Submission to the ACM Java Task Force
A Class Hierarchy for Programs

Eric Roberts
Stanford University
eroberts@cs.stanford.edu

Summary

This proposal describes the **`Program`** class hierarchy, a simple set of Java classes with three main goals:

1. To reduce the pedagogical problems associated with the static **`main`** method used in Java applications
2. To eliminate the distinction between applets and applications, making it possible to write code that runs in both domains
3. To provide a compelling example of an inheritance hierarchy that can easily be explained and justified to novices

The behavior of classes in the **`Program`** hierarchy is illustrated by the program shown in Figure 1, which uses the **`ConsoleProgram`** subclass to duplicate the effect of a classic imperative program that adds two numbers and displays their result. The essential idea is that **`ConsoleProgram`**, like any **`Program`**, is in fact both an applet and an application. If it is invoked as an applet, the public **`main`** method is invoked at the conclusion of the applet startup process; if it is invoked as an application, the static **`main`** method that is part of the **`Program`** class creates the necessary frames, instantiates an instance of the appropriate target class, and then invokes the **`main`** method for that class. As a result, the two models end up having the same effect, thereby allowing the same code to work in the context of a browser or in an application.

1. The problem

One of the most common complaints identified in the Java Task Force's draft report on the problems in teaching Java [2]—listed as problem L1 in the taxonomy—is the fact that every Java application must include a **`main`** method with the signature

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

Figure 1. Example of the `ConsoleProgram` class

```
/* File: Add2.java */

import acm.program.*;

/**
 * Reads two numbers from the user and
 * prints their sum.
 */

public class Add2 extends ConsoleProgram
{
  public void main()
  {
    int n1, n2, total;

    println("Program to add two numbers.");
    print("Enter n1: ");
    n1 = readInt();
    print("Enter n2: ");
    n2 = readInt();
    total = n1 + n2;
    println("The total is " + total + ".");
  }
}
```

From a pedagogical perspective, an obvious shortcoming of this declaration is that it includes several syntactic elements—the keywords **`public`** and **`static`**, the types **`void`** and **`String`**, the method parameter **`args`**, and the square brackets designating an array—none of which are meaningful to the novice. This problem, however, is not really that serious, given that introductory students can usually be convinced to accept a certain amount of code beyond their understanding as something of a magical template, particularly if they are told that explanation for the arcane bits of syntax are forthcoming later in the course.

The more significant problem with the required existence of a **`main`** method is that Java applications begin their execution in a static environment in which no objects yet exist. The **`main`** method in an application class cannot call other methods in that class unless those methods are also declared as static. To support an object-oriented approach, the **`main`** method must instead allocate new objects and then either send messages to those objects directly or install them in frames that will allow them to listen for user events. These operations often require a nontrivial amount of code, which adds further complexity to the startup template that students must use without fully understanding.

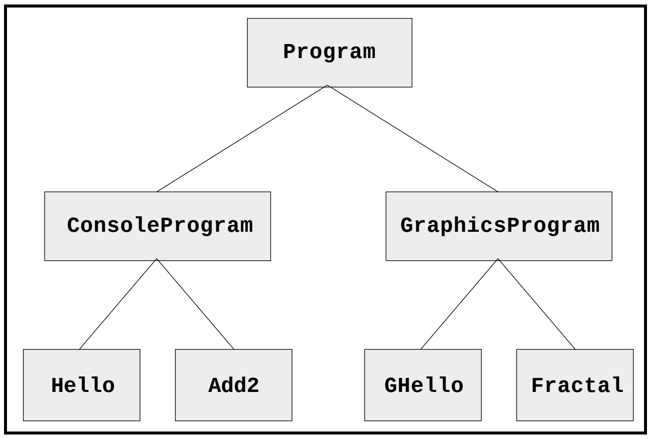
The computational paradigm of an application, moreover, is substantially different from that of an applet. In an applet, the browser automatically instantiates a new instance of the target class, installs that instance in a frame, and then invokes the necessary initialization code. The applet approach is much cleaner in terms of its adherence to principles of object-oriented design, but is falling out of favor as browser support for applets fails to keep pace with the latest versions of Java.

The difference between the application and applet paradigms also creates complications for authors of Java textbooks. To ensure that beginners are not confused by two independent models of computation, many textbooks adopt one strategy or the other, thereby limiting their audience to those who have chosen that particular curricular strategy. As noted in the summary, one of the principal goals of the **`Program`** class hierarchy is to eliminate this distinction by allowing the same code to work in both the application and applet paradigms.

2. Pedagogical advantages of the `Program` class

As it happens, the **`Program`** class hierarchy—while developed initially to address the problems raised by the static **`main`** method in applications—turns out to solve another pedagogical problem in Java that may in fact be more important. One of the challenges that instructors face in developing an objects-first approach is to identify compelling examples of class hierarchies that make intuitive 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 this context: all horses are mammals, all mammals are animals, but there are some mammals that are not horses and some animals that are not mammals. So far, so good. The trouble comes in trying to reflect this same notion in a context that makes sense in programming.

Figure 2. One possible design of the `Program` hierarchy



Most textbooks I've seen tend to use a hierarchy of graphical objects to provide similar models of this hierarchical relationship: all squares are rectangles, all rectangles are shapes, but not all shapes are rectangles, let alone squares.

While the graphical object hierarchy works perfectly well as an example, 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 very first concepts they encounter. By establishing a hierarchy of programs, students can see how their particular example program is an instance of a particular program category, which is itself one of several subcategories of a larger, more generic class of programs.

One possible structure for the **`Program`** hierarchy is illustrated in Figure 2. This version of the hierarchy shows two subclasses, **`ConsoleProgram`** and **`GraphicsProgram`**, which represent different categories of programs that behave in certain respects like any program, but which have particular characteristics appropriate to their class. A **`ConsoleProgram`**, as represented by the **`Add2`** example shown in Figure 1, starts up with a console in the program window and includes methods for reading and writing data to and from that console. A **`GraphicsProgram`**, as represented by the **`GHello`** example in Figure 3, starts up with some kind of graphical canvas in the program window and includes an entirely different set of methods that make it possible to draw images on that window.

Figure 3. Example of the `GraphicsProgram` class

```
/* File: GHello.java */

import acm.program.*;

/**
 * Displays a text message on the screen.
 */

public class GHello extends GraphicsProgram
{
  public static final String MSG =
      "Hello, world.";

  public void main()
  {
    double x, y;

    x = (getWidth() - stringWidth(MSG)) / 2;
    y = getHeight() / 2;
    drawString(MSG, x, y);
  }
}
```

It is important to note that the specific subclasses included here are not formally a part of this proposal and may change depending on what other libraries the committee chooses to adopt. The **`GHello`** example as written assumes the graphics library I have proposed in a separate submission, but one could just as easily imagine having the **`GraphicsProgram`** class instead serve the function of the **`WindowController`** class from the Williams graphics library. [1]

3. Implementation overview

The **`Program`** class in the current implementation of the library is a subclass of **`JApplet`** that also defines a static **`main`** method allowing it to be used as an application. The overall goal is to implement the **`Program`** class so that the behavior of a program is as similar as possible no matter whether it is invoked as an applet or as application. To achieve this goal, the implementation adopts one of the two following strategies:

• If the program is invoked as an applet, it already has an instantiated object as required by the semantics of the **`Program`** class. In this case, the applet machinery invokes the **`init`** method just as it normally would. The call to the **`main`** method is triggered by a separate thread initiated by the **`start`** method, which is also called during the standard applet invocation.

• If the program is invoked as an application, the implementation of the static **`main`** method in the **`Program`** class identifies the subclass to be executed using the strategy outlined late in this section, instantiates an object of that class, creates a frame to house the program, and then invokes the **`init`** and **`main`** methods in the instantiated program object.

Individual program subclasses use the same basic mechanism, but override various hook procedures in the **`Program`** class to produce their specific behavior.

Unfortunately, Java does not in fact allow the **`Program`** startup logic to identify the specific subclass the user intends to invoke. The current implementation therefore must have access to the **`index.html`** file used by the applet, which it parses to determine both the identity of the target class and the desired sizes. Most Java programming environments can automate the creation of the **`index.html`** file as part of a standard template for defining new projects. Moreover, for programs that seek to live without this crutch, it is also possible to add the following template definition, which achieves the same effect, as shown here for the **`Add2`** program:

```
public static void main(String[] argv)
{
  main(new Add2(), argv);
}
```

4. Validation

The **`Program`** class, along with the specific subclasses described in this proposal, has been used successfully over the past four years in the secondary school curriculum developed by the Stanford Bermuda Project (**`http://bermuda.stanford.edu`**), which is in place at all Bermuda public schools.

References

1. 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, Charlotte, NC, February 2001, pp. 6-10.
2. Eric Roberts. Taxonomy of problems in teaching Java. ACM Java Task Force, February 27, 2004.