Input/Output and Functions

CS 121: Data Structures

Attendance Quiz

Attendance Quiz: Arrays

- Scan the QR code, or find today’s attendance quiz under the “Quizzes” tab on Canvas
- Password: announced in class
- After five minutes, we will discuss the answers

Attendance Quiz: Arrays

- Write your name
- Translate the following pseudocode into a Java program, Arrays.java

```
Create an array of strings containing the course IDs of courses you’re enrolled in this
semester (e.g., "CS121")

For each course ID:

  Print "Course #N: COURSE_ID" (e.g., "Course #1: CS121”, "Course #2: CS…", …)
```

Considering Subscribing to the “CS Interest” Mailing List

- Announcements of events like:
  - Welcome back luncheon
  - Leetcode programming practice

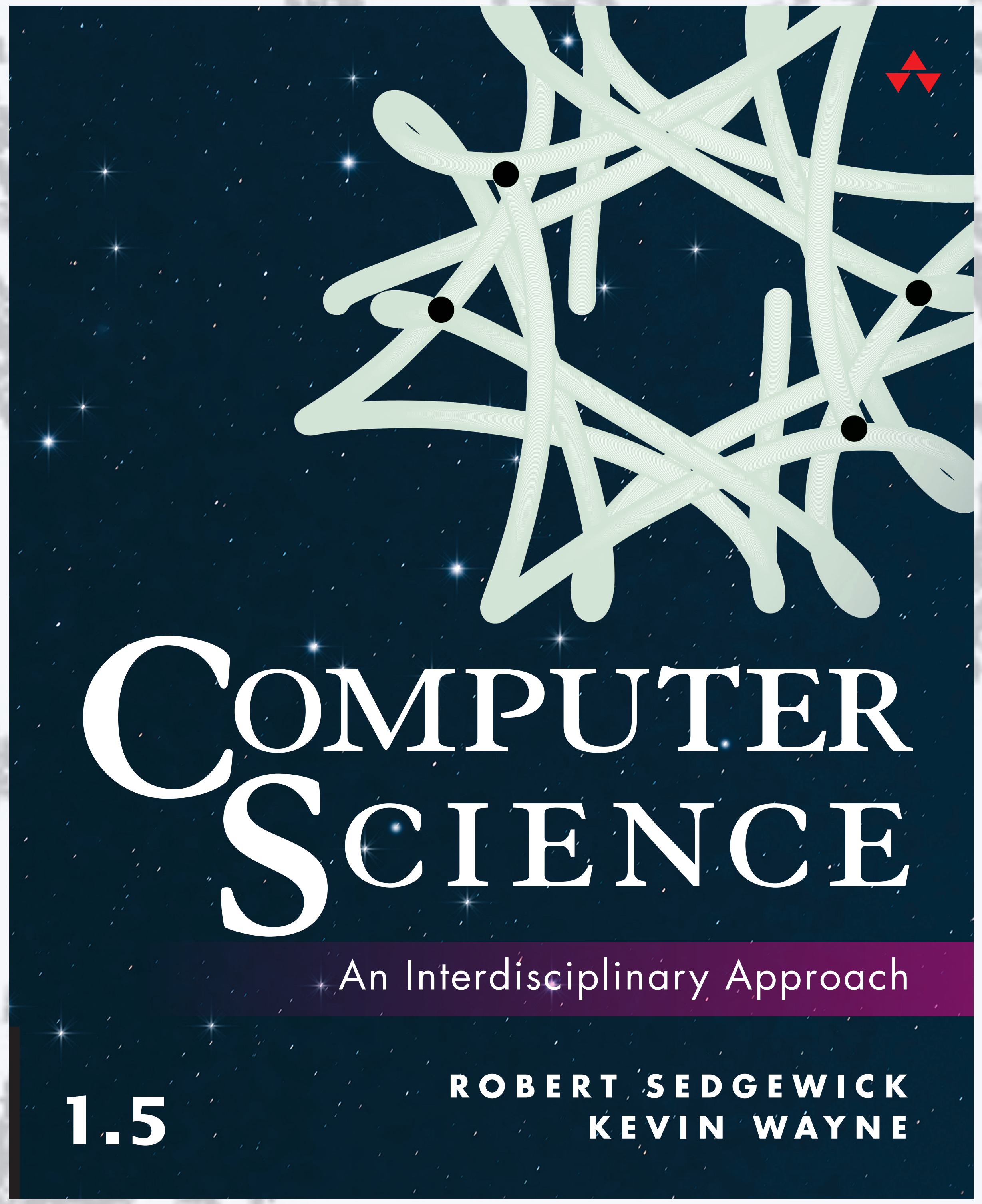
https://lists.clarku.edu/subscribe/csinterest

START RECORDING

Outline

- Attendance quiz
- Standard input and output
- Standard drawing
- Functions and libraries: Basic concepts
- Modular programming and libraries

COMPUTER SCIENCE
SEDGEWICK / WAYNE
PART I: PROGRAMMING IN JAVA



http://introcs.cs.princeton.edu

4. Input and Output

4. Input and Output

- Standard input and output
- Standard drawing
- Fractal drawings
- Animation

Basic building blocks for programming

any program you might want to write

objects

functions and modules

graphics, sound, and image I/O

arrays

conditionals and loops

Math | text I/O

primitive data types | assignment statements

Ability to interact with the outside world

Input and output

Goal: Write Java programs that interact with the outside world via *input* and *output* devices.

Typical INPUT devices


Keyboard


Trackpad


Storage


Network


Camera


Microphone

Typical OUTPUT devices


Display


Storage


Network


Printer


Speakers

Our approach.

- Define input and output *abstractions*.
- Use operating system (OS) functionality to connect our Java programs to actual devices.

Abstraction

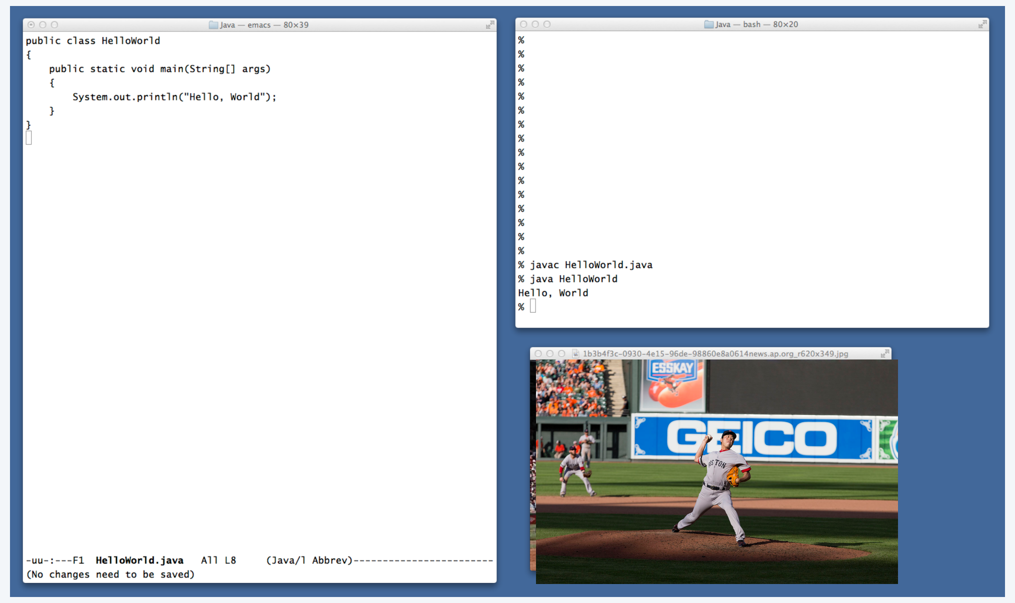
plays an *essential* role in understanding computation.

Interested in thinking more deeply about this concept?
Consider taking a philosophy course.

An *abstraction* is something that exists only as an idea.

Example: "Printing" is the idea of a program producing text as output.

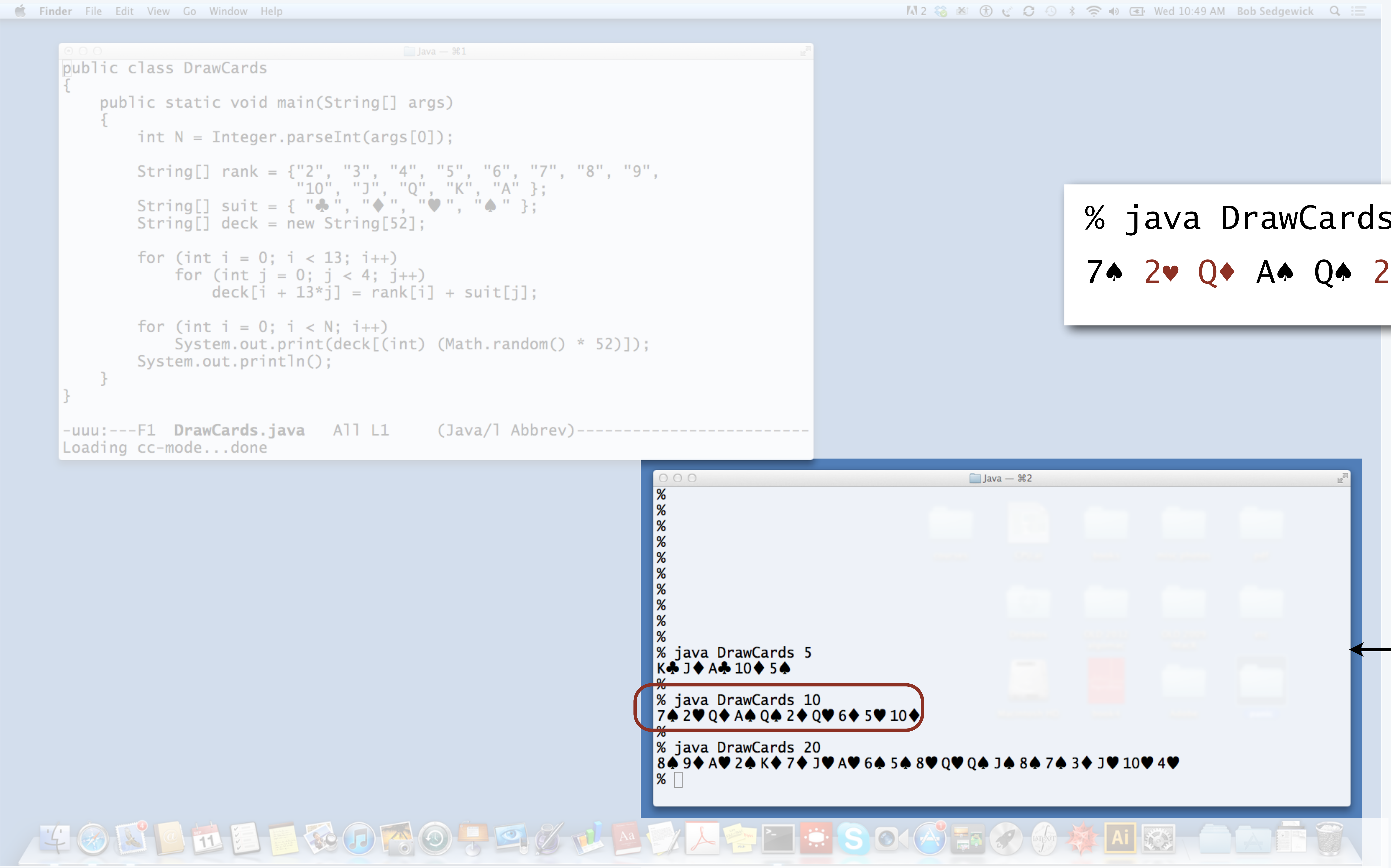
Good abstractions *simplify* our view of the world, by *unifying* diverse real-world artifacts.



This lecture. Abstractions for delivering input to or receiving output from our programs.

Quick review

Terminal. An abstraction for providing input and output to a program.



Input from command line

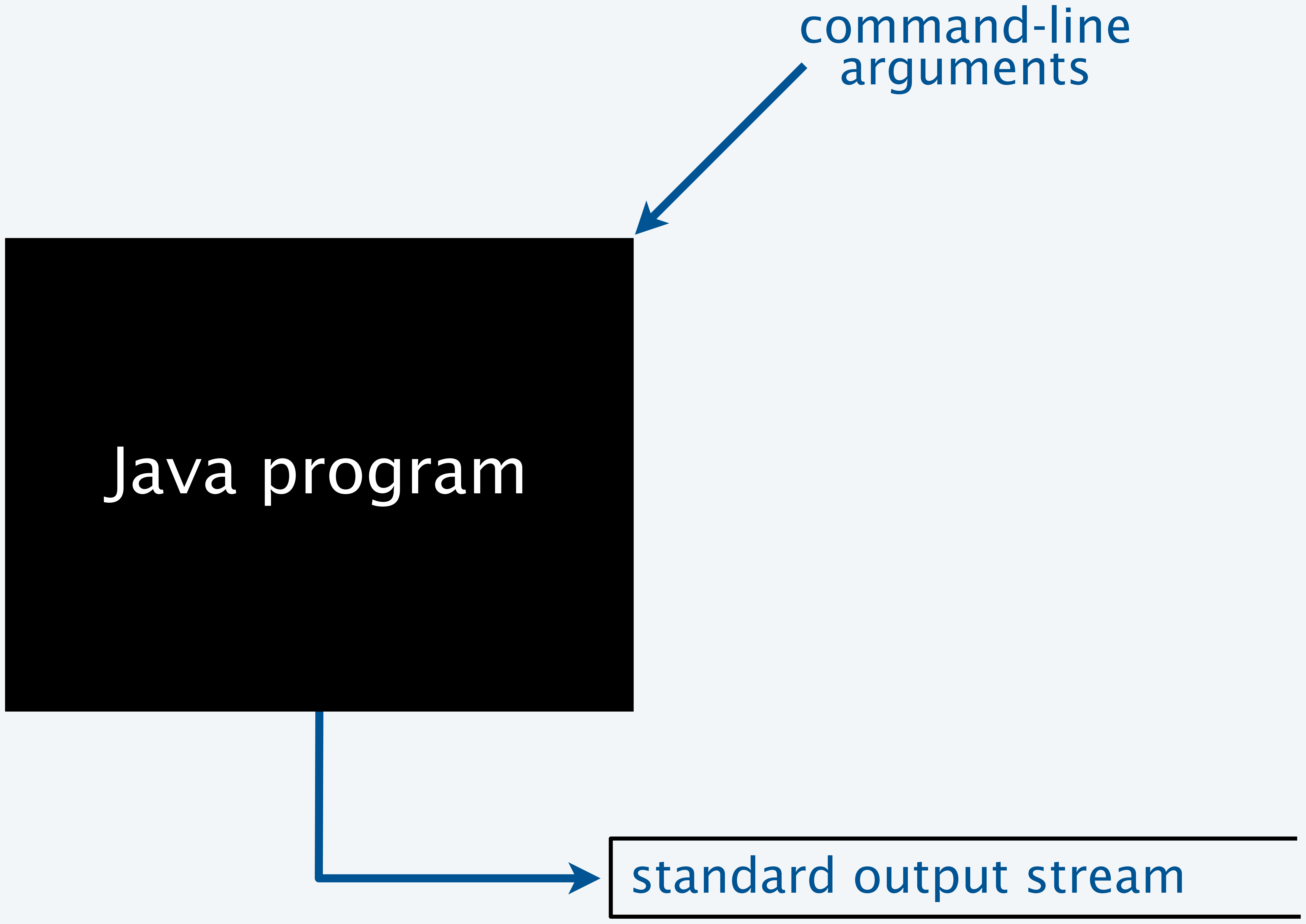
```
% java DrawCards 10
7♠ 2♥ Q♦ A♠ Q♠ 2♦ Q♥ 6♦ 5♥ 10♦
```

Output to standard output stream

Virtual VT-100 terminal

Input-output abstraction (so far)

A mental model of what a Java program does.



Review: command-line input

Command-line input. An abstraction for providing arguments (strings) to a program.

Basic properties

- Strings you type after the program name are available as `args[0]`, `args[1]`, ... at *run* time.
- Arguments are available when the program *begins* execution.
- Need to call system conversion methods to convert the strings to other types of data.

```
public class RandomInt
{
   public static void main(String[] args)
   {
      int N = Integer.parseInt(args[0]);
      double r = Math.random();
      int t = (int) (r * N);
      System.out.println(t);
   }
}
```

```
% java RandomInt 6
3

% java RandomInt 10000
3184
```

Review: standard output

Infinity. An abstraction describing something having no limit.

Standard output stream. An abstraction for an infinite output sequence.

Basic properties

- Strings from `System.out.println()` are added to the end of the standard output stream.
- Standard output stream is sent to terminal application by default.

```
public class RandomSeq
{
   public static void main(String[] args)
   {
      int N = Integer.parseInt(args[0]);
      for (int i = 0; i < N; i++)
         System.out.println(Math.random());
   }
}
```

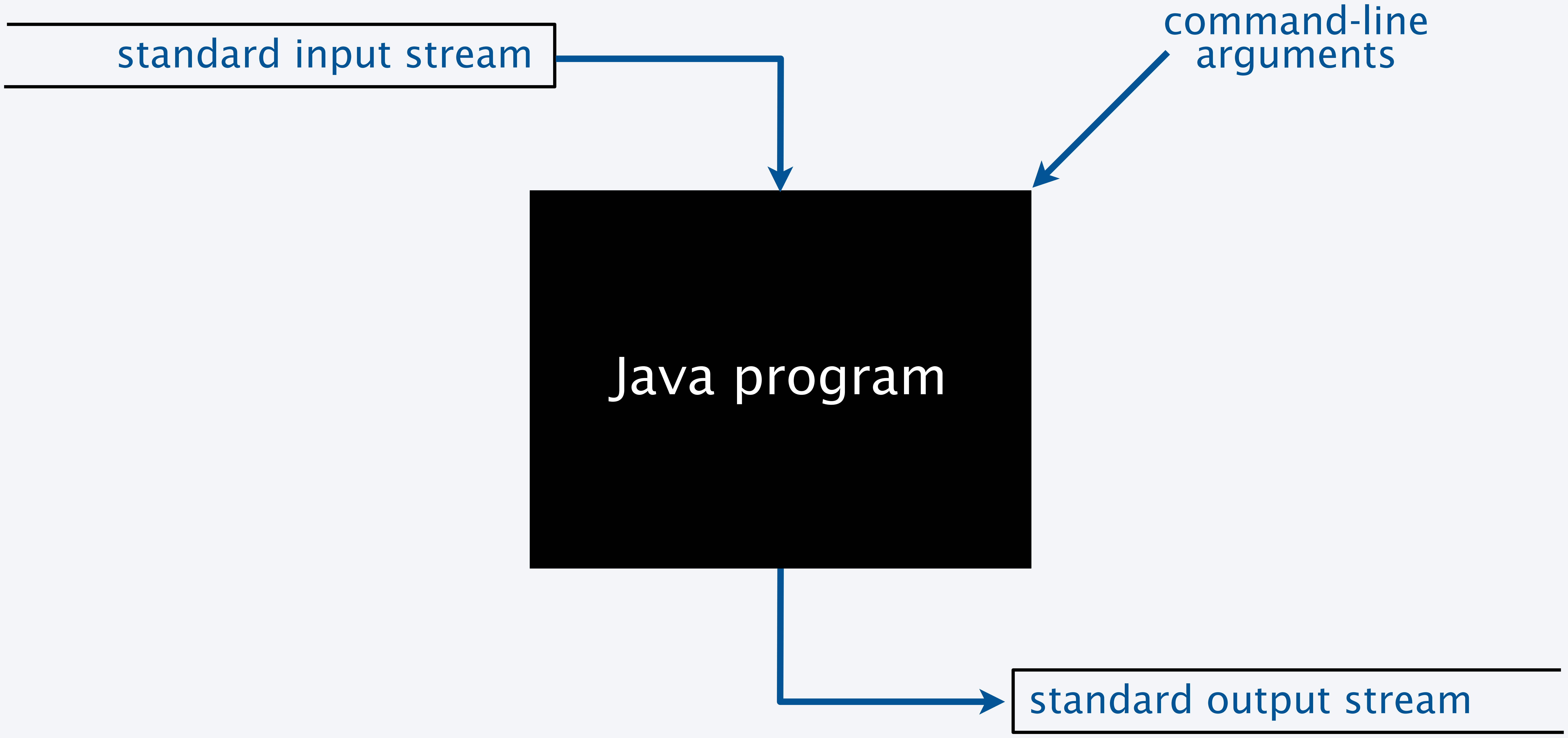
```
% java RandomSeq 4
0.9320744627218469
0.4279508713950715
0.08994615071160994
0.6579792663546435
```

```
% java RandomSeq 1000000
0.09474882292442943
0.2832974030384712
0.1833964252856476
0.2952177517730442
0.8035985765979008
0.7469424300071382
0.5835267075283997
0.3455279612587455
...
```

No limit on amount of output

Improved input-output abstraction

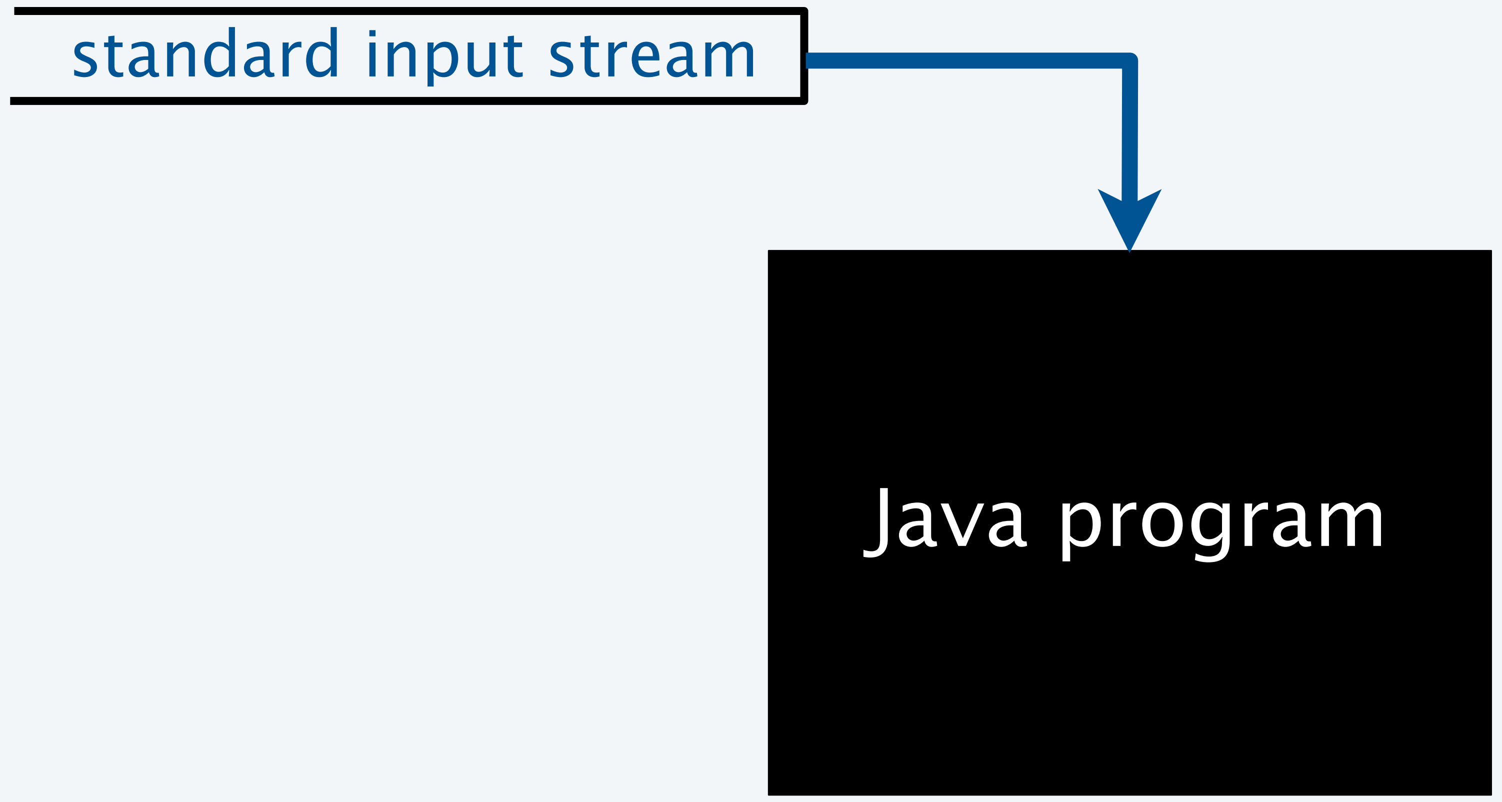
Add an infinite *input* stream.



Standard input

Infinity. An abstraction describing something having no limit.

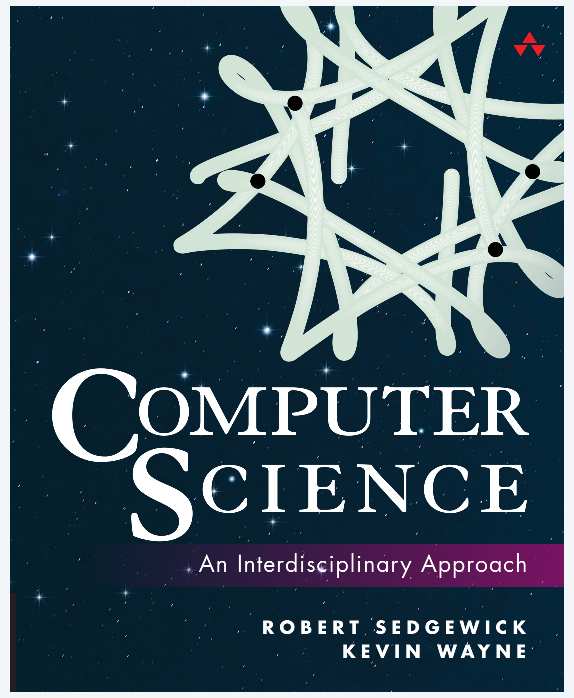
Standard input stream. An abstraction for an infinite *input* sequence.



Advantages over command-line input

- Can provide new data *while* the program is executing.
- No limit on the amount of data we can input to a program.
- Conversion to primitive types is explicitly handled (stay tuned).

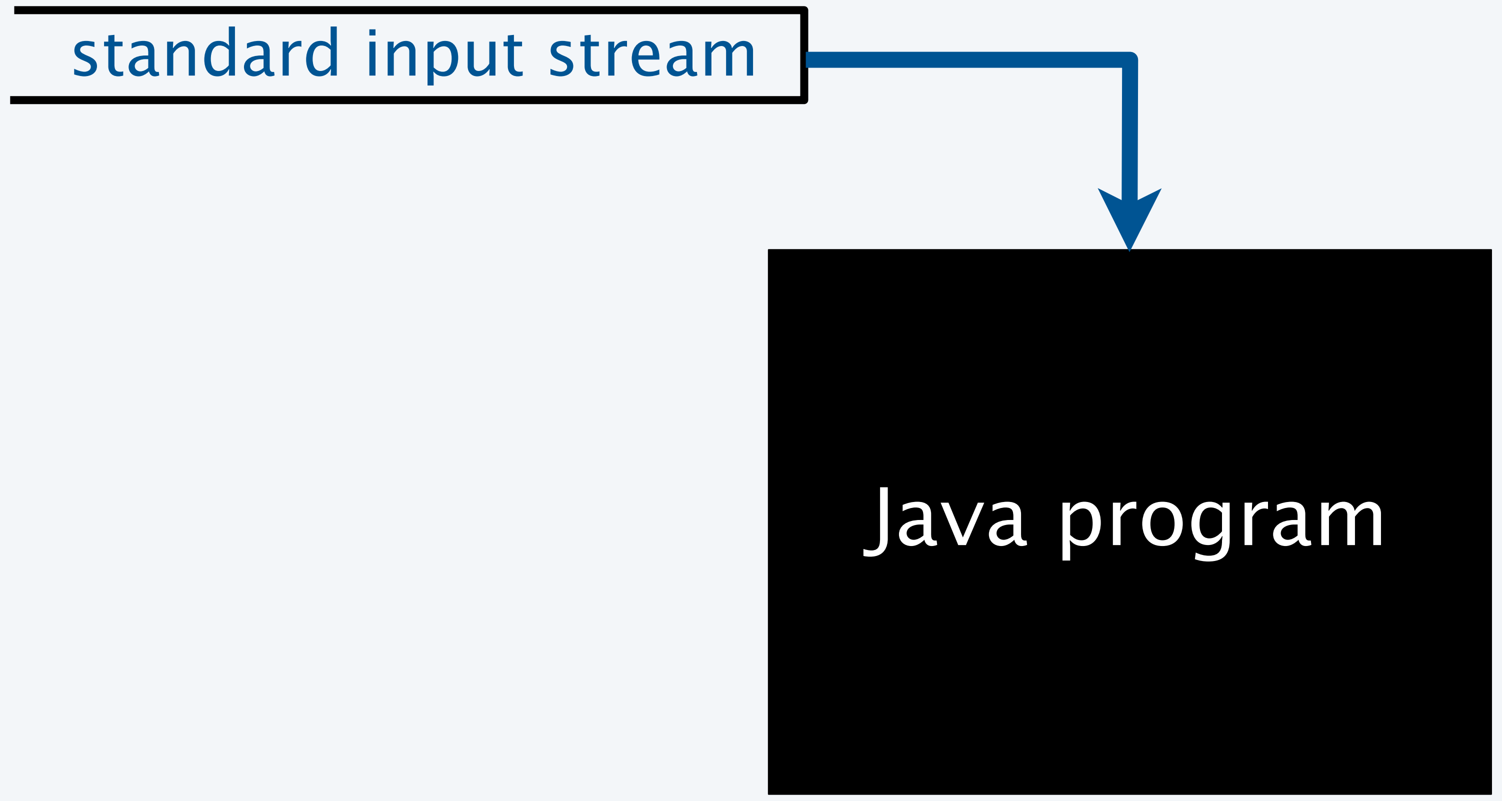
StdIn library



Developed for this course, but broadly useful

- Implement abstractions invented for UNIX in the 1970s.
- Available for download at booksite.
- Included in `introcs` software you downloaded at the beginning of the course.

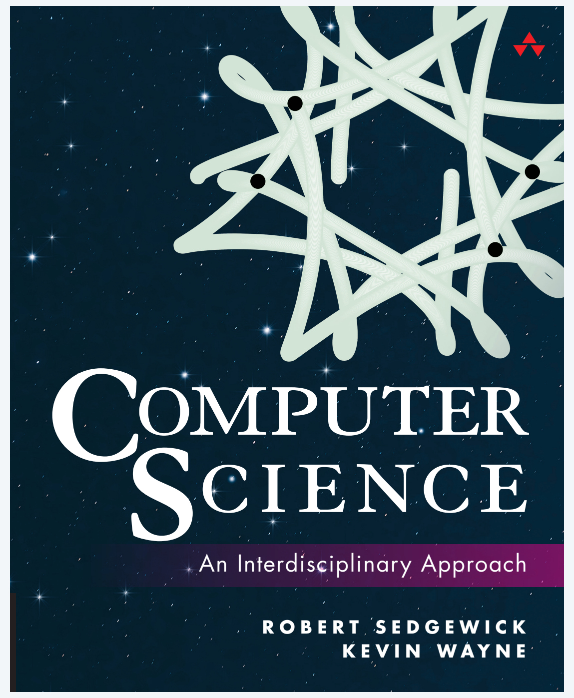
`public class StdIn`	
`boolean isEmpty()`	`true` *iff no more values*
`int readInt()`	*read a value of type* `int`
`double readDouble()`	*read a value of type* `double`
`long readLong()`	*read a value of type* `long`
`boolean readBoolean()`	*read a value of type* `boolean`
`char readChar()`	*read a value of type* `char`
`String readString()`	*read a value of type* `String`
`String readAll()`	*read the rest of the text*



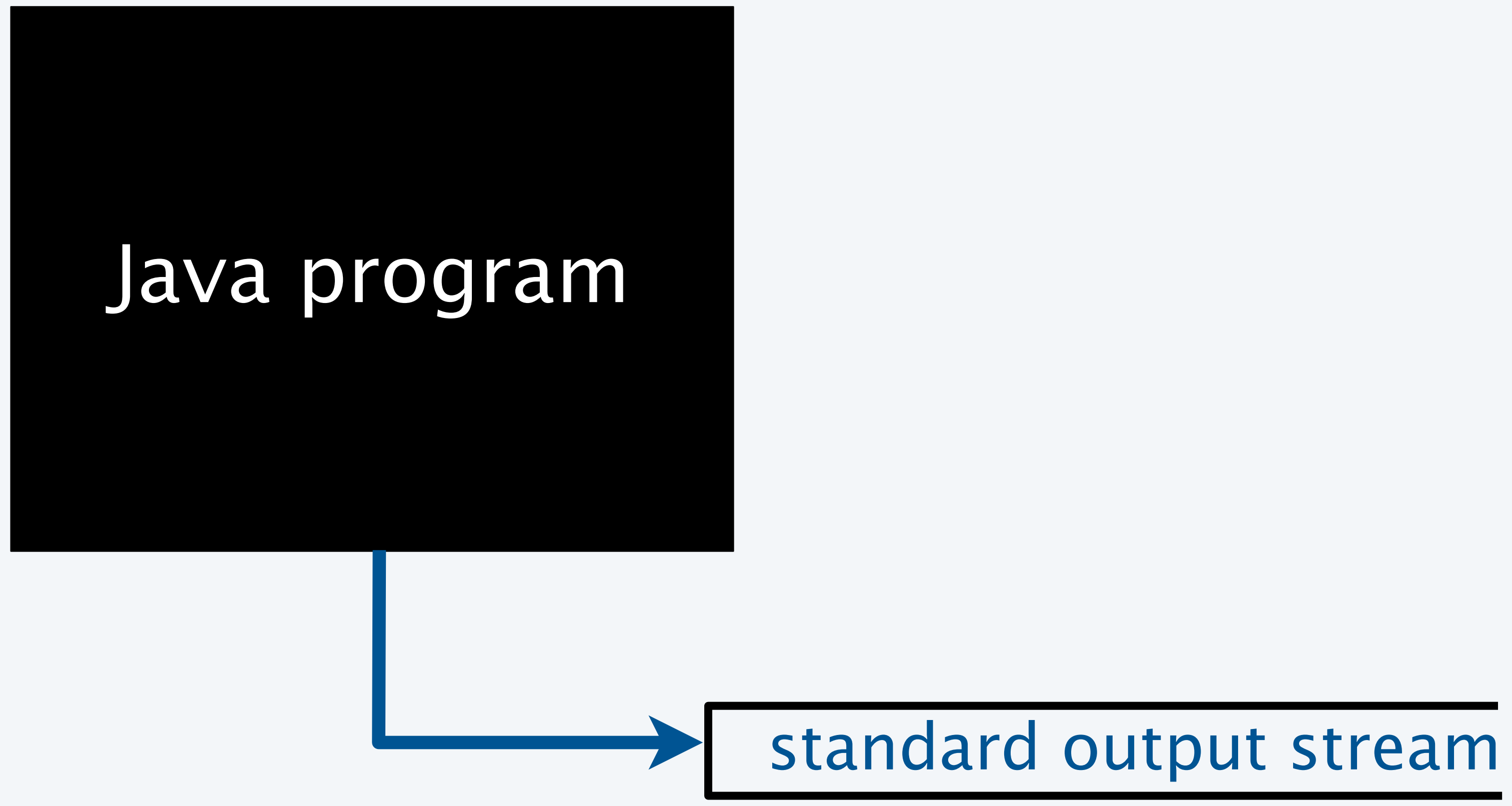
StdOut library

Developed for this course, but broadly useful

- Implement abstractions invented for UNIX in the 1970s.
- Available for download at booksite.
- Included in `introcs` software you downloaded at the beginning of the course.



`public class StdOut`	
`void print(String s)`	*put `s` on the output stream*
`void println()`	*put a newline on the output stream*
`void println(String s)`	*put `s`, then a newline on the stream*
`void printf(String f, ...)`	*formatted output*



Q. These are the same as `System.out`. Why not just use `System.out`?

A. We can make output *independent* of system, language, and locale.

A. Less typing!

StdIn/StdOut warmup

Interactive input

- Prompt user to type inputs on standard input stream.
- Mix input stream with output stream.

```
public class AddTwo
{
   public static void main(String[] args)
   {
      StdOut.print("Type the first integer: ");
      int x = StdIn.readInt();
      StdOut.print("Type the second integer: ");
      int y = StdIn.readInt();
      int sum = x + y;
      StdOut.println("Their sum is " + sum);
   }
}
```

```
% java AddTwo
Type the first integer:  1
Type the second integer: 2
Their sum is 3
```

StdIn application: average the numbers on the standard input stream

Average

- Read a stream of numbers.
- Compute their average.

Q. How do I specify the end of the stream?

A. <Ctrl-d> (standard on macOS, Linux)

A. <Ctrl-z> then <Enter> (Windows)

```
public class Average
{
   public static void main(String[] args)
   {
      double sum = 0.0;  // cumulative total
      int n = 0;         // number of values
      while (!StdIn.isEmpty())
      {
         double x = StdIn.readDouble();
         sum = sum + x;
         n++;
      }
      StdOut.println(sum / n);
   }
}
```

Key points

- No limit on the size of the input stream.
- Input and output can be interleaved.

```
% java Average
10.0 5.0  6.0
 3.0 7.0 32.0
<Ctrl-d>
10.5
```

Summary: prototypical applications of standard output and standard input

StdOut: Generate a stream of random numbers

```
public class RandomSeq
{
   public static void main(String[] args)
   {
      int N = Integer.parseInt(args[0]);
      for (int i = 0; i < N; i++)
         StdOut.println(Math.random());
   }
}
```

StdIn: Compute the average of a stream of numbers

```
public class Average
{
   public static void main(String[] args)
   {
      double sum = 0.0;   // cumulative total
      int n = 0;          // number of values
      while (!StdIn.isEmpty())
      {
         double x = StdIn.readDouble();
         sum = sum + x;
         n++;
      }
      StdOut.println(sum / n);
   }
}
```

Both streams are *infinite* (no limit on their size).

Q. Do I always have to type in my input data and print my output?

A. No! Keep data and results in *files* on your computer, or use *piping* to connect programs.

Redirection: keep data in files on your computer

Redirect standard output to a file

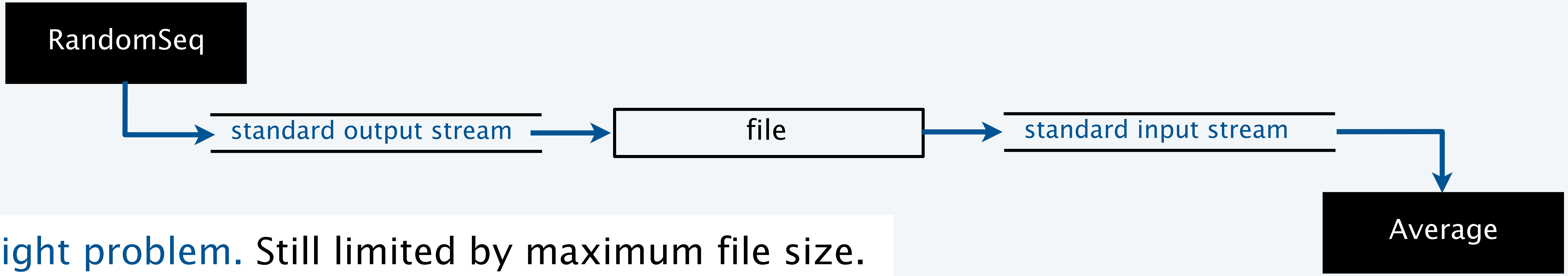
```
% java RandomSeq 1000000 > data.txt

% more data.txt
0.09474882292442943
0.2832974030384712
0.1833964252856476
0.2952177517730442
0.8035985765979008
0.7469424300071382
0.5835267075283997
0.3455279612587455
...
```

"redirect standard output to"

Redirect from a file to standard input

```
% java Average < data.txt
0.4947655567740991
```

"take standard input from"



Slight problem. Still limited by maximum file size.

Piping: entirely avoid saving data

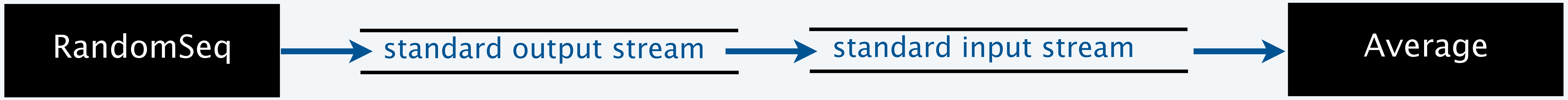
Q. There's no room for a huge file on my computer. Now what?

A. No problem! Use *piping*.

Piping. Connect standard output of one program to standard input of another.

set up a pipe

```
% java RandomSeq 1000000 | java Average
0.4997970473016028

% java RandomSeq 1000000 | java Average
0.5002071875644842
```



Critical point. No limit *within programs* on the amount of data they can handle.

It is the job of the *system* to collect data on standard output and provide it to standard input.

Early computing

- Amount of available memory was much smaller than amount of data to be processed.
- *But* dramatic increases happened every year.
- Redirection and piping enabled programs to handle much more data than computers could store.

Modern computing

- Amount of available memory *is* much smaller than amount of data to be processed.
- Dramatic increases *still* happen every year.
- *Streaming algorithms* enable our programs to handle much more data than our computers can store.

Lesson. Avoid limits *within your program* whenever possible.

Image sources

http://www.digitalreins.com/wp-content/uploads/2013/05/Binary-code.jpg

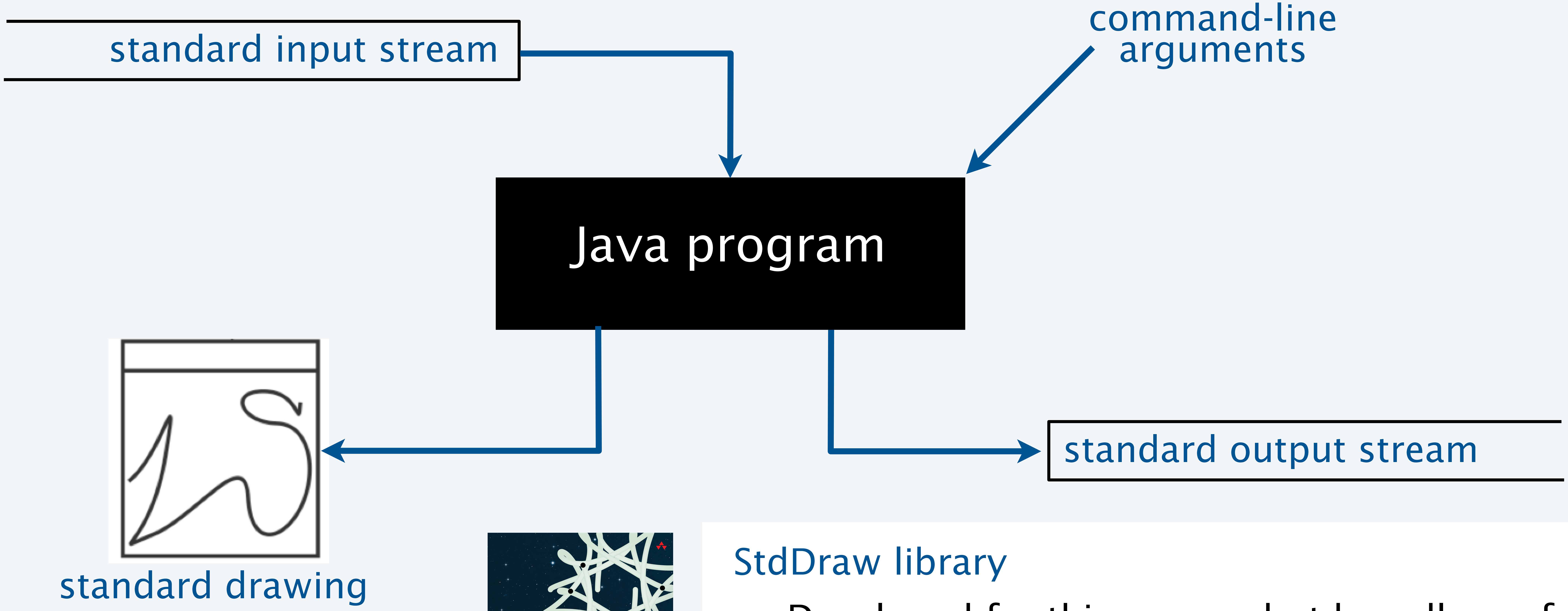
http://en.wikipedia.org/wiki/Punched_tape#mediaviewer/File:Harwell-dekatron-witch-10.jpg

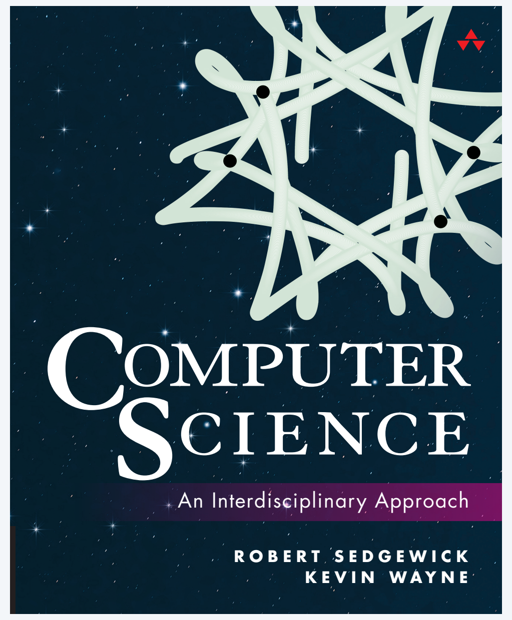
4. Input and Output

- Standard input and output
- **Standard drawing**
- Fractal drawings
- Animation

Further improvements to our I/O abstraction

Add the ability to create a *drawing*.





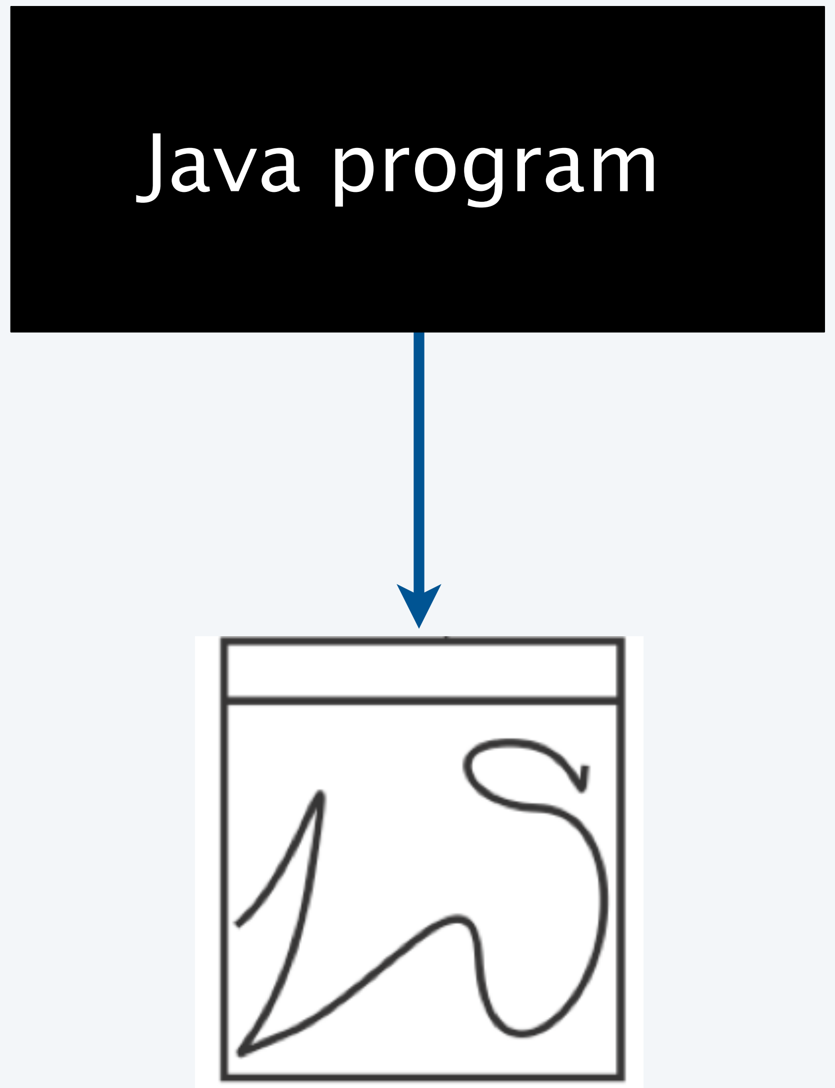
StdDraw library

- Developed for this course, but broadly useful.
- Available for download at booksite.
- Included in `introcs` software.

StdDraw library

`public class StdDraw`	
`void line(double x0, double y0, double x1, double y1)`	
`void point(double x, double y)`	
`void text(double x, double y, String s)`	
`void circle(double x, double y, double r)`	
`void square(double x, double y, double r)`	
`void polygon(double x, double y, double r)`	
`void picture(double x, double y, String filename)`	*place* .gif, .jpg *or* .png *file*
`void setPenRadius(double r)`	
`void setPenColor(Color c)`	
`void setXscale(double x0, double x1)`	*reset x range to* $[x_0, x_1)$
`void setYscale(double y0, double y1)`	*reset y range to* $[y_0, y_1)$
`void show(int dt)`	*show all; pause* dt *millisecs*

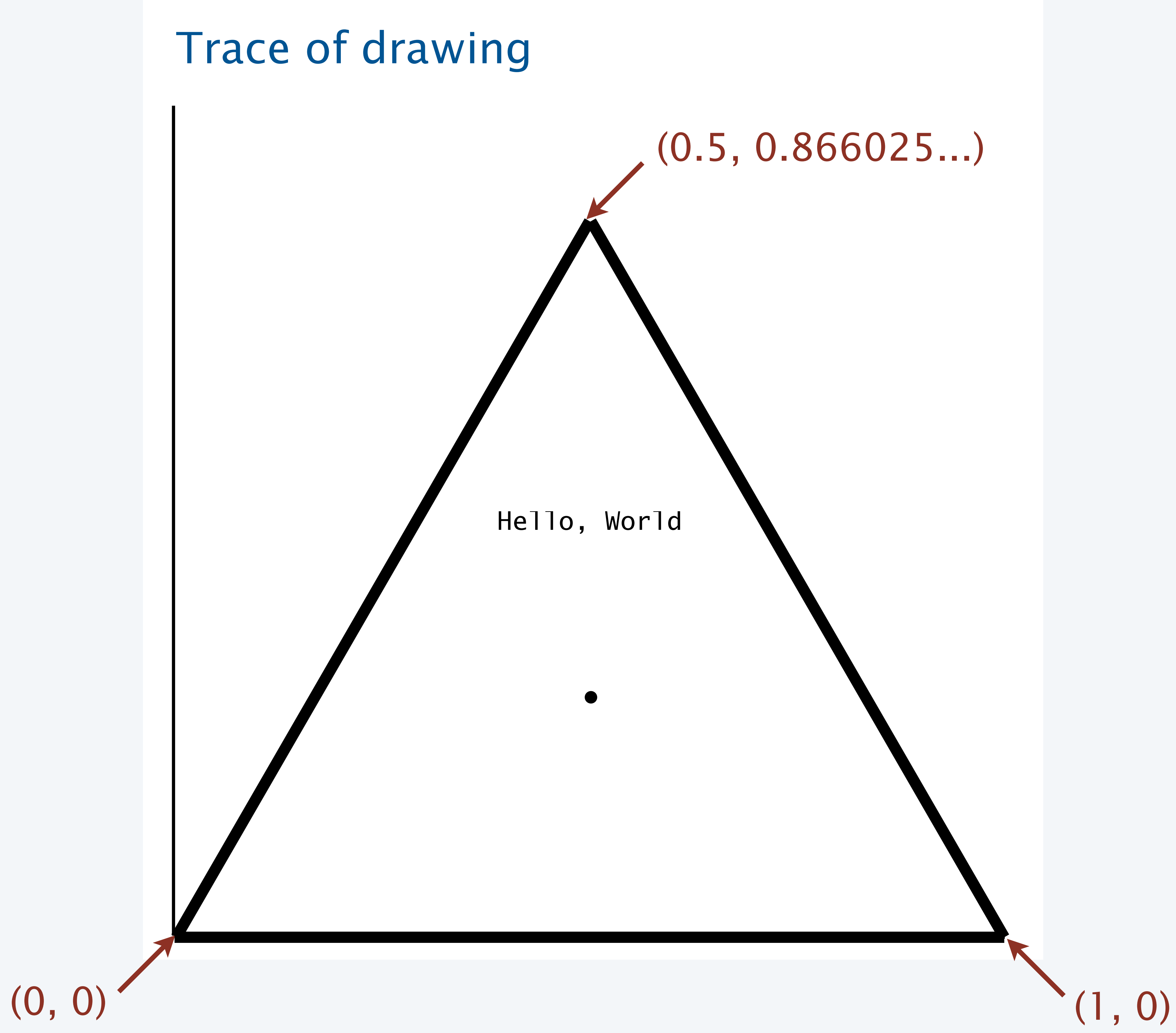
also filledCircle(), filledSquare(), and filledPolygon()



standard drawing

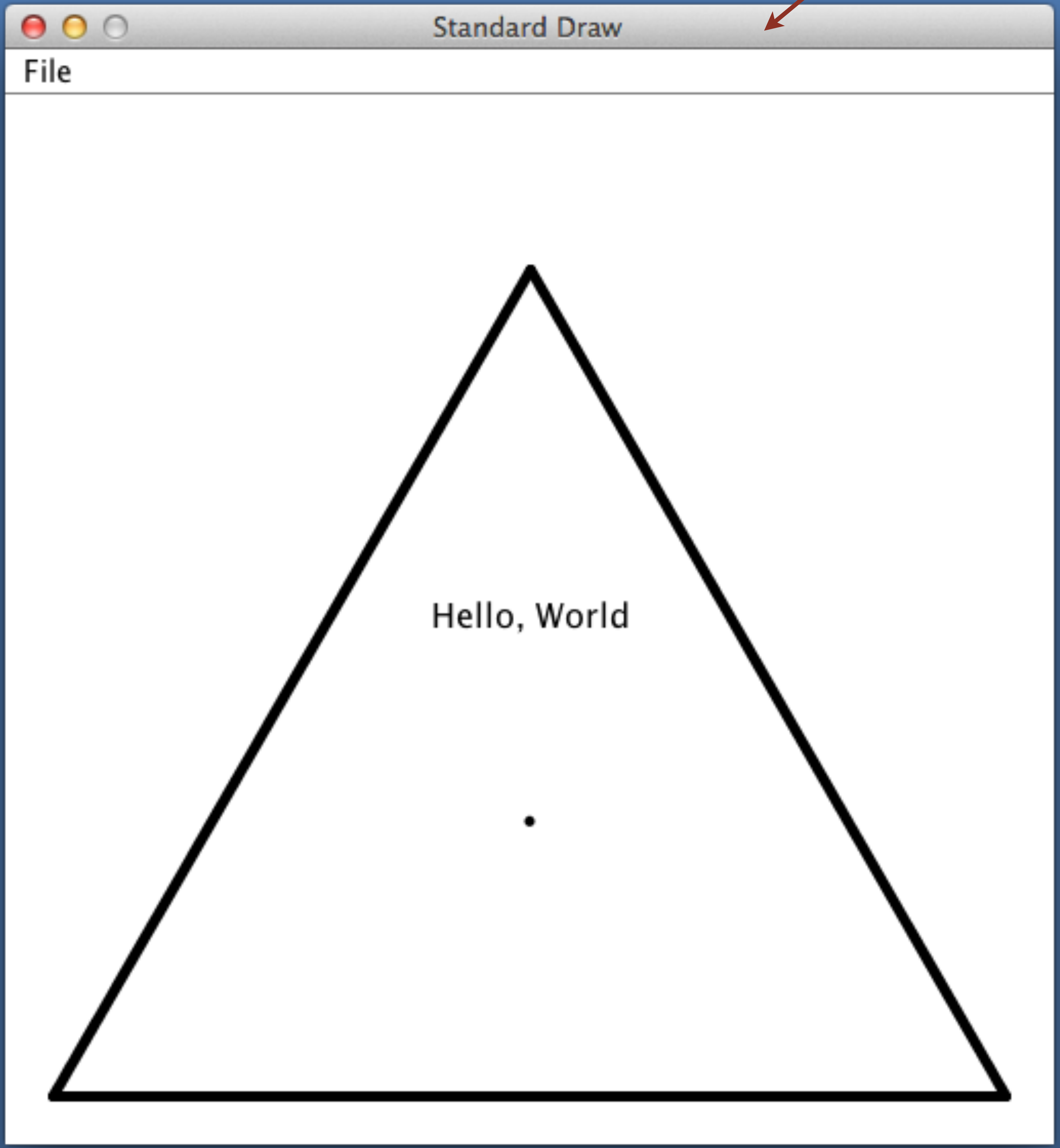
“Hello, World” for StdDraw

```
public class Triangle
{
   public static void main(String[] args)
   {
      double c = Math.sqrt(3.0) / 2.0;
      StdDraw.setPenRadius(0.01);
      StdDraw.line(0.0, 0.0, 1.0, 0.0);
      StdDraw.line(1.0, 0.0, 0.5, c);
      StdDraw.line(0.5, c, 0.0, 0.0);
      StdDraw.point(0.5, c/3.0);
      StdDraw.text(0.5, 0.5, "Hello, World");
   }
}
```



“Hello, World” for StdDraw

window for standard drawing



virtual terminal for editor

Java — ⌘2

```java
public class Triangle
{
   public static void main(String[] args)
   {
     double c = Math.sqrt(3.0) / 2.0;
     StdDraw.setPenRadius(0.01);
     StdDraw.line(0.0, 0.0, 1.0, 0.0);
     StdDraw.line(1.0, 0.0, 0.5, c);
     StdDraw.line(0.5, c, 0.0, 0.0);
     StdDraw.point(0.5, c/3.0);
     StdDraw.text(0.5, 0.5, "Hello, World");
   }
}
```

Java — ⌘3

```
%
% javac Triangle.java
% java Triangle
```

virtual terminal for OS commands

StdDraw application: data visualization

```
public class PlotFilter
{
   public static void main(String[] args)
   {
      double xmin = StdIn.readDouble();
      double ymin = StdIn.readDouble();
      double xmax = StdIn.readDouble();
      double ymax = StdIn.readDouble();
      StdDraw.setXscale(xmin, xmax);
      StdDraw.setYscale(ymin, ymax);
      while (!StdIn.isEmpty())
      {
         double x = StdIn.readDouble();
         double y = StdIn.readDouble();
         StdDraw.point(x, y);
      }
   }
}
```

read coords of bounding box → (the four `double ... = StdIn.readDouble();` lines)

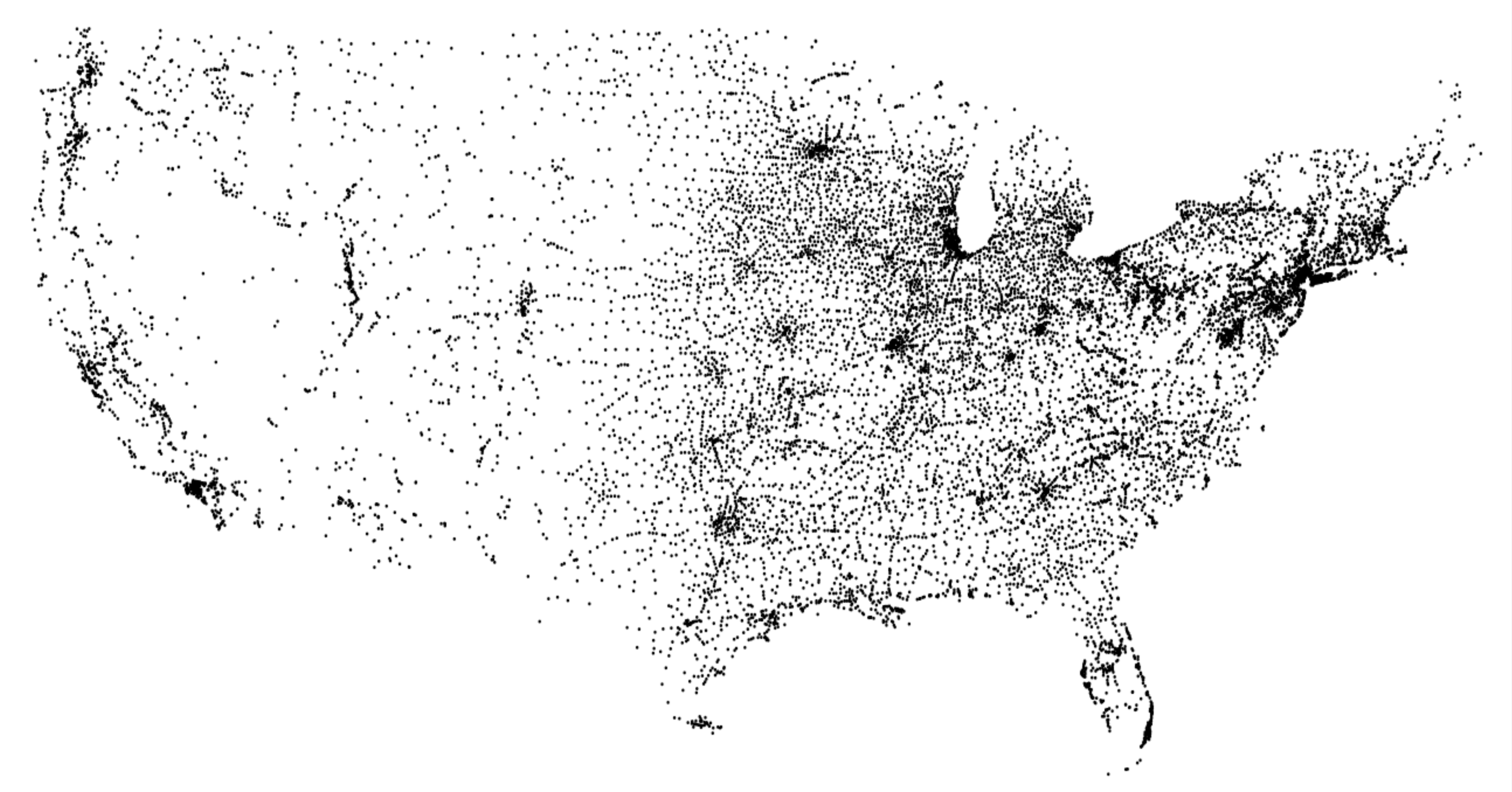
rescale → (the two `StdDraw.set...scale` lines)

read and plot a point → (the loop body)

bounding box coords

```
% more < USA.txt
669905.0 247205.0 1244962.0 490000.0
 1097038.8890  245552.7780
 1103961.1110  247133.3330
 1104677.7780  247205.5560
 ...

% java PlotFilter < USA.txt
```

sequence of point coordinates (13,509 cities)



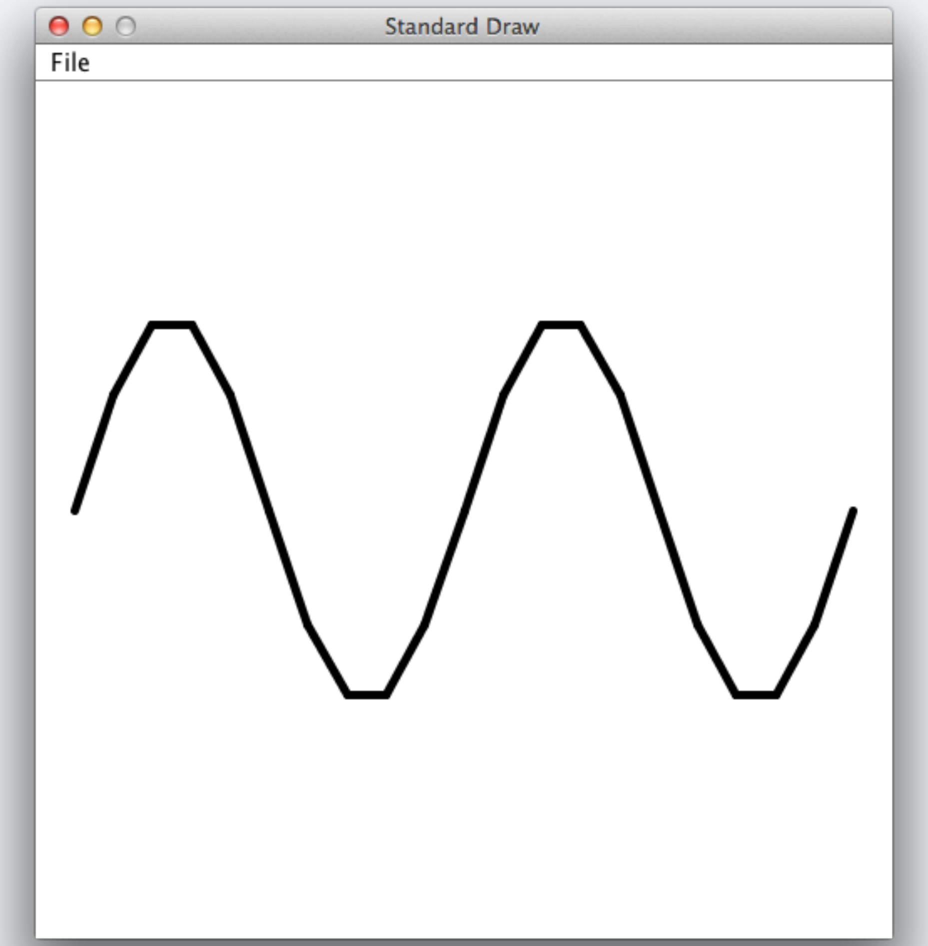
StdDraw application: plotting a function

Goal. Plot $y = \sin(4x) + \sin(20x)$ in the interval $(0, \pi)$.

Method. Take N samples, regularly spaced.

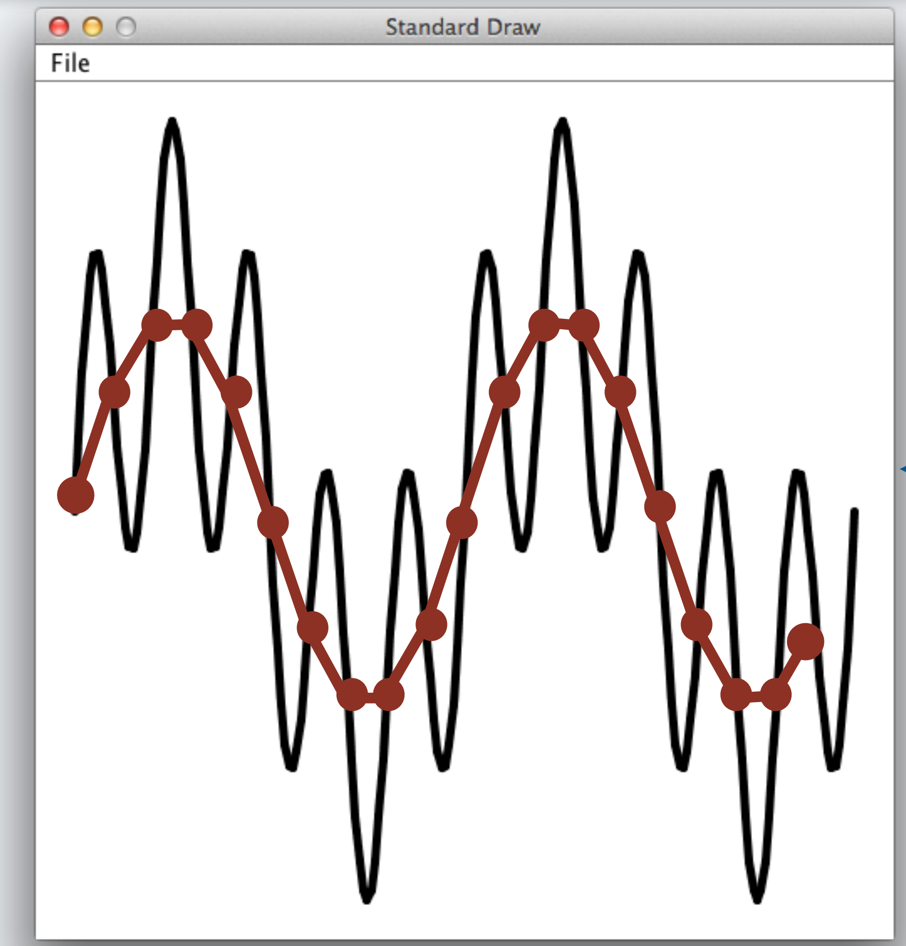
```
public class PlotFunctionEx
{
   public static void main(String[] args)
   {
      int N = Integer.parseInt(args[0]);
      double[] x = new double[N+1];
      double[] y = new double[N+1];
      for (int i = 0; i <= N; i++)
      {
         x[i] = Math.PI * i / N;
         y[i] = Math.sin(4*x[i]) + Math.sin(20*x[i]);
      }
      StdDraw.setXscale(0, Math.PI);
      StdDraw.setYscale(-2.0, +2.0);
      for (int i = 0; i < N; i++)
         StdDraw.line(x[i], y[i], x[i+1], y[i+1]);
   }
}
```

```
% java PlotFunctionEx 20
```



Lesson 1: Plotting is easy. →

```
% java PlotFunctionEx 200
```



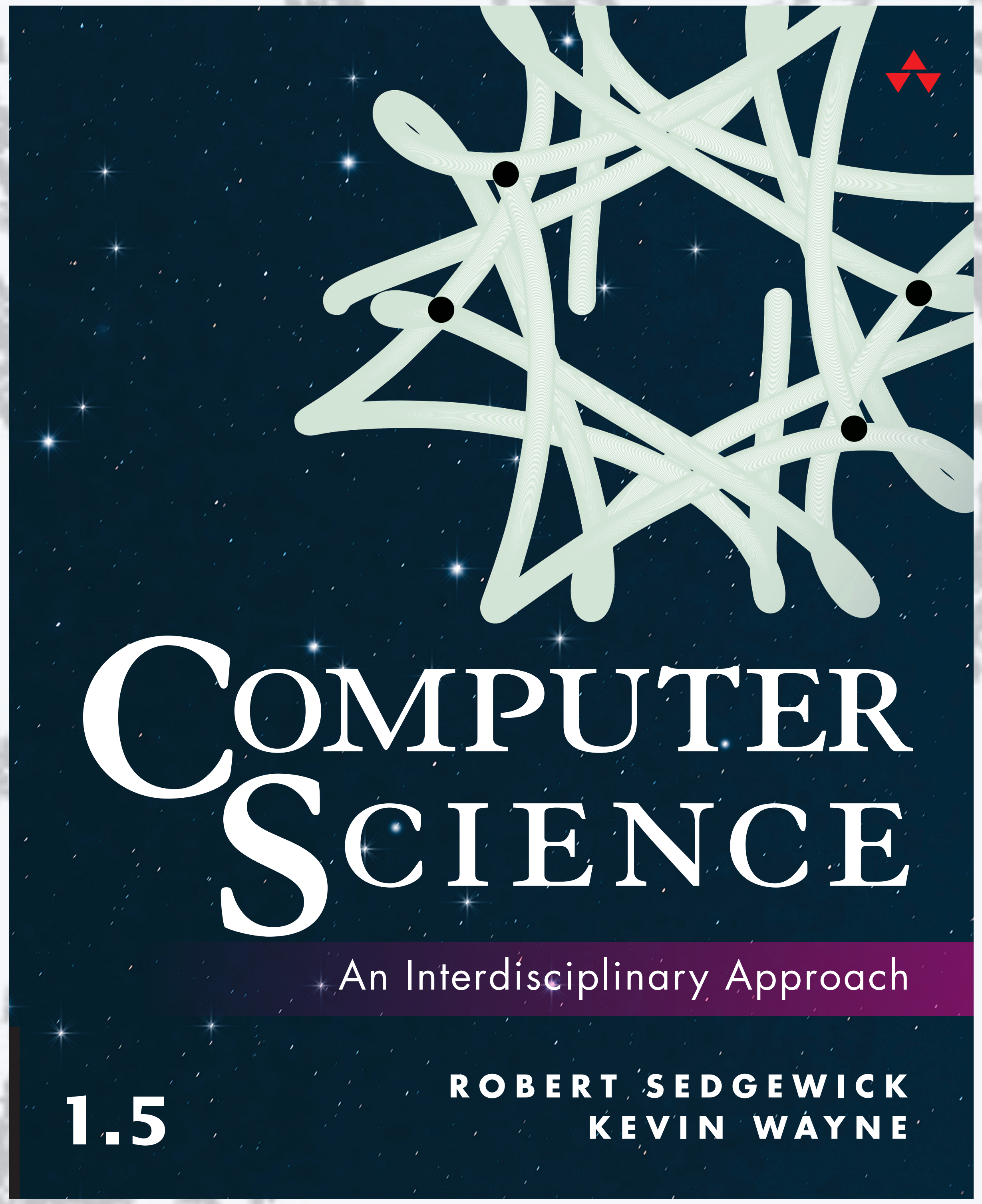
← Lesson 2: Take a sufficiently large sample—otherwise you might miss something!

COMPUTER SCIENCE

SEDGEWICK/WAYNE

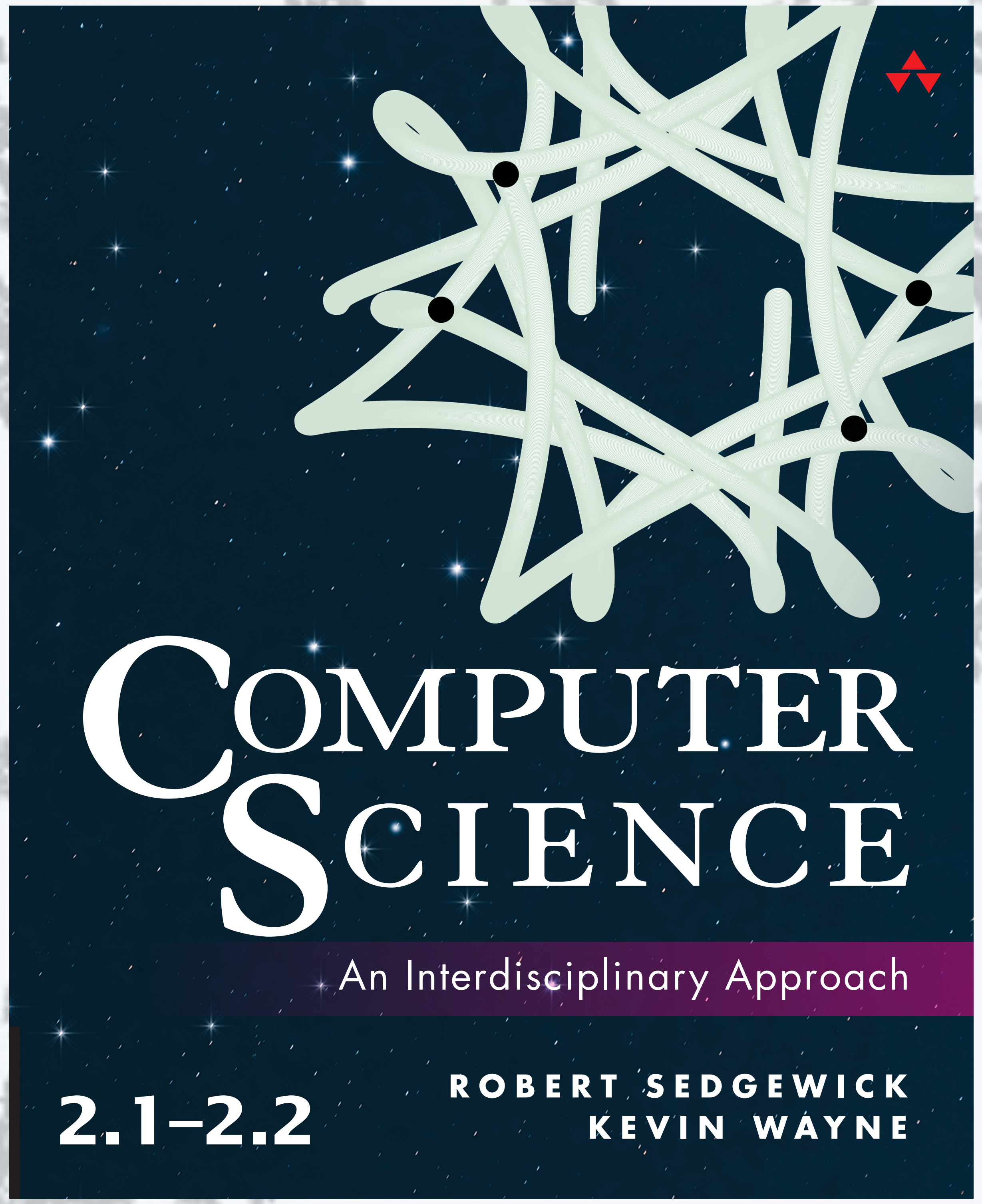
PART I: PROGRAMMING IN JAVA

CS.4.B.IO.Drawing



http://introcs.cs.princeton.edu

4. Input and Output



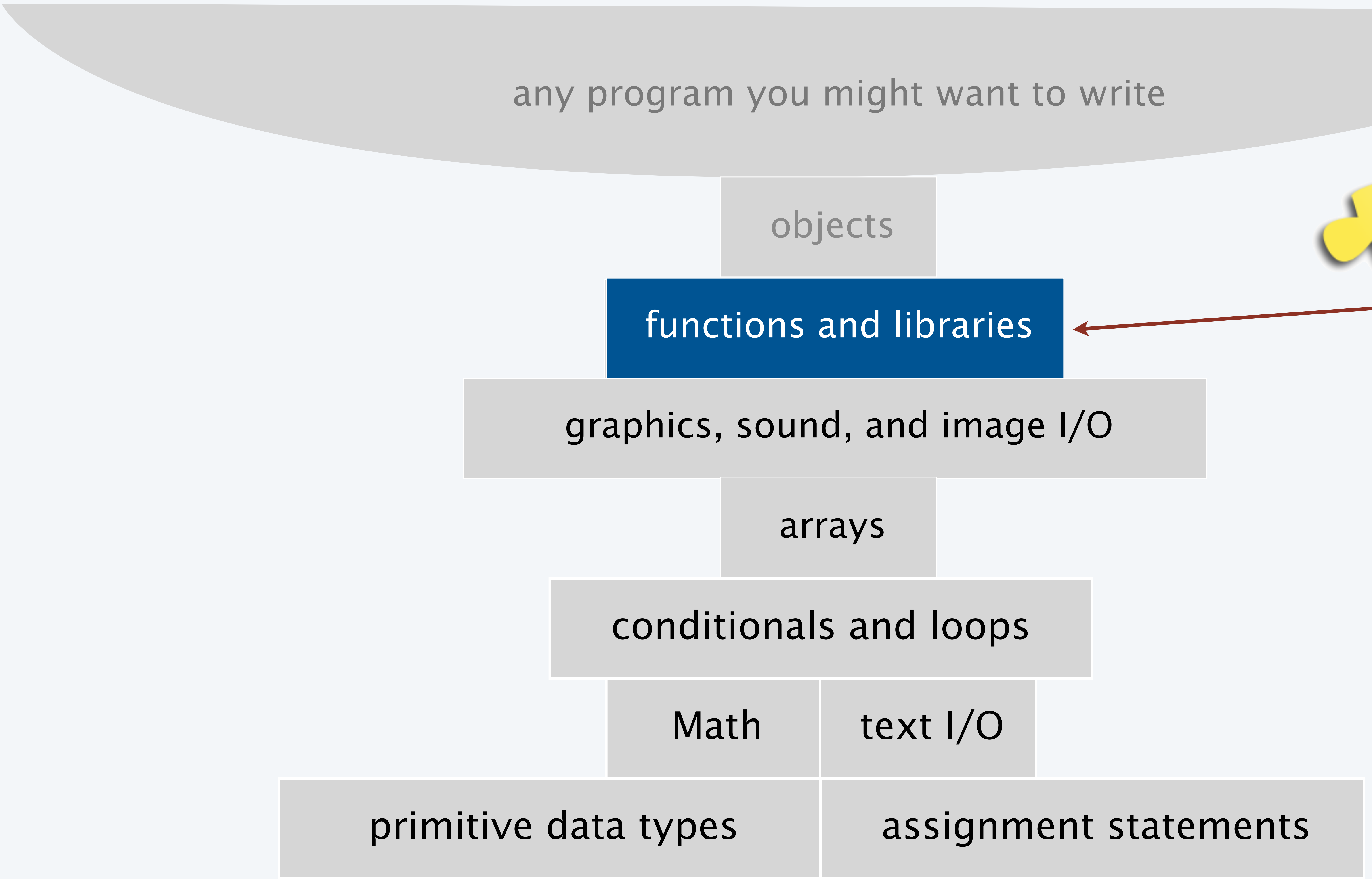
http://introcs.cs.princeton.edu

5. Functions and Libraries

5. Functions and Libraries

- **Basic concepts**
- Case study: Digital audio
- Application: Gaussian distribution
- Modular programming and libraries

Context: basic building blocks for programming



This lecture:
Reuse code to build big programs from small pieces

Functions, libraries, and modules

Modular programming

- Organize programs as independent modules that do a job together.
- Why? Easier to share and reuse code to build bigger programs.

Facts of life

- Support of modular programming has been a holy grail for decades.
- Ideas can conflict and get highly technical in the real world.

Def. A library is a set of functions.

↑ for purposes of this lecture

Def. A module is a `.java` file.

↑ for purposes of this course

For now. Libraries and modules are the *same thing*: `.java` files containing sets of functions.

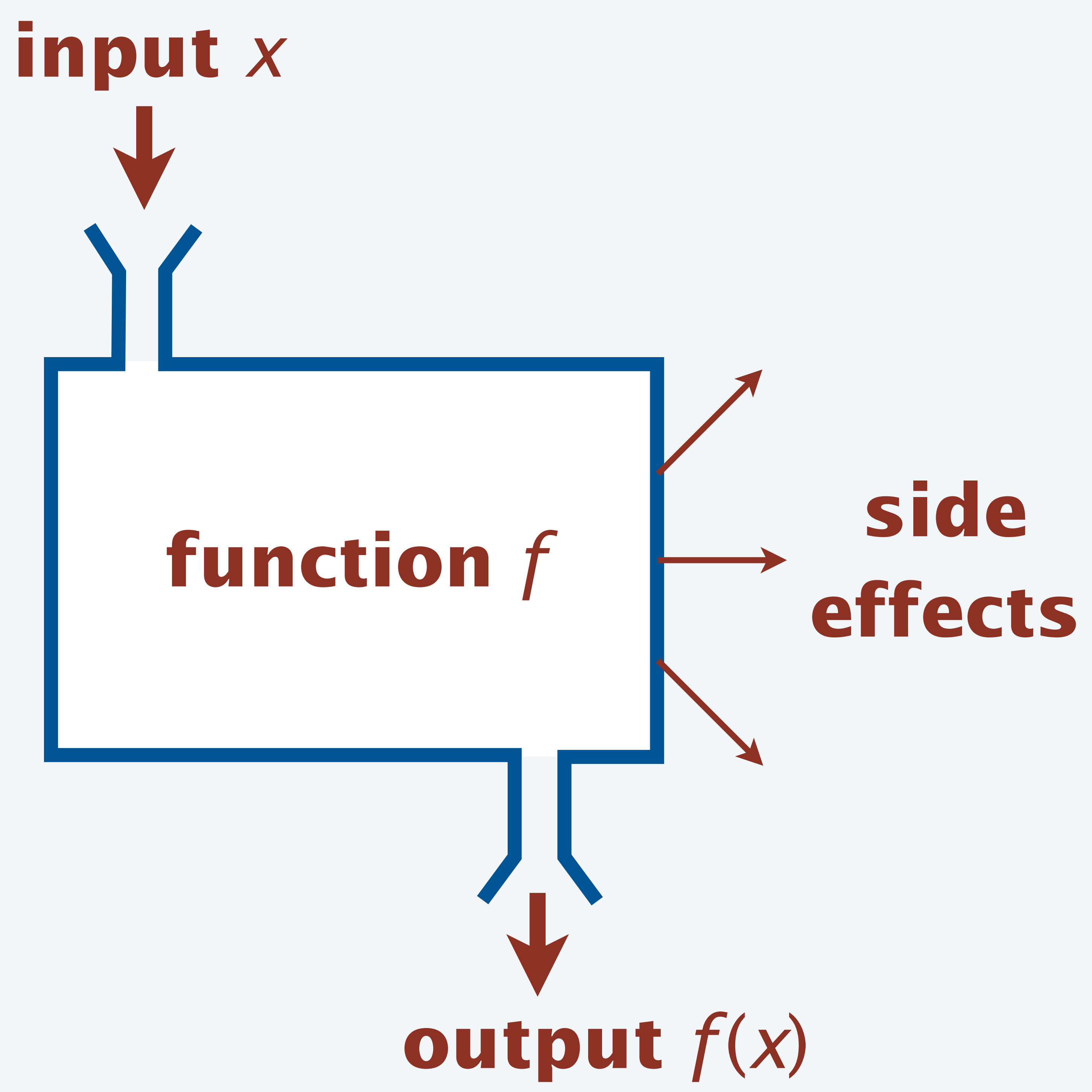
Later. Modules implement *data structures* (stay tuned).

Functions (static methods)

Java *function* ("aka *static method*")

- Takes zero or more *input* arguments.
- Returns zero or one *output* value.
- May cause *side effects* (e.g., output to standard draw).

Java functions are *more general* than mathematical functions



Applications

- Scientists use mathematical functions to calculate formulas.
- Programmers use functions to build modular programs.
- *You* use functions for both.

Examples seen so far

- Built-in functions: `Math.random()`, `Math.abs()`, `Integer.parseInt()`.
- Our I/O libraries: `StdIn.readInt()`, `StdDraw.line()`, `StdAudio.play()`.
- User-defined functions: `main()`.

Anatomy of a Java static method

To implement a function (static method)

- Create a *name*.
- Declare type and name of *argument*(*s*).
- Specify type for *return value*.
- Implement *body* of method.
- Finish with *return* statement.

return type

method name

argument declarations

the method's *signature*

```
public static double sqrt(double c, double eps)
{
    if (c < 0) return Double.NaN;
    double t = c;
    while (Math.abs(t - c/t) > eps * t)
       t = (c/t + t) / 2.0;
    return t;
}
```

body of sqrt()

return statement

Anatomy of a Java library

A **library** is a set of functions.

Note: We are using our `sqrt()` *from earlier to illustrate the basics with a familiar function.*

Our focus is on control flow here. See earlier slides for technical details.

You can use `Math.sqrt()`.

module named `Newton.java`

```
public class Newton          ← library/module name
{
   public static double sqrt(double c, double eps)      ← sqrt() method
   {
      if (c < 0) return Double.NaN;
      double t = c;
      while (Math.abs(t - c/t) > eps * t)
         t = (c/t + t) / 2.0;
      return t;
   }

   public static void main(String[] args)               ← main() method
   {
      double[] a = new double[args.length];
      for (int i = 0; i < args.length; i++)
         a[i] = Double.parseDouble(args[i]);
      for (int i = 0; i < a.length; i++)
         StdOut.println(sqrt(a[i], 1e-3));
   }
}
```

Key point. Functions provide a *new way* to control the flow of execution.

Scope

Def. The scope of a variable is the code that can refer to it by name.

```
public class Newton
{
   public static double sqrt(double c, double eps)
   {
      if (c < 0) return Double.NaN;
      double t = c;
      while (Math.abs(t - c/t) > eps * t)
         t = (c/t + t) / 2.0;
      return t;
   }

   public static void main(String[] args)
   {
      double[] a = new double[args.length];
      for (int i = 0; i < args.length; i++)
         a[i] = Double.parseDouble(args[i]);
      for (int i = 0; i < a.length; i++)
         StdOut.println(sqrt(a[i], 1e-3));
   }
}
```

scope of c and eps

scope of t

cannot refer to a or i in this code

scope of a

cannot refer to c, eps, or t in this code

two *different* variables named i each with scope limited to a single for loop

In a Java library, a variable's scope is the code following its declaration, in the same block.

Best practice. Declare variables so as to *limit* their scope.

Flow of control

```
public class Newton
{
   public static double sqrt(double c, double eps)
   {
      if (c < 0) return Double.NaN;
      double t = c;
      while (Math.abs(t - c/t) > eps * t)
         t = (c/t + t) / 2.0;
      return t;
   }
   public static void main(String[] args)
   {
      double[] a = new double[args.length];
      for (int i = 0; i < args.length; i++)
         a[i] = Double.parseDouble(args[i]);
      for (int i = 0; i < a.length; i++)
      {
         double x = sqrt(a[i], 1e-3);
         StdOut.println(x);
      }
   }
}
```

Summary of flow control for a function call

- Control transfers to the function code.
- Argument variables are declared and initialized with the given values.
- Function code is executed.
- Control transfers back to the calling code (with return value assigned in place of the function name in the calling code).

"pass by value"
(other methods used in other systems)

Note. OS calls main() on java command

Function call flow of control trace

```
public class Newton
{
   public static double sqrt(double c, double eps)
   {
      if (c < 0) return Double.NaN;
      double t = c;
      while (Math.abs(t - c/t) > eps * t)
         t = (c/t + t) / 2.0;
      return t;
   }
   public static void main(String[] args)
   {
      double[] a = new double[args.length];
      for (int i = 0; i < args.length; i++)
         a[i] = Double.parseDouble(args[i]);
      for (int i = 0; i < a.length; i++)
      {
         double x = sqrt(a[i], 1e-3);
         StdOut.println(x);
      }
   }
}
```

c	t
3.0	3.0
	2.0
	1.75
	1.732

i	a[i]	x
0	1.0	1.000
1	2.0	1.414
2	3.0	1.732
3		

```
% java Newton 1 2 3
1.000
1.414
1.732
```

Pop quiz 1a on functions

Q. What happens when you compile and run the following code?

```
public class PQfunctions1a
{
   public static int cube(int i)
   {
      int j = i * i * i;
      return j;
   }
   public static void main(String[] args)
   {
      int N = Integer.parseInt(args[0]);
      for (int i = 1; i <= N; i++)
         StdOut.println(i + " " + cube(i));
   }
}
```

Pop quiz 1a on functions

Q. What happens when you compile and run the following code?

```
public class PQfunctions1a
{
   public static int cube(int i)
   {
      int j = i * i * i;
      return j;
   }
   public static void main(String[] args)
   {
      int N = Integer.parseInt(args[0]);
      for (int i = 1; i <= N; i++)
         StdOut.println(i + " " + cube(i));
   }
}
```

A. Takes *N* from the command line, then prints cubes of integers from 1 to *N*

```
% javac PQfunctions1a.java
% java PQfunctions1a 6
1 1
2 8
3 27
4 64
5 125
6 216
```

Pop quiz 1b on functions

Q. What happens when you compile and run the following code?

```
public class PQfunctions1b
{
   public static int cube(int i)
   {
      int i = i * i * i;
      return i;
   }
   public static void main(String[] args)
   {
      int N = Integer.parseInt(args[0]);
      for (int i = 1; i <= N; i++)
         StdOut.println(i + " " + cube(i));
   }
}
```

Pop quiz 1b on functions

Q. What happens when you compile and run the following code?

```
public class PQfunctions1b
{
   public static int cube( int i )
   {
      int i = i * i * i;
      return i;
   }
   public static void main(String[] args)
   {
      int N = Integer.parseInt(args[0]);
      for (int i = 1; i <= N; i++)
         StdOut.println(i + " " + cube(i));
   }
}
```

A. Won't compile. Argument variable i is declared and initialized for function block, so the name cannot be reused.

```
% javac PQfunctions1b.java
PQfunctions1b.java:5: i is already defined in cube(int)
      int i = i * i * i;
          ^
1 error
```

Pop quiz 1c on functions

Q. What happens when you compile and run the following code?

```
public class PQfunctions1c
{
   public static int cube(int i)
   {
      i = i * i * i;
   }
   public static void main(String[] args)
   {
      int N = Integer.parseInt(args[0]);
      for (int i = 1; i <= N; i++)
         StdOut.println(i + " " + cube(i));
   }
}
```

Pop quiz 1c on functions

Q. What happens when you compile and run the following code?

```
public class PQ6_1c
{
   public static int cube(int i)
   {
      i = i * i * i;
   }
   public static void main(String[] args)
   {
      int N = Integer.parseInt(args[0]);
      for (int i = 1; i <= N; i++)
         StdOut.println(i + " " + cube(i));
   }
}
```

A. Won't compile. Need return statement.

```
% javac PQfunctions1c.java
PQfunctions1c.java:6: missing return statement
   }
   ^
1 error
```

Pop quiz 1d on functions

Q. What happens when you compile and run the following code?

```
public class PQfunctions1d
{
   public static int cube(int i)
   {
      i = i * i * i;
      return i;
   }
   public static void main(String[] args)
   {
      int N = Integer.parseInt(args[0]);
      for (int i = 1; i <= N; i++)
         StdOut.println(i + " " + cube(i));
   }
}
```

Pop quiz 1d on functions

Q. What happens when you compile and run the following code?

```
public class PQfunctions1d
{
   public static int cube(int i)
   {
      i = i * i * i;
      return i;
   }
   public static void main(String[] args)
   {
      int N = Integer.parseInt(args[0]);
      for (int i = 1; i <= N; i++)
         StdOut.println(i + " " + cube(i));
   }
}
```

A. Works. The `i` in `cube()` is

- Declared and initialized as an argument.
- Different from the `i` in `main()`.

BUT changing values of function arguments is sufficiently confusing to be deemed bad style for this course.

```
% javac PQfunctions1d.java
% java PQfunctions1d 6
1 1
2 8
3 27
4 64
5 125
6 216
```

Pop quiz 1e on functions

Q. What happens when you compile and run the following code?

```
public class PQfunctions1e
{
   public static int cube(int i)
   {
      return i * i * i;
   }
   public static void main(String[] args)
   {
      int N = Integer.parseInt(args[0]);
      for (int i = 1; i <= N; i++)
         StdOut.println(i + " " + cube(i));
   }
}
```

Pop quiz 1e on functions

Q. What happens when you compile and run the following code?

```
public class PQfunctions1e
{
   public static int cube(int i)
   {
      return i * i * i;
   }
   public static void main(String[] args)
   {
      int N = Integer.parseInt(args[0]);
      for (int i = 1; i <= N; i++)
         StdOut.println(i + " " + cube(i));
   }
}
```

A. Works fine. Preferred (compact) code.

```
% javac PQfunctions1e.java
% java PQfunctions1e 6
1 1
2 8
3 27
4 64
5 125
6 216
```

Image sources

http://upload.wikimedia.org/wikipedia/commons/b/ba/Working_Together_Teamwork_Puzzle_Concept.jpg

http://pixabay.com/en/ball-puzzle-pieces-of-the-puzzle-72374/

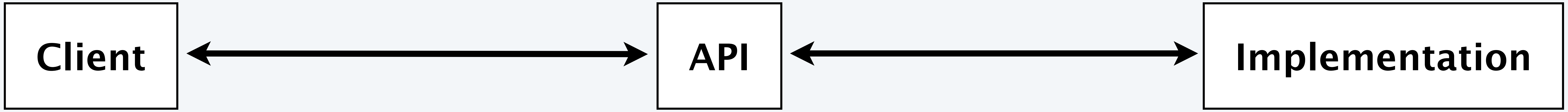
http://upload.wikimedia.org/wikipedia/commons/e/ef/Ben_Jigsaw_Puzzle_Puzzle_Puzzle.png

http://en.wikipedia.org/wiki/Function_(mathematics)#mediaviewer/File:Function_machine2.svg

5. Functions and Libraries

- Basic concepts
- Case study: Digital audio
- Application: Gaussian distribution
- **Modular programming**

Fundamental abstractions for modular programming



Client
Module that calls a library's methods.

Applications programming interface (API)
Defines signatures, describes methods.

Implementation
Module containing library's Java code.

```
public class GaussianPlot
{
   ...
   y[i] = Gaussian.pdf(x[i]);
   ...
}
```

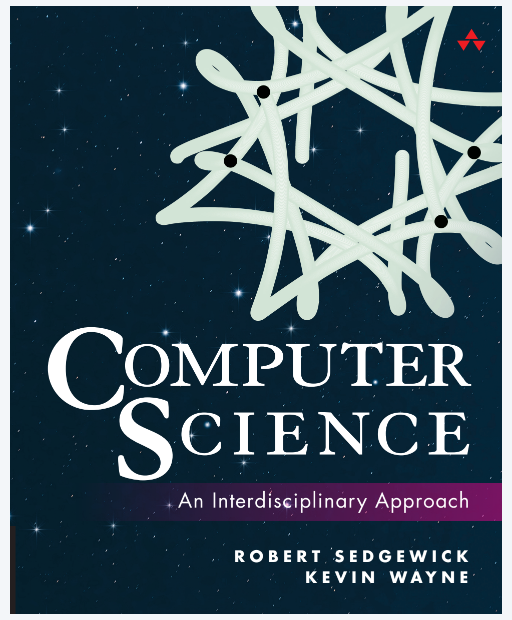
public class Gaussian	
double pdf(double x)	*Gaussian probability density function*
double cdf(double x)	*Gaussian cumulative distribution function*

```
public class Gaussian
{
  public static double pdf(double x)
  {
    double val = Math.exp(-x*x / 2);
    val /=  Math.sqrt(2 * Math.PI);
    return val
  }
  ...
}
```

Example: StdRandom library

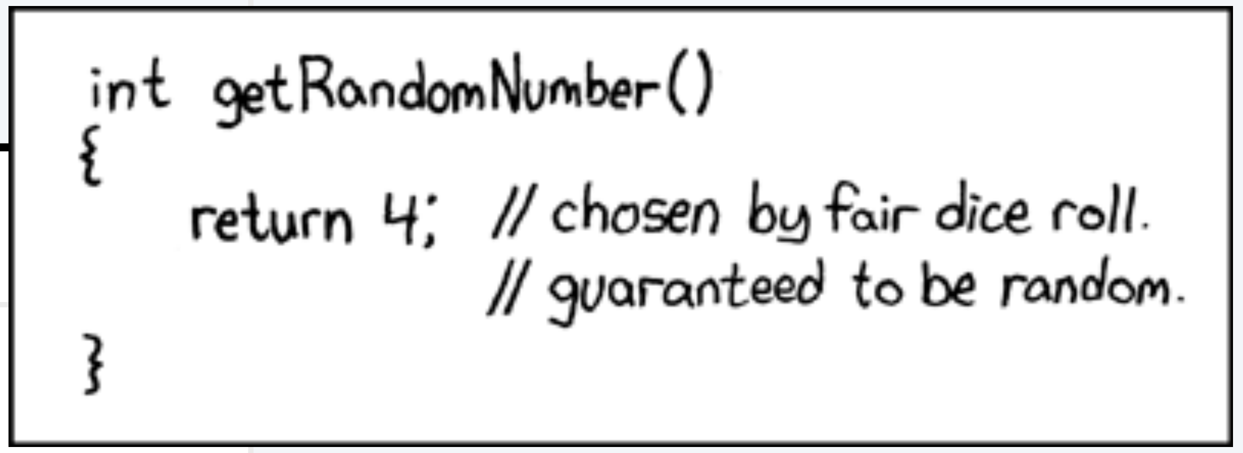
Developed for this course, but broadly useful

- Implement methods for generating random numbers of various types.
- Available for download at booksite (and included in introcs software).



API

public class StdRandom	
`int uniform(int N)`	*integer between* `0` *and* `N-1`
`double uniform(double lo, double hi)`	*real between* `lo` *and* `hi`
`boolean bernoulli(double p)`	`true` *with probability* `p`
`double gaussian()`	*normal with mean* `0`, *stddev* `1`
`double gaussian(double m, double s)`	*normal with mean* `m`, *stddev* `s`
`int discrete(double[] a)`	`i` *with probability* `a[i]`
`void shuffle(double[] a)`	*randomly shuffle the array* `a[]`



First step in developing a library: Articulate the API!

StdRandom details

Implementation

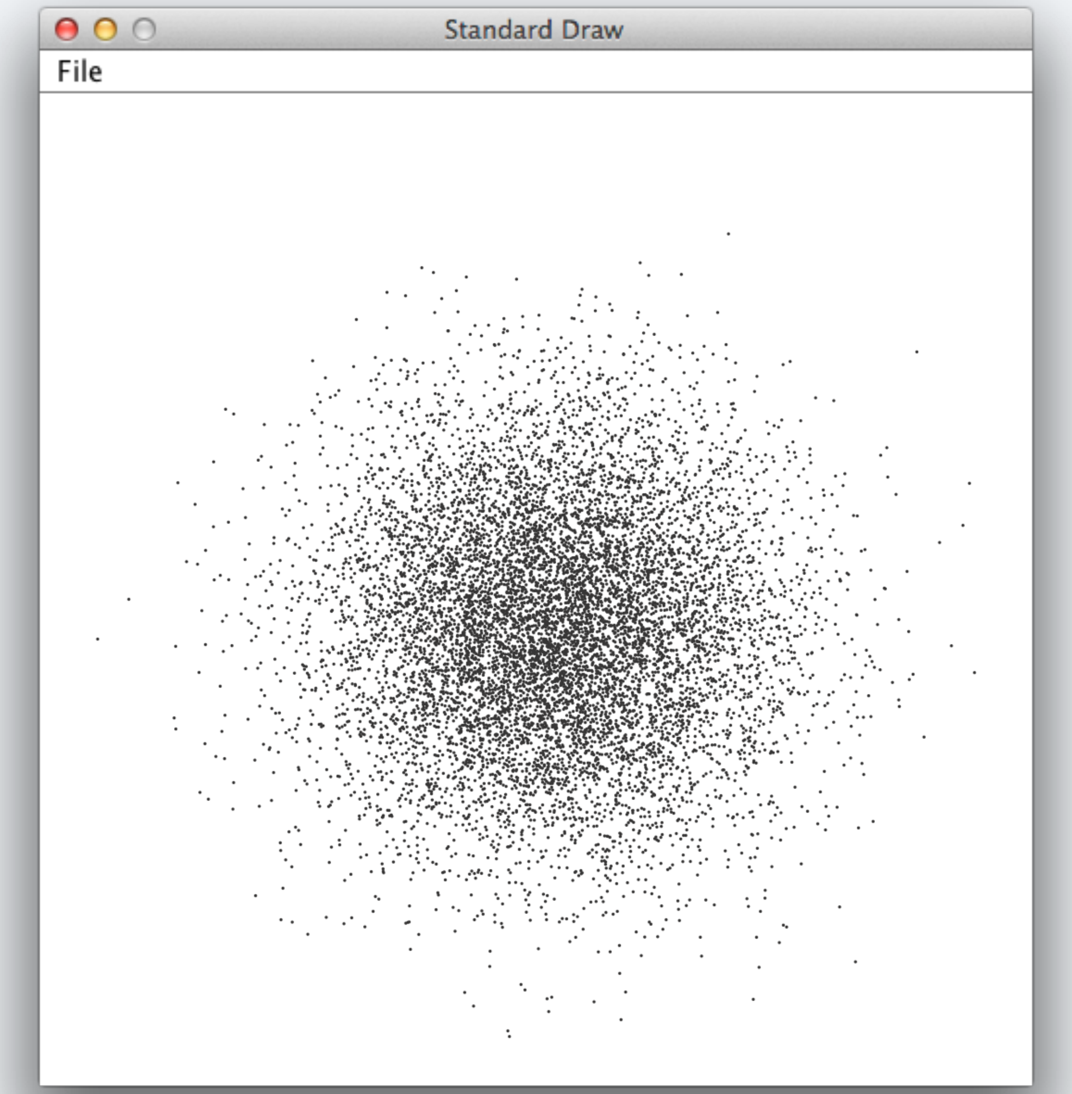
```
public class StdRandom
{
   public static double uniform(double a, double b)
   {  return a + Math.random() * (b-a);  }

   public static int uniform(int N)
   {  return (int) (Math.random() * N);  }

   public static boolean bernoulli(double p)
   {  return Math.random() < p;  }

   public static double gaussian()
      /* see Exercise 1.2.27 */

   public static double gaussian(double m, double s)
   {  return mean + (stddev * gaussian());  }
   ...
}
```

You *could* implement many of these methods, but now you don't have to!

Typical client

```
public class RandomPoints
{
   public static void main(String[] args)
   {
      int N = Integer.parseInt(args[0]);
      for (int i = 0; i < N; i++)
      {
         double x = StdRandom.gaussian(0.5, 0.2);
         double y = StdRandom.gaussian(0.5, 0.2);
         StdDraw.point(x, y);
      }
   }
}
```

```
% java RandomPoints 10000
```



Best practices

Small modules

- Separate and classify small tasks.
- Implement a layer of abstraction.

Independent development

- Code client *before* coding implementation.
- Anticipate needs of future clients.

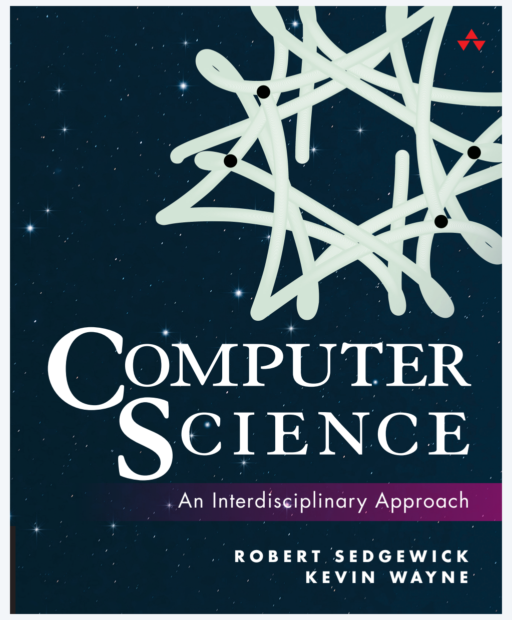
Test clients

- Include `main()` test client in each module. ← run all code at least once!
- Do more extensive testing in a separate module.

```
public class StdRandom
{
   ...
   public static void main(String[] args)
   {
      int N = Integer.parseInt(args[0]);
      for (int i = 0; i < N; i++) {
         StdOut.printf(" %2d " , uniform(100));
         StdOut.printf("%8.5f ", uniform(10.0, 99.0));
         StdOut.printf("%5b "  , bernoulli(.5));
         StdOut.printf("%7.5f ", gaussian(9.0, .2));
         StdOut.println();
      }
   }
}
```

```
% java StdRandom 5
 61 21.76541  true 9.30910
 57 43.64327 false 9.42369
 31 30.86201  true 9.06366
 92 39.59314  true 9.00896
 36 28.27256 false 8.66800
```

Example: StdStats library

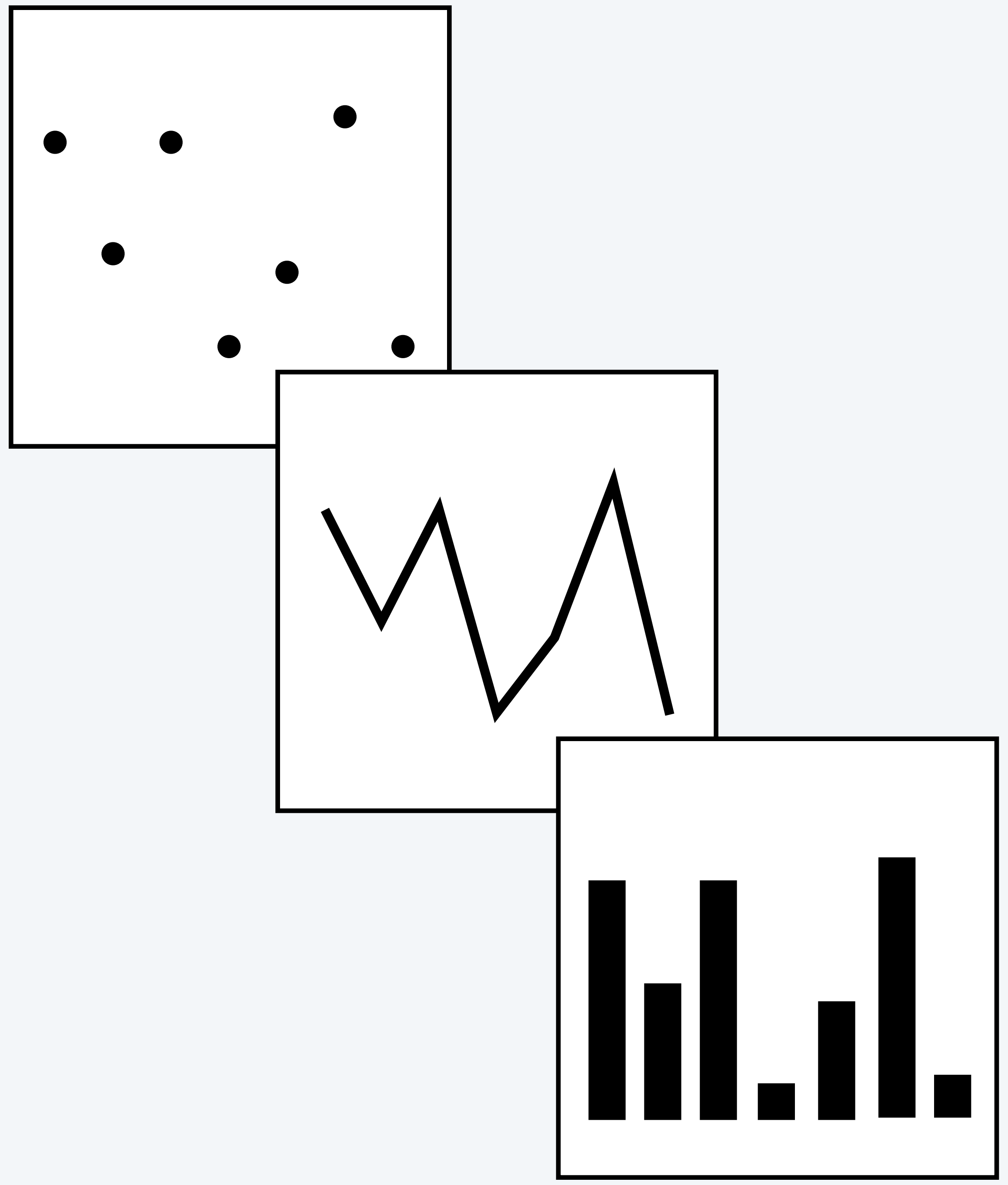


Developed for this course, but broadly useful

- Implement methods for computing statistics on arrays of real numbers. ← and plotting on StdDraw
- Available for download at booksite (and included in introcs software).

API

public class StdStats	
`double max(double[] a)`	*largest value*
`double min(double[] a)`	*smallest value*
`double mean(double[] a)`	*average*
`double var(double[] a)`	*sample variance*
`double stddev(double[] a)`	*sample standard deviation*
`double median(double[] a)`	*plot points at* `(i, a[i])`
`void plotPoints(double[] a)`	*plot points at* `(i, a[i])`
`void plotLines(double[] a)`	*plot lines connecting points at* `(i, a[i])`
`void plotBars(double[] a)`	*plot bars to points at* `(i, a[i])`



Easy to implement, but easier to use! ← one reason to develop a library: clarify client code

Example of modular programming: StdStats, StdRandom, and Gaussian client

Experiment

- Flip *N* coins.
- How many heads?
- Prediction: Expect *N*/2.

Prediction (more detailed)

- Run experiment *trials* times.
- How many heads?

```
public static int binomial(int N)
{
   int heads = 0;
   for (int j = 0; j < N; j++)
      if (StdRandom.bernoulli(0.5))
         heads++;
   return heads;
}
```

Goal. Write a program to validate predictions.

Example of modular programming: Bernoulli trials

```
public class Bernoulli
{
   public static int binomial(int N)
   // See previous slide.

   public static void main(String[] args)
   {
      int N = Integer.parseInt(args[0]);
      int trials = Integer.parseInt(args[1]);

      int[] freq = new int[N+1];
      for (int t = 0; t < trials; t++)
         freq[binomial(N)]++;

      double[] normalized = new double[N+1];
      for (int i = 0; i <= N; i++)
         normalized[i] = (double) freq[i] / trials;
      StdStats.plotBars(normalized);

      double mean = N / 2.0;
      double stddev = Math.sqrt(N) / 2.0;
      double[] phi  = new double[N+1];
      for (int i = 0; i <= N; i++)
         phi[i] = Gaussian.pdf(i, mean, stddev);
      StdStats.plotLines(phi);
   }
}
```

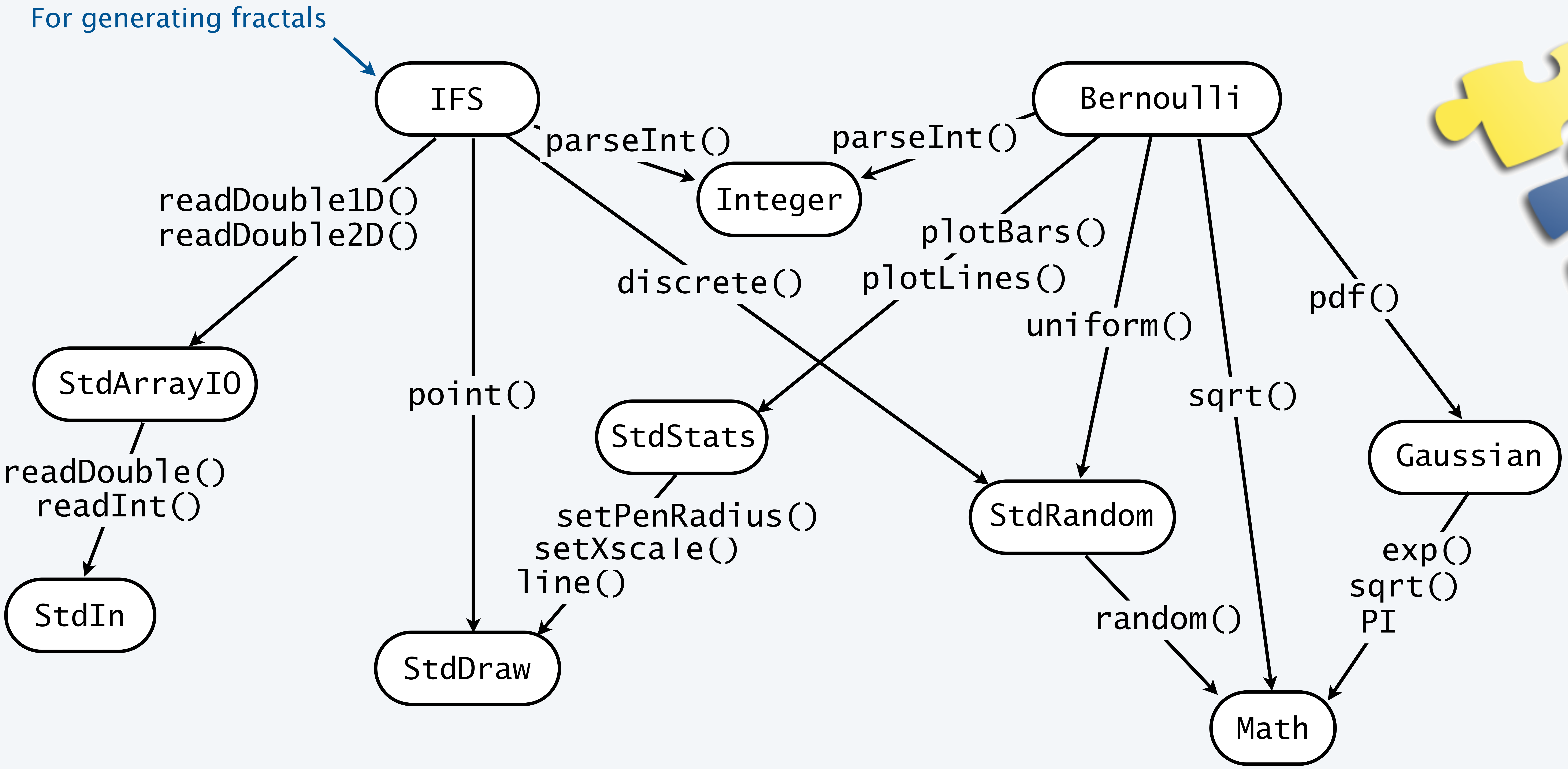
Bernoulli simulation

- Get command-line arguments (*trials* experiments of *N* flips).
- Run experiments. Keep track of frequency of occurrence of each return value.
- Normalize to between 0 and 1. Plot histogram.
- Plot theoretical curve.

```
% java Bernoulli 20 10000
```

Modular programming

enables development of complicated programs via simple independent modules.



Advantages. Code is easier to understand, debug, maintain, improve, and reuse.

Why modular programming?

Modular programming enables

- Independent development of small programs.
- Every programmer to develop and share layers of abstraction.
- Self-documenting code.

Fundamental characteristics

- Separation of client from implementation benefits all *future* clients.
- Contract between implementation and clients (API) benefits all *past* clients.

Challenges

- How to break task into independent modules?
- How to specify API?

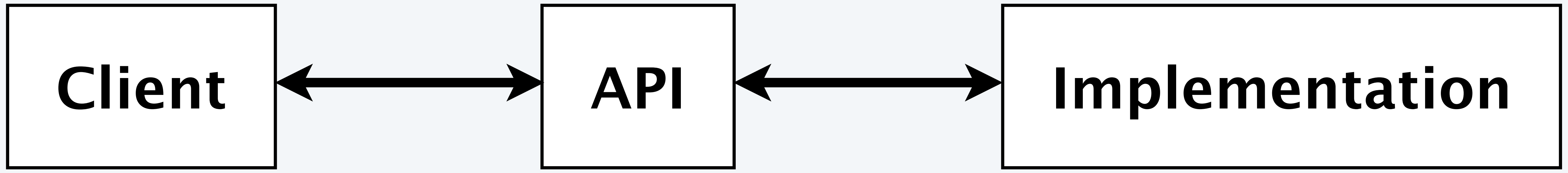


Image sources

http://xkcd.com/221/

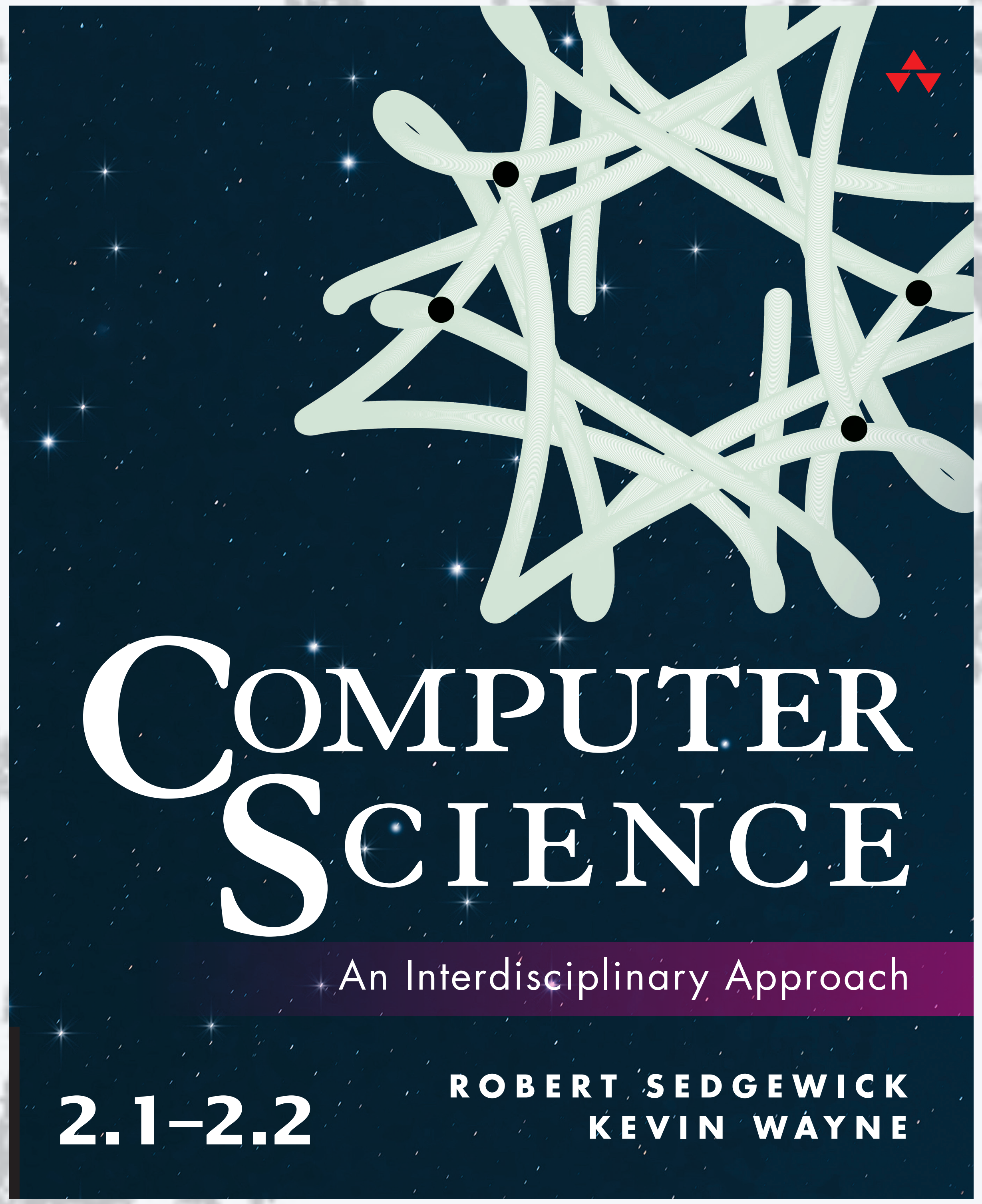
http://upload.wikimedia.org/wikipedia/commons/b/ba/Working_Together_Teamwork_Puzzle_Concept.jpg

http://upload.wikimedia.org/wikipedia/commons/e/ef/Ben_Jigsaw_Puzzle_Puzzle_Puzzle.png

COMPUTER SCIENCE

SEDGEWICK / WAYNE

PART I: PROGRAMMING IN JAVA



http://introcs.cs.princeton.edu

5. Functions and Libraries

Introduce HW3
Discuss collaboration policy
Discuss Gradescope