```
      double a = control.getDouble("lower limit a");
      double b = control.getDouble("upper limit b");
      int n = control.getInt("number of intervals n");
      Function f;
      try {
         f = new ParsedFunction(fstring);
      } catch(ParserException ex) {
         control.println(ex.getMessage());
         plotFrame.clearDrawables();
         return;
      }
      plotFrame.clearDrawables();
      // sets domain of x to integration limits
      plotFrame.setPreferredMinMaxX(a, b);
      plotFrame.addDrawable(new FunctionDrawer(f));
      RectangularApproximation approximate =
           new RectangularApproximation(f, a, b, n);
      plotFrame.addDrawable(approximate);
      plotFrame.setMessage("area = "
           +decimalFormat.format(approximate.sum));
      control.println("approximate area under curve = "
           +approximate.sum);
   }

   public static void main(String[] args) {
      CalculationControl.createApp(new NumericalIntegrationApp());
   }
}
```

We consider the accuracy of the rectangular approximation for the integral of $f(x) = \cos x$ from $x = 0$ to $x = \pi/2$ by comparing the numerical results shown in Table 11.1 with the exact answer of unity. Note that the error decreases as n^{-1}. This observed n^{-1} dependence of the error is consistent with the analytic derivation of the n-dependence of the error obtained in Appendix 11A. We explore the n-dependence of the error associated with other numerical integration methods in Problems 11.1 and 11.2.

Problem 11.1 The rectangular and midpoint approximations

(a) Test `NumericalIntegrationApp` by reproducing the results in Table 11.1.

(b) Use the rectangular approximation to determine the approximate definite integrals of $f(x) = 2x + 3x^2 + 4x^3$ and $f(x) = e^{-x}$ for $0 \le x \le 1$ and $f(x) = 1/x$ for $1 \le x \le 2$. What is the approximate n-dependence of the error in each case?

(c) A straightforward modification of the rectangular approximation is to evaluate $f(x)$ at the *midpoint* of each interval. Define a `MidpointApproximation` class by making the necessary modifications and approximate the integral of $f(x) = \cos x$ in the interval $0 \le x \le \pi/2$. Remember that you need to change how the rectangles are drawn. How does the magnitude of the error compare with the results shown in Table 11.1? What is the approximate dependence of the error on n?

(d) Use the midpoint approximation to determine the definite integrals considered in part (b). What is the approximate n-dependence of the error in each case? ■

Table 11.1 Rectangular approximations of the integral of $\cos x$ from $x = 0$ to $x = \pi/2$ as a function of n, the number of intervals. The error Δ_n is the difference between the rectangular approximation and the exact result of unity. Note that the error Δ_n decreases approximately as n^{-1}; that is, if n is increased by a factor of 2, Δ_n is decreased by a factor 2.

n	F_n	Δ_n
2	1.34076	0.34076
4	1.18347	0.18347
8	1.09496	0.09496
16	1.04828	0.04828
32	1.02434	0.02434
64	1.01222	0.01222
128	1.00612	0.00612
256	1.00306	0.00306
512	1.00153	0.00153
1024	1.00077	0.00077

Problem 11.2 The trapezoidal approximation

(a) Modify your program so that the trapezoidal approximation is computed and determine the n-dependence of the error for the same functions as in Problem (11.1). What is the approximate n-dependence of the error in each case? Which approximation yields the best results for the same computation time?

(b) It is possible to double the number of intervals without losing the benefit of the previous calculations. For $n = 1$ the trapezoidal approximation is proportional to the average of $f(a)$ and $f(b)$. In the next approximation, the value of f at the midpoint is added to this average. The next refinement adds the values of f at the $1/4$ and $3/4$ points. Modify your program so that the number of intervals is doubled each time and the results of previous calculations are used. The following should be helpful:

```
if (n == 1) {
   double midpoint = a + 0.5*(b-a);
   sum = (b - a)*(0.5*f(a) + 0.5*f(b) + f(midpoint));
   n = 2;
} else {
   double delta = (b - a)/n;      // separation between new points
   double x = a + 0.5*delta;
   double intermediateSum = 0.0;
   for (int i = 0; i < n; i++) {
      intermediateSum += f(x);
      x += delta;
   }
   // 0.5*delta = separation between points
   sum = 0.5*(intermediateSum*delta + sum);
   n *= 2;
}
```
■