemble the output in pieces. Displaying each individual value and comma in a separate dialog box would destroy the conceptual integrity of the output and render the entire mechanism unusable. A more satisfactory approach is to have **print** buffer the output inside the dialog until a call to **println** arrives to signal the end of the line. This interpretation has the effect, therefore, of allowing

```
int array = {2, 3, 5, 7, 11};
showDialog(new IODialog(), array);
```

to generate a single dialog box, as follows:



Similar care must be applied if calls to **print** are followed by an input operation before a **println** call occurs. In such situations, the buffered output is added at the beginning of the prompt string. This treatment provides a sensible interpretation for code in which the programmer explicitly prints a prompt rather than including it as a parameter to the input call.

Another difficult decision in the design of both **IODialog** and **IOConsole** was the question of how to handle illegal input data. For novices, the best strategy is for the input method simply to display an error message and request new input, either by bringing up another dialog or by repeating the prompt on the console. For more sophisticated users, however, it is more appropriate to allow the program to gain control at this point. By default, both the **IODialog** and **IOConsole** classes adopt the novice-friendly approach and request new input. The more advanced programmer, however, can change this

behavior by calling **`setExceptionOnError(true)`**, in which case the input methods throw an **`ErrorException`** on invalid input.

A related issue arises in defining reasonable semantics for the "Cancel" button, which is part of the typical input dialog mechanism available with **`JOptionPane`**. In this case, it is not appropriate to have the program bring up a new dialog after the user clicks the "Cancel" button, which is presumably exactly what the user did *not* want. To avoid this conceptual problem, the **`IODialog`** mechanism does not display "Cancel" buttons as part of the dialog unless the client has made it clear that special handling of dialog cancellation is desired. If the client calls **`setAllowCancel(true)`**, the input methods display a "Cancel" button that throws a **`CancelledException`** when it is clicked.

Chapter 5
The **acm.graphics** Package

One of the most widely cited problems in teaching Java—identified as problem A2 in the taxonomy from Chapter 3—is the lack of a graphics facility that is simple enough for novices to use. This problem has been identified as critical by several authors [Bruce01, Martin98, Roberts98]; Nick Parlante goes so far as to suggest that it is the only problem that rises above his critical threshold:

> Java has many little features that I thought might be a problem in CS1, but in almost every case they worked out fine. Graphics was the one exception. [Parlante04a]

Given the perceived importance of the problem, we were not surprised to find that graphical packages were heavily represented among the solution strategies submitted to the Java Task Force. These packages fall into three categories:

1. Object-based packages in which the user creates objects that draw themselves in a window [Bruce04a, Parlante04a, SandersK04a, SandersK04b]
2. Packages that present graphics in the context of a "microworld" [Lambert04b, SandersD04a]
3. Packages that offer a traditional Cartesian-based graphical model [Roberts04e]

After discussing these strategies at a Task Force meeting, we decided to pursue only the first approach, although we also provide enabling technology for "turtle graphics" along the lines suggested by Ken Lambert and Martin Osborne in their submission to the Task Force [Lambert04b]. The problem with adopting a complete microworld implementation is that there are many existing designs, and it does not seem appropriate for the Java Task Force to privilege one model over the others. In addition, most microworld packages are stand-alone and can be adopted easily even if the Java Task Force does not include them in its collection of APIs. Our reason for rejecting the Cartesian-based package is simply that it differs too radically from Java's own design. Teaching students a graphics model that differs from Java's in any substantive way runs the risk of confusing them all the more when they start to use the facilities in the standard Java packages.

We were, however, convinced of the value of supporting a simple, object-based graphics facility that allows students to assemble figures by creating a graphical canvas and then adding graphical objects to it. The advantages of having such a package include:

- It enables students to use graphics from the very beginning of the first course. Students today, having grown up with graphical interfaces, are much more interested in this type of program than they are in the more traditional input/process/output style of program.
- An object-based graphics model simplifies enormously the task of responding to repaint requests. In both the Abstract Windowing Toolkit (AWT) and the more powerful model provided by Swing, the programmer is responsible for repainting the contents of a window whenever an update is necessary. Implementing that repaint operation under the standard graphics model forces students to maintain enough state information to regenerate the display, which typically requires some sophistication

with data structures. Our graphical objects, however, are designed so that they repaint themselves, thereby automating the repaint process.

- A graphical object hierarchy provides a compelling illustration of how object-oriented programming works. If the package is well designed, it can serve as an early illustration of general notions like subclassing and inheritance along with more specific implementation strategies such as abstract classes and interfaces. The desire to use the graphics class hierarchy as a pedagogical tool provides a strong incentive to keep the package small and consistent in its design.

Although we were convinced that an object-based graphics package had to be one of our deliverables, none of the submissions in response to our call for proposals was in fact ideal. Each of these proposals has strengths and weaknesses, as outlined in Figure 5-1. The proposals therefore served as a foundation for the design of a new package that incorporates the best features of each. The result of that design effort is the **`acm.graphics`** package, which defines an extensible hierarchy of graphical objects rooted at the **`GObject`** class. The subsections that follow offer an overview of **`acm.graphics`** and describe the design decisions involved in its development.

5.1 The coordinate system

One of the fundamental design decisions in the development of any graphics package is the choice of the graphical model, primarily in terms of its coordinate system. Although one of the proposals [Roberts04e] argued for changing the coordinate system to a Cartesian framework similar to that used in Adobe PostScript, the consensus of the Task Force was that such a model represented too much of a change from the standard Java approach and would therefore force students to learn two incompatible graphics models.

Figure 5-1. Strengths and weaknesses of the proposed graphics packages

objectdraw [Bruce04a]	+ Successful experience at multiple institutions + Clean and consistent animation model + Sufficient functionality to define interesting pictures ± Mouse events are available through subclassing, but not by using listeners – Package is relatively large (44 classes and interfaces) – Naming conventions are not consistent – Class hierarchy is not intuitive (**`FramedOval`** extends **`Rect`**, for example) – Drawable classes are not easily extendable by students
OOPS [SandersK04a]	+ Successful experience at multiple institutions + Reasonably small (20 classes and interfaces of which students use eight) ± Mouse events are available through subclassing, but not by using listeners – Some classes (**`QuitButton`**, **`ConversationBubble`**) don't fit the model. – Some names clash with those in standard libraries (**`Frame`**, **`Image`**, **`Shape`**) – The package has no general mechanism for displaying strings – Extending the set of graphical objects requires knowledge of **`Graphics2D`**
Stanford Graphics [Parlante04a]	+ Successful experience (but only at Stanford) + Very small (six classes) and easy to learn + Consistent naming scheme for graphical object classes + Easily extended by students to create new **`DShape`** objects ± Allows composition, but permits non-intuitive containment – Missing some basic functionality, such as drawing lines and arcs – Rectangles and ovals are always filled, never outlined – Objects cannot field mouse events

We therefore made the decision to stick with a graphics model that for the most part matches that used in Java: origin in the upper-left corner, *y* values increasing downward, and coordinates measured in pixels.

There was, however, one change that the Task Force decided—after extensive debate—would be worth adopting. All but one of the graphics proposals we received chose to use **double**s rather than **int**s to specify coordinate values. The principal advantage in adopting a **double**-based paradigm is that the abstract model typically makes more sense in a real-valued world. Physics, after all, operates in the continuous space of real numbers and not in the discrete world of integers, which is simply an artifact of the pixel-based character of graphical displays. Animating an object so that it has velocity or acceleration pretty much forces the programmer to work with real numbers at some point. Keeping track of object coordinates in double-precision typically makes it possible for programmers to use the stored coordinates of a graphical object as the sole description of its position. If the position of a graphical object is stored as an integer, many applications will require the programmer to keep track of an equivalent real-valued position somewhere else within the structure. On the other hand, the disadvantage of using real-valued coordinates is that doing so represents something of a break from the standard Java model (although less of one that it might at first appear, as discussed later in this section).

The Task Force gave detailed consideration to three options for the coordinate system:

1. *The all-integer model*. In this model, coordinate values are stored as **int**s, just as they are in the methods provided by the **Graphics** class in **java.awt**. Locations, sizes, and bounds for graphical objects are stored using the standard **Point**, **Dimension**, and **Rectangle** classes, which use integer-valued coordinates. This option is the easiest to implement and maintains consistency with the methods provided in the **Graphics** class. At the same time, it suffers from the serious deficiency of forcing students to maintain a parallel set of **double**-valued parameters for any applications that involve animation or have other dependencies on a mathematically precise physical world.
2. *The all-double model*. In this model, coordinate values are stored as **double**s, thereby making it possible for students to store locations and sizes in double-precision so that the values can correspond with parameters in the physical world. This model requires aggregate structures that describe locations, sizes, and bounds in double precision and that are therefore analogous to **Point**, **Dimension**, and **Rectangle** in the integer world. Although **double**-valued implementations of these types exist in the **java.awt.geom** package, the class names **Point2D.Double**, **Dimension2D.Double**, and **Rectangle2D.Double** seem certain to generate confusion. If nothing else, these names expose the notion of inner classes, which is not generally taken to be an introductory concept. In light of that complexity, the **acm.graphics** package includes definitions for the classes **GPoint**, **GDimension**, and **GRectangle**, which provide the same functionality.
3. *The doubles-only-for-individual-coordinates model*. As a compromise between the two earlier strategies, the committee also considered an intermediate position in which individual coordinates are stored using type **double**, but without carrying that decision through to the aggregate values. Under this model, individual coordinates and sizes would be available to the client in double-precision using the methods **getX**, **getY**, **getWidth**, and **getHeight**. The built-in methods **getLocation**, **getSize**, and **getBounds**, however, would return standard **Point**, **Dimension**, and **Rectangle** values in which the coordinates are rounded to the closest integer.

The Task Force experimented with each of these options but ultimately chose to go with Option #2. Although the introduction of the new double-precision classes **GPoint**,

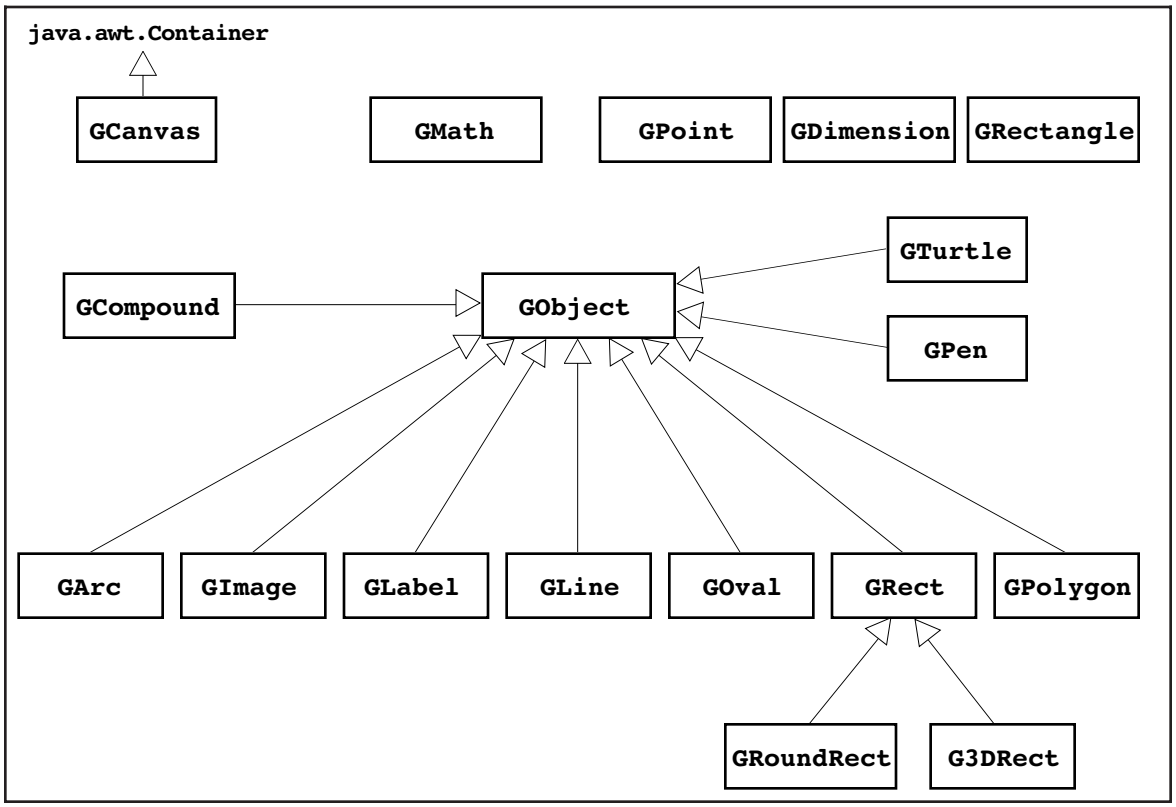
GDimension, and **GRectangle** adds some complexity, the conceptual integrity of using **double**s consistently throughout the implementation ended up carrying the day.

5.2 The graphics class hierarchy

The classes that make up the **acm.graphics** hierarchy are shown in Figure 5-2. The central class in the hierarchy shown in the diagram is the abstract **GObject** class, which is the common superclass of all graphical objects that can be displayed on a graphical canvas. The implementation for the graphical canvas is in turn supplied by the **GCanvas** class, which is described in section 5.5. Descending from **GObject** is a set of classes—collectively referred to as *shape classes*—that correspond to the figures one can draw in the original definition of the **java.awt.Graphics** class. Thus, if the **Graphics** class includes a **draw**—— method for some figure ——, the **acm.graphics** package includes a corresponding **GObject** subclass with the name **G**——. The only deviation from this naming convention arose primarily for nontechnical reasons: a majority of the Task Force felt that it was prudent to substitute the name **GLabel** for **GString**, both to avoid the slightly racy interpretation and to emphasize that a **GLabel** object is not really a **String**. After making this substitution, the hierarchy includes nine shape classes, which constitute the built-in set of graphical objects: **GArc**, **GImage**, **GLabel**, **GLine**, **GOval**, **GPolygon**, **GRect**, **GRoundRect**, and **G3DRect**.

The design of the hierarchy, however, was not without controversy. Because many of the shape classes share common structural characteristics (**GImage**, **GOval**, and **GRect**, for example, all share rectangular frames), changing the class hierarchy to reflect those similarities offers an opportunity for code sharing. To take advantage of that opportunity, both the OOPS and objectdraw submissions include intermediate classes that enable code-sharing for those classes that support common operations. This strategy leads to a

Figure 5-2. Class diagram for the acm.graphics package



hierarchy that includes ovals and images under an abstract intermediate class, which is called **RectangularShape** in OOPS and **Rect** in the submitted version of objectdraw.

The Task Force concluded that the definition of such intermediate classes was problematic for the following reasons:

1. The resulting hierarchy is counterintuitive, particular for novices who are struggling to understand the conceptual model of inheritance. On the one hand, it seems reasonable for a **GRoundRect** to be a subclass of **GRect** because a rounded rectangle is still conceptually a rectangle. On the other hand, having **GOval** be a subclass of **GRect** (or even of **RectangularShape**) violates that intuition because an oval is not a rectangle.
2. The additional classes increase the complexity of the package structure, which reduces its accessibility to introductory students.
3. Organizing the shape classes according to their common method suites does not yield a linear hierarchy. A **GPolygon**, for example, has significant similarities to **GOval** and **GRect** in the sense that all three classes represent closed, fillable shapes. Despite those similarities, **GPolygon** responds to a different set of methods from the other two. As an example, the bounds for a **GOval** or a **GRect** are established by specifying the enclosing rectangle, which is not a useful strategy for the **GPolygon** class. Conversely, the **GImage** type takes rectangular bounds in exactly the way that **GOval** and **GRect** do but does not support filling. The fact that the hierarchy is not linear reduces the attractiveness of using intermediate classes, given that Java does not support multiple inheritance, which would allow classes to inherit methods from more than one superclass.

Given these concerns, the design we finally adopted has the simple class hierarchy shown in Figure 5-2. No intermediate classes exist in the package, thereby simplifying its conceptual structure. The common characteristics shared by sets of classes are instead specified by the interfaces **GFillable**, **GResizable**, and **GScalable**, which are discussed in section 5.3.

The shape classes included in the **acm.graphics** package do not by themselves constitute a complete graphics framework. In order to display a **GObject** on the screen, the student needs to add it to some sort of "canvas" that is part of the standard window system. That role is fulfilled in **acm.graphics** by the **GCanvas** class, which is described in section 5.5. We expect most adopters of the package to use the **GCanvas** that is automatically provided by the **GraphicsProgram** class, which is part of the **Program** class hierarchy described in Chapter 6. It is, however, straightforward to instantiate a **GCanvas** and then embed it inside a standard **JFrame**.

The **acm.graphics** package diagram shown in Figure 5-2 contains several additional classes beyond those that have already been described. The **GMath** class provides an extended set of mathematical methods that are useful for graphical programs and is described in section 5.4. The **GCompound** class makes it possible to define one **GObject** as a collection of others. This facility is described in section 5.7. The **GPen** class provides a simple mechanism for constructing line drawings that adopts a conceptual model of a pen drawing on the canvas. The **GTurtle** class is similar in many respects, but offers a somewhat more restricted (and arguably more intuitive) graphical model based on the "turtle graphics" paradigm described in Seymour Papert's *Mindstorms* [Papert80]. Each of these classes is described in section 5.9.

5.3 The `GObject` class

The question of what methods graphical objects should support required considerable time for the Task Force to reach closure. The first step toward determining the appropriate functionality for graphical objects was to survey the capabilities of the various graphics packages that had been proposed. From there, our next challenge was to understand how best to implement that functionality in a way that would be simple, consistent, and versatile. The next two subsections describe each of these activities in more detail, and the remaining subsections describe design decisions associated with specific shape classes.

Survey of operations provided by the graphics proposals

The proposals received by the Java Task Force differ in many respects, but nonetheless share a similar underlying model. In each of the submitted designs, the student creates pictures by instantiating graphical objects, installing them (sometimes implicitly by including the canvas as a parameter to the constructor) on a canvas of some sort, and then manipulating the display by sending messages to the objects to change their state and appearance.

A comparison of the methods available to graphical objects in the various models appears in Figure 5-3. As the table shows, every package made it possible for the student to specify the location, size, and color of an object. Beyond those basic capabilities, however, the methods supported by the individual packages differed in many ways. What the Task Force sought to do was to choose the best features of each proposed package, as long as those features could be implemented consistently and cleanly. The **`acm.graphics`** package supports all the features shown in Figure 5-3 with the exception

Figure 5-3. Comparison of features provided in the submissions

	objectdraw [Bruce04a]	OOPS [SandersK04a]	Stanford Graphics [Parlante04a]
Reset location and size	yes	yes	yes
Set object color	yes	yes	yes
Set fill	fill is fixed for class **`FramedRect`**/**`FilledRect`**	yes, with independent fill and border color	no
Set visibility	yes, but uses deprecated names **`show`** and **`hide`**	yes, but uses deprecated names **`show`** and **`hide`**	no
Check containment	yes	yes	no
Change *z*-ordering	yes, but you can't find out what the *z*-order is	no	no
Respond to mouse events	yes, but not with standard Java listeners	yes, but not with standard Java listeners	click-to-proceed only
Features not adopted for **`acm.graphics`**	**`overlaps`** method	supports rotation can set frame thickness	**`inset`** method all objects are nestable

of those in the last line, which were not included in the final package design for the following reasons:

- The objectdraw package allows an object to determine whether its bounding box overlaps that of another through the inclusion of an **overlaps** method. This method would be straightforward to add, but the functionality is easily obtainable by calling **intersects** on the bounding rectangles returned by **getBounds**. Making the user retrieve the bounding box has the advantage that the definition of overlap is then more explicit. The fact that two circles can "overlap" in objectdraw even when they are not touching (the bounding boxes will typically overlap before the circles do) seems likely to cause confusion.
- The ability to rotate arbitrary objects as in OOPS would be useful, but it depends on using **Graphics2D** for its internal operation. Using the **Graphics2D** interface instead of the more primitive **Graphics** interface has two downsides. First, writing code that uses it becomes more difficult, particularly in terms of its mathematical sophistication. That fact makes it unlikely that students would be able to write their own shape classes. Second, many browsers supply graphics contexts that don't implement **Graphics2D**, which limits the range of platforms on which such code can run. In the **acm.graphics** design, only the **GPolygon** class supports rotation, where it can be implemented without recourse to **Graphics2D**. The capability of changing the size of the frame seems to move the package away from the model provided by **java.awt.Graphics** and does not appear to generate benefits that are worth the cost.
- The Stanford Graphics proposal includes an **inset(dx, dy)** method that shrinks an object by the specified displacements along each edge. This method is therefore analogous to—but opposite in direction from—the **grow** method in **Rectangle** (suggesting at a minimum that it be replaced by one with a more standard name). The underlying function, however, seems of minor utility and can easily be achieved by calling

  ```
  gobj.setBounds(gobj.getBounds().grow(-dx, -dy));
  ```

  That proposal also makes every graphical object capable of containing other graphical objects. The notion of nested graphical objects seems defensible as an idea, but not in such an undisciplined way. The Task Force strategy for implementing nested objects is outlined in section 5.8.

When we adopted features from the various submissions, we often implemented them in a somewhat different way than the initial conception. Our principal reasons for making such changes were to provide more extensive functionality or to maintain greater consistency among the different parts of the package. The most important design decisions along these lines are as follows:

- *Mouse interaction*. In contrast to the packages submitted to the Task Force, the **acm.graphics** package allows graphical objects to respond to mouse events in exactly the way that components do. That mechanism and its underlying rationale are discussed in section 5.6.
- *Method naming*. The method names used in all **acm.graphics** classes follow as closely as possible the patterns for method naming used in modern releases of the JDK. Several of the submissions used method names that have been deprecated for the **Component** class, such as **show** and **hide**, which are particularly problematic. The **acm.graphics** package replaces **show** and **hide** with **setVisible**, which has been the standard name for this operation since JDK 1.1. Similarly, all mutator and accessor methods follow the Java Bean convention of using the **set** and **get** prefixes (plus **is** for booleans).

- *Filling*. The model for filling an object differs in the various submissions. The minimalist Stanford Graphics proposal ignores the issue entirely: all shapes are filled. OOPS uses a model similar to the one proposed here in which students can turn filling off for individual objects; it also provides methods to set the fill color independently from that of the frame, which has been adopted for **acm.graphics**. The objectdraw package provides two parallel sets of classes, one filled and one framed, which seems unnecessarily complicated.

Designing the functionality of the GObject class

One of the central issues in the design of the **GObject** class and the standard shape classes arises from the fact that the various shape classes have different behavioral characteristics and therefore require individualized sets of methods. Although it makes perfect sense to fill a **GRect** or **GOval**, filling is not appropriate for **GImage** or **GLabel**. Several members of the Java Task Force felt it was important to define each shape class so that it responded only to messages appropriate to graphical objects of that type. Such a design makes it possible to catch at compile time any attempts to invoke inappropriate methods. Adopting that principle ruled out an earlier design in which the **GObject** class defined an expansive set of methods, with each subclass either ignoring or generating runtime errors for methods that were inappropriate to that class.

The strategy that we eventually adopted was to include in **GObject** only those methods that are common to all graphical objects and to defer to each subclass the additional methods that make sense only in that domain. These additional methods are collected into standard suites identified by interfaces. As an example, the shape classes that implement **GFillable** respond to the methods **setFilled**, **isFilled**, **setFillColor**, and **getFillColor**. The public methods defined for all graphical objects appear in Figure 5-4, and the additional methods specified by interfaces appear in Figure 5-5.

Note that the collection of methods enumerated in Figure 5-4 includes a set of static methods to simplify trigonometric calculations. The reason for including these methods (which are also defined in the **GraphicsProgram** class defined in Chapter 6) is that many graphical figures are defined most naturally in terms of trigonometric specification. Unfortunately, these calculations are made more complicated for novices by the fact that the **Math** class introduces additional complexity. One of the problems is that using the mathematical functions in the **Math** class forces students to learn a new syntax for the invocation of static methods; if the student reasons by analogy to other invocations, the expression **Math.sin(theta)** seems to be sending the **sin** message to an object named **Math**, when in fact something quite different is occurring. A second problem is that the trigonometric functions in the **Math** class use radian measure instead of the degree-based geometry used in the graphics library. Adding the static methods in Figure 5-4 reduces the severity of these problems.

The GRect class and its subclasses

The simplest and most intuitive of the shape classes defined in **acm.graphics** is the **GRect** class, which represents a rectangular box. This class implements the **GFillable**, **GResizable**, and **GScalable** interfaces, but otherwise includes no other methods except its constructors.

The one important design decision to observe in the implementation not only of **GRect** but the other shape classes as well concerns the relationship between the screen area covered by the filled and outlined versions of the corresponding shapes. In the standard **Graphics** class, a rectangle outlined using **drawRect** does not cover the same pixels as the one generated by **fillRect**; the filled rectangle is one pixel smaller in each

Figure 5-4. Methods common to all graphical objects

Methods to set and retrieve standard properties of the object

`void setLocation(double x, double y)` *or* **`setLocation(GPoint pt)`**
Sets the location of this object to the specified point.

`void move(double dx, double dy)`
Moves the object using the displacements **`dx`** and **`dy`**.

`void movePolar(double r, double theta)`
Moves the object **`r`** units in direction **`theta`**, measured in degrees.

`double getX()`
Returns the x-coordinate of the object.

`double getY()`
Returns the y-coordinate of the object.

`double getWidth()`
Returns the width of the object.

`double getHeight()`
Returns the height of the object.

`GPoint getLocation()`
Returns the location of this object as a **`GPoint`**.

`GDimension getSize()`
Returns the size of this object as a **`GDimension`**.

`GRectangle getBounds()`
Returns the bounding box of this object (the smallest rectangle that covers the figure).

`boolean contains(double x, double y)` *or* **`contains(GPoint pt)`**
Checks to see whether a point is inside the object.

`void setColor(Color c)`
Sets the color of the object.

`Color getColor()`
Returns the object color. If this value is **`null`**, the package uses the color of the container.

`void setVisible(boolean visible)`
Sets whether this object is visible.

`boolean isVisible()`
Returns **`true`** if this object is visible.

Methods to allow objects to respond to events

`void addMouseListener(MouseListener listener)`
Specifies a listener to process mouse events for this graphical object.

`void removeMouseListener(MouseListener listener)`
Removes the specified mouse listener from this graphical object.

`void addMouseMotionListener(MouseMotionListener listener)`
Specifies a listener to process mouse motion events for this graphical object.

`void removeMouseMotionListener(MouseMotionListener listener)`
Removes the specified mouse motion listener from this graphical object.

`void addActionListener(ActionListener listener)`
Specifies a listener to process action events for this graphical object.

`void removeActionListener(ActionListener listener)`
Removes the specified action listener from this graphical object.

`void fireActionEvent(String actionCommand)` *or* **`fireActionEvent(ActionEvent e)`**
Notifies any registered listeners that an action event has occurred.

Miscellaneous methods

`abstract void paint(Graphics g)`
Paints the object using the graphics context **`g`**. This operation is usually invisible to students.

`void pause(double milliseconds)`
Delays the caller for the specified interval (like **`Thread.sleep`** but without any exceptions).

`GContainer getParent()`
Returns the parent of this object, which is the container in which it is enclosed.

`void sendToFront()` *or* **`sendToBack()`**
Moves this object to the front (or back) of its container.

`void sendForward()` *or* **`sendBackward()`**
Moves this object forward (or backward) one position in the *z* ordering.

Figure 5-5. Additional methods specified by interfaces

`GFillable` (implemented by **`GArc`**, **`GOval`**, **`GPen`**, **`GPolygon`**, and **`GRect`**)

`void setFilled(boolean fill)` Sets whether this object is filled (**`true`** means filled, **`false`** means outlined).
`boolean isFilled()` Returns **`true`** if the object is filled.
`void setFillColor(Color c)` Sets the color used to fill this object. If the color is **`null`**, filling uses the color of the object.
`Color getFillColor()` Returns the color used to fill this object.

`GResizable` (implemented by **`GImage`**, **`GOval`**, and **`GRect`**)

`void setSize(double width, double height)` Changes the size of this object to the specified width and height.
`void setSize(GDimension size)` Changes the size of this object as specified by the **`GDimension`** parameter.
`void setBounds(double x, double y, double width, double height)` Changes the bounds of this object as specified by the individual parameters.
`void setBounds(GRectangle bounds)` Changes the bounds of this object as specified by the **`GRectangle`** parameter.

`GScalable` (implemented by **`GArc`**, **`GCompound`**, **`GImage`**, **`GLine`**, **`GOval`**, **`GPolygon`**, and **`GRect`**)

`void scale(double sf)` Resizes the object by applying the scale factor in each dimension, leaving the location fixed.
`void scale(double sx, double sy)` Scales the object independently in the *x* and *y* dimensions by the specified scale factors.

dimension because the filled figure does not include the framing pixels on the right and bottom edges of the outline. In the **`acm.graphics`** package, a filled shape is, in essence, both framed and filled, and therefore covers exactly the same pixels in either mode. This definition was necessary to support separate fill and frame colors and is also likely to generate less confusion for students.

The two specialized forms of rectangles—**`GRoundRect`** and **`G3DRect`**—appear in the hierarchy as subclasses of **`GRect`**. The purpose of introducing this additional layer in the hierarchy (which could equally easily have been implemented as a flat collection of the various shapes) was to provide an intuitively compelling illustration of the nested hierarchies. Just as all the shape classes are graphical objects (and therefore subclasses of **`GObject`**), the **`GRoundRect`** and **`G3DRect`** classes are graphical rectangles (and therefore subclasses of **`GRect`**). Organizing the hierarchy in this way emphasizes the *is-a* relationship that defines subclassing.

The **`GRoundRect`** and **`G3DRect`** classes include additional method definitions that allow clients to set and retrieve the properties that define their visual appearance. For **`GRoundRect`**, these properties are the specifications of corner curvature, expressed exactly as in the **`drawRoundRect`** method; for **`G3DRect`**, the additional methods allow the client to indicate whether the rectangle should appear raised.

The `GOval` class

The **`GOval`** class represent an elliptical shape which is defined so that the parameters of its constructor match the arguments to the **`drawOval`** and **`fillOval`** methods in the standard Java **`Graphics`** class. Once students understand that an oval is specified by its bounding rectangle and not by its center point, using the **`GOval`** class is quite straightforward. Like **`GRect`**, the **`GOval`** class implements the **`GFillable`**, **`GResizable`**, and **`GScalable`** interfaces but otherwise includes no methods that are specific to the class.

The complication in **GOval** lies in its implementation. In many implementations of the Java **Graphics** class across a range of platforms, the pixels drawn by a call to **fillOval** do not precisely match the pixels in the interior of the shape drawn by **drawOval**. As a result, it is often the case that a filled **GOval** rendered using the standard methods from the Java **Graphics** class ends up with unpainted pixels just inside the boundary. When we discovered this problem, the Task Force decided that we had to fix things so that ovals and arcs were always completely filled. Adopters of the package would undoubtedly regard the unsatisfying appearance of ovals and arcs as a bug in the ACM libraries, even if the actual source of that bug was inside the standard Java libraries.

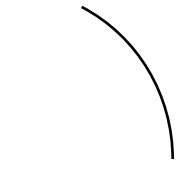
To circumvent this rendering bug, the **acm.graphics** package ordinarily draws **GOval**s and **GArc**s and **GRoundRect**s using polygons rather than the native methods. The result is a much more consistent screen image, potentially at a small cost in execution efficiency. The **GCanvas** class includes an option setting to disable this behavior.

The GLine class

The **GLine** class makes it possible for clients of the **acm.graphics** package to construct arbitrary line drawings and seems to be an important capability for any graphics package to support, even though lines are different in many respects from the other shape classes. This difference in conceptual model is reflected in the methods supported by the **GLine** class, which implements **GScalable** (relative to the starting point of the line), but not **GFillable** or **GResizable**. It is, moreover, important to modify the notion of containment for lines, since the idea of being within the boundary of the line is not well defined. In the abstract, of course, a line is infinitely thin and therefore contains no points in its interior. In practice, however, it makes sense to define a point as being contained within a line if it is "close enough" to be considered as part of that line. The motivation for this definition comes from the fact that one of the central uses of the **contains** method is to determine whether a mouse click applies to a particular object. As is the case in any drawing program, selecting a line with the mouse is indicated by clicking within some pixel distance of the abstract line. In the **acm.graphics** package, that distance is specified by the constant **LINE_TOLERANCE** in the **GLine** class, which is defined to be a pixel and a half.

Given that some students are likely to have preconceptions that the "width" of a line refers to its thickness, it is important to emphasize that the methods **getWidth**, **getHeight**, and **getSize** are defined for the **GLine** class in precisely the way that they are for all other **GObject**s. Thus, these methods return the appropriate dimensions of the bounding box that encloses the line.

The **GLine** class also contains several methods for determining and changing the endpoints of the line. The **setStartPoint** method allows clients to change the first endpoint of the line without changing the second; conversely, **setEndPoint** gives clients access to the second endpoint without affecting the first. These methods are therefore different in their operation from **setLocation**, which moves the entire line without changing its length and orientation.


Unfilled arc

The GArc class

The **GArc** class raises a variety of interesting design issues, particularly in terms of seeking to understand the relationship between a filled arc and an unfilled one. To ease the transition to the standard Java graphics model, we chose to implement the **GArc** class so that its operation was as consistent as possible with the **drawArc** and **fillArc** methods in the standard **Graphics** class. An unfilled arc, therefore, displays only the pixels on the arc boundary; a filled arc fills the entire wedge-shaped region, as shown on the right.


Filled arc

Adopting the standard Java interpretation of arc filling has implications for the design of the **GArc** object. Most notably, the **contains** method for the **GArc** class returns a result that depends on whether the arc is filled. For an unfilled arc, containment implies that the arc point is actually on the arc, subject to the same interpretation of "closeness" as described for lines in the preceding section. For a filled arc, containment implies inclusion in the wedge. This definition of containment is necessary to ensure that mouse events are transmitted to the arc in a way that matches the user's intuition.

To match Java's interpretation, the bounds for a **GArc** object are indicated by specifying the enclosing rectangle along with the starting angle and sweep extent of the arc. The constructor for **GArc** therefore has the following signature:

```
public GArc(double x, double y, double width, double height,
            double start, double sweep)
```

Given that the constructor specifies rectangular bounds, it would at first seem appropriate to have **GArc** implement **GResizable** and support operations like **setSize** and **setBounds**. Doing so, however, could easily create confusion. In contrast to the situation for ovals, rectangles, and images, the rectangle used to define the size of a **GArc** is not the same as the bounding box for that object. Thus, supporting the method **setBounds** in the obvious way would generate the surprising situation that a subsequent call to **getBounds** would typically return a smaller rectangle.

To avoid this possible source of confusion, **GArc** does not implement **GResizable**, but instead implements the methods **setFrameRectangle** and **getFrameRectangle**. These methods allow clients to change or retrieve the rectangle that defines the arc while minimizing the likelihood of confusion with **getBounds**, which is defined in the **acm.graphics** package to return the bounding box.

The **GArc** class includes methods that enable clients to manipulate the angles defining the arc (**setStartAngle**, **getStartAngle**, **setSweepAngle**, and **getSweepAngle**) as well as methods to return the points at the beginning and end of the arc (**getStartPoint** and **getEndPoint**).

The GImage class

The **GImage** class turns out to be relatively easy to design. In Java, the bounds of an image are specified by a rectangle, which makes it easy to define **GImage** class so that it implements both **GResizable** and **GScalable**, but not **GFillable**, which doesn't make sense in the context of an image. Resizing or scaling an image has the effect of stretching or compressing the pixels in the image and is implemented by the standard **drawImage** code in **java.awt.graphics**.

One interesting question that arose in the Task Force discussion was what effect, if any, the **setColor** method should have for the **GImage** class. Initially, some members of the Task Force had suggested that the inclusion of the **GImage** class in the hierarchy meant that we needed to define an interface called **GColorable** that all shape classes *except* **GImage** would implement. It turns out, however, that the **drawImage** method in the **Graphics** class offers a straightforward interpretation of the color of an image as the background color that shows through any transparent or translucent pixels. By using the color of the object to specify this background, clients can use **setColor** to tint an image as long as that image includes pixels that are less than fully opaque. If the image contains only opaque pixels, setting its color has no effect.

In contrast to the design of the abstract interface, the implementation of **GImage** involves some complexity, mostly in terms of having the package know how to load

images from the environment. The implementation of search paths for images and the associated mechanisms to ensure that images work in both applet and application contexts is described in the discussion of the **MediaTools** class in Chapter 8.

The GLabel class

The **GLabel** class differs in almost every respect from the other classes in the **acm.graphics** package. As we have chosen to implement it, **GLabel** implements none of the standard **GFillable**, **GResizable**, and **GScalable** interfaces. The size of a label is determined instead by setting its font, just as is done in Java either for the **drawString** method in **Graphics** or for the **JLabel** class in the Swing hierarchy. Objects of type **GLabel**, therefore, must respond to the methods **setFont** and **getFont**, just as a **JLabel** does. Moreover, to simplify positioning text strings on the display, the **GLabel** class includes methods to return information about the font metrics (**getAscent**, **getDescent**, and **getLeading**). The width and height of the entire string are available through the standard **GObject** methods **getWidth** and **getHeight**. Making this information accessible directly makes it possible for students to use the **GLabel** class without having to understand how **FontMetrics** works.

To provide yet another simplification for introductory students, the **GLabel** class includes an overloaded definition of **setFont** that takes a string rather than a **Font** object. The string is interpreted in the manner specified by **Font.decode**. Thus, to set the font of a **GLabel** variable called **label** to be an 18-point, boldface, sans-serif font (typically Helvetica), the student could simply write

```
label.setFont("SansSerif-bold-18");
```

As an extension to the semantics of **Font.decode**, the **setFont** method interprets an asterisk in the string as signifying the previous value. Thus, students can set the style of a label to be italic by writing

```
label.setFont("*-italic-*");
```

without changing its family or size.

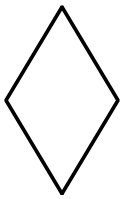
The GPolygon class

The shape that represents the greatest deviation from the traditional Java model is the **GPolygon** class, which is used to draw closed polygonal shapes. Although the inspiration for **GPolygon** comes from the **drawPolygon** method in the standard **Graphics** class, the **acm.graphics** package does not adopt the same model as **java.awt**, for a variety of sound pedagogical reasons:

1. The **Polygon** class in **java.awt** uses integer coordinates. This design is incompatible with the overall paradigm used in the **acm.graphics** package in which coordinate values are specified using **double**s. The use of **double**s is arguably more important for the **GPolygon** class than for any other part of the package because it is particularly hard for students to define a closed polygon using integer arithmetic.
2. The **Polygon** class uses parallel arrays to hold the points. This design is inappropriate for the **acm.graphics** package, both because novices may not have learned about arrays and because it seems clearly more appropriate to use an array of points here.
3. The vertices in a **Polygon** object are specified using absolute coordinates. This design is much harder to use than one in which the polygon has its own origin, and the vertices are defined relative to that origin.

The **GPolygon** class adopts a model that avoids each of these problems. The basic idea is that the student creates a **GPolygon**, which is initially empty. Starting with that empty

polygon, the student adds new vertices using a set of methods (**addVertex**, **addEdge**, and **addPolarEdge**) that add vertices to the polygon.



These methods are easiest to illustrate by example. The simplest method to explain is **addVertex(x, y)**, which adds a vertex at the point (**x**, **y**) relative to the location of the polygon. For example, the following code defines a diamond-shaped polygon in terms of its vertices, as shown in the diagram at the right:

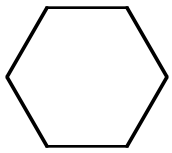
```
GPolygon diamond = new GPolygon();
diamond.addVertex(-22, 0);
diamond.addVertex(0, 36);
diamond.addVertex(22, 0);
diamond.addVertex(0, -36);
```

The diamond is drawn so that its center is at the point (0, 0) in the coordinate space of the polygon. Thus, if you were to add **diamond** to a **GCanvas** and set its location to the point (300, 200), the diamond would be centered at that location.

The **addEdge(dx, dy)** method is similar to **addVertex**, except that the parameters specify the displacement from the previous vertex to the current one. One could therefore draw the same diamond by making the following sequence of calls:

```
GPolygon diamond = new GPolygon();
diamond.addVertex(-22, 0);
diamond.addEdge(22, 36);
diamond.addEdge(22, -36);
diamond.addEdge(-22, -36);
diamond.addEdge(-22, 36);
```

Note that the first vertex must still be added using **addVertex**, but that subsequent ones can be defined by specifying the edge displacements. Moreover, the final edge is not explicitly necessary because the polygon is automatically closed before it is drawn.



Some polygons are easier to define by specifying vertices; others are more easily represented by edges. For many polygonal figures, however, it is even more convenient to express edges in polar coordinates. This mode of specification is supported in the **GPolygon** class by the method **addPolarEdge**, which is identical to **addEdge** except that its arguments are the length of the edge and its direction expressed in degrees counterclockwise from the +x axis. This method makes it easy to create figures with more sophisticated structure, such as the centered hexagon generated by the following method (as shown on the right using 36 pixels as the value of **side**):

```
GPolygon createHexagon(double side) {
   GPolygon hex = new GPolygon();
   hex.addVertex(-side, 0);
   for (int i = 0; i < 6; i++) {
      hex.addPolarEdge(side, 60 - i * 60);
   }
   return hex;
}
```

The **GPolygon** class implements the **GFillable** and **GScalable** interfaces, but not **GResizable**. It also supports the method **rotate(theta)**, which rotates the polygon theta degrees counterclockwise around its origin.

One of the most important applications of the **GPolygon** class consists of creating extended **GObject** subclasses that have a particular shape. That technique is described in section 5.8.

5.4 Static definitions and the GMath class

In the February 2005 release of the **acm.graphics** package, the **GObject** class included a number of static constant definitions (primarily the color names such as **RED**) along with a collection of static methods intended to make it easier for students to calculate trigonometric relationships for angles measured in degrees. In the current release, the constants have been eliminated, and the static methods have been moved into a separate class called **GMath**, which exports the methods shown in Figure 5-6. The reasons that led us to make these change are as follows:

- The small simplification one gains by allowing students to write **RED** instead of **Color.RED** is overwhelmed by the problems that arise once students begin to write their own classes and no longer have direct access to the constant definitions in **GObject**. Based on the experience of the early users, it is far better to have students use the fully qualified names from the beginning
- Declaring static constants and methods requires code duplication, given that students expect these definitions to appear not only in **GObject**, but also in the **GCanvas** and **GraphicsProgram** classes as well.
- The **GMath** class is more closely parallel to the standard **Math** class in **java.lang**. Although having to write **GMath.sinDegrees(theta)** is a little more cumbersome than **sinDegrees(theta)**, students quickly get used to including the name of the class in static method calls. Such calls do not seem to represent a significant source of confusion.
- Including static methods in class definitions made it more difficult to use those classes with the BlueJ environment. To minimize this incompatibility, we moved most static methods into classes of their own.
- Java Standard Edition 5.0 includes the new **import static** feature, which makes it possible to eliminate the class name on static methods and fields. As Java SE 5.0 becomes more widespread, the problem of forcing students to write qualified constant and method names will presumably begin to disappear.

Figure 5-6. Static methods in the GMath class

Trigonometric methods in degrees

static double sinDegrees(double angle) Returns the trigonometric sine of an angle measured in degrees.
static double cosDegrees(double angle) Returns the trigonometric cosine of an angle measured in degrees.
static double tanDegrees(double angle) Returns the trigonometric tangent of an angle measured in degrees.
static double toDegrees(double radians) Converts an angle from radians to degrees.
static double toRadians(double degrees) Converts an angle from degrees to radians.

Polar coordinate conversion methods

double distance(double x, double y) Returns the distance from the origin to the point (**x**, **y**).
double distance(double x0, double y0, double x1, double y1) Returns the distance between the points (**x0**, **y0**) and (**x1**, **y1**).
double angle(double x, double y) Returns the angle between the origin and the point (**x**, **y**), measured in degrees.

Convenience method for rounding to an integer

static int round(double x) Rounds a **double** to the nearest **int** (rather than to a **long** as in the **Math** class).

5.5 The GCanvas class

The **GCanvas** class provides the link between the world of graphical objects and the Java windowing system. Conceptually, the **GCanvas** class acts as a container for graphical objects and allows clients of the package to add and remove elements of type **GObject** from an internal display list. When the **GCanvas** is repainted, it forwards **paint** messages to each of the graphical objects it contains. Metaphorically, the **GCanvas** class acts as the background for a collage in which the student, acting in the role of the artist, positions shapes of various colors, sizes, and styles.

The methods supported by the implementation of the **GCanvas** class in the **acm.graphics** package are shown in Figure 5-7.

Figure 5-7. Public methods in the GCanvas class

Constructor

GCanvas() Creates a new **GCanvas** that contains no objects.

Methods to add and remove graphical elements from a canvas

void add(GObject gobj) Adds a graphical object to the canvas.
void add(GObject gobj, double x, double y) *or* **add(GObject gobj, GPoint pt)** Adds a graphical object to the canvas at the specified location.
void remove(GObject gobj) Removes the specified graphical object from the canvas.
void removeAll() Removes all graphical objects and components from the canvas.

Methods to add and remove components from the canvas

void add(Component c) Adds a component to the canvas, where it floats above the graphical elements.
void add(Component c, double x, double y) *or* **add(Component c, GPoint pt)** Adds a component to the canvas at the specified location.
void remove(Component c) Removes the specified component from the canvas.

Methods to determine the contents of the canvas

Iterator iterator() Returns an iterator that runs through the graphical elements from back to front.
Iterator iterator(int direction) Returns an iterator that runs in the specified direction (**BACK_TO_FRONT** or **FRONT_TO_BACK**).
int getElementCount() Returns the number of graphical objects contained on the canvas.
GObject getElement(int i) Returns the graphical object at the specified index, numbering from back to front.
GObject getElementAt(double x, double y) *or* **getElementAt(GPoint pt)** Returns the topmost object containing the specified point, or **null** if no such object exists.

Option setting

void setAutoRepaintFlag(boolean state) Determines whether repaint requests occur automatically when a graphical object changes.
boolean getAutoRepaintFlag() Returns whether automatic repainting is enabled.
void setNativeArcFlag(boolean state) Determines whether arcs and ovals are rendered using polygons (**false**) or the JDK (**true**).
boolean getNativeArcFlag() Returns whether arcs and ovals are rendered using polygons (**false**) or the JDK (**true**).

Adding and removing objects from a canvas

The first set of methods in Figure 5-7 is straightforward given the fact that a **GCanvas** is conceptually a container for **GObject** values. The container metaphor explains the functionality provided by the **add**, **remove**, and **removeAll** methods in Figure 5-7, which are analogous to the identically named methods in **JComponent** or **Container**.

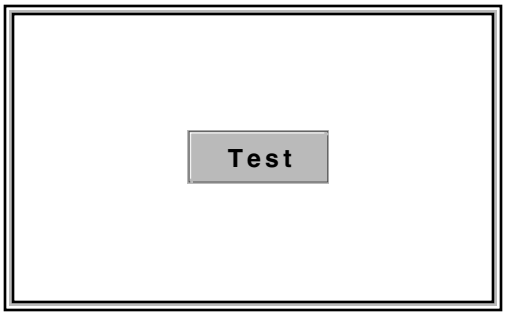
Interestingly, none of the three submitted proposals chose to implement the relationship between graphical objects and their canvases using this metaphor of adding objects to a container. In each of the packages, the container is specified—implicitly in the case of OOPS—at the time the graphical object is constructed. Such an design seems less than ideal for three reasons. First, the strategy runs counter to the model provided by components and containers and will therefore provide no help when the student moves on Swing. Second, such a restriction seems to violate the intuitive notion of a collage, in which one creates freestanding objects and then pastes them on a canvas. Third, you often need to have the object to know where to put it in the container, as illustrated by the following code, which centers a **GLabel** object in the **GCanvas** named **gc**:

```
GLabel glabel = new GLabel("Hello");
double x = (gc.getWidth() - glabel.getWidth()) / 2;
double y = (gc.getHeight() - glabel.getHeight()) / 2;
gc.add(glabel, x, y);
```

It's not clear how you would put the object in the right place without being able to instantiate the freestanding object. The best you could do would be add it somewhere else (possibly invisibly, as in OOPS) and then move it to the center.

Adding and removing components to a canvas

The next set of methods from Figure 5-7 provide analogous capabilities for adding and removing an AWT or Swing component from a **GCanvas**. Students who have gotten into the habit of adding a **GObject** to a **GCanvas** will find it natural to add some other graphical object—a **JButton** perhaps—to a **GCanvas** as well. As an example, the code




```
JButton test = new JButton("Test");
test.setLocation(75, 50);
gc.add(test);
```

produces a display such as the one shown on the right. The **JButton** is fully active and can accept listeners, just as it can in any other context.

Given that **GCanvas** is a subclass of **Container**, the simple versions of the **add** and **remove** methods already exist by inheritance, although it is useful to reimplement these methods slightly to give them more useful semantics. The first change is that **GCanvas** objects should use **null** as their default layout manager so that the positioning of the components is under client control, just as is true for graphical objects. Second, the **add** method must check the size of the component and set it to its preferred size if its bounds are empty. These simple changes seem to do exactly what students would want.

Determining the contents of a canvas

The third set of methods in Figure 5-7 make it possible for clients to figure out what graphical elements have been added to a **GCanvas** and provide a couple of approaches for doing so. The strategy that fits best with modern disciplines for using Java is to use an iterator that runs through the elements of a collection. The **iterator** provides that functionality in the conventional Java form, making it possible to write

```
for (Iterator i = gc.iterator(); i.hasNext(); ) {
   GObject gobj = (GObject) i.next();
   . . . code for this element . . .
}
```

or, in Java 5.0, the abbreviated form

```
for (GObject gobj : gc) {
   . . . code for this element . . .
code
```

Unfortunately, there are two reasonable orders in which to process the elements of a **GCanvas**. If you are doing something analogous to painting, you want to go through the elements from back to front so that the elements at the front of the stacking order correctly obscure those behind them. On the other hand, if you are fielding mouse events, you want to go through the elements from front to back so that primacy on receiving the event goes to the graphical objects in front. (Both painting and mouse-event dispatching are provided by the package, so students need not code these two mechanisms, but might sometime want to do something similar.) To permit both strategies, the **iterator** method takes an optional argument that can be either **BACK_TO_FRONT** or **FRONT_TO_BACK** to specify the desired order along the *z*-axis. The default is **BACK_TO_FRONT**.

To support instructors who choose not to introduce iterators as early as they cover the **acm.graphics** package, the **GCanvas** class also exports the methods **getElementCount** and **getElement**, which make it possible to sequence through the elements in any order.

5.6 Responding to mouse events

The evolutionary history of Java provides two distinct models for responding to mouse events. The model that was defined for JDK 1.0 was based on callback methods whose behavior was defined through subclassing. Thus, to detect a mouse click on a component, you need to define a subclass that overrides the definition of **mouseDown** with a new implementation having the desired behavior. That model was abandoned in JDK 1.1 in favor of a new model based on event listeners. Under this paradigm, you add the necessary suite of methods to implement **MouseListener** to some class and then add some object of that class as a listener on the component whose events you want to catch. These models are quite different, and there is no obvious sense in which learning one helps prepare you to learn the other.

The two submitted packages that offer mouse support each provide a mechanism for responding to mouse events whose design use a callback model rather than listeners. The graphical object types in the OOPS proposal, for example, define methods for **mousePressed**, **mouseReleased**, **mouseClicked**, and **mouseDragged** (but not, interestingly, **mouseMoved**, **mouseEntered**, or **mouseExited**) that are invoked whenever the specified event is detected over that object. In objectdraw, mouse events generated on the graphical canvas are eventually handled in methods contained in class **WindowController**, which is the extension of **JApplet** that contains that canvas. Students write a class extending **WindowController** and then override the event-handling methods whose behavior they wish to specify.

There are certainly some ways in which the callback strategy is simpler for novices. At the same time, we believe that many potential adopters will insist on using Java's event listeners to ensure that students can more easily make the transition to the standard Java paradigm. It is, of course, possible to use listeners under any of these packages simply by adding the appropriate listener to whatever class plays the role of the canvas

and fielding the events from there. The difficulty, however, is that the most intuitive model for event handling—and the one that corresponds to the behavior of the **JComponent** hierarchy—would have the graphical *objects* be the source for the events rather than the *canvas* in which those objects are contained.

To illustrate the distinction, it is useful to consider how one might use the **mouseEntered** and **mouseExited** methods in a canvas-based listener. Although one can construct situations in which you need to notice that the mouse has entered the canvas, what most students are likely to want is some way of detecting when the mouse has entered one of their objects. Such a capability enables, for example, the simulation of roll-over buttons, which students seem to love.

To support that style of interaction, the **acm.graphics** enables mouse event listening for the **GObject** class. As an illustration of this behavior, the methods

```
public void mouseEntered(MouseEvent e) {
   ((GObject) e.getSource()).setColor(Color.RED);
}

public void mouseExited(MouseEvent e) {
   ((GObject) e.getSource()).setColor(Color.BLACK);
}
```

define listener methods that implement rollover behavior for any **GObject** that adds a mouse listener containing this code. Moving the mouse into the containment region of the object turns the object red; moving the mouse out again turns it black.

Figure 5-8 provides a more extensive example of the use of mouse events in objects by presenting the complete code for an application that installs two shapes—a red rectangle and a green oval—and allows the user to drag either shape using the mouse. The program also illustrates *z*-ordering by moving the current object to the front on a mouse click. This demo is available as an applet on the JTF web site.

From the programmer's perspective, the design of the event-handling mechanism in the **acm.graphics** package remains unchanged from what it was at the time of the February 2005 draft. The underlying implementation, however, is considerably more efficient. In the initial release, the location of each mouse event was checked against every **GObject** in the canvas to see if it was interested. The new release maintains a separate data structure of **GObject**s that are listening for mouse events, resulting in a much faster search.

5.7 The GCompound class

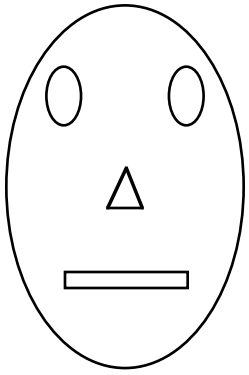
The shape classes that appear at the bottom of Figure 5-2 represent "atomic" shapes that have no internal components. Although the **GCanvas** class makes it possible to position these shapes on the display, it is often useful to assemble several atomic shapes into a "molecule" that you can then manipulate as a unit. The need to construct this type of compound units provides the motivation behind the inclusion of the **GCompound** class, in the **acm.graphics** package. The methods available for **GCompound** are in some sense the union of those available to the **GObject** and **GCanvas** classes. As a **GObject**, a **GCompound** responds to method calls like **setLocation** and **scale**; as an implementer (like **GCanvas**) of the **GContainer** interface, it supports methods like **add** and **remove**.

Figure 5-8. Program to illustrate mouse dragging in an application

```
/** This class displays a mouse-draggable rectangle and oval */
public class ObjectDragExample extends JFrame
   implements MouseListener, MouseMotionListener {

/** Initializes and installs the objects in the JFrame */
   public ObjectDragExample() {
      gc = new GCanvas();
      GRect rect = new GRect(100, 100, 150, 100);
      rect.setFilled(true);
      rect.setColor(Color.RED);
      rect.addMouseListener(this);
      rect.addMouseMotionListener(this);
      gc.add(rect);
      GOval oval = new GOval(300, 115, 100, 70);
      oval.setFilled(true);
      oval.setColor(Color.GREEN);
      oval.addMouseListener(this);
      oval.addMouseMotionListener(this);
      gc.add(oval);
      getContentPane().add(BorderLayout.CENTER, gc);
      setSize(500, 300);
   }

/** Called on mouse press to record the coordinates of the click */
   public void mousePressed(MouseEvent e) {
      last = new GPoint(e.getPoint());
   }

/** Called on mouse drag to reposition the object */
   public void mouseDragged(MouseEvent e) {
      GObject gobj = (GObject) e.getSource();
      GPoint pt = new GPoint(e.getPoint());
      gobj.move(pt.getX() - last.getX(), pt.getY() - last.getY());
      last = pt;
   }

/** Called on mouse click to move this object to the front */
   public void mouseClicked(MouseEvent e) {
      ((GObject) e.getSource()).sendToFront();
   }

   public void mouseReleased(MouseEvent e) { }
   public void mouseEntered(MouseEvent e) { }
   public void mouseExited(MouseEvent e) { }
   public void mouseMoved(MouseEvent e) { }

/** Standard entry point for the application */
   public static void main(String[] args) {
      new ObjectDragExample().show();
   }

/* Private state */
   private GCanvas gc;
   private GPoint last;

}
```

A simple example of GCompound

To understand how the **GCompound** class works, it is easiest to start with a simple example. Imagine that you wanted to assemble the following face on the canvas:



For the most part, this figure is easy to create. All you need to do is create a new **GOval** for the head, two **GOval**s for the eyes, a **GRect** for the mouth, and a **GPolygon** for the nose. If you put each of these objects on the canvas individually, however, it will be hard to manipulate the face as a whole. Suppose, for example, that you wanted to move the face around as a unit. Doing so would require moving every piece independently. It would be better simply to tell the entire face to move.

The code in Figure 5-9 uses the **GCompound** class to define a **GFace** class that contains the necessary components. These components are created and then added in the appropriate places as part of the **GFace** constructor. Once you have defined the class, you could construct a new **GFace** object and add it to the center of the canvas using the following code:

```
GFace face = new GFace(100, 150);
add(face, getWidth() / 2, getHeight() / 2);
```

The GCompound coordinate system

The general paradigm for using **GCompound** is to create an empty instance of the class and then to add other graphical objects to it. The coordinates at which these objects appear are expressed relative to the origin of the **GCompound** itself, and not to the canvas in which the compound will eventually appear. This strategy means that you can add a compound object to a canvas and the move all its elements as a unit simply by setting the location of the compound. Thus, once you had created the **GFace** object described in the preceding section, you could move the entire face 20 pixels to the right by executing the following method:

```
face.move(20, 0);
```

In some cases—most notably when you need to translate the coordinates of a mouse click, which are expressed in the global coordinate space of the canvas—it is useful to be able to convert coordinates from the local coordinate space provided by the **GCompound** to the coordinate space of the enclosing canvas, and vice versa. These conversions are implemented by the methods

```
GPoint getCanvasPoint(localPoint)
```

and

```
GPoint getLocalPoint(canvasPoint)
```

The complete set of methods available for the **GCompound** class appears in Figure 5-10.

Figure 5-9. Program to create a `GFace` class by extending `GCompound`

```
/*
 * File: GFace.java
 * ----------------
 * This file defines a compound GFace class.
 */

import acm.graphics.*;

/**
 * This code defines a new class called GFace, which is a compound
 * object consisting of an outline, two eyes, a nose, and a mouth.
 * The origin point for the face is the center of the figure.
 */
public class GFace extends GCompound {

/** Construct a new GFace object with the specified dimensions. */
   public GFace(double width, double height) {
      head = new GOval(width, height);
      leftEye = new GOval(EYE_WIDTH * width, EYE_HEIGHT * height);
      rightEye = new GOval(EYE_WIDTH * width, EYE_HEIGHT * height);
      nose = createNose(NOSE_WIDTH * width, NOSE_HEIGHT * height);
      mouth = new GRect(MOUTH_WIDTH * width, MOUTH_HEIGHT * height);
      add(head, -width / 2, -height / 2);
      add(leftEye, -0.25 * width - EYE_WIDTH * width / 2,
                   -0.25 * height - EYE_HEIGHT * height / 2);
      add(rightEye, 0.25 * width - EYE_WIDTH * width / 2,
                    -0.25 * height - EYE_HEIGHT * height / 2);
      add(nose, 0, 0);
      add(mouth, -MOUTH_WIDTH * width / 2,
                 0.25 * height - MOUTH_HEIGHT * height / 2);
   }

/** Creates a triangle for the nose */
   private GPolygon createNose(double width, double height) {
      GPolygon poly = new GPolygon();
      poly.addVertex(0, -height / 2);
      poly.addVertex(width / 2, height / 2);
      poly.addVertex(-width / 2, height / 2);
      return poly;
   }

/* Constants specifying feature size as a fraction of the head size */
   private static final double EYE_WIDTH    = 0.15;
   private static final double EYE_HEIGHT   = 0.15;
   private static final double NOSE_WIDTH   = 0.15;
   private static final double NOSE_HEIGHT  = 0.10;
   private static final double MOUTH_WIDTH  = 0.50;
   private static final double MOUTH_HEIGHT = 0.03;

/* Private instance variables */
   private GOval head;
   private GOval leftEye, rightEye;
   private GPolygon nose;
   private GRect mouth;
}
```

Figure 5-10. Public methods in the `GCompound` class (plus those inherited from `GObject`)

Constructor

`GCompound()` Creates a new **`GCompound`** that contains no objects.

Methods specified by `GScalable`

`void scale(double sf)` Resizes the compound by applying the scale factor in each dimension relative to its origin.
`void scale(double sx, double sy)` Resizes the compound independently in the *x* and *y* dimensions by the specified scale factors.

Methods to add and remove graphical objects from a compound

`void add(GObject gobj)` Adds a graphical object to the compound.
`void add(GObject gobj, double x, double y)` *or* **`add(GObject gobj, GPoint pt)`** Adds a graphical object to the compound at the specified location.
`void remove(GObject gobj)` Removes the specified graphical object from the compound.
`void removeAll()` Removes all graphical objects and components from the compound.

Methods to determine the contents of the compound

`Iterator iterator()` Returns an iterator that runs through the graphical objects from back to front.
`Iterator iterator(int direction)` Returns an iterator that runs in the specified direction (**`BACK_TO_FRONT`** or **`FRONT_TO_BACK`**).
`int getElementCount()` Returns the number of graphical objects contained in the compound.
`GObject getElement(int i)` Returns the graphical object at the specified index, numbering from back to front.
`GObject getElementAt(double x, double y)` *or* **`getElementAt(GPoint pt)`** Returns the topmost object containing the specified point, or **`null`** if no such object exists.

Miscellaneous methods

`void markAsComplete()` Marks this compound as complete to prohibit any further changes to its contents.
`GPoint getLocalPoint(double x, double y)` *or* **`getLocalPoint(GPoint pt)`** Returns the point in the local coordinate space corresponding to **`pt`** in the canvas.
`GPoint getCanvasPoint(double x, double y)` *or* **`getCanvasPoint(GPoint pt)`** Returns the point on the canvas corresponding to **`pt`** in the local coordinate space.

The argument for including the `GCompound` class

The question of whether to include **`GCompound`** in the **`acm.graphics`** package prompted considerable discussion within the Java Task Force. The arguments against including it are predicated largely on the principle of minimizing complexity: all other things being equal, it's best to define as few classes as possible to achieve the desired goal of producing a usable object-based graphics package. The **`GCompound`** class, by definition, provides no new capabilities in terms of what clients of the package can draw on the screen. By contrast, the arguments in its favor are:

- The **`GCompound`** class provides a powerful abstraction mechanism for creating new objects that maintain their conceptual integrity.
- **`GCompound`** serves as the base class for a wide spectrum of useful classes that many adopters might want to use (such as the **`GVariable`** class used in Figure 5-12 later in this chapter). The online tutorial supplied with the ACM libraries uses many of these.

- **GCompound** serves as the base class for a wide spectrum of useful classes that many adopters might want to use, such as the **GVariable** class used in the example. The online tutorial supplied with the ACM libraries uses many of these.
- The **GCompound** class makes it possible to shift the virtual origin of any of the other figures. If you wanted, for example, to define an object whose appearance was a circle with its origin at the center, you could not rely simply on the **GOval** class because doing so would leave the origin in the upper-left corner of the bounding box. However, you can achieve the desired effect by embedding a **GOval** at the appropriate location within a **GCompound**. In essence, this strategy relies on the fact that **GCompound** objects define their own local coordinate system, which offers significant expressive power.
- The inclusion of this class and the associated **GContainer** interface make it possible for teachers to use the **acm.graphics** package as a paradigmatic example of containment as well as subclassing. The understanding that students gain through this process will help them understand the relationship between the **Component** and **Container** classes in the AWT, which is directly parallel to this design. A **Container** is a **Component** that also contains **Component**s, just as a **GCompound** is a **GObject** that also contains **GObject**s.
- It is not really possible to add the **GCompound** class as an independent extension because the design of the rest of the package depends to a certain extent on whether the **GCompound** class exists. If there were no **GCompound** class, for example, there would be no obvious need for the **GContainer** interface, even though that interface is integral to a clean design of the compound object facility. The only possible parent for a **GObject** would be the **GCanvas** in which it resides, and it would be difficult to explain the existence of the **GContainer** interface except as a gateway to future extensions "left to the reader as an exercise." Such a package would appear unfinished.
- There are many aspects of the design and implementation of **GCompound** that are hard to get right, and it seems dangerous to have such a useful facility implemented independently by individual adopters of the package. In particular, handling mouse events that occur in the region of a compound object requires some subtlety to retain consistent semantics.

5.8 Extensibility

With only 18 classes and four interfaces, the **acm.graphics** package is small enough for teachers and students to understand the whole of the design in a relatively short amount of time. Despite its modest size, however, the facilities provided by the package enable students to construct extremely sophisticated designs. For simple applications, the classes provided by the package are sufficient. As applications become larger and more complex, clients of the **acm.graphics** package will find it useful to extend the basic functionality by defining new classes that build on the existing ones. The next few subsections describe four strategies for extending the predefined object hierarchy, as follows:

1. Extending the existing shape classes
2. Extending the **GPolygon** class
3. Creating compound objects using **GCompound**
4. Deriving subclasses directly from **GObject**

Extending the existing shape classes

The simplest form of extension available to students is creating subclasses of the existing shapes. Such subclasses are useful, for example, in those applications in which you need

to create many shapes that share a common property. Such properties can be set in the subclass constructor, thereby freeing you from the need to set them for each individual object. Consider, for example, the following pair of class definitions:

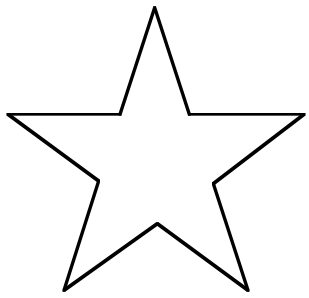
```
class GFilledSquare extends GRect {
   public GFilledSquare(double x, double y, double size) {
      super(x, y, size, size);
      setFilled(true);
   }
}

class GRedSquare extends GFilledSquare {
   public GRedSquare(double x, double y, double size) {
      super(x, y, size);
      setColor(Color.RED);
   }
}
```

In this example, the class **GFilledSquare** represents a rectangle object that is initially square (in the sense that only a single dimension is required for both width and height) and filled; **GRedSquare** extends that class to create a filled square that is initially red.

While this simple style of extension is accessible to introductory students, it is problematic for two reasons. First, its use is limited to those cases in which an existing class can be made to draw the desired figure. Second, this extension model creates class names that imply properties for an object without any guarantee that those properties can be maintained. Since **GFilledSquare** and **GRedSquare** both inherit all the methods of **GRect** and **GObject**, clients would be able to change the initial properties at will. Thus, an object declared to be a **GRedSquare** could end up being a green rectangle after a couple of client calls. It therefore seems preferable—at least for pedagogical reasons—to adopt one of the alternative extension models described in a subsequent section.

Extending the GPolygon class

As it turns out, the extension style that has proven to be most useful in practice consists of defining new subclasses of **GPolygon**, which was described in section 5.3. The basic strategy is to use the **GPolygon** class as the base class for a new shape class whose outline is a polygonal region. This technique is illustrated by the **GStar** class in Figure 5-11, which draws the five-pointed star shown on the right. The only complicated part of the class definition is the geometry required to compute the coordinates of the starting point of the figure. This code also illustrates two other useful strategies, as follows:



- The figure is drawn inside a unit square and then scaled to the size given in the parameters. This strategy for figure drawing is common in PostScript and emphasizes the idea that a figure can be drawn independent of scale.
- The constructor for the **GStar** class includes a call to the protected **GPolygon** method **markAsComplete**, which prohibits clients from adding more vertices to the polygon. This call protects the integrity of the class and makes it impossible for clients to change the shape of a **GStar** object to something else.

Creating compound objects using GCompound

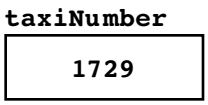
Although the **GPolygon** class is useful in a variety of contexts, it is limited to those applications in which the desired graphical object can be described as a simple closed polygon. Many graphical objects that one might want to define don't fit this model.

Figure 5-11. Class definition for a five-pointed star

```
/** Defines a graphical object that appears as a five-pointed star */
public class GStar extends GPolygon {

/** Constructs a star centered in a square of the specified size */
   public GStar(double size) {
      double sinTheta = GMath.sinDegrees(18);
      double b = 0.5 * sinTheta / (1.0 + sinTheta);
      double edge = 0.5 - b;
      addVertex(-0.5, -b);
      int angle = 0;
      for (int i = 0; i < 5; i++) {
         addPolarEdge(edge, angle);
         addPolarEdge(edge, angle + 72);
         angle -= 72;
      }
      scale(size);
      markAsComplete();
   }

/** Constructs a star centered at (x, y) */
   public GStar(double x, double y, double size) {
      this(size);
      setLocation(x, y);
   }
}
```

Fortunately, the **GCompound** class described in section 5.7 provides the basis for a much more powerful extension mechanism. Given that the predefined shape classes defined in **acm.graphics** are precisely aligned with the drawing capabilities available in the **Graphics** class from **java.awt**, anything that one could draw using the traditional Java paradigm can be described as a collection of shape objects, which makes it possible to use **GCompound** to encapsulate figure Java could draw as a single graphical object.

As a practical application of the extension capabilities provide by **GCompound**, consider the code in Figure 5-12. The idea behind the **GVariable** class is to create a new graphical object that represents a traditional box diagram of a variable, such as the one shown at the right. In terms of its graphical representation, the **GVariable** has three components: a **GRect** that displays the box, a **GLabel** to display the variable name, and another **GLabel** representing the value. To the client, however, the **GVariable** offers a conceptual interface that supports setting and retrieving the value of the variable, which is represented as an arbitrary object. The box diagram shown, for example, might be generated by the following statements:




```
GVariable taxiNumber = new GVariable("taxiNumber");
taxiNumber.setValue(new Integer(1729));
```

Deriving subclasses directly from GObject

The strategies presented earlier all have the advantage—particularly for novices—that they allow subclass designers to remain entirely in the **acm.graphics** world, without exposing the implementation structures underneath. It is also possible to extend **GObject** directly. In this case, however, the subclass must provide a definition of the method

```
public abstract void paint(Graphics g);
```

which means that whoever writes that subclass has to understand **java.awt**'s **Graphics** class and its methods. That said, it turns out to be relatively straightforward to write such classes, and we certainly expect teachers and more advanced students to do so.

Figure 5-12. The GVariable class implemented using GCompound

```
/* Class: GVariable */
/**
 * This class represents a labeled box with contents that can be
 * set and retrieved by the client.  It is intended for displaying
 * the value of a variable.
 */
class GVariable extends GCompound {

   public static final double VARIABLE_WIDTH = 75;
   public static final double VARIABLE_HEIGHT = 25;
   public static final Font FONT = new Font("Monospaced",
                                            Font.BOLD, 10);

/** Constructs a GVariable object with the specified name. */
   public GVariable(String name) {
      GRect box = new GRect(VARIABLE_WIDTH, VARIABLE_HEIGHT);
      box.setFilled(true);
      box.setFillColor(Color.white);
      add(box);
      GLabel label = new GLabel(name);
      label.setFont(FONT);
      label.setLocation(0, -label.getDescent() - 1);
      add(label);
      contents = new GLabel("");
      contents.setFont(FONT);
      add(contents);
      centerContents();
      markAsComplete();
   }

/** Sets the value of the variable. */
   public void setValue(Object value) {
      this.value = value;
      contents.setLabel(value.toString());
      centerContents();
   }

/** Returns the value of the variable. */
   public Object getValue() {
      return value;
   }

/** Centers the contents string in the variable box. */
   private void centerContents() {
      double x = (VARIABLE_WIDTH - contents.getWidth()) / 2;
      double y = (VARIABLE_HEIGHT + contents.getAscent()) / 2;
      contents.setLocation(x, y);
   }

/* Private instance variables */
   private GLabel label, contents;
   private Object value;
}
```

To offer at least one example of how such extensions might work, the code in Figure 5-13 reimplements the **GVariable** example from the preceding section as a direct subclass of **GObject**. Much of the code is the same. The difference is that the painting of the object (including the centering of the value string) is now supplied by an explicit **paint** method. For Java programmers, the resulting code is not too difficult. The class

Figure 5-13. The GVariable class implemented as a direct GObject subclass

```
/* Class: GVariable */
/**
 * This class represents a labeled box with contents that can be
 * set and retrieved by the client.  It is intended for displaying
 * the value of a variable.
 */
class GVariable extends GObject {

   public static final double VARIABLE_WIDTH = 75;
   public static final double VARIABLE_HEIGHT = 25;
   public static final Font FONT = new Font("Monospaced",
                                            Font.BOLD, 10);

/** Constructs a GVariable object with the specified name. */
   public GVariable(String name) {
      setSize(VARIABLE_WIDTH, VARIABLE_HEIGHT);
      this.name = name;
      value = null;
   }

/** Sets the value of the variable. */
   public void setValue(Object value) {
      this.value = value;
      repaint();
   }

/** Returns the value of the variable. */
   public Object getValue() {
      return value;
   }

/** Draws the object using the graphics context g. */
   public void paint(Graphics g) {
      Rectangle r = getBounds().toRectangle();
      g.setColor(Color.white);
      g.fillRect(r.x, r.y, r.width, r.height);
      g.setColor(getColor());
      g.drawRect(r.x, r.y, r.width, r.height);
      g.setFont(FONT);
      FontMetrics fm = g.getFontMetrics();
      g.drawString(name, r.x, r.y - fm.getDescent() - 1);
      String str = (value == null) ? "" : value.toString();
      int x = r.x + (r.width - fm.stringWidth(str)) / 2;
      int y = r.y + (r.height + fm.getAscent()) / 2;
      g.drawString(str, x, y);
   }

/* Private instance variables */
   private String name;
   private Object value;
}
```

definition still fits on one page, and the implementation of **paint** is reasonably straightforward. For a novice, however, this implementation requires coming to understand several new concepts, including the classes **Graphics** and **FontMetrics**; the code based on **GCompound** hides these implementation details.

Managing extensions

The existence of any strategy for defining new **GObject** subclasses, whether or not a **GCompound** class exists, invites adopters of the package to create new classes, some of which will undoubtedly be of interest to the broader community. One of the lessons of the extensible language movements of the 1970s—and indeed one of the underlying reasons for the success of very rich software development environments like the JDK—is that programmers are generally less concerned with extensibility than they are with extensions. Being able to build your own extensions is of interest to some; picking up extensions that someone else has developed is of great interest to a much broader spectrum of users. It therefore seems reasonable to expect that educators (and possibly others) who adopt the **acm.graphics** package will be willing both to supply new **GObject** classes to a common repository and to make use of ones that others put there.

Understanding how to manage such a repository needs to be part of the follow-on work envisioned by the Java Task Force. Presumably, it would make sense to establish some vetting process by which submissions could be evaluated and checked for consistency. Those that pass muster—and that come with a required set of materials including a **javadoc** description and source code distributable under some form of open source license—could then be put into a repository somewhere. Such a repository would increase community buy-in to the ACM libraries generally and make available an array of tools beyond anything that the Task Force itself could undertake to do.

At the same time, it seems important that the extended classes in the repository *not* become part of the **acm.graphics** package but remain as compatible supplements to it. Trying to manage a series of staged releases to accommodate new community-supplied classes seems like a fool's errand and would reintroduce the instability that has plagued Java itself. As long as they stay within the 18 classes defined in the **acm.graphics** package, textbook authors and other resource developers can feel confident that the materials they develop will not quickly become dated. Similarly, teachers will not have to worry that the curricular materials they develop during one year might become obsolete in the next.

5.9 The GPen and GTurtle classes

The remaining two classes in the **acm.graphics** package are the **GPen** and **GTurtle** classes, which don't really fit into the other categories. Their purpose is to provide students with a simple mechanism for drawing figures using a paradigm that is more aligned with the pen-on-paper model that the felt-board model used in the rest of the package. The two models, however, are integrated in the sense that **GPen** and **GTurtle** keep track of their own path and can therefore respond to any repaint requests, just as the other graphical object classes do.

The GPen class

The **GPen** class models a pen that remembers its current location on the **GCanvas** on which it is installed. The most important methods for **GPen** are **setLocation** (or **move** to specify relative motion) and **drawLine**. The former corresponds to picking up the pen and moving it to a new location; the latter represents motion with the pen against the canvas, thereby drawing a line. Each subsequent line begins where the last one ended, which makes it very easy to draw connected figures. The **GPen** object also remembers the

path it has drawn, making it possible to redraw the path when repaint requests occur. The full set of methods for the **GPen** class are shown in Figure 5-14.

The graphics model provided by the **GPen** class provides a straightforward mechanism for generating a variety of recursive figures, such as the Koch fractal ("snowflake" curve) or the Sierpinski triangle. The following code, for example, uses the **GPen** class to draw a Koch fractal of the indicated order and size (horizontal extent) with its geometric center at the point (**x**, **y**):

```
void drawKochFractal(double x, double y, double size, int order) {
   GPen pen = new GPen(x, y);
   pen.move(-size / 2, -size / (2 * Math.sqrt(3)));
   drawFractalLine(pen, size, 0, order);
   drawFractalLine(pen, size, -120, order);
   drawFractalLine(pen, size, +120, order);
}
```

Figure 5-14. Public methods in the GPen class

Constructors

GPen() Creates a new **GPen** object with an empty path.
GPen(double x, double y) Creates a new **GPen** object whose initial location is the point (**x**, **y**).

Methods to reposition and draw lines with the pen

void setLocation(double x, double y) *or* **setLocation(GPoint pt)** Moves the pen to the specified absolute location.
void move(double dx, double dy) Moves the pen using the displacements **dx** and **dy**.
void movePolar(double r, double theta) Moves the pen **r** units in the direction **theta**, measured in degrees.
void drawLine(double dx, double dy) Draws a line with the specified displacements, leaving the pen at the end of that line.
void drawPolarLine(double r, double theta) Draws a line **r** units in the direction **theta**, measured in degrees.

Methods to define a filled region bounded by pen strokes

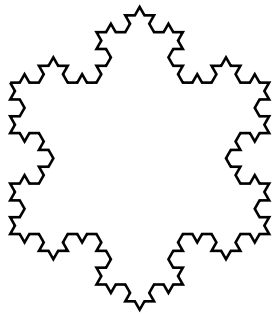
void startFilledRegion() Fills the polygon formed by lines drawn between here and the next **endFilledRegion**.
void endFilledRegion() Closes and fills the region begun by **startFilledRegion**.

Miscellaneous methods

void showPen() Makes the pen itself visible, making it possible to see where the pen moves.
void hidePen() Makes the pen invisible.
boolean isPenVisible() Returns **true** if the pen is visible, and **false** otherwise.
void setPenImage(Image image) Sets an image to display the pen. By default, the pen appears as a quill.
Image getPenImage() Returns the image currently used to display the pen.
void setSpeed(double speed) Sets the speed of the pen, which must be a number between 0 (slow) and 1 (fast).
double getSpeed() Returns the speed last set by **setSpeed**.
void erasePath() Removes all lines from this pen's path.

```
void drawFractalLine(GPen pen, double len, int theta, int order) {
   if (order == 0) {
      pen.drawPolarLine(len, theta);
   } else {
      drawFractalLine(pen, len / 3, theta, order - 1);
      drawFractalLine(pen, len / 3, theta + 60, order - 1);
      drawFractalLine(pen, len / 3, theta - 60, order - 1);
      drawFractalLine(pen, len / 3, theta, order - 1);
   }
}
```

Calling **drawKochFractal(cx, cy, 100, 3)** (where **cx** and **cy** are the coordinates of the center of the figure) generates the following picture:



The applet that produces this figure is available on the JTF web site.

Both the **GPen** class and the **GTurtle** class described in the following section are often used to create animated displays. To provide clients with some control over the speed of the animation, both classes include a **setSpeed** method, which takes a number between 0.0 and 1.0 as its argument. Calling **setSpeed(0.0)** means that the animation crawls along at a very slow pace; calling **setSpeed(1.0)** makes it proceed as fast as the system allows. Intermediate values are interpreted so as to provide a smoothly varying speed of operation. Thus, if the speed value is associated with a scrollbar whose ends represent the values 0.0 and 1.0, adjusting the scrollbar will cause the animation to speed up or slow down in a way that seems reasonably natural to users.

The speed mechanism is implemented by delaying the calling thread each time the **GPen** or **GTurtle** object is moved. If that calling thread is Java's event-handling thread, any other user-generated events will be held up during the period in which that thread is delayed. Given that the delays are relatively short even on the slowest speed, this implementation does not usually cause problems for novices writing animations. More sophisticated users, however, should create a separate thread to animate these objects.

The GTurtle class

The **GTurtle** class is similar to **GPen** but uses a "turtle graphics" model derived from the Project Logo turtle described in Seymour Papert's *Mindstorms* [Papert80]. In the turtle graphics world, the conceptual model is that of a turtle moving on a large piece of paper. A **GTurtle** object maintains its current location just as a **GPen** does, but also maintains a current direction. The primary methods to which a **GTurtle** responds are **forward(***distance***)**, which moves the turtle forward the specified distance, and the directional methods **left(***angle***)** and **right(***angle***)**, which rotate the turtle the indicated number of degrees in the appropriate direction. The path is created by a pen located at the center of the turtle. If the pen is down, calls to **forward** generate a line; if the pen is up, such calls simply move the turtle without drawing a line. The full list of methods is shown in Figure 5-15.

Figure 5-15. Public methods in the `GTurtle` class

Constructors

`GTurtle()` Creates a new .**`GTurtle`** object with an empty path.
`GTurtle(double x, double y)` Creates a new .**`GTurtle`** object whose initial location is the point (**`x`**, **`y`**).

Methods to move and rotate the turtle

`void setLocation(double x, double y)` *or* **`setLocation(GPoint pt)`** Moves the turtle to the specified absolute location without drawing a line.
`void forward(double distance)` Moves the turtle **`distance`** units in the current direction, drawing a line if the pen is down.
`void setDirection(double direction)` Sets the direction (in degrees counterclockwise from the *x*-axis) in which the turtle is moving.
`double getDirection()` Returns the current direction in which the turtle is moving.
`void right(double angle)` *or* **`right()`** Turns the turtle direction the specified number of degrees to the right (default is 90).
`void left(double angle)` *or* **`left()`** Turns the turtle direction the specified number of degrees to the left (default is 90).

Miscellaneous methods

`void penDown()` Tells the turtle to lower its pen so that it draws a track. The pen is initially up.
`void penUp()` Tells the turtle to raise its pen so that it stops drawing a track
`boolean isPenDown()` Returns **`true`** if the pen is down, and **`false`** otherwise.
`void showTurtle()` Makes the turtle visible. The turtle itself is initially visible.
`void hideTurtle()` Makes the turtle invisible.
`boolean isTurtleVisible()` Returns **`true`** if the turtle is visible, and **`false`** otherwise.
`void setSpeed(double speed)` Sets the speed of the pen, which must be a number between 0 (slow) and 1 (fast).
`double getSpeed()` Returns the speed last set by **`setSpeed`**.
`void erasePath()` Removes all lines from the turtle's path.

Although **`GTurtle`** and **`GPen`** have similar capabilities, they are likely to be used in different ways. The **`GTurtle`** class is designed to be used at the very beginning of a course and must be both simple and evocative as intuitive model. The **`GTurtle`** therefore has somewhat different defaults than its **`GPen`** counterpart does. The image of the **`GTurtle`**, for example, is initially visible, while the **`GPen`** image is hidden. Moreover, the **`GTurtle`** does not actually draw lines until the pen is lowered. The **`GPen`** offers no option; the pen is always down. These defaults make the **`GTurtle`** consistent with the Logo model, in which students move the turtle first and then start drawing pictures with it.

The turtle graphics model is illustrated in section 6.8, which also describes how to use objects of the **`GTurtle`** classes as standalone programs.

Chapter 6
The **acm.program** Package

In the taxonomy of problems presented in Chapter 3, one of the issues that rose to the fore (as part of the discussion under problem L1) is that Java applications start by invoking a method whose signature is

```
public static void main(String[] args)
```

The structure of the **main** method in an application has several negative effects. One of the most commonly cited is that the method signature for **main** includes several concepts—static methods and array parameters, for example—that are difficult to present to novices. This shortcoming, however, is only part of the problem. The Java Task Force views the following problems as being of at least as much concern:

- *The use of a static method as the entry point encourages students to operate outside the object-oriented domain.* It is unfortunate that **main**—which is the first method students typically write in an application-based introduction to Java—is a static method. The extent that the **main** method contains any code other than the instantiation of some object, that code runs in a purely procedural framework that has no association with the object-oriented world. Moreover, if the static **main** method is decomposed into subsidiary methods, those methods must also be declared as static, thereby delaying the introduction of the object-oriented approach. To avoid this problem, the Java Task Force believes strongly that instructors—even if they choose not to adopt the **acm.program** package—should encourage their students to code **main** so that its only function is to instantiate an object and then pass control to it by invoking one of its methods.
- *The paradigm used to define an application is substantially different from that used for applets.* Instructors who teach applets instead of applications have the advantage that student code runs in the object-oriented world from the beginning. When a browser encounters an applet, it instantiates a new instance of that applet and then sends it messages to trigger its operation. Applets, however, are problematical both because they are declining in popularity within the Java community and because many browsers do not support the most recent implementations of Java. These drawbacks make it impossible to recommend using applets in all contexts, despite their pedagogical advantages.
- *Applications provide little support for many of the facilities that programmers would like to employ.* Although programming environments can in some cases extend the capabilities of Java applications, programs running in their most basic form look much more like programs from the 1970s than they resemble programs of the 21st century. The standard paradigm for an application consists of a single-threaded program whose only connection to the user's environment is a set of traditional I/O streams (**System.in**, **System.out**, and **System.err**). Unless the programmer includes explicit code to do so, an application creates no windows or interactors of the sort that today's students associate with modern applications. And although it is not that difficult to put up a simple window, creating one that includes, for example, a menu bar is much harder to achieve.

To minimize these negative effects, the Java Task Force offers a new **Program** class as part of its standard library packages. The central idea behind the proposal is simply that the existence of such a class—coupled with several specific subclasses that define

specific program types—gives instructors and students a standard framework for creating better, more functional applications.

We believe that the facilities offered by the **acm.program** package will have a profoundly simplifying effect on introductory Java examples. The major advantages associated with the use of this package can be summarized as follows:

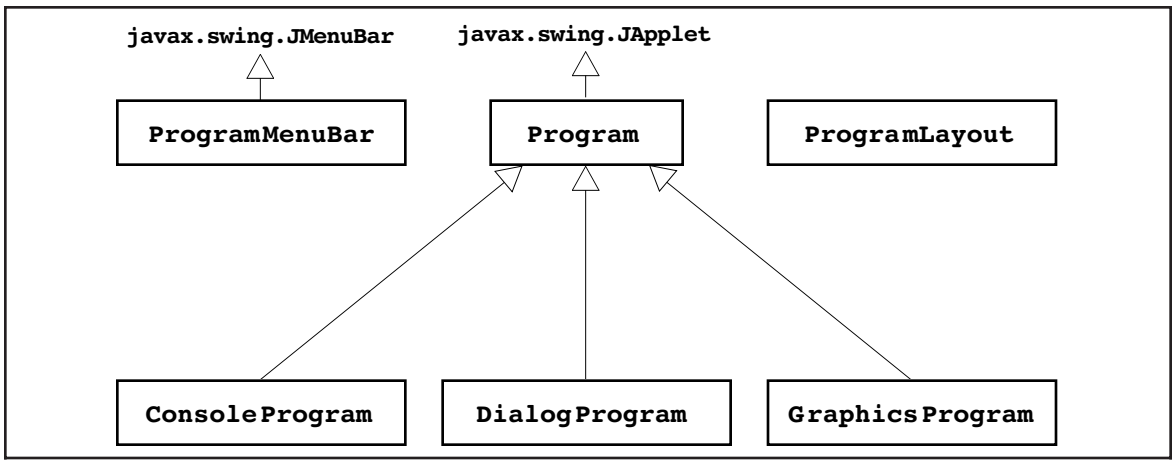
1. The conventional pattern of use associated with the **acm.program** package moves students away from the procedural style of **public static void main** into a more pedagogically defensible environment in which students are always working in the context of an object.
2. The **Program** class allows Java applications to double as applets, thereby making it possible for instructors to use either paradigm in a consistent way. Even for those instructors that choose to focus on the application paradigm, using the **Program** class makes it much easier for students to make their code available on the web. Moreover, because the **Program** class is defined to be a subclass of **JApplet** (and indirectly therefore from the basic **Applet** class) applications that extend **Program** can take advantage of such applet-based features as loading audio clips and images from a code base.
3. The **Program** class includes several features that make instruction easier, such as menu bars that support operations like printing and running programs with an automated test script.
4. The classes in the **acm.program** package offer a compelling example of an inheritance hierarchy that introductory students can understand and appreciate right from the beginning of their first course.

As is often the case, however, these advantages are much easier to understand if one has some familiarity with the general idea of the proposal. This chapter, therefore, begins by illustrating the **Program** class with a few simple examples, and then describes in more detail the subclasses available in the standard **acm.program** hierarchy.

6.1 Simple examples of the Program class

The **acm.program** package defines a small hierarchy of classes, as illustrated in Figure 6-1. Every class in the package is an abstract class and is therefore instantiated only in the form of a specific client-defined subclass. A student who writes a program using this package picks the **Program** subclass most appropriate for the problem at hand and then writes a new class definition that extends that base.

Figure 6-1. Class diagram for the acm.program package



As an example, Figure 6-2 shows a program that descends directly from the **Program** class and has the effect of reading two integers from the user and displaying their sum. Because this program makes very little use of the object-oriented paradigm, it is by no means representative of the program that the Java Task Force encourages, but will nonetheless serve as a useful baseline for illustrating the operation of the class.

The **Add2** class in Figure 6-2 defines a single method called **run**, which specifies the code that needs to be executed when the program is run. In this example, the **run** method reads two integers, adds them together, and displays the result. Because this implementation is a direct subclass of **Program**, input is read from the standard input stream (**System.in**) and output appears on the standard output stream (**System.out**). No windows are created by the **Program** class itself, which means that the **Add2** program might produce the following typescript in a command-line environment, shown here as it appears in a terminal window running on Mac OS X:

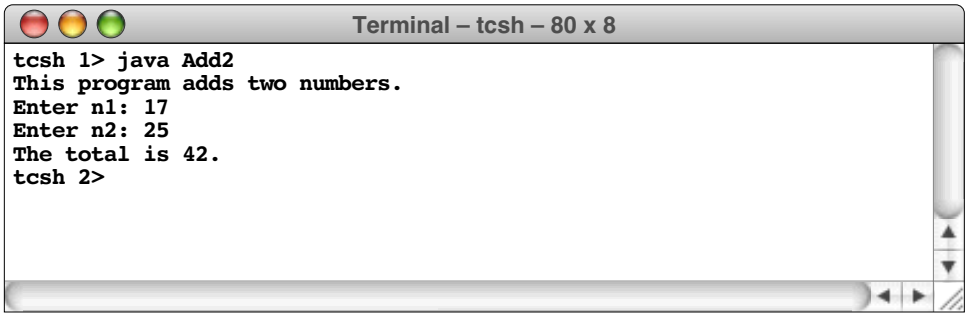


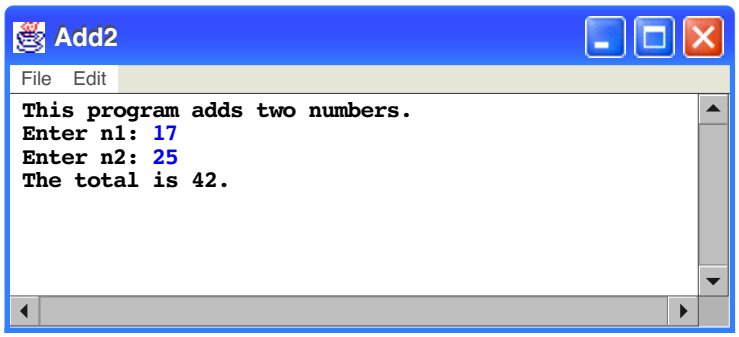
Figure 6-2. Simple program to add two numbers

```
/*
 * File: Add2.java
 * ---------------
 * This program adds two numbers and prints their sum.
 */

import acm.program.*;

/**
 * This class adds two numbers entered by the user and displays
 * their sum.  Because this version is a Program, input and
 * output are assigned to System.in and System.out.
 */

public class Add2 extends Program {

/**
 * Runs the program.  Here, the run method adds two numbers
 * entered by the user.
 */
   public void run() {
      println("This program adds two numbers.");
      int n1 = readInt("Enter n1: ");
      int n2 = readInt("Enter n2: ");
      int total = n1 + n2;
      println("The total is " + total + ".");
   }
}
```

The **`Program`** subclasses at the bottom of Figure 6-1 make it possible to change the behavior of this program in relatively small but highly useful ways. If the first line of the class definition is changed to

```
public class Add2 extends ConsoleProgram
```

then the same program runs in a window that contains an **`IOConsole`** as defined in Chapter 4. That program might look like this if it were run under Windows XP:

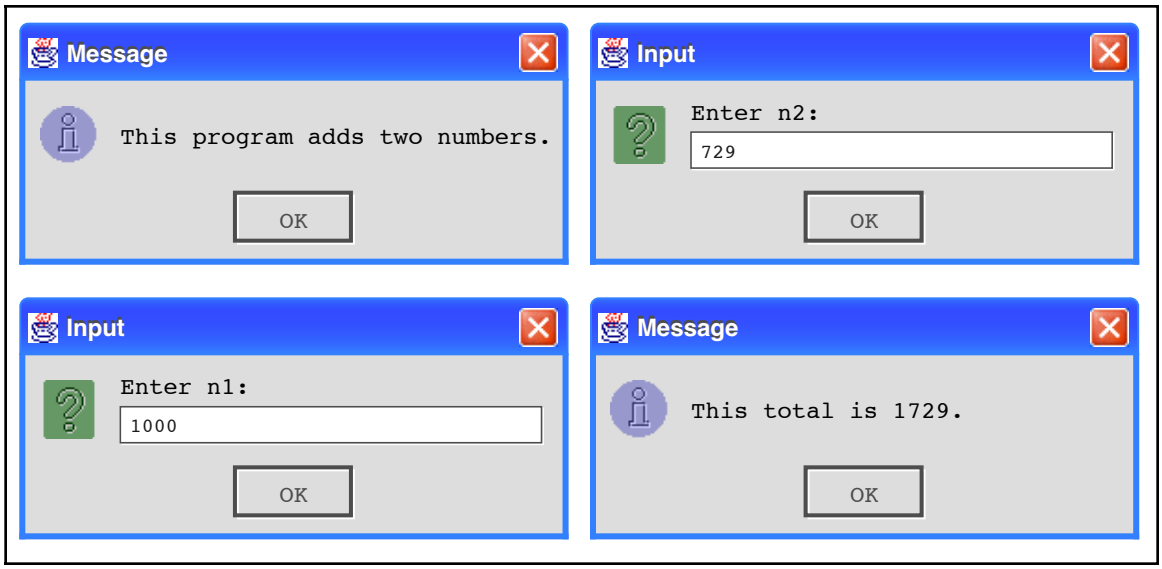


If the definition of the **`Add2`** class were instead changed to extend **`DialogProgram`**, the program would then operate by putting up a series of dialog boxes, as illustrated in Figure 6-3. (The **`GraphicsProgram`** class is not appropriate to the add-two-numbers example and will be discussed in section 6.4.)

6.2 Programs as a paradigm for inheritance

The primary point of introducing the examples in the preceding section is to illustrate an important pedagogical advantage of the **`acm.program`** class hierarchy. One of the challenges that instructors face in using an objects-first approach is finding examples of class hierarchies that make sense in the programming domain. The real world has no shortage of such examples. The classification of the biological kingdom, for example, offers a wonderful model of the class idea and the notion of subclassing. An individual horse is an instance of the general class of horses, which is itself a subclass of mammal, which is in turn a subclass of animal. Students have no trouble understanding the subclass relationship in the biological context: all horses are mammals, all mammals are

Figure 6-3. Dialogs produced by the `DialogProgram` version of `Add2`



animals, but there are some mammals that are not horses and some animals that are not mammals.

So far, so good. The trouble for teachers trying to offer new students a compelling justification for object-oriented design comes in trying to reflect the intuitive notion of class hierarchies in a programming context. Textbooks often struggle to find appropriate, programming-related examples. Most end up relying on a graphical hierarchy similar to the one outlined in Chapter 5. That hierarchy provides examples that are similar to the biological model: a **GStar** (as defined in Figure 5-11) is a subclass of **GPolygon**, which is in turn a subclass of **GObject**. The structure of the graphics classes supports the intuition that students derive from real-world hierarchies: all stars are polygons, all polygons are graphical objects, but not all graphical objects are polygons, let alone stars.

While the hierarchy from **acm.graphics** works well as an example, the Task Force believes that it is possible to illustrate this same idea even earlier by making the intuitive notion of a "program" into a hierarchy in its own right. Students are then exposed to the notions of objects, classes, and subclass hierarchies as the first concepts they encounter. By establishing a hierarchy of programs, students can see how their particular program is an instance of a specific program category, which is itself a subcategory of a larger, more generic class of programs. In particular, students will quickly come to understand that every **ConsoleProgram** is also a **Program** and can perform the operations that a **Program** does, subject to any extensions at the **ConsoleProgram** level. This understanding of the fundamental *is-a* relationship should help to emphasize the idea of inheritance.

6.3 The structure of the acm.program package

The most important thing to understand about the design of the **acm.program** package is the merging of the applet and application roles. The **Program** class extends **JApplet**, and is therefore available for use by a web browser. When invoked as an applet, the browser instantiates an instance of the main applet class and invokes its **init** method. Each **Program** class, however, also includes an implementation of a method with the familiar signature

```
public static void main(String[] args)
```

which allows it to be invoked as an application. As part of its startup operation, the application version of a **Program** also calls the **init** method, which makes it possible to include initialization code in a way that is analogous to the applet mode of operation. The **main** method in the **Program** class identifies the class that invoked it, instantiates a new instance of that class, and then calls the **init** and **start** methods of that program, just as a browser running an applet would do. Thus, in both the applet and application frameworks, the program runs in the context of an instantiated object rather than in the static domain.

Irrespective of whether the program is invoked as an applet or application, the initialization code establishes a component hierarchy that matches the one used in the Swing **JApplet** class. Moreover, as in Java 2 Standard Edition 5.0, the **add** methods for the **Program** class automatically forward any **add** requests to the content pane, so that it is now possible to write

```
add(component);
```

instead of the novice-unfriendly

```
getContentPane().add(component);
```

By default, the **Program** class itself doesn't add anything to the content pane, although subclasses are free to do so. If the content pane has any components at the end of the initialization phase, the implementation calls **setVisible(true)** on the program frame to display it on the screen. If the content pane is empty, that call does not occur. Thus, programs that are written to use only the standard input streams or popup dialogs for their user interactions can run without displaying a window.

Program startup

After setting up the standard **JApplet** component hierarchy, the **Program** startup code executes the following three steps:

1. Calls the **init** method to initialize the application.
2. Calls **validate** on the content pane to ensure that its components are arranged correctly on the screen.
3. Creates a new thread to execute the **run** method.

Although it can be useful to include both methods in a single program, most student programs will define either an **init** method or a **run** method, but not both. The **init** method is used for interactive applications in which the program creates and arranges a set of components and then waits for the user to trigger some action, typically by clicking on a button. The **run** method is intended for programs that have an independent thread of control unrelated to the actions of the user. Programs that animate a graphical object or set up a console interaction are usually of this form.

Consider, for example, the **Add2** program presented at the beginning of this chapter. That program establishes a dialog with the user through the console that requires sequential operation and therefore requires a **run** method, which for the **Add2** program looks like this:

```
public void run() {
   println("This program adds two numbers.");
   int n1 = readInt("Enter n1: ");
   int n2 = readInt("Enter n2: ");
   int total = n1 + n2;
   println("The total is " + total + ".");
}
```

The **run** method in a **Program** is analogous to the **main** method in a Java application. The key difference is that the **run** method is not static and therefore has an object to work with. It can use instance variables in that object and can invoke other methods that are not themselves static. This change alone represents the central advantage of the **acm.program** package.

The rationale behind the `init` method

The February 2005 release of the **acm.program** package did not describe the **init** method, although it was there as part of the standard applet paradigm. At the time, we proposed that initialization occur in a constructor for the class. That strategy, however, did not prove viable. The constructor for a program is invoked very early and therefore has no access to any contextual information. Inside the constructor, for example, it is impossible to determine the dimensions of the applet frame, to perform any operations that require a code base such as loading an image, or even to determine whether one is running in an application or applet context. Students found these restrictions very difficult to understand. The **init** method is invoked after the object has been instantiated

and installed in the frame, which means that the program has access to the required information.

Public methods in the `Program` class

A list of the public methods in the **`Program`** class appears in Figure 6-4. A large fraction of these methods implement the **`IOModel`** interface or one of the event interfaces from **`java.awt.event`**. To save space in the table, these methods are not described in full. The details of the **`IOModel`** methods are described in Chapter 4.

Input and output from programs

Every **`Program`** object is given both an **`IOConsole`** and **`IODialog`** object at initialization time. By default, the console is **`IOConsole.SYSTEM_CONSOLE`** and the dialog is a default instance of **`IODialog`**. Subclasses can substitute specially tailored instances of these classes by overriding the protected factory methods **`createConsole`** and **`createDialog`**. The **`ConsoleProgram`** subclass, for example, overrides **`createConsole`** as follows:

```
protected IOConsole createConsole() {
   return new IOConsole();
}
```

The **`Program`** class also defines two public methods—**`getInputModel`** and **`getOutputModel`**—to specify the source and destination for the input and output methods, respectively. In the **`Program`** class itself, these methods return the result of **`getConsole`**. The **`DialogProgram`** subclass overrides these methods to return the result of **`getDialog`** instead. Indeed, those two one-line methods are the only code in the **`DialogProgram`** subclass.

The varieties of object-oriented experience

Depending on your perception of what properties make a program object-oriented, the code for the **`Add2`** example presented earlier in this chapter may not seem to meet those requirements. The method calls in the body of the **`run`** method have no explicit receivers, depending instead on the fact that the **`Program`** class itself implements the **`IOModel`** interface and therefore defines methods like **`println`** and **`readInt`**. Such an approach is simpler for novices—as well as for teachers with little background in the object-oriented style of programming—but it doesn't emphasize the idea of sending messages to objects. It is possible, however, to rewrite the program so that make those object references explicit. If an instructor chooses to have all method calls be tied to an object, the **`run`** method for **`Add2`** could be rewritten like this:

```
public void run() {
   IOModel io = this.getConsole();
   io.println("This program adds two numbers.");
   int n1 = io.readInt("Enter n1: ");
   int n2 = io.readInt("Enter n2: ");
   int total = n1 + n2;
   io.println("The total is " + total + ".");
}
```

Judging from the range of opinions on the Task Force, different instructors will have strong preferences for one or the other of these paradigms, and it does not seem wise to allow only one of these styles. The decision to support both approaches is also consistent with the first principle put forward in Chapter 2, in which we make a commitment "to ensure that our library packages and tools are usable with other pedagogical approaches" besides the objects-first approach.

Figure 6-4. Public methods common to all `Program` objects

Methods involved in running a program

`void run()` Contains the code for the program, which is called after initialization.
`void init()` Contains applet-style initialization code; most programming styles will not use this method.

Methods for setting and retrieving the I/O context of the program

`void getConsole(IOConsole console)` Sets the **`IOConsole`** assigned to this program.
`IOConsole getConsole()` Returns the **`IOConsole`** assigned to this program.
`IODialog getDialog()` Returns the **`IODialog`** assigned to this program.
`PrintWriter getWriter()` Returns a writer that can be used to write to the console.
`BufferedReader getReader()` Returns a reader that can be used to read from the console.
`IOModel getInputModel()` Returns the input model, which is typically either the standard console or dialog.
`IOModel getOutputModel()` Returns the output model, which is typically either the standard console or dialog.

Methods for manipulating layout regions of the program

`void setLayout(String region, LayoutManager manager)` Sets the layout manager for the specified program region.
`LayoutManager getLayout(String region)` Returns the layout manager for the specified program region.
`void removeAll(String region)` Removes all the components from the specified program region.

Miscellaneous methods

`void pause(double milliseconds)` Sleeps for the specified number of milliseconds but never throws an exception.
`void exit()` Exits from the program.
`void setTitle(String title)` Sets the title used for the application title bar.
`String getTitle()` Returns the title used for the application title bar.
`boolean isAppletMode()` Returns **`true`** if this applet is running in a browser (not available in the constructor).
`void addActionListeners()` Adds the **`Program`** as an **`ActionListener`** to every button it contains.

Methods specified by the `IOModel` interface

`void print(…)` **`void println(…)`** **`String readLine(…)`** **`int readInt(…)`** **`double readDouble(…)`** **`boolean readBoolean(…)`** **`void showErrorMessage(String msg)`**

Methods specified by the `MouseListener`, `MouseMotionListener`, and `ActionListener` interfaces

`void mouseClicked(MouseEvent e)` **`void mousePressed(MouseEvent e)`** **`void mouseReleased(MouseEvent e)`** **`void mouseEntered(MouseEvent e)`** **`void mouseExited(MouseEvent e)`** **`void mouseMoved(MouseEvent e)`** **`void mouseDragged(MouseEvent e)`** **`void actionPerformed(ActionEvent e)`**

6.4 The GraphicsProgram subclass

The **GraphicsProgram** subclass simplifies the use of the **acm.graphics** package from Chapter 5 and is quite straightforward in its design. During initialization, a **GraphicsProgram** allocates a new **GCanvas** and installs it in the content pane of the **Program** window. Subclasses that extend **GraphicsProgram** can then use either of two strategies to gain access to that canvas:

1. Use the **getGCanvas** method to obtain the **GCanvas** and then invoke methods directly on that canvas.
2. Invoke wrapper methods in the **GraphicsProgram** class, each of which performs that delegation automatically.

The first paradigm is illustrated by the following program, which draws the string "Hello, world." at the center of the program window:

```
public class HelloWorld extends GraphicsProgram {
   public void run() {
      GCanvas gc = this.getGCanvas();
      GLabel label = new GLabel("Hello, world.");
      gc.add(label, (this.getWidth() - label.getWidth()) / 2,
                    this.getHeight() / 2);
   }
}
```

The second paradigm is simpler but uses a seemingly less object-oriented style, given that the receivers of the messages are all implicit:

```
public class HelloWorld extends GraphicsProgram {
   public void run() {
      GLabel label = new GLabel("Hello, world.");
      add(label, (getWidth() - label.getWidth()) / 2,
                 getHeight() / 2);
   }
}
```

The **GraphicsProgram** class implements wrappers for all the methods in **GCanvas**, with the exception of the overloaded version of the **add** method that takes a **Component** instead of a **GObject**. That method would have an ambiguous interpretation, because there would be no way to differentiate the operation of adding a component to the **GCanvas** or adding that component to the content pane of the program. This method is unlikely to be used by very many students and can easily be invoked unambiguously by calling

```
getGCanvas().add(component);
```

Responding to events in a GraphicsProgram

The **GraphicsProgram** class supports two models for mouse events. The first is simply to assign mouse listeners to the graphical objects as described in Chapter 5. The second model, which may prove easier for students, is to use the **GraphicsProgram** itself as a listener. As a convenience for new students, the **Program** class declares itself to be an implementor for the standard mouse listeners, even though the methods that implement each of these interfaces are empty. If a student wants to use a **GraphicsProgram** subclass as a listener, the steps required are:

1. Write new versions of any listener methods that need to be overridden for this particular application.
2. Invoke the method **addMouseListeners** when enough of the initialization has been performed to make it safe to call the listeners. Forcing students to include this call

explicitly eliminates the timing issues that arise in event models based on a callback strategy.

This event-handling discipline is illustrated in Figure 6-5, which updates the object-dragging example from Chapter 5 to use this program-based listener model.

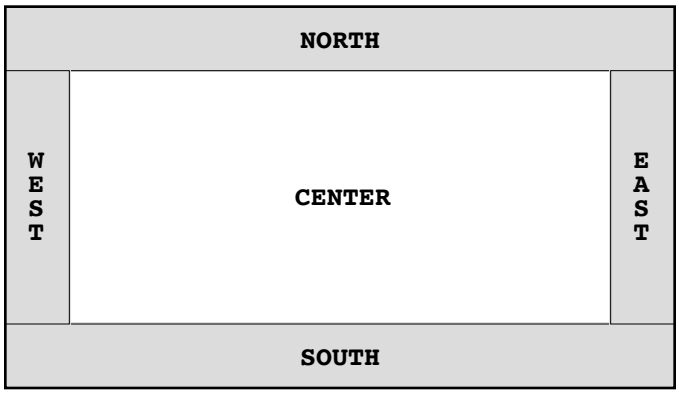
Figure 6-5. A rewrite of the object-dragging program that uses the program as a listener

```
/* File: ObjectDragExample.java */

import java.awt.*;
import java.awt.event.*;
import acm.graphics.*;
import acm.program.*;

/** This class displays a mouse-draggable rectangle and oval */
public class ObjectDragExample extends GraphicsProgram {

/** Runs the program */
   public void run() {
      GRect rect = new GRect(100, 100, 150, 100);
      rect.setFilled(true);
      rect.setColor(Color.RED);
      add(rect);
      GOval oval = new GOval(300, 115, 100, 70);
      oval.setFilled(true);
      oval.setColor(Color.GREEN);
      add(oval);
      addMouseListeners();
   }

/** Called on mouse press to record the coordinates of the click */
   public void mousePressed(MouseEvent e) {
      last = new GPoint(e.getPoint());
      gobj = getElementAt(last);
   }

/** Called on mouse drag to reposition the object */
   public void mouseDragged(MouseEvent e) {
      if (gobj != null) {
         gobj.move(e.getX() - last.getX(), e.getY() - last.getY());
         last = new GPoint(e.getPoint());
      }
   }

/** Called on mouse click to move this object to the front */
   public void mouseClicked(MouseEvent e) {
      if (gobj != null) gobj.sendToFront();
   }

/* Private instance variables */
   private GObject gobj;          /* The object being dragged   */
   private GPoint last;           /* The last mouse position    */
}
```

6.5 Layout strategy for the `Program` class

Much of the development work between the first and second draft releases of the Java Task Force packages was directed toward finding strategies to simplify the creation of graphical user interfaces, or GUIs. The results of that work and the rationale that led up to it are outlined in Chapter 7, which describes the **`acm.gui`** package. One aspect of that design, however, seemed to fit more easily in the **`acm.program`** package, because it is logically associated with the **`Program`** class itself. The essential idea behind the design is that programs often need to be able to create simple control bars that present the user with a simple list of interactors. Unfortunately, given the facilities provided by the standard Java packages, building this type of control bar and populating it with interactors is often too difficult for novices. As at least a partial solution to this problem, the Java Task Force decided to implement a new layout manager for programs whose primary function is to simplify the creation of control bars.

The **`Program`** class automatically assigns a **`BorderLayout`** manager to the content pane and initializes each of the five components—**`NORTH`**, **`SOUTH`**, **`EAST`**, **`WEST`**, and **`CENTER`**—to be an empty **`JPanel`**. The individual **`JPanel`** objects are arranged as in the standard **`BorderLayout`** paradigm, which looks like this:



Each of the regions along the edges is assigned its preferred size, which means that it disappears entirely as long as it is empty. The center region gets all the remaining space.

The **`Program`** class overrides the standard definitions of the **`add`** methods so that components are added to the center region by default, but to one of the side regions if the constraint matches the appropriate region constant. Thus, calling

```
add(new JButton("OK"), SOUTH);
```

will add a new button labeled **`OK`** to the **`SOUTH`** region.

As with any container, the components that makes up the regions of a **`Program`** object are rearranged whenever the container is validated, which happens automatically when the container is resized but which can also be triggered explicitly by calling **`validate`** on the container. The **`Program`** logic automatically calls **`validate`** after the **`init`** method returns, which means that most students will never have to worry about forcing an update of the layout structure.

Each of the **`JPanel`**s that form the five program regions has a default layout manager, as follows:

`NORTH`, `SOUTH`	`new TableLayout(1, 0, 5, 5)`	Single horizontal row.
`EAST`, `WEST`	`new TableLayout(0, 1, 5, 5)`	Single vertical column.
`CENTER`	`new GridLayout(1, 0)`	Equally spaced horizontal row.

You can, however, change any of these layout managers by calling

```
setLayout(region, layout);
```

The details of the `TableLayout` class are discussed in Chapter 7.

Simple examples that use the layout regions

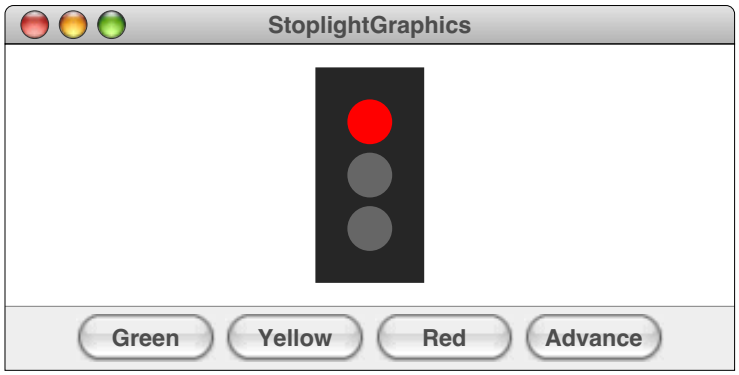
The code in Figure 6-6 offers a simple example of the layout strategy provided by the `Program` class. The code defines three buttons that simply display their name on the console whenever they are activated.

The three calls to `add` in the constructor assign these buttons to the `SOUTH` region of the layout, as specified by the constraint parameter in the `add` call. Because the default layout for the `SOUTH` region is a `TableLayout` manager with a single row and a five-pixel gap between adjacent interactors. The result is therefore a program window that looks like this, which shows the state of the console window after the user has clicked each of the buttons in order:

Figure 6-6. A `ConsoleProgram` version of a stoplight

```
/*
 * File: StoplightConsole.java
 * ---------------------------
 * This program illustrates the construction of a simple GUI.
 */

import acm.program.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * This class displays three buttons at the south edge of the window.
 * The name of the button is echoed on the console each time a button
 * is pressed.
 */

public class StoplightConsole extends ConsoleProgram {

/** Create the GUI program. */
   public StoplightConsole() {
      add(new JButton("Green"), SOUTH);
      add(new JButton("Yellow"), SOUTH);
      add(new JButton("Red"), SOUTH);
      addActionListeners();
   }

/** Listen for a button action. */
   public void actionPerformed(ActionEvent e) {
      println(e.getActionCommand());
   }

}
```

Positioning interactors around the border is equally useful with **Program** subclasses other that **ConsoleProgram**. The code in Figure 6-7 shows a similar application redesigned as a **GraphicsProgram** in which the stoplight is represented graphically on the display, like this:



The code for the **Stoplight** class itself appears in Figure 6-8. The **Stoplight** class extends **GCompound** to create an object that responds to the messages **setState(color)** and **advance()**.

6.6 Using menu bars with the Program class

Applications that are running in a modern window-based environment typically include menu bars to give the user ready access to a variety of useful operations, typically including both file-based operations such as saving or printing and editor-based operations such as cut, copy, and paste. The **Program** class makes it very easy to include menu bars in applications. Every **Program** instance includes a standard menu bar by default, and it is easy to extend that menu bar to include application-specific menu items.

The default menu bar

By default, the menu bar associated with each program includes a **File** and an **Edit** menu, which have the structure shown in Figure 6-9. The **File** menu includes the following items:

- **Print**. Prints a copy of the window image for the program.
- **Print Console**. Prints a log of the console, dividing it up into pages as necessary.
- **Save** and **Save As**. Saves the typescript of the console as a text file.
- **Script**. Opens a dialog and allows the user to select a text file. That file is then used as the source for all console input. That input is echoed to the console window so that the typescript shows the full session.

The **Edit** menu includes items for the **Cut**, **Copy**, **Paste**, and **Select All** options, all of which are standard in modern computing platforms.

Figure 6-7. A `GraphicsProgram` version of a stoplight

```
/*
 * File: StoplightGraphics.java
 * ----------------------------
 * This program illustrates the construction of a simple GUI using a
 * GraphicsProgram as the main class.
 */

import acm.graphics.*;
import acm.program.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * This class displays three buttons at the south edge of the window.
 * Pressing a button lights the indicated lamp in the stoplight.
 */

public class StoplightGraphics extends GraphicsProgram {

/** Create the GUI program. */
   public StoplightGraphics() {
      add(new JButton("Green"), SOUTH);
      add(new JButton("Yellow"), SOUTH);
      add(new JButton("Red"), SOUTH);
      add(new JButton("Advance"), SOUTH);
   }

/** Runs the program to create the stoplight. */
   public void run() {
      signal = new Stoplight();
      add(signal, getWidth() / 2, getHeight() / 2);
      addActionListeners();
   }

/** Listen for a button action. */
   public void actionPerformed(ActionEvent e) {
      String command = e.getActionCommand();
      if (command.equals("Advance")) {
         signal.advance();
      } else if (command.equals("Red")) {
         signal.setState(Color.RED);
      } else if (command.equals("Yellow")) {
         signal.setState(Color.YELLOW);
      } else if (command.equals("Green")) {
         signal.setState(Color.GREEN);
      }
   }

/* Private instance variables */

   private Stoplight signal;

}
```

Figure 6-8. Graphical implementation of the `Stoplight` class

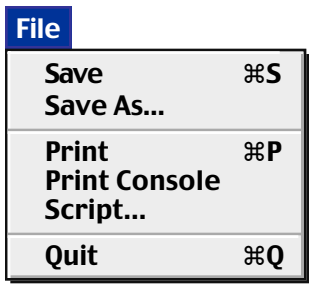
```
/*
 * File: Stoplight.java
 * --------------------
 * This class implements a stoplight as a compound graphical object.
 */

import acm.graphics.*;
import acm.util.*;
import java.awt.*;

/**
 * This class represents a graphical stoplight with its origin point
 * at the center.  The Stoplight class exports the following methods:
 *
 * <ul>
 * <li>getState()      - Returns the current state of the Stoplight
 * <li>setState(color) - Sets the Stoplight to the specified state
 * <li>advance()       - Advances the Stoplight to the next state
 * </ul>
 */

public class Stoplight extends GCompound {

/* Public constants for the colors */
   public static final Color RED = Color.RED;
   public static final Color YELLOW = Color.YELLOW;
   public static final Color GREEN = Color.GREEN;

/** Creates a new Stoplight object, which is initially red. */
   public Stoplight() {
      GRect frame = new GRect(STOPLIGHT_WIDTH, STOPLIGHT_HEIGHT);
      frame.setFilled(true);
      frame.setColor(Color.DARK_GRAY);
      add(frame, -STOPLIGHT_WIDTH / 2, -STOPLIGHT_HEIGHT / 2);
      redLamp = createLamp(0, -STOPLIGHT_HEIGHT / 4);
      yellowLamp = createLamp(0, 0);
      greenLamp = createLamp(0, STOPLIGHT_HEIGHT / 4);
      add(redLamp);
      add(yellowLamp);
      add(greenLamp);
      setState(Color.RED);
   }

/** Changes the state of the stoplight to the indicated color */
   public void setState(Color color) {
      state = color;
      redLamp.setColor((state == RED) ? RED : Color.GRAY);
      yellowLamp.setColor((state == YELLOW) ? YELLOW : Color.GRAY);
      greenLamp.setColor((state == GREEN) ? GREEN : Color.GRAY);
   }
```

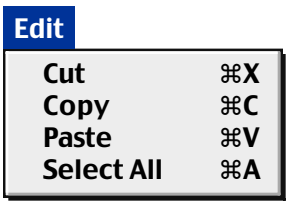
Figure 6-8. Graphical implementation of the `Stoplight` class (continued)

```
/** Returns the current state of the stoplight */
   public Color getState() {
      return state;
   }

/** Advances the stoplight to the next state */
   public void advance() {
      if (state == RED) {
         setState(GREEN);
      } else if (state == YELLOW) {
         setState(RED);
      } else if (state == GREEN) {
         setState(YELLOW);
      } else {
         throw new ErrorException("Illegal stoplight state");
      }
   }

/** Creates a new GOval to represent one of the three lamps */
   private GOval createLamp(double x, double y) {
      GOval lamp = new GOval(x - LAMP_RADIUS, y - LAMP_RADIUS,
                             2 * LAMP_RADIUS, 2 * LAMP_RADIUS);
      lamp.setFilled(true);
      return lamp;
   }

/* Private constants */
   private static final double STOPLIGHT_WIDTH = 50;
   private static final double STOPLIGHT_HEIGHT = 100;
   private static final double LAMP_RADIUS = 10;

/* Private instance variables */
   private Color state;
   private GOval redLamp;
   private GOval yellowLamp;
   private GOval greenLamp;

}
```

The **`Print`** and **`Quit`** items are handled by the implementation of the **`Program`** class itself. The other menu items are passed to the **`IOConsole`** that has the keyboard focus, if any.

Figure 6-9. Menus provided by the standard `Program` class





The menu bars for a **`Program`** follow the standard design for the platform on which they appear. The menus are shown here as they appear on the Macintosh.

Extending the menu bar

The menu bar for a program is created by calling the factory method **createMenuBar**, which has the following definition in the **Program** class:

```
protected ProgramMenuBar createMenuBar() {
   return new ProgramMenuBar(this);
}
```

In order to create a different menu bar, the application programmer can override this method so that it returns a different **ProgramMenuBar**, which will typically be an instance of a programmer-defined subclass of the base class defined in the **acm.program** package. For example, a specific application program could redefine the factory method as

```
protected ProgramMenuBar createMenuBar() {
   return new MySpecialMenuBar(this);
}
```

With this definition, the program would create its menu bar by constructing an instance of **MySpecialMenuBar** instead.

The **ProgramMenuBar** class is a subclass of the standard Swing **JMenuBar** class. You can add **JMenu**s and **JMenuItem**s to it, just as you would assemble any menu bar. To include the standard menu items, you can use any of the convenience methods shown in Figure 6-10. For example, to include a **Spell Check** menu item in the **Edit** menu, you could override the **createEditMenu** method like this:

```
protected JMenu createEditMenu() {
   JMenu menu = new JMenu("Edit");
   addEditItems(menu);
   menu.addSeparator();
   JMenuItem spellCheckItem = new JMenuItem("Spell Check");
   spellCheckItem.addActionListener(this);
   menu.add(spellCheckItem);
   return menu;
}
```

In addition to illustrating how the menu is created, this code example also shows how the programmer can specify the response to the menu item. As in the standard menu-handling paradigm in Swing, the strategy is to add an action listener to the menu item. When the menu item is activated, any associated action listeners will be sent an **actionPerformed** method.

Figure 6-10. Methods included in ProgramMenuBar to simplify menu construction

Factory methods to create an entire menu
protected JMenu createFileMenu() Creates the standard **File** menu; subclasses can override this method to change this menu.
protected JMenu createEditMenu() Creates the standard **Edit** menu; subclasses can override this method to change this menu.
Methods for adding standard menu item sets
protected void addSaveItems(JMenu menu) Adds the standard **Save** and **Save As** items to the specified menu.
protected void addPrintItems(JMenu menu) Adds the standard **Print**, **Print Console**, and **Script** items to the specified menu.
protected void addQuitItem(JMenu menu) Adds the standard **Quit** item (called **Exit** under Windows) to the specified menu.
protected void addEditItem(JMenu menu) Adds the standard **Cut**, **Copy**, **Paste**, and **Select All** items to the specified menu.

Implementation issues with respect to menu bars

Menu bars proved to be one of the most difficult aspect of the **acm.program** package to get right. In many ways, the overarching design criterion for menu bars is that they should behave so intuitively that the user does not have to think about them at all. Unfortunately, achieving this goal means that menu bars must be implemented in a system dependent way because each platform has a particular notion of what constitutes intuitive behavior. On Microsoft Windows and most flavors of Unix that support window systems, the menu bars are included in the program frame. On the Apple Macintosh, the menu bar appears at the top of the entire screen. In that context, having two menu bars—one for the window and one for the application—only generates confusion. When a program is running as an applet, the appropriate structure for a menu bar depends on the capabilities of the browser.

At the same time, it is essential to shield that platform dependence from programmers who are coding at the novice level. The strategy that we eventually chose hides that complexity in the code for the **ProgramMenuBar** class. On most systems for which a **JMenuBar** is the appropriate structure, the **Program** class simply installs the menu bar in the appropriate place in the **JFrame** used in the application. On the Macintosh, the **Program** class installs an AWT **MenuBar** instead but ensures that the effect of that **MenuBar** is what the application designer expects. Each **MenuItem** in the AWT **MenuBar** listens to the corresponding **JMenuItem** in the **ProgramMenuBar** and changes when that item changes. Thus, if the programmer enables or disables a menu item in the **ProgramMenuBar**, that change will be reflected in the Macintosh menus as well.

6.7 Other features of the Program class

The **Program** class includes a few important features beyond those outlined in the preceding sections. Although each of these features was mentioned briefly at the end of the introduction to this chapter, it is useful to describe them in more detail.

Automatic identification of the main class

One of the most important simplifications offered by the **Program** class is the elimination of the static **main** method that has been cited by so many instructors as a source of difficulty. As noted in section 6.4, the **Program** class includes its own **public static void main** method, which students therefore do not have to write. The implementation of the **main** method in the **Program** class executes the following steps:

1. Determines the identity of the actual class that the user intended to run.
2. Instantiates a new instance of that class by calling its default constructor.
3. Creates a new **JFrame** for the program if its content pane has any components.
4. Resizes the frame as described in the following section.
5. Invokes the **init** method, as in the **Applet**/**JApplet** paradigm.
6. Invokes **validate** to ensure that components are correctly sized in the frame.
7. Invokes the **run** method in the **Program** object being executed.

At the time of the February 2005 release of this rationale document, the Task Force believed that it was not possible to implement this strategy on all of the major platforms. The only problematic step is the first: determining the identity of the class that the user intended to run, which we will identify as the *main class*. While this problem might seem straightforward, the current design of Java provides no platform-independent mechanism to do so. Fortunately, there are platform-specific mechanisms that work for Microsoft Windows, Mac OS X, and Linux, which are the primary platforms we expect adopters to use. The current implementation of the **Program** class supports each of these platforms.

If you are using a platform not included in this list or are working with an Integrated Development Environments (IDE) that has not been updated to support the Java Task Force packages, you may have to adopt a slightly more cumbersome strategy. Even before attempting its platform-dependent algorithm to identify the main class, the **Program** initialization code parses the arguments passed to **main**. Any parameter that has the form

name=*value*

is interpreted as a parameter definition and made available to the program through the **getParameter** method defined by the **Applet** class. The parameters that appear in the HTML **APPLET** tag are treated specially at startup time and can be used to designate not only the main class but other such useful parameters as the width and height of the window. These parameters can be passed explicitly on the command line like this:

```
> java HelloWorld code=HelloWorld.class width=300 height=200
```

This strategy can also be used in IDEs that make it possible to pass an explicit list of command-line arguments to the program being invoked. In those environments, the instructor can provide students with a template for assignments that includes the **code** parameter.

If the strategy of passing parameters does not work, it is always possible to add an explicit **public static void main** method to the **Program** code. The code is entirely canonical and looks like this:

```
public static void main(String[] args) {
   new MainClass().start(args);
}
```

where *MainClass* is the name of the main class. Including this method ensures maximum portability but significantly increases the conceptual overhead necessary for students to write their first programs.

We have also taken steps to ensure that this problem can eventually be solved in a much cleaner way. Together with Cay Horstmann, we submitted a request to the Java Community Process to include a new property called **java.main** that can be obtained through **System.getProperty**. The current implementation of the **Program** class already looks for this property. When it is implemented, a great deal of unnecessary code can be eliminated.

Setting the size and location of the application window

In an applet, the size of the window is determined by the **width** and **height** parameters to the **applet** tag in an HTML file. Applications, however, have no direct counterpart. The traditional strategy for building applications requires the programmer to set the bounds explicitly with a **setSize** or **setBounds** call applied to the **Frame** or **JFrame** object that encloses the application. When one is using the **acm.program** package, setting up that frame is under the control of the **Program** startup code, which suggests that a different approach is necessary to setting size and location.

The **Program** startup code chooses the size and location of the frame by applying the following strategies in order:

1. Look for command-line parameter definitions with the names **x**, **y**, **width**, and **height**. If these parameters are supplied, the integer value that follows is used. Thus, if you invoke an application using

   ```
   java HelloWorld code=HelloWorld.class width=300 height=200
   ```

the width and height of the application frame will be set to 300 and 200, respectively.

2. If no command line parameters are defined for these values, look for static constant definitions of the following variables in the main class: **APPLICATION_X**, **APPLICATION_Y**, **APPLICATION_WIDTH**, and **APPLICATION_HEIGHT**. If any of these variables is defined, the program uses its value to set the corresponding parameter.
3. If neither of these strategies succeeds, use default values for these parameters as specified in the **Program** class.

Applet capabilities

Because **Program** is an indirect subclass of **Applet**, programs have access to the methods provided by Java applets even when they run as applications. Of these capabilities, the most useful are the following:

- *Code base resources*. Applets have the ability to read resources—typically images and audio clips—from their code or document base. When a program runs as an application, its code and document base are set to the current directory. Although these facilities do make it possible to load resources in the way that applets traditionally have, the **acm.util** package described in Chapter 8 includes a **MediaTools** class that offers a more robust set of tools that works in both the applet and application environments.
- *Simplified parameter access*. Applets can be given parameters using the **param** tag in the HTML file. Programs running as applications automatically scan the command-line arguments and interpret any argument of the form

  name=*value*

  as a parameter definition.
- *Status display*. The **showStatus** method in an applet is redefined so that programs running as applications print the status line on the standard output stream.

6.8 Strategies for using the Program class

One of the most important learning experience to come out of the Task Force was that different people teaching introductory computer science have widely divergent views about how to teach programming in an object-oriented language such as Java. Many instructors put off any significant coverage of objects until late in the introductory course, after the students have already learned to use the more traditional procedural paradigm. Even among those who believe that it is important to introduce objects early—the curriculum strategy that *Computing Curricula 2001* [ACM01] called the *objects-first approach*—there are significant strategic and tactical differences as to how to do so.

The central point of contention is which aspects of object-oriented programming need to be brought out early, and which can be deferred. Some instructors, for example, believe that it is important to emphasize the notion of message passing and, in particular, to ensure that students are comfortable with the receiver-based syntax used in object-oriented languages to invoke a method on an object:

receiver`.`*method*`(`*arguments*`)`

For instructors who support this view, the specification of a receiver object is the key concept to learn early. Some instructors emphasize the role of the receiver by requiring students to specify **this** when an object makes calls to its own methods.

Another group of instructors holds that the critical concept to present early is the overall hierarchical structure of objects and classes. In this view, students should learn

early the difference between objects and classes along with the idea that objects of one class are also objects of its various superclasses. This contingent tends to present examples of inheritance early, emphasizing the notion that subclasses extend the behavior of their superclasses. By contrast, instructors in the community that focuses on message-passing typically regard inheritance as less essential in the early part of the term.

Although strong opinions were expressed within the Task Force on both sides of this debate, it seems clear that each model has been used successfully to introduce object-oriented programming to students. It would not make sense for the Task Force to endorse one model over the other. Instead, it is important for the ACM packages to support each of these models well and enable students to use either style from the very beginning of the course. The following section offers several implementations of a flower-drawing program that illustrates each of these styles.

A simple graphical example: Drawing a flower using `GTurtle`

During the first weeks of an introductory programming course, many instructors introduce some sort of microworld, such as the LOGO system described by Seymour Papert in *Mindstorms* [Papert80] or Richard Pattis's *Karel the Robot* [Pattis94]. Although the Task Force did not develop a complete microworld system, the graphical objects—particularly **`GTurtle`**—can fulfill much the same role. Figure 6-11, for example, shows one approach to writing a graphics program that uses **`GTurtle`** to draw a flower consisting of 36 squares with a 10-degree rotation between each one.

Figure 6-11. A single-method program that draws a turtle flower

```
/*
 * File: DrawTurtleFlower.java
 * ---------------------------
 * This program draws a turtle flower using receiver-based
 * invocations in a single, undecomposed run method.
 */

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

public class DrawTurtleFlower extends GraphicsProgram {

/**
 * Runs the program.  This program creates a GTurtle object,
 * puts it in the center of the screen, and then draws a flower.
 * The flower consists of 36 squares, with a 10-degree rotation
 * between each one.  The squares, in turn, are drawn by drawing
 * four line segments interspersed with 90-degree rotations.
 */
   public void run() {
      GTurtle turtle = new GTurtle();
      add(turtle, getWidth() / 2, getHeight() / 2);
      turtle.penDown();
      for (int i = 0; i < 36; i++) {
         for (int j = 0; j < 4; j++) {
            turtle.forward(100);
            turtle.left(90);
         }
         turtle.left(10);
      }
   }
}
```

Although this implementation illustrates the idea of sending a message to a **GTurtle** object and gives students practice with **for** loops, it takes no advantage of the power of decomposition, which is one of the fundamental programming principles that microworlds make so easy to illustrate. Unfortunately, breaking the **run** method down into components is not as simple as it might appear. The primary issue you need to resolve is how to share the **GTurtle** object with the subsidiary methods. Passing it as a parameter to each level has stylistic advantages but seems likely to add more complexity than novice programmers can easily assimilate. The other obvious approach is to declare **turtle** as an instance variable for the class, which gives rise to the code in Figure 6-12.

Figure 6-12. A first attempt at decomposing the DrawTurtleFlower program

```
/*
 * File: DrawTurtleFlower.java
 * ---------------------------
 * This program draws a turtle flower using receiver-based
 * invocations and two levels of decomposition.
 */

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

public class DrawTurtleFlower extends GraphicsProgram {

/**
 * Runs the program.  This program creates a GTurtle object,
 * puts it in the center of the screen, and then draws a flower.
 */
   public void run() {
      turtle = new GTurtle();
      add(turtle, getWidth() / 2, getHeight() / 2);
      turtle.penDown();
      drawFlower();
   }

/** Draws a flower with 36 squares separated by 10-degree turns. */
   private void drawFlower() {
      for (int i = 0; i < 36; i++) {
         drawSquare();
         turtle.left(10);
      }
   }

/** Draws a square with four lines separated by 90-degree turns. */
   private void drawSquare() {
      for (int i = 0; i < 4; i++) {
         turtle.forward(100);
         turtle.left(90);
      }
   }

/** Holds the GTurtle object as an instance variable */
   private GTurtle turtle;

}
```

The problem with this coding is that the structure seems unnatural to programmers who are predisposed to an early introduction of inheritance. Conceptually, this program defines a turtle. You tell the turtle to turn left by writing

```
turtle.left();
```

but draw a square by writing

```
drawSquare();
```

The asymmetry is confusing. Why isn't **drawSquare** also a message to the turtle in the way that **left** is?

The answer, of course, is simply that the methods **drawFlower** and **drawSquare** are defined as methods in the **DrawTurtleFlower** class and not in the **GTurtle** object stored in the variable **turtle**. That object knows how to respond to the **left** message but not to the **drawFlower** message. It is, however, easy enough to reorganize the program so that the main program instantiates a new **FlowerTurtle** subclass that has these capabilities. That program appears in Figure 6-13. Note that the new coding no longer requires an instance variable and that it has a consistent calling structure. The **DrawTurtleFlower** class creates a new **FlowerTurtle** object and sends it the **drawFlower** message using the receiver-based syntax

```
turtle.drawFlower();
```

Inside the definition of **FlowerTurtle**, however, all methods are invoked without an explicit receiver object, because they operate on the current instance of **FlowerTurtle**. Some instructors regard this coding as emblematic of good object-oriented design; others regard the absence of receivers as a sure indication that the program is not really object-oriented at all. The point is that the two coding styles are likely to appeal to instructors with different emphases, and it is important for the Java Task Force to support both constituencies.

Using GTurtle subclasses as standalone programs

Although the code in Figure 6-13 offers an example of how inheritance can be employed in a useful way, it may be beyond the level of what one can teach at the very beginning of an introductory course. If nothing else, the program contains two separate classes: the **DrawTurtleFlower** class that acts as the main program and **FlowerTurtle** that extends **GTurtle** so that it can draw a flower. It's hard enough to get new students to understand a single class at the beginning of the course, and introducing two might send some over the edge.

To make this style of coding easier for students and teachers to use, the **GTurtle** class includes a special hook that allows it to operate as a standalone program. If a **GTurtle** or one of its subclasses is specified as the main class, its implementation of **main** starts up the program by performing the following operations:

1. Instantiate an object of the desired **GTurtle** subclass using exactly the strategy described in the earlier subsection entitled "Automatic identification of the main class."
2. Instantiate a new **GraphicsProgram** object and installing it in the frame, just as if the main class were itself a graphics program and not a graphical object.
3. Add the **GTurtle** object at the center of the canvas installed in the **GraphicsProgram**.
4. Invoke the **run** method in the **GTurtle** subclass.

Figure 6-13. A DrawTurtleFlower implementation using a FlowerTurtle subclass

```
/*
 * File: DrawTurtleFlower.java
 * ---------------------------
 * This program draws a turtle flower using a GTurtle subclass
 * that knows how to draw flowers.
 */

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

public class DrawTurtleFlower extends GraphicsProgram {

/**
 * Runs the program.  This program creates a FlowerTurtle object,
 * centers it in the screen, and then asks it to draw a flower.
 */
   public void run() {
      GTurtle turtle = new FlowerTurtle();
      add(turtle, getWidth() / 2, getHeight() / 2);
      turtle.penDown();
      turtle.drawFlower();
   }
}

/**
 * A GTurtle subclass that knows how to draw a flower.
 */

class FlowerTurtle extends GTurtle {

/** Draws a flower with 36 squares separated by 10-degree turns. */
   public void drawFlower() {
      for (int i = 0; i < 36; i++) {
         drawSquare();
         left(10);
      }
   }

/** Draws a square with four lines separated by 90-degree turns. */
   private void drawSquare() {
      for (int i = 0; i < 4; i++) {
         forward(100);
         left(90);
      }
   }
}
```

This interpretation provides a very simple framework for creating animated graphical programs in the first few days of the term, as shown in Figure 6-14.

Figure 6-14. Using the FlowerTurtle subclass as a standalone program

```
/*
 * File: FlowerTurtle.java
 * -----------------------
 * This program draws a turtle flower by invoking a GTurtle
 * object as if it were a program.
 */

import acm.graphics.*;

public class FlowerTurtle extends GTurtle {

/** Runs the program. */
   public void run() {
      penDown();
      drawFlower();
   }

/** Draws a flower with 36 squares separated by 10-degree turns. */
   private void drawFlower() {
      for (int i = 0; i < 36; i++) {
         drawSquare();
         left(10);
      }
   }

/** Draws a square with four lines separated by 90-degree turns. */
   private void drawSquare() {
      for (int i = 0; i < 4; i++) {
         forward(100);
         left(90);
      }
   }
}
```

6.9 The decision to use the `Applet` model

As the discussion in this chapter emphasizes, the **`Program`** class attempts to combine the best features of applications and applets and to make the facilities of both mechanisms available to students. Most development environments today support the application paradigm, but the ability to run programs as applets offers the obvious advantage of making it possible to run programs on the web. That capability makes it easier for the Java Task Force to create demonstration programs, but also holds forth the promise of enabling students to make their programs easily available to friends and family.

Despite these advantages, the decision to use the applet paradigm flies in the face of the direction that Java seems to be taking. Support for applets is eroding in the Java community, as evidenced by the fact that Sun's Java tutorial no longer uses applets for its demos. In light of this decline in the fortunes of the applet paradigm, it is important for us to outline why we have chosen to embrace applets in the design of the **`acm.program`** package. The sections that follow recount the evolution of that decision and offer several reasons why we believe that this strategy will provide a stable approach to the problem of creating web-accessible programs.

The rise and fall of the applet paradigm

Ever since its introduction in 1995, educators have recognized that Java represents an attractive environment for the development of interactive teaching materials, primarily because of its integration with the web through the applet mechanism. The ITiCSE conferences in both 1997 and 1998 included working groups seeking to develop Java as a resource for creating visualizations and interactive animations for instructional use [Naps97, Bergin98]. The 1997 report offers the following assessment of the opportunities that Java provides:

> Visualization has long been an important pedagogical tool in CS education. The widespread use of the Web and the introduction of Java, with its ability to present interactive animated applets and other types of animation, all provide opportunities to expand the availability of visualization-based teaching and learning. [Bergin98]

In recent years, however, the use of web-based resources in Java—and particularly those based on the applet paradigm—have declined in popularity as it became more difficult to maintain compatibility among applets running in an ever-expanding array of browsers that often implement radically different versions of the Java Development Kit (JDK). These compatibility problems have caused applets to fall out of favor in the Java community, where they have been replaced in many environments by plug-ins that implement the Java virtual machine or by the Java Web Start technology supported by Sun [Sun05]. That compatibility issues and the continuing evolution of the JDK are at the root of this change is clearly reflected in the following introductory passage from a 2001 article in *Java World:*

> Java applets fueled Java's initial growth. The ability to download code over the network and run it on a variety of desktops offering a rich user interaction proved quite compelling. However, Java's Write Once, Run Anywhere (WORA) promise soon became strained as browsers began to bloat and several incompatibilities emerged that were caused by the Java language itself. [Srinivas01]

The strategy of using Java Web Start or browser-specific plugins does not represent a real solution to the problem of browser incompatibility. Java Web Start, for example, is not implemented on all platforms and, even when implemented, typically requires the user to download and install the appropriate plug-in. More importantly, Java Web Start applications run only if the version of the Java runtime on the user's system is up to date with respect the web-based code. In a sense, the abandonment of the applet paradigm has not fostered any increase in compatibility but merely made it possible for both Sun Microsystems and the browser vendors to declare those incompatibilities to be, in effect, "someone else's problem." What has changed is that there is no longer an expectation that applets will work compatibly in different browsers, forcing the adoption of an alternate strategy.

Making applets work

The reality of the situation with regard to applets, however, is not quite as gloomy as the preceding section implies. Although it is impossible to ensure that *any* Java applet will run correctly in all browsers, it is possible to engineer *specific* applets so that they run acceptably well in most of them. The difficulty, however, lies in the fact that making applets run in a wide range of browsers forces one to adopt a lowest-common-denominator strategy that runs counter to the modern approaches to Java that one wants to offer to students.

To illustrate the severity of the problem, it is still the case that almost any browser you encounter in a commercial Internet-access storefront will be running the 1.1 version of the JDK. If you walk into your local Kinko's outlet or the nearest easyInternetCafe, JDK 1.1 is all you find. Given that JDK 1.2 was released in December 1998, these browsers are more than eight years out of date, which is an eternity in the fast-paced world of computing technology. JDK 1.1 lacks support for Swing, for collections, and for many other tools that modern Java programmers cannot live without. The cost of living in that world seems to far outweigh any possible benefits.

It is, of course, quite legitimate to ask why anyone would care about old browsers such as those in Internet storefronts. That is not, presumably, where most students work, nor is it the place from which prospective adopters are most likely to look at the Java Task Force materials. Anyone who is prepared to teach Java must be willing to do so in a more modern Java environment, and it would be wrong to encourage them to do otherwise.

That view, however, may be too optimistic. While it is unlikely that students in a well-endowed university find themselves faced with obsolete technology, the same cannot be said for schools in low-income areas or institutions that focus on distance-learning and consequently have less control over the hardware and software environments their students use. In those environments, it is not at all unlikely that someone will boot up their machine, invoke the browser that comes for free on the desktop, and discover that it doesn't run any of their programs. The failure mode associated with running a new applet in an old browser is also particularly bad from a pedagogical point of view. The applet simply fails to load, and the error message—if it is visible at all—tends to be entirely unhelpful.

But there is yet another consideration that militates in favor of allowing programs to run in older environments. One of the great things about allowing student programs to run as applets is that doing so permits them to share those programs easily with relatives and friends. Think for a moment about a student from the inner city, the first in the family to attend college, who takes an introductory computing course and completes an assignment that generates a great deal of pride. Such students should be encouraged to put their programs up on the web and to tell their parents about them. The parents may not have access to the web outside of a Kinko's or a public library, and might very well be unable to obtain a better, more modern browser. If the student's applet runs in that environment, the parents will be able to see it; if not, they are out of luck.

Evolution of the Java Task Force strategy

As noted in Chapter 1, the February 2005 release of the Java Task Force materials used only JDK 1.1 constructs in the implementation of the library itself, thereby making it possible for applets built using the libraries to run in the most commonly available browsers. This approach generated considerable criticism. Much of that criticism reflected the mistaken belief that this decision in some way constrained adopters to adhere to the same restrictions. In fact, the initial implementation of the packages did take advantage of JDK 1.2 features if they were available. If they were not, the packages tried to do something reasonable to ensure that the programs would run on old browsers that did not support those features.

At its meeting in June 2005, the Task Force decide that it had to abandon that position and adopt a more modern Java baseline for its own implementation. After some deliberation, we chose JDK 1.4 as the appropriate compromise in that it supports more of Java than our earlier design but does not insist on a version that is still unsupported on many common platforms. Our reasons for making that decision were as follows:

- The introduction of the **acm.gui** package essentially forced us to use some Swing features. Earlier implementations of the JTF packages did not need to refer to Swing classes because they worked in what was largely an orthogonal domain. The GUI components, however, were firmly rooted in the Swing model.
- It was impossible to use the code for the library packages as a model if we continued to adhere to the JDK 1.1 rule. Although the code used no deprecated features, it also did not take advantage of newer features that we would like students to use. Moreover, to the extent that the code attempted to use more advanced features, it did so through reflection, thereby rendering the code much more difficult to follow.
- We discovered through experience that it was nearly impossible to dispel the misconceptions about the reasons for adopting the JDK 1.1 strategy. Too many people decided that our implementation decision inevitably meant that the JTF packages were obsolete. Although we could convince most detractors otherwise if we had the opportunity to explain our position on the issue, there was nothing we could do about potential adopters who chose to reject the JTF packages based on a misconception that we had no opportunity to correct.

At the same time, we did not wish to abandon the goal of allowing the JTF packages to function in the widest possible set of browsers. In particular, we wanted to code the libraries using JDK 1.4 but somehow have it run in JDK 1.1 browsers. At first glance, this goal sounds impossible. One possible approach that might have succeeded was to develop two versions of the **acm.jar** library, one for use with JDK 1.4, and one for use with JDK 1.1 to provide legacy browser support. That strategy, however, requires maintaining two separate versions of the library code, which would represent a maintenance nightmare. We regarded that strategy as unacceptable and did not pursue it.

The best of both worlds

After giving up on the idea of supporting legacy browsers on several occasions, we eventually hit upon a strategy that provides exceptional browser compatibility while allowing a modern coding style. Although it at first seems difficult to imagine, the strategy offers all of the following advantages:

1. The library packages themselves are written in a pure JDK 1.4 style without special hooks to ensure backward compatibility. As a result, the library packages can serve as a model for other developers.
2. The code can be compiled in a way that allows it to run in JDK 1.1 browsers.
3. Only one version of the library code exists.

The key to making this strategy work lies in providing additional classes that implement simplified versions of the JDK 1.4 classes on which the library depends. If the program and libraries are compiled with these additional classes, the resulting code runs on JDK 1.1 browsers. If the programs and libraries are compiled alone, the program relies instead on the system implementation of these classes. Operationally, students can include the compatibility code by compiling their programs with the **acm11.jar** file instead of the standard **acm.jar**. Because the **acm11.jar** file includes the additional classes designed to ensure backward compatibility, the resulting program runs in JDK 1.1 environments.

The essential character of this strategy is easiest to illustrate by example. Suppose that a student wants to include a button in an application. In any modern version of Java, that button will be implemented using Swing's **JButton** class. Unfortunately, JDK 1.1 browsers don't know about the **JButton** class, which was introduced in JDK 1.2. It is

hardly appropriate to suggest that students use the older **Button** class instead, because doing so would require them to use a Java paradigm that has long been obsolete. On the other hand, it is perfectly acceptable for students to write their code using **JButton** but to provide that class in the **acm11.jar** file. The student's code is written precisely as it should be, but compatibility can nonetheless be maintained by simulating the **JButton** class in terms of more primitive AWT features.

The **acm11.jar** file contains implementations for every class that is defined in Chapter 9 as part of the JTF subset but which is not already part of JDK 1.1. These classes include the collection classes in the **java.util** package and the most common interactors defined in **javax.swing**. It does not, however, need to implement those features of Swing that lie outside the JTF subset. The resulting simplification turns out to be enormous. The code to accomplish the goal of simulating the new classes in the JTF subset, while not at all trivial, is surprisingly small. Given that the subset includes only a very small fraction of the classes involved in the Swing domain, the basic functionality of those classes can be simulated using less than two percent of the source code required for the full implementation of Swing.

In point of fact, this strategy is not quite as easy to implement as it sounds. The new classes defined as part of the **acm11.jar** file cannot actually live in the **java** and **javax** package hierarchy because the security manager for applets ordinarily makes it illegal to load user classes into those packages. (Being able to define new classes in system packages would indeed represent a significant security loophole in that such classes would have access to the protected information for that package.) Thus, the new classes provided by **acm11.jar**—such as the backward-compatible implementation of **JButton**—have to live somewhere else. At first glance, that restriction suggests that the user's code would need to include additional **import** statements to gain access to those classes, which would violate the principle that there be only one version of the source. In actuality, the problem is even worse than that. If the **acm11.jar** file did assign the compatibility classes to a new package, the references to those classes would typically ambiguous, given that a **JButton** class, for example, would then exist in both **javax.swing** and the package containing the compatibility classes.

There is, however, a simple stratagem that avoids both the ambiguity and the need for additional **import** statements. If the compatibility classes are defined in the same package as the one in which the reference occurs, the compiler will use those definitions before looking at the imports. The **acm11.jar** library therefore defines the set of classes required for compatibility as part of the unnamed package typically used in introductory courses. If a student program is compiled using the **acm.jar** library, any references to **JButton** will be resolved to the **javax.swing** package. If the program is instead compiled with **acm11.jar**, references to **JButton** will be resolved within the unnamed package. The **acm11.jar** uses a similar strategy to ensure that the code for the ACM packages themselves have access to the full set of classes in the recommended JTF subset. If one of the ACM packages uses a class that is not available in JDK 1.1, the **acm11.jar** defines a class with that name as part of the appropriate ACM package to ensure that the code will run on any JDK 1.1 browser.

Chapter 7
The **acm.gui** Package

Of all the problems reported by the computing education community that arise from using Java at the introductory level, the most difficult one for the Task Force to solve was the lack of appropriate components for creating simple graphical user interfaces, more commonly known as GUIs. In the taxonomy presented in Chapter 3, this problem was expressed as follows:

A3. GUI components inappropriate for beginners

From our initial search for strategies to address this problem, we concluded that the chance of finding a solution that would appeal to any large segment of the education community was remote and that the tools already provided by Java were likely to prove more successful than any alternative we could provide. In fact, the original decision of the Task Force was to abandon the search for a solution to this problem after several failed attempts to come up with a satisfactory design. In the first draft of this Rationale document, we summarized our failure to identify a workable solution as follows:

> Unfortunately, the Task Force has not been able to come up with a satisfactory design. What's more, some members of the Task Force have become convinced that it is not possible to create a design that a significant fraction of the potential audience would accept as a standard. That conclusion may be incorrect. There may be a design out there around which a consensus might emerge. As of this release, however, the Task Force has not been able to find it.[JTF05]

In the feedback that we received in response to the draft proposal, however, we were strongly encouraged to look harder for a solution to this problem, which was clearly an important one for a significant fraction of our prospective audience. In response to that demand, we went back to the drawing board to see if we could fashion a synthesis of the sort we had found for the **acm.graphics** package described in Chapter 5. After experimenting with several designs, we were able to engineer an **acm.gui** package that simplifies the creation and layout of GUI components in a way that is simple for novices and, at the same time, allows for a straightforward transition to the standard Java layout managers. This chapter describes the final design of that package, along with the general principles that led the Task Force to adopt it.

Although the design of the **acm.gui** package is largely independent of the **acm.program** package described in Chapter 6, we expect that most adopters will use the GUI tools in the context of a **Program** object. The examples in this chapter therefore are designed to use **acm.program** as their application framework. Adapting those examples so that they use a **JFrame** object instead of a **Program** is straightforward, but is not spelled out in detail.

7.1 What's missing from Java's standard GUI libraries

In many ways, developing a simple GUI package for Java is difficult not because Java's existing libraries are lacking essential functionality but rather because they contain so much that is good. A large fraction of the classes in the **javax.swing** package are specifically designed to support the development of graphical user interfaces and work well at that task. The collection of GUI widgets available in Swing is extraordinarily rich and, moreover, seems on the whole to be well designed. Students have access to buttons, scrollbars, sliders, text areas, selectable lists, popup menus, and a host of other interactor

classes that enable them to write graphical user interfaces containing most if not all of the interactor patterns they are likely to see in professional software. And while it may require some sophistication to use the more advanced features of these interactors, most of the Swing classes specify reasonable defaults that make it easy to use these interactors in conventional ways. Novices, for example, have little trouble using **`JButton`**s in their code once they master the standard listener paradigm Java uses for all event handling. From the outset, the Task Force recognized that trying to offer a replacement for **`JButton`** would be a losing proposition. The standard Swing class is entirely serviceable, and it would make no sense to ask students to learn some parallel structure that they would quickly have to abandon.

At the same time, educators had certainly complained about the inappropriateness of Java's GUI components for novices. Clearly, something was missing. The important question for the Task Force was to figure out what those missing elements might be.

Input from the community

To answer the question of what was missing from Java's standard packages in terms of GUI support, the Task Force began by looking at the proposals we had received to see what aspects of the GUI-construction process those packages had sought to change. The three proposals that most directly addressed this question were the following:

1. The Java Power Tools (JPT) collection developed at Northeastern University [Raab00]. The JPT package offers many programming tools that address a much wider range of issues than the simple GUI-support issue. Two of the submissions from the JPT community—a pair of classes for creating tabular component layouts [Rasala04c] and a collection of GUI widgets [Rasala04d]—addressed the question of GUI development directly.
2. The BreezySwing package developed by Ken Lambert and Martin Osborne [Lambert04a]. This package provided a few simple interactor classes along with a framework for placing components in a two-dimensional grid. The goal of the package was to make it easier for students to write realistic, event-driven programs than it would be using Swing classes alone. At the same time, the authors sought to avoid modal dialogs that distort the classical paradigms of GUI-based programming and to provide a lightweight API that was sufficiently close to the Swing paradigms to avoid having students learn a style of programming that was substantially at odds with the standard approach.
3. The KlassroomSwing package written by Dean Sanders at Northwest Missouri State University [SandersD04]. In his problem statement, Sanders notes that this package "grew out of frustration with existing packages," several of which solve certain aspects of the GUI-creation problem, but of which "none are wholly suitable." The KlassroomSwing package offers a set of classes that closely parallels the standard interactor classes in Swing, but in a simplified form. It also includes support for several additional capabilities, most notably images, audio, and video, which extend the package beyond the narrow confines of the GUI domain.

As was the case in our design of the **`acm.graphics`** package, the Java Task Force identified strengths in each of these proposals, but felt that none were suitable as they stood. The common problem in each of these packages was that they differed much more substantially from the Swing standard than we were prepared to accept. We also found the strongest features of each package occurred in complementary aspects of their designs, making it possible to create an improved package by adopting the best features of each.

The important realization that we derived from studying these proposals in detail is that the critical problem does not lie with the Swing interactor classes themselves, but rather with the facilities that the **java.awt** package uses for placing those interactors on the screen, which is largely unchanged in the Swing world. Richard Rasala offers the following succinct expression of this principle in the introduction to one of his proposals to the Task Force:

> A central problem in GUI construction is the composition of the individual GUI widgets into some organized arrangement within a panel or frame. To avoid manual pixel positioning, Java uses *layout managers* to layout components. Unfortunately, the layout managers provided by Java are very problematic. The simple ones do not do enough and the advanced ones are very hard to use and often do not produce the expected results after all the work. [Rasala04c]

The essential problem can be expressed more precisely as follows: Given the standard facilities available in Java's AWT and Swing packages, there is no good way to create a two-dimensional layout in which the preferred sizes of the components determine the sizes of the cells in the grid. The problem is not that the necessary functionality is unavailable but rather that the standard classes are difficult for novices to use. Java's **GridBagLayout** class provides the necessary capability, but is not suitable for novices for the reasons described in the following section.

The shortcomings of GridBagLayout

In accordance with our general principle of minimizing the number of new classes outside the standard Java package framework, the Task Force would have been happier not to create any new layout manager classes. Unfortunately, after looking closely at the problems associated with **GridBagLayout**, the Task Force—along with the various people who submitted proposals in this area—became convinced that the difficulty of using **GridBagLayout** and the bad habits it encourages outweigh the advantage of maintaining compatibility.

The most significant problems associated with **GridBagLayout** are as follows:

- *It is too complex for novices*. In order to use **GridBagLayout**, clients have to create a constraint object using the class **GridBagConstraints**, assign values to the fields of that **GridBagConstraints** object, and then pass that object as a parameter to the **add** method. Using this approach forces students to understand object construction and assignment to member variables, which students may not have at the point in the class at which GUI programming is introduced.
- *It requires the use of programming practices that many instructors would prefer to avoid*. In modern object-oriented programming, using an object as if it were a traditional record variable has largely fallen out of favor. Doing so undermines the essential concept of encapsulation by making it possible for clients to reach inside the object and change its field values with no regard for the consistency of the resulting object. To avoid such violations of conceptual integrity, most object-first curricula today emphasize the idea of using accessor and mutator methods for any such access. The **GridBagConstraints** class does not include such methods and therefore forces its clients to write explicit field assignments like

  ```
  gbc.gridwidth = 2;
  ```

- *The implementation of the constraint mechanism seems to violate the rules of object behavior*. Consider, for a moment, the following lines of code, which are adapted from the example in the **GridBagConstraints** documentation:

```
GridBagConstraints gbc = new GridBagConstraints();
gbc.gridwidth = 3;
add(new JButton("Button1"), gbc);
gbc.gridwidth = 1;
add(new JButton("Button2"), gbc);
```

The effect of the code seems straightforward enough once the student understands the purpose of the **gridwidth** field: the code installs two buttons, the first of which takes up three columns and the second which takes up only one. Unfortunately, if students think about the code more carefully, they might recognize that there is only a single **GridBagConstraints** object involved. Each assignment to the **gridwidth** field updates the object stored in the variable **gbc**, so it might seem as if the constraint for both buttons (since it is the same object) would have **gridwidth** equal to 1 at the end of the code fragment. The code has the intuitive effect only because **GridBagLayout** copies the constraint whenever a new component is added. Were it not for this unadvertised behavior, none of the standard **GridBagLayout** code would work. In a pedagogical sense, however, the mysterious behavior of **GridBagConstraints** will make it harder for them to understand the semantics of objects.

Given these problems, the general desiderata for a new class to replace **GridBagLayout** are for the most part clear. To be successful, the new class should

1. Be much simpler to use than **GridBagLayout**
2. Avoid the design flaws in **GridBagLayout** that interfere with the development of good object-oriented style
3. Maintain compatibility with **GridBagLayout** to ease the eventual transition to the standard Java classes
4. Provide as much of the functionality of **GridBagLayout** as possible

The first three design criteria follow immediately from the principles that the Task Force articulated in Chapter 2. The last design goal, by contrast, is not necessarily as obvious. The external submissions we received typically eliminated some functionality from **GridBagLayout** as a way of simplifying it. Our concern, however, was that such an approach would give some potential adopters a reason for staying with **GridBagLayout**, despite its shortcomings. If the Task Force packages were incapable of duplicating a layout constraint that someone was already using, that person would be less likely to adopt the alternative design. If, on the other hand, one can do everything with the new class that had previously been possible with **GridBagLayout**, much of the resistance to change could be eliminated.

Our solution was to implement a **TableLayout** manager class that offers the functionality of **GridBagLayout** in a way that is much easier for novices to understand. Its motivations are therefore similar to those that gave rise to the BreezySwing package [Lambert04a] and the table capabilities of Java Power Tools [Rasala04c], both of which served as models. The principal differences in our implementation of **TableLayout** are that we wanted to ensure that clients had access to the full set of capabilities from **GridBagLayout** and that we felt it was important for the package to support a smooth transition to the standard **GridBagLayout** approach. The details of the **TableLayout** class are outlined in section 7.3.

Other extensions

Although the single most important problem in the existing Java packages is the lack of a simple tabular layout manager, the Task Force also concluded that the standard Java classes have two additional shortcomings that were important to correct:

1. The existing Java classes make it difficult to read numeric data from the user. For the most part, the problems are the same as those involved in reading numeric data from a console or dialog box, as discussed in Chapter 4. If students are required to perform their own numeric conversion, they must first master such difficult conceptual issues as the use of wrapper classes for numeric types and the details of exception handling. Hiding that complexity simplifies such operations considerably. To do so, the Task Force decided to add two new classes—**`IntField`** and **`DoubleField`**—to simplify the development of applications that require numeric input. These classes are discussed in detail in section 7.2.
2. In many cases, students do not want to design the user interface for an entire application but rather provide some interactors that make it easy to control the activity of some other type of program. For example, it might be useful to create a **`GraphicsProgram`** controlled by a few buttons located at the periphery of the display. This capability is provided by the **`ProgramLayout`** class and has already been described in section 6.5.

7.2 Numeric interactors

As noted in the preceding section, the Task Force decided that it was necessary to provide **`IntField`** and **`DoubleField`** classes to simplify the development of applications that require numeric input. Each of these classes extends **`JTextField`** but provides additional methods to hide the complexity involved in numeric conversion and exception handling. The additional methods available for **`DoubleField`** appear in Figure 7-1; The methods for **`IntField`** are the same except for the expected changes in the argument and result types.

The **`IntField`** and **`DoubleField`** classes are closely related to a similar pair of classes proposed by Lambert and Osborne [Lambert04a]. The versions supplied by the **`acm.gui`**

Figure 7-1. Methods defined in the `DoubleField` class

Constructors

`DoubleField()` Creates a **`DoubleField`** object with no initial value.
`DoubleField(double value)` Creates a **`DoubleField`** object with the specified initial value.
`DoubleField(double low, double high)` Creates a **`DoubleField`** object whose value is constrained to the specified limits.
`DoubleField(double value, double low, double high)` Creates a **`DoubleField`** object with the specified initial value and limits.

Methods to set and retrieve the value of the field

`void setValue(double value)` Sets the value of the field and updates the display.
`double getValue()` Returns the value in the field. If the value is out of range, errors or retries occur here.

Methods to control formatting

`void setFormat(String format)` Sets the format string for the field as specified in the **`DecimalFormat`** class in **`java.text`**.
`String getFormat()` Returns the current format string.

Additional methods (unlikely to be used by novices)

`void setExceptionOnError(boolean flag)` Sets the error-handling mode: **`false`** means retry on error, **`true`** means raise an exception.
`boolean getExceptionOnError()` Returns the error-handling mode, as defined in **`setExceptionOnError`**.

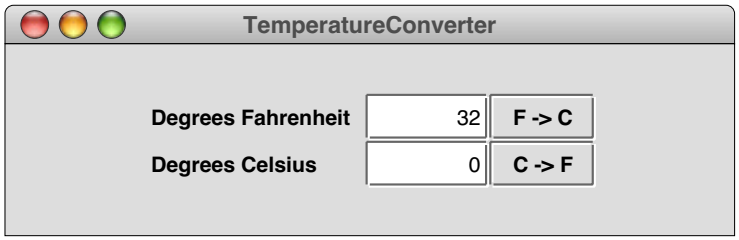
package have been redesigned to make the behavior of those classes more consistent with the **readInt** and **readDouble** methods defined for the **IOConsole** class in **acm.io**. In particular, the **IntField** and **DoubleField** classes adopt the paradigm of their **acm.io** counterparts in that their default behavior gives the user a chance to reenter values in the case of illegal input but allows more sophisticated applications to catch the exception that occurs in such cases.

The only significant design decision for these classes was how to control format. It turns out that the **DoubleField** class is useful in practice only if the client has some way of specifying the format of the displayed result. In the absence of format control, the display typically contains so many digits as to become unreadable. The **setFormat** and **getFormat** methods shown in Figure 7-1 eliminate this problem by allowing the client to specify the output format. The format itself is specified using a string as defined in the **DecimalFormat** class in **java.text**. The use of format codes is illustrated in the currency converter program shown in Figure 7-2 later in this chapter.

7.3 The TableLayout Class

The easiest way to understand how the **TableLayout** class works is to look at a simple example. The usual strategy for building applications that use **TableLayout** to create the user interface is to implement a class that extends **Program** with an individually designed constructor that assembles the interactors into the desired arrangement. The first line of the constructor is typically a call to **setLayout**, which establishes the number of rows and columns in the tabular grid. The rest of the constructor then creates the necessary interactors and adds them to the table, filling each row from left to right and then each row from top to bottom. This strategy is illustrated in Figure 7-2, which implements a simple temperature converter.

The user interface for the **TemperatureConverter** program looks like this:



The user can type values into either of the **IntField** interactors and then perform the conversion to the other scale either by hitting the appropriate button or by hitting the ENTER key in the interactor itself. Each of these actions generates an **ActionEvent** whose action command is either the string **"F -> C"** or **"C -> F"** depending on which button or interactor generated the event. These events are fielded by the **actionPerformed** method in the class, which performs the necessary conversion and then updates the value of the corresponding field.

In terms of understanding what the **TableLayout** framework provides, all of the important code appears in the constructor. The line

```
setLayout(new TableLayout(2, 3));
```

defines the layout for the program as a whole to be a **TableLayout** object with two rows and three columns. The rest of the constructor adds interactors to the table layout in order, filling each horizontal row from left to right, and then proceeding through each row

Figure 7-2. Temperature conversion program

```
/*
 * File: TemperatureConverter.java
 * -------------------------------
 * This program allows users to convert temperatures
 * back and forth from Fahrenheit to Celsius.
 */

import acm.gui.*;
import acm.program.*;
import java.awt.event.*;
import javax.swing.*;

/** This class implements a temperature converter. */
public class TemperatureConverter extends Program {

/** Constructor to layout the components. */
   public TemperatureConverter() {
      setLayout(new TableLayout(2, 3));
      fahrenheitField = new IntField(32);
      fahrenheitField.setActionCommand("F -> C");
      fahrenheitField.addActionListener(this);
      celsiusField = new IntField(0);
      celsiusField.setActionCommand("C -> F");
      celsiusField.addActionListener(this);
      add(new JLabel("Degrees Fahrenheit"));
      add(fahrenheitField);
      add(new JButton("F -> C"));
      add(new JLabel("Degrees Celsius"));
      add(celsiusField);
      add(new JButton("C -> F"));
      addActionListeners();
   }

/** Listen for a button action. */
   public void actionPerformed(ActionEvent e) {
      String cmd = e.getActionCommand();
      if (cmd.equals("F -> C")) {
         int f = fahrenheitField.getValue();
         int c = (int) Math.round((5.0 / 9.0) * (f - 32));
         celsiusField.setValue(c);
      } else if (cmd.equals("C -> F")) {
         int c = celsiusField.getValue();
         int f = (int) Math.round((9.0 / 5.0) * c + 32);
         fahrenheitField.setValue(f);
      }
   }

/* Private instance variables. */
   private IntField fahrenheitField;
   private IntField celsiusField;
}
```

from top to bottom. In this example, each row contains a **JLabel** indicating the temperature scale for that row, an **IntField** that allows the user to enter a value, and a **JButton** to trigger the conversion. The code that sets the action command for the **IntField** and adds the program as a listener is required only to enable the ENTER key in those interactors. The buttons would be active even in the absence of those calls.

The feature that sets **TableLayout** apart from the simpler **GridLayout** mechanism is that the sizes of each component in the table are adjusted according to their preferred sizes and the constraints imposed by the grid. The **JLabel** objects are of different sizes, but the implementation of **TableLayout** makes sure that there is enough space in the first column to hold the longer of the two labels. By default, each component added to a **TableLayout** container is expanded to fills its grid cell, although this behavior can be changed by specifying constraints as described in the following section.

What the **TemperatureConverter** program illustrates is that the student can rely on the **TableLayout** to arrange the elements appropriately on the screen without having to specify the detailed constraint information that would be required using **GridBagLayout**. By defining the appropriate defaults, all the student usually has to do is count the number of rows and columns in the user interface and add the interactors in the appropriate order.

Specifying constraints

Unlike the packages that were proposed to the Task Force, the **TableLayout** class allows the programmer to specify constraints that provide fine-grained control over the layout process. For many applications, such constraints are unnecessary. In other cases, however, it is important to be able to specify specific field sizes, to control the alignment of a cell, or to merge cells horizontally or vertically so that they span several element positions. The **GridBagLayout** paradigm uses a separate class called **GridBagConstraints** to represent these constraints. The primary advantage of **TableLayout** is that it hides the **GridBagConstraints** object from the programmer.

Instead of allocating a constraints object explicitly, the **TableLayout** design allows clients to specify the constraints in string form. For every field in Java's **GridBagConstraints**, the **TableLayout** manager accepts a constraint string in the form

field=*value*

where *field* is the name of the **GridBagConstraints** field and *value* is a value appropriate for that field. For example, to duplicate the effect of setting the **gridwidth** field of a constraints object to 2 (thereby specifying a two-column field), adopters of the **acm.gui** package can simply specify the constraint string

```
"gridwidth=2"
```

instead of the more confusing process of allocating a **GridBagConstraints** object and setting the appropriate field, like this:

```
GridBagConstraints gbc = new GridBagConstraints();
gbc.gridwidth = 2;
```

The strings used as constraint objects can set several fields at once by including multiple field/value pairs separated by spaces. Moreover, for those fields whose values are defined by named constants in the **GridBagConstraints** class, **TableLayout** allows that name to be used as the value field of the constraint string. For example, the following string indicates that a field should span two columns but that the component should fill space only in the *y* direction:

```
"gridwidth=2 fill=VERTICAL"
```

Constraint strings are checked at run time to make sure that the fields and values are defined and are consistent. The case of letters, however, is ignored, which makes it possible to name the fields in a way that is consistent with Java's conventions. Thus, if an instructor wants to emphasize the case convention that has each word within a multiword identifier begin with an uppercase letter, it is equally effective to write

```
"gridWidth=2 fill=VERTICAL"
```

The complete list of constraints supported by the **`TableLayout`** class is shown in Figure 7-3. The first block shows the constraints that adopters are likely to use; the second block consists of constraints that are included only to maintain symmetry with the **`GridBagConstraints`** class.

To emphasize the advantage of using the **`TableLayout`** model, Figure 7-4 and 7-5 offer two implementations of a program that lays out **`Button`**s objects (the code comes

Figure 7-3. Constraints supported by the `TableLayout` class

Constraints that clients will often find useful

`gridwidth=`*columns* *or* **`gridheight=`***rows*
Indicates that this table cell should span the indicated number of columns or rows.

`width=`*pixels* *or* **`height=`***pixels*
The **`width`** specification indicates that the width of this column should be the specified number of pixels. If different widths are specified for cells in the same column, the column width is defined to be the maximum. In the absence of any **`width`** specification, the column width is the largest of the preferred widths. The **`height`** specification is interpreted symmetrically for row heights.

`weightx=`*weight* *or* **`weighty=`***weight*
If the total size of the table is less than the size of its enclosure, **`TableLayout`** will ordinarily center the table in the available space. If any of the cells, however, are given nonzero **`weightx`** or **`weighty`** values, the extra space is distributed along that axis in proportion to the weights specified. As in the **`GridBagLayout`** model, the weights are floating-point values and may therefore contain a decimal point.

`fill=`*fill*
Indicates how the component in this cell should be resized if its preferred size is smaller than the cell size. The legal values are **`NONE`**, **`HORIZONTAL`**, **`VERTICAL`**, and **`BOTH`**, indicating the axes along which stretching should occur. The default is **`BOTH`**.

`anchor=`*anchor*
If a component is not being filled along a particular axis, the **`anchor`** specification indicates where the component should be placed in its cell. The default value is **`CENTER`**, but any of the standard compass directions (**`NORTH`**, **`SOUTH`**, **`EAST`**, **`WEST`**, **`NORTHEAST`**, **`NORTHWEST`**, or **`SOUTHEAST`**, **`SOUTHWEST`**) may also be used.

Less useful constraints included for compatibility with `GridBagLayout`

`top=`*pixels* *or* **`bottom=`***pixels* *or* **`left=`***pixels* *or* **`right=`***pixels*
Supplies an inset value for the specified edge of the component in the cell.

`ipadx=`*pixels* *or* **`ipady=`***pixels*
Specifies additional padding internal to the component that increases its preferred size along the indicated axis. The extra pixels are added on both sides of the component, so the size increases by twice the value supplied.

`gridx=`*column* *or* **`gridy=`***row*
These constraints specify the location of the component within the grid and will not ordinarily be used in the **`TableLayout`** model, unless you are trying to teach how **`GridBagLayout`** works.

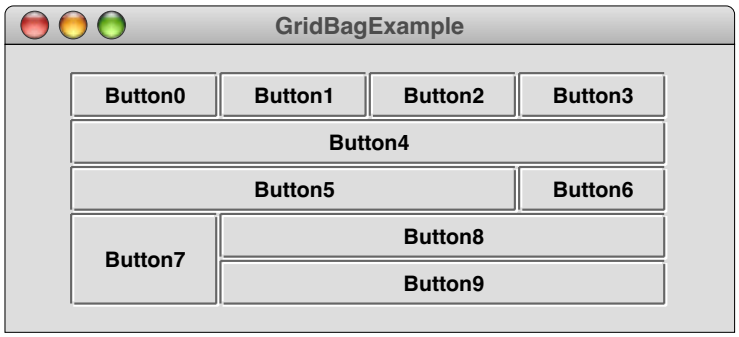
Figure 7-4. Example from the `GridBagLayout` documentation

```
/*
 * File: GridBagExample.java
 * -------------------------
 * This application uses GridBagLayout to construct a
 * two-dimensional table of buttons.  The example is
 * adapted from the GridBagLayout documentation.
 */

import java.awt.*;
import javax.swing.*;

public class GridBagExample extends JPanel {

   public GridBagExample() {
      GridBagLayout gridbag = new GridBagLayout();
      GridBagConstraints c = new GridBagConstraints();
      setLayout(gridbag);
      c.fill = GridBagConstraints.BOTH;
      addButton("Button0", gridbag, c);
      addButton("Button1", gridbag, c);
      addButton("Button2", gridbag, c);
      c.gridwidth = GridBagConstraints.REMAINDER;
      addButton("Button3", gridbag, c);
      addButton("Button4", gridbag, c);
      c.gridwidth = 3;
      addButton("Button5", gridbag, c);
      c.gridwidth = GridBagConstraints.REMAINDER;
      addButton("Button6", gridbag, c);
      c.gridwidth = 1;
      c.gridheight = 2;
      addButton("Button7", gridbag, c);
      c.gridwidth = GridBagConstraints.REMAINDER;
      c.gridheight = 1;
      addButton("Button8", gridbag, c);
      addButton("Button9", gridbag, c);
   }

   private void addButton(String name,
                          GridBagLayout gridbag,
                          GridBagConstraints c) {
      Button button = new Button(name);
      gridbag.setConstraints(button, c);
      add(button);
   }

   public static void main(String[] args) {
      GridBagExample example = new GridBagExample();
      JFrame frame = new JFrame("GridBagExample");
      Container contentPane = frame.getContentPane();
      contentPane.setLayout(new BorderLayout());
      contentPane.add(BorderLayout.CENTER, example);
      frame.setSize(350, 200);
      frame.setVisible(true);
   }
}
```

Figure 7-5. Similar application using `TableLayout`

```
/*
 * File: TableExample.java
 * -----------------------
 * This application uses the ACM TablePanel class to
 * construct a two-dimensional table of buttons that
 * matches the one from GridBagExample.
 */

import acm.gui.*;
import java.awt.*;

public class TableExample extends Program {

   public TableExample() {
      setLayout(new TableLayout(5, 4));
      add(new Button("Button0"));
      add(new Button("Button1"));
      add(new Button("Button2"));
      add(new Button("Button3"));
      add(new Button("Button4"), "gridwidth=REMAINDER");
      add(new Button("Button5"), "gridwidth=3");
      add(new Button("Button6"));
      add(new Button("Button7"), "gridheight=2");
      add(new Button("Button8"), "gridwidth=REMAINDER");
      add(new Button("Button9"), "gridwidth=REMAINDER");
   }

}
```

from the **`GridBagLayout`** documentation, which still uses the JDK 1.1 style) in the following grid:



As the code for the figures makes clear, the **`TableLayout`** version is less than half the size of the **`GridBagLayout`** example and is significantly easier to read. At the same time, the underlying discipline is compatible, which should allow users to migrate from one model to the other.

Controlling spacing

Several of the layout managers in the standard Java packages—including, for example, **`FlowLayout`**, **`BorderLayout`**, and **`GridLayout`**—make it possible to control the spacing between components in the display by setting **`hgap`** and **`vgap`** parameters to specify the spacing in the horizontal and vertical dimensions, respectively. The standard **`GridBagLayout`** mechanism does not do so, even though it would be extremely useful.

The **TableLayout** manager does support these parameters, which can be set either by using the four-argument version of the constructor

```
public TableLayout(int rows, int columns, int hgap, int vgap)
```

or by calling the **setHgap** and **setVgap** methods on an existing layout. Setting these parameters ensures that each row or column in the table is separated from the next by the specified number of pixels, making it easy to add spacing around the interactors.

Although the **hgap** and **vgap** parameters are typically positive, it turns out that the value –1 can be extremely useful in this setting. If you specify –1 as a gap value, each component will be positioned so that it overlaps the preceding one by one pixel. If the components have a one-pixel border, this strategy ensures that the border running between the components will be only one pixel wide. If no gap applied, each component would have a one-pixel border, which would result in a two-pixel divider on the screen.

7.4 The TablePanel Classes

The examples presented so far in this chapter use **TableLayout** as the layout manager for the central region of a program, which is likely to be its most common application in the introductory curriculum. The **TableLayout** manager, however, can be used with any container and is extremely useful in assembling patterns of interactors.

To make it easier to assemble nested containers hierarchically, the **acm.gui** package includes three convenience classes that extend **JPanel** but install an appropriate **TableLayout** manager. These classes and their constructor patterns appear in Figure 7-6.

The **HPanel** and **VPanel** classes make it easy to create complex assemblages of interactors by decomposing them hierarchically into rows and columns. In this respect, they have at least a common purpose with the **BoxLayout** manager introduced in the **javax.swing** package. The panel **HPanel** and **VPanel** classes, however, offer far more flexibility because they have the full power of the **TableLayout** class. The **BoxLayout** manager, by contrast, makes it difficult to do anything except to string together components in a linear form with no control over spacing or format.

Figure 7-6. Convenience classes based on TableLayout

TablePanel constructors

public TablePanel(int rows, int columns) Creates a **JPanel** with the indicated number of rows and columns.
public TablePanel(int rows, int columns, int hgap, int vgap) Creates a **JPanel** with the specified dimensions and gaps.

HPanel constructors

public HPanel() Creates a **JPanel** consisting of a single horizontal row.
public HPanel(int hgap, int vgap) Creates an **HPanel** with the specified gaps (**vgap** applies above and below the row).

VPanel constructors

public VPanel() Creates a **JPanel** consisting of a single vertical column.
public VPanel(int hgap, int vgap) Creates an **VPanel** with the specified gaps (**hgap** applies to the left and right of the column).

Chapter 8
The **acm.util** Package

The **acm.util** package includes a set of utilities that are either simple standalone tools or that are shared resources for the other packages. The package consists of the following classes, presented here in an order that roughly reflects the likelihood that someone adopting the ACM packages would use them:

- **ErrorException**—Allows errors to be reported in a consistent way
- **CancelledException**—Allows a dialog to signal clients that it has been cancelled
- **RandomGenerator**—A more pedagogically defensible random number interface
- **Animator**—Implements a **Thread** subclass with an exceptionless **pause** method
- **Platform**—Contains several static methods to support platform-specific code
- **MediaTools**—Implements a flexible mechanism for loading images and sounds
- **JTFTools**—Contains a set of static methods used in several of the JTF packages
- **OptionTable**—Implements a facility for parsing option values from a string

The sections that follow provide a brief overview of each mechanism without describing the class in detail. For more details, please see the **javadoc** documentation on the web site.

8.1 The ErrorException class

The **ErrorException** class is an extremely simple class whose only purpose is to provide the ACM packages with a consistent way to report errors. When errors are detected in the package, or in any other code that adopts the same convention, those errors are indicated by throwing an exception like this:

```
throw new ErrorException(message);
```

Any caller in the dynamic execution chain can catch this error if they want to handle it; if it is uncaught, the program will terminate with an unhandled exception error. The key advantage is that **ErrorException** is a subclass of **RuntimeException** and therefore need not be declared in **throws** clauses.

8.2 The CancelledException class

The **CancelledException** class is used to represent an exception that signals cancellation of an operation. Although the only class that throws this exception is the **IODialog** class in **acm.io**, we decided to put it in acm.util to make it more generally available. In particular, it is possible that future extensions to the **Animator** class might need to signal cancellation.

8.3 The RandomGenerator class

In response to our call for community input last spring, Alyce Brady of Kalamazoo College offered a pair of simple proposals that resonated with many of the Task Force members. After noting that random numbers play an important role in many introductory courses, Alyce made the following observation:

> The **java.util.Random** class, with its **nextInt**, **nextDouble**, and **nextBoolean** methods is a good choice to use, but students often fall into the trap of constructing multiple **Random** instances in order to generate multiple random numbers. This can lead to behavior that appears far from random if the different **Random** objects generate equivalent or similar sequences of random numbers. Another problem with the **Random** class, which is not as insignificant as it sounds, is that the name is misleading. A **Random** instance is actually a random number generator, not a random object. [Brady04a]

For precisely the reasons that Alyce describes here, we have included a **RandomGenerator** class in the **acm.util** package. **RandomGenerator** is a subclass of **java.util.Random** and therefore provides all the methods that are available in that class. In addition, however, the **RandomGenerator** class offers the additional methods shown in Figure 8-1.

Figure 8-1. Additional methods in the RandomGenerator class

int nextInt(int low, int high) Returns a random integer in the specified range (inclusive).
double nextDouble(double low, double high) Returns a random **double** in the specified range.
boolean nextBoolean(double p) Returns a random **boolean** that is **true** with probability **p** (0 = never, 1 = always).
Color nextColor() Returns a random opaque color.

8.4 The Animator class

The **Animator** class is a simple subclass of **Thread** designed to support simpler animation. The only important new behavior that it adds to the standard **Thread** class is a method called **pause(***time***)** that suspends for the specified number of milliseconds. This method is therefore similar to **Thread.sleep** but is easier for novices to use because it never throws an exception. Thus, an animator can delay its own operation by writing

```
pause(time);
```

rather than the ungainly

```
try {
   Thread.sleep(time);
} catch (InterruptedException ex) {
   /* Empty */
}
```

The **Animator** class was suggested by Kim Bruce. [Bruce04b]

8.5 The Platform class

The **Platform** class implements a variety of convenience methods for the ACM packages that allow the implementations to determine properties about the environment. The intent of these facilities is not to produce programs that behave differently on different platforms, but instead to overcome any known platform incompatibilities to ensure that code works in as portable a way as possible. The package also includes several methods that look at the runtime environment to determine what capabilities are supported by the virtual machine. These facilities are used to take advantage of modern Java facilities while remaining compatible with earlier versions of the JDK.

8.6 The `MediaTools` class

The **`MediaTools`** class provides several static methods that support the creation and loading of images and audio clips in a convenient and portable way. Please see the package documentation for details.

8.7 The `JTFTools` class

The **`JTFTools`** class implements a collection of static methods that are used throughout the ACM packages. Several of these methods are potentially useful to clients outside of those packages, but the motivation behind the package is simply to centralize common methods used in the packages to avoid duplication of code.

8.8 The `OptionTable` class

The **`OptionTable`** class was added to the **`acm.util`** package as part of the implementation of the **`TableLayout`** class introduced in section 7.3. It has the responsibility of parsing the string version of the constraint specification, which consists of a set of key/value pairs. The **`OptionTable`** package checks to make sure that a key is valid and makes it easy to specify default options. Although **`TableLayout`** is currently the only client, other developers may want to use the same mechanism.

8.9 Oh `Scanner`, where art thou?

The February 2005 release of the **`acm.util`** package included a **`Scanner`** class that provided most of the features of the Java 2 Standard Edition 5.0 class while remaining compatible with JDK 1.1. Putting that class in the **`acm.util`** package, however, did not turn out to be a workable strategy. Clients working in older Java environments could indeed use this class to gain access to most features of the new **`Scanner`**. Unfortunately, when those clients later upgraded to a J2SE 5.0 system, the old code would often fail to compile. Given that many Java programmers tend to import everything in a package in a single **`import`** line, the decision to call both classes **`Scanner`** made it impossible to include both of the import lines

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

without incurring an ambiguity to **`Scanner`**. At the same time, calling our simulated **`Scanner`** class something else would eliminate much of the reason for its inclusion, which was motivated in large measure by a desire to let students see code that was as close as possible to what they would soon be able to write.

Fortunately, the simulated **`Scanner`** class has not entirely disappeared. Although it no longer exists in the **`acm.util`** package, the same functionality is now available in the supplemental package that allows the ACM libraries to run even in JDK 1.1 environments. The strategy that enables this level of compatibility is described in section 6.3.

Chapter 9
The JTF Subset

In the public announcement of the ACM Java Task Force at SIGCSE 2004, the first item included in our list of deliverables was

> *A definition of a subset of the standard Java APIs appropriate for first-year computer science*. This subset would involve restricting both the number of classes used as well as the number of public methods made visible within those classes. Note that this subset must be sufficient to have students write significant applications using Java. To this end, it will presumably be a superset of the AP Java subset [Astrachan00], which seeks to define what aspects of the language will be tested on the AP exam. [Roberts04b]

This task is clearly a perilous one, particularly given that any decisions we make are sure to upset those instructors who feel compelled to use some piece of the standard Java API that did not make it into the Task Force subset. At the same time, it is also true that the Task Force has a responsibility to provide guidance in this area. In a very real way, the most significant problem with Java is not any of the specific shortcomings that we have sought to address with the ACM packages but rather the enormous scale of the languages and its APIs.

In some ways, however, the issue can be made less contentious by redefining the goal. Rather than defining any sort of official "standard," our current strategy is to define a subset that we agree to support as effectively as possible. That support has three components:

1. Define a set of the classes from the standard Java packages that we agree to use in our own development of JTF materials. We will then agree to use only those classes in any publications, online tutorials, downloadable examples, and extension packages. That subset is described in section 9.1.
2. Create documentation for the JTF subset that is more student-friendly than the existing **javadoc** material. The simplified **javadoc** design appears in section 9.2.
3. Enable implementors who stay within the JTF subset to make their code compatible with older versions of the JDK dating back to the 1.1 implementation. The strategies that make this goal possible were described in section 6.9 earlier in this document.

9.1 The JTF subset

The February 2005 release of this rationale document offered a preliminary proposal for a set of classes that would serve as the JTF subset. Through the feedback we received on the web forum, we have added several classes to that list and taken away a few as well. The current list appears in Figure 9-1.

In putting this list together, we undertook a mechanical survey of the code that is published on the web for several of the leading text books to see what classes are used in practice. This survey provided us with considerable insights into how classes are used, but required some analysis to separate out various artifacts from the useful data. For example, one of the leading classes on the list is **`java.lang.StringBuffer`**, not because these textbooks refer to it directly, but because Java compilers generate references to **`StringBuffer`** (or its newer **`StringBuilder`** counterpart) whenever they process string constants. Thus, it was important to analyze the data and look for this type of outlier. In

Figure 9-1. Current listing of the JTF subset

```
java.applet:
  Applet
  AudioClip

java.awt:
  BorderLayout
  Color
  Component
  Container
  Dimension
  Event
  FlowLayout
  Font
  FontMetrics
  Frame
  Graphics
  GridLayout
  Image
  LayoutManager
  MediaTracker
  Point
  Rectangle
  Toolkit

java.awt.event:
  ActionEvent
  ActionListener
  AdjustmentEvent
  AdjustmentListener
  ComponentEvent
  ComponentListener
  FocusEvent
  FocusListener
  KeyEvent
  KeyListener
  MouseEvent
  MouseListener
  MouseMotionListener
  WindowEvent
  WindowListener

java.io:
  BufferedReader
  BufferedWriter
  File
  FileReader
  FileWriter
  IOException
  PrintWriter
  Reader
  StreamTokenizer
  StringReader
  StringWriter
  Writer

java.lang:
  Boolean
  Character
  Class
  Cloneable
  Comparable
  Double
  Enum
  Float
  Integer
  Long
  Math
  Number
  Object
  Runnable
  String
  System
  Thread

java.math:
  BigInteger

java.net:
  URL
  URLConnection

java.text:
  DateFormat
  DecimalFormat
  NumberFormat

java.util:
  ArrayList
  Arrays
  BitSet
  Collection
  Comparator
  HashMap
  HashSet
  Iterator
  LinkedList
  List
  ListIterator
  Map
  PriorityQueue
  Random
  Scanner
  Set
  SortedMap
  SortedSet
  Stack
  StringTokenizer

javax.swing:
  Box
  BoxLayout
  ButtonGroup
  JApplet
  JButton
  JCheckBox
  JComboBox
  JComponent
  JDialog
  JFileChooser
  JFrame
  JLabel
  JList
  JOptionPane
  JPanel
  JPopupMenu
  JRadioButton
  JScrollBar
  JScrollPane
  JSlider
  JSpinner
  JTable
  JTextArea
  JTextField
  JTextPane
  JToggleButton
  JWindow
  KeyStroke
  Timer

javax.swing.event:
  ChangeEvent
  ChangeListener
  ListDataEvent
  ListDataListener
  ListSelectionEvent
  ListSelectionListener
```

the end, we used the list developed by the automatic survey as a "sanity check" as to what classes one might reasonably present in the first-year curriculum.

We expect that this list will evolve slowly over time, but believe that it is likely to remain stable through the official release of the materials in early 2006.

9.2 Simplifying the javadoc presentation

The idea of using the documentation to define the JTF subset was suggested in our original list of deliverables:

> *A public web site containing an updated javadoc reference manual for the approved Java subset*. This web site would make it possible for students to browse the standard classes and methods defined in the subset without being overwhelmed by classes, methods, and concepts they are unlikely to use. For the classes and methods that are included, the web site will contain more examples and tutorial material than is currently supplied with the Java APIs.

After all, given that Java's corporate creator assures us that "the network is the computer," what matters is what's described on the web rather than a standard buried in a bookcase somewhere.

Our proposal to use the **javadoc** documentation to define the JTF subset includes the following goals:

1. Create a web site for the Task Force that includes an abridged form of the standard Java documentation called the **student view** from which classes and methods unlikely to be used in an introductory course have been eliminated. The student view contains pointers to the **complete view**, which contains the full documentation.
2. Publish the documentation of the Java Task Force packages using the same split between the student and complete views. Students can browse a less overwhelming collection of documentation, while teachers and advanced users have access to the full capabilities of the class.
3. Make available the tools we use to produce the documentation, including the driver files that determine which parts of the complete documentation are included in the student view. In that way, any adopter who feels that something is missing can generate new documentation in which that class or method has been restored. And, perhaps more important in practice, instructors who believe that the Task Force has revealed too much can generate even more compact documentation that covers only the elements used in the local curriculum.

The current draft release of the Task Force materials implements the split between the student and complete view only for the documentation of the ACM packages. When you go to the web site, you come up in the student view. Each page in the student view contains a link in the navigation bar to the complete view for the same page (and vice versa), making it very easy to move back and forth between the two views.

The primary characteristic of the student view is that it hides extraneous information. Figure 9-2, for example, shows the top of the page in the student view for the **`GLabel`** class in **`acm.graphics`**. What you see in the figure looks familiar enough and also illustrates the level of documentation we plan to associate with each of the classes. But what's more important is what you don't see. You don't, for example, see a navigation tab directing you to the discussion of inner classes or deprecated methods. Similarly, you don't discover that **`GLabel`** implements the mysterious interfaces **`Cloneable`** and **`Serializable`**. You do, of course, see these features in the complete view, but they are not getting in the way here.

Another important change in the student documentation view is that any inherited methods that the documentation designer chooses to include are listed in the summary section using a format similar to that of the methods implemented by this class. This feature is illustrated in Figure 9-3, which shows the first few entries in the summaries of

Figure 9-2. Beginning of documentation page for the `GLabel` class

Overview Package **Student** Complete Tree Index Help

PREV CLASS NEXT CLASS FRAMES NO FRAMES
SUMMARY: FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

acm.graphics
Class GLabel

java.lang.Object
  |
  +--acm.graphics.GObject
        |
        +--**acm.graphics.GLabel**

```
public class GLabel extends GObject
```

The **GLabel** class is a graphical object whose appearance consists of a text string.

To use the **GLabel** class, the first step is to construct a new **GLabel** object and add it to an existing **GCanvas** (assumed here to be stored in the variable **gc**), as follows:

```
GLabel glabel = new GLabel("Hello, world.", 50, 60);
gc.add(glabel);
```

This code creates a **GLabel** containing the string **"Hello, world."** with its origin at the point (50, 60) and installs the label in the **GCanvas** as shown in the figure to the right.

Most graphical objects in Java use the upper left corner of the figure to define their location. Strings, however, work differently. The location at which a string is displayed is always taken to be at the leftmost edge of the first character along what is called the **baseline**, which is the line on which the uppercase letters sit. Some lowercase letters (**g**, **j**, **p**, **q**, and **y**) descend below the baseline, as do several special characters like the comma. The location of the start of the string and the concept of the baseline are illustrated in the diagram to the right.

the locally defined methods and those that are inherited from superclasses. These methods tend to have equal weight in the student's mind, and it is useful to present them symmetrically. Contrast the view shown in Figure 9-3 with the standard presentation shown in Figure 9-4. In that form, the student has no sense of the structure of these methods and is exposed to several—the three versions of **`wait`** are an obvious example—that offer no help to novices.

The only other significant change in the student view is the inclusion of a "usage" line in the detailed presentation of methods, preceding the declaration of parameters and results. Student find a paradigmatic example extremely useful, particularly when they can cut it out of the documentation and paste it into their own code. This line is illustrated in Figure 9-5.

Figure 9-3. Selections from the `GLabel` method summary

Method Summary

double	**getAscent()** Returns the distance this string extends above the baseline.
GRectangle	**getBounds()** Returns a **GRectangle** that specifies the bounding box for the string.
double	**getDescent()** Returns the distance this string descends below the baseline.
Font	**getFont()** Returns the font in which the **GLabel** is displayed.
FontMetrics	**getFontMetrics()** Returns a **FontMetrics** object describing the dimensions of this string.

. . .

Inherited Method Summary

void	**addMouseListener(MouseListener listener)** Adds a mouse listener to this graphical object.
void	**addMouseMotionListener(MouseMotionListener listener)** Adds a mouse motion listener to this graphical object.
boolean	**contains(GPoint pt)** Checks to see whether a point is inside the object.
boolean	**contains(double x, double y)** Checks to see whether a point is "inside" the string, which is defined to be inside the bounding rectangle.

Figure 9-4. Standard javadoc presentation of inherited methods

Methods inherited from class acm.graphics.GObject

addMouseListener, addMouseMotionListener, angle, contains, contains, cosD, distance, distance, getColor, getLocation, getParent, getSize, getX, getY, isVisible, main, move, movePolar, pause, removeMouseListener, removeMouseMotionListener, round, sendBackward, sendForward, sendToBack, sendToFront, setColor, setLocation, setLocation, setVisible, sinD, tanD, toDegrees, toRadians, toString

Methods inherited from class java.lang.Object

equals, getClass, hashCode, notify, notifyAll, wait, wait, wait

Figure 9-5. Standard javadoc presentation of inherited methods

Method Detail

```
public double getAscent()
```

Returns the distance this string extends above the baseline.

Usage: `double ascent = glabel.getAscent();`
Returns: The ascent of this string in pixels

References

[ACM01] The ACM/IEEE-CS Joint Task Force on Computing Curricula. *Computing Curricula 2001: Final Report of the Joint ACM/IEEE-CS Task Force on Computer Science Education,* Eric Roberts and Gerald Engel (editors). Los Alamitos, CA: IEEE Computer Society Press, December 2001.
http://www.acm.org/sigcse/cc2001/.

[Allen02] Eric Allen, Robert Cartwright, and Brian Stoler. DrJava: a lightweight pedagogic environment for Java. Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education, Cincinnati, KY, February 2002, pp. 137-141.
http://doi.acm.org/10.1145/563340.563395.

[Astrachan00] Owen Astrachan, Robert (Corky) Cartwright, Gail Chapman, David Gries, Cay Horstmann, Richard Kick, Frances Trees, Henry Walker, and Ursula Wolz. Recommendations of the AP Computer Science ad hoc committee, October 2000.
http://cbweb2s.collegeboard.org/ap/pdf/adhoc_report_surveys.pdf.

[Austin04] Calvin Austin. J2SE 1.5 in a nutshell. **http://developers.sun.com**, February 2004.
http://java.sun.com/developer/technicalArticles/releases/j2se15/.

[Bergin98] Joseph Bergin, Thomas L. Naps, Constance G. Bland, Stephen J. Hartley, Mark A. Holliday, Pamela B. Lawhead, John Lewis, Myles F. McNally, Christopher H. Nevison, Cheng Ng, George J. Pothering, Tommi Teräsvirta. Java resources for computer science instruction: Report of the ITiCSE '98 working group on curricular opportunities of java based software development. Proceedings of the 3rd Annual Conference on Innovation and Technology in Computer Science Education, Dublin, Ireland, August 1998, pp. 14-34.
http://doi.acm.org/10.1145/316572.358291.

[Biddle98] Robert Biddle and Ewan Tempero. Java pitfalls for beginners. SIGCSE Bulletin, 30:2, June 1998, pp. 48-52.
http://doi.acm.org/10.1145/292422.292441.

[Brady04a] Alyce Brady. Random Number Generator class that supports a singleton generator. Submission to the Java Task Force, May 1, 2004.

[Brady04b] Alyce Brady. A class to make generating random colors easy. Submission to the Java Task Force, May 1, 2004.

[Brady05] Alyce Brady. Range checking for **readInt**, **readDouble**. Posting to the Java Task Force web forum, April 8, 2005.

[Bruce01] Kim B. Bruce, Andrea Danyluk, and Thomas Murtagh. A library to support a graphics-based object-first approach to CS1. Proceedings of the Thirty-second SIGCSE Technical Symposium on Computer Science Education, Charlotte, NC, February 2001, pp. 6-10.
http://doi.acm.org/10.1145/364447.364527.

[Bruce04a] Kim Bruce. Graphics. Submission to the Java Task Force, April 30, 2004.

[Bruce04b] Kim Bruce. Event handling. Submission to the Java Task Force, April 30, 2004.

[Cooper03] Erb Cooper. Java 1.5: The end of Java? Java.net web log, July 21, 2003.
http://weblogs.java.net/pub/wlg/261.

[Cross04] James H. Cross II, Dean Hendrix, and David Umphress. jGRASP: An integrated development environment with visualizations for teaching Java in CS1, CS2, and beyond. Workshop description for the Thirty-Fifth SIGCSE Technical Symposium on Computer Science Education, Norfolk, VA, March 2004. Further information on jGRASP is available at **http://jgrasp.org**.
http://www.csis.gvsu.edu/~wolffe/SIGCSE2004-Workshops.html.

[Eck02] David Eck. *Introduction to Programming Using Java,* fourth edition, July 2002.
http://math.hws.edu/javanotes/.

[Grissom00] Scott Grissom. A pedagogical framework for introducing Java I/O in CS1. SIGCSE Bulletin, December 2000, pp. 57-59.
http://doi.acm.org/10.1145/369295.369326.

[Hadjerrouit98] Said Hadjerrouit. Java as first programming language: a critical evaluation. SIGCSE Bulletin, June 1998, pp. 43-47.
http://doi.acm.org/10.1145/292422.292440.

[Hartley98] Stephen J. Hartley. "Alfonse, your Java is ready!" Proceedings of the Twenty-Ninth SIGCSE Technical Symposium on Computer Science Education, Atlanta, GA, March 1998, pp. 247-251.
http://doi.acm.org/10.1145/273133.274306.

[Heiss03] Janice J. Heiss. New language features for ease of development in the Java 2 Platform, Standard Edition 1.5: A conversation with Joshua Bloch. Article on **http://developers.sun.com**, May 8, 2003.
http://java.sun.com/developer/technicalArticles/releases/j2se15/.

[Hume00] J. N. Patterson Hume and Christine Stephenson. *Introduction to Programming in Java*. Toronto: Holt Software Associates Inc., 2000.

[Hosch96] Frederick Hosch. Java as a first language: an evaluation. SIGCSE Bulletin, September 1996, pp. 45-50.
http://doi.acm.org/10.1145/234867.234877.

[King97] K. N. King. The case for java as a first language. Proceedings of the 35th Annual ACM Southeast Conference, Murfreesboro, TN, April, 1997.
http://www.gsu.edu/~matknk/java/reg97.htm.

[Koffman01] Elliot Koffman and Ursula Wolz. A simple java package for GUI-like interactivity. Proceedings of the Thirty-second SIGCSE Technical Symposium on Computer Science Education, Charlotte, NC, February 2001, pp. 11-15.
http://doi.acm.org/10.1145/364447.364528.

[Kölling00] Michael Kölling and John Rosenberg. Objects first with Java and BlueJ. Proceedings of the Thirty-first SIGCSE Technical Symposium on Computer Science Education, Austin, TX, March 2000, p. 429.
http://doi.acm.org/10.1145/330908.331912.

[Lambert04a] Ken Lambert and Martin Osborne. BreezySwing. Submission to the Java Task Force, March 12, 2004.

[Lambert04b] Ken Lambert. Turtle graphics. Submission to the Java Task Force, March 17, 2004.

[Lambert04c] Ken Lambert. I/O utilities. Submission to the Java Task Force, March 17, 2004.

[Lea99] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns,* second edition. Boston: Addison-Wesley, 1999.

[Martin98] Peter Martin. Java, the good, the bad and the ugly. SIGPLAN Notices, April 1998, pp. 34-39.
http://doi.acm.org/10.1145/278283.278288.

[Naps97] Thomas Naps, Joseph Bergin, Ricardo Jiménez-Peris, Myles F. McNally, Marta Patiño-Martínez, Viera K. Proulx, Jorma Tarhio. Using the WWW as the delivery mechanism for interactive, visualization-based instructional modules (report of the ITiCSE '97 working group on visualization). Proceedings of the 2nd Annual Conference on Innovation and Technology in Computer Science Education, Uppsala, Sweden, September 1997, pp. 13-26.
http://doi.acm.org/10.1145/266057.266062.

[Papert80] Seymour Papert. *Mindstorms*. New York: Basic Books, 1980.

[Parlante04a] Nick Parlante. Graphics. Submission to the Java Task Force, March 30, 2004.

[Parlante04b] Nick Parlante. CS1-Java—Love and chaos. Posting to **SIGCSE-MEMBERS**, May 13, 2004.
See **http://listserv.acm.org/archives/sigcse-members.html**.

[Pattis94] Richard Pattis. *Karel The Robot: A Gentle Introduction to the Art of Programming*, 2nd edition. New York: John Wiley, 1994.

[Poet00] Java on the MscIT course at the University of Glasgow. Java in the Computing Curriculum 4, January 2000.
See **http://www.ics.ltsn.ac.uk/pub/Jicc4/poet.doc**.

[Raab00] Jeff Raab, Richard Rasala, and Viera K. Proulx. Pedagogical Power Tools for Teaching Java. Proceedings of the 5th Annual Conference on Innovation and Technology in Computer Science Education, Helsinki, Finland, July 2000, pp. 156-159.
http://doi.acm.org/10.1145/343048.343155.

[Rasala00] Richard Rasala. Toolkits in first year computer science: a pedagogical imperative. Proceedings of the Thirty-first SIGCSE Technical Symposium on Computer Science Education, Austin, TX, March 2000, pp. 185-191.
http://doi.acm.org/10.1145/330908.331852.

[Rasala04a] Richard Rasala. Java Power Framework. Submission to the Java Task Force, April 9, 2004.

[Rasala04b] Richard Rasala. I/O in JPT. Submission to the Java Task Force, April 23, 2004.

[Rasala04c] Richard Rasala. GUI Composition: TableLayout and TablePanel. Submission to the Java Task Force, April 29, 2004.

[Rasala04d] Richard Rasala. GUI widgets. Submission to the Java Task Force, April 30, 2004.

[Reges00] Stuart Reges. Conservatively radical Java in CS1. Proceedings of the Thirty-first SIGCSE Technical Symposium on Computer Science Education, Austin, TX, March 2000, pp. 85-89.
http://doi.acm.org/10.1145/330908.331821.

[Reges02] Stuart Reges. Can C# replace Java in CS1 and CS2? Proceedings of the 7th Annual Conference on Innovation and Technology in Computer Science Education, Aarhus, Denmark, June 2002, pp. 4-8.
http://doi.acm.org/10.1145/544414.544419.

[Roberts98] Eric Roberts and Antoine Picard. Designing a Java graphics library for CS 1. Proceedings of the 3rd Annual Conference on Innovation and Technology in Computer Science Education, Dublin, Ireland, August 1998, pp. 213-218.
http://doi.acm.org/10.1145/282991.283129.

[Roberts01] Eric Roberts. An overview of MiniJava. Proceedings of the Thirty-Second SIGCSE Technical Symposium on Computer Science Education, Charlotte, NC, February 2001, pp. 1-5.
http://doi.acm.org/10.1145/364447.364525.

[Roberts04a] Eric Roberts. The dream of a common language: The search for simplicity and stability in computer science education. Proceedings of the Thirty-Fifth SIGCSE Technical Symposium on Computer Science Education, Norfolk, VA, March 2004.
http://doi.acm.org/10.1145/971300.971343.

[Roberts04b] Eric Roberts. Resources to support the use of Java in introductory computer science. Proceedings of the Thirty-Fifth SIGCSE Technical Symposium on Computer Science Education, Norfolk, VA, March 2004.
http://doi.acm.org/10.1145/971300.971384.

[Roberts04c] Eric Roberts. Program class hierarchy. Submission to the Java Task Force, April 30, 2004.

[Roberts04d] Eric Roberts. Console class. Submission to the Java Task Force, April 30, 2004.

[Roberts04e] Eric Roberts. Simple graphics. Submission to the Java Task Force, April 30, 2004.

[SandersD04a] Dean Sanders. Jeroo. Submission to the Java Task Force, April 26, 2004.

[SandersD04b] Dean Sanders. Klassroom Swing. Submission to the Java Task Force, April 30, 2004.

[SandersK04a] Kathryn Sanders. OOPS graphics. Submission to the Java Task Force, April 28, 2004.

[SandersK04b] Kathryn Sanders. NGP graphics. Submission to the Java Task Force, April 28, 2004.

[Srinivas01] Raghavan N. Srinivas. Java Web Start to the rescue. Java World, July 2001.
http://www.javaworld.com/javaworld/jw-07-2001/jw-0706-webstart.html.

[Stein98] Lynn Stein. What we've swept under the rug: Radically rethinking CS1. Computer Science Education 8 (2):1998, pp. 118-129.
http://faculty.olin.edu/~las/2001/07/www.ai.mit.edu/people/las/papers/rug.html.

[Stephenson98] Chris Stephenson and Tom West. Language choice and key concepts in introductory computer science courses. Journal of Research on Computing in Education, Fall 1998, pp. 89-95.

[Sun04] Sun Microsystems. Core Java J2SE 5.0. September 2004.
http://java.sun.com/j2se/1.5.0/.

[Sun05] Sun Microsystems. Java™ Web Start Overview. May 2005.
http://java.sun.com/products/javawebstart/overview.html.

[Tyma98] Paul Tyma. Why are we using Java again? Communications of the ACM, June 1998, pp. 38-42.
http://doi.acm.org/10.1145/276609.27667

[Wallace97] Chris Wallace, Peter Martin, and Bob Lang. Not whether Java but how Java. Paper presented at the Java in the Computing Curriculum conference, London, January 1997.
http://www.ulst.ac.uk/cticomp/not.html.

[Weiss98] Mark Allen Weiss. Experiences teaching data structures with Java. Proceedings of the Twenty-eighth SIGCSE Technical Symposium on Computer Science Education, San Jose, CA, March 1997, pp. 164-168.
http://doi.acm.org/10.1145/268084.268143.

[Wirth02] Niklaus Wirth. Computing science education: The road not taken. Proceedings of the 7th Annual Conference on Innovation and Technology in Computer Science Education, Aarhus, Denmark, June 2002.
http://doi.acm.org/10.1145/544414.544415.

[Wolz99] Ursula Wolz and Elliot Koffman. simpleIO: a Java package for novice interactive and graphics programming. Proceedings of the 4th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education, Cracow, Poland, June 1999, pp. 139-142.
http://doi.acm.org/10.1145/305786.305896.

[Zukowski02] John Zukowski. Java 2 Platform, Standard Edition, Version 1.4 overview. Article on **http://developers.sun.com**, February 2002.
http://java.sun.com/developer/technicalArticles/releases/j2se1.4/.