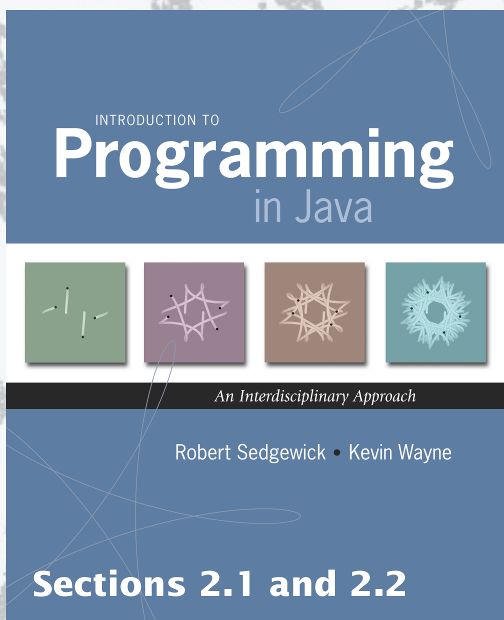
**COMPUTER SCIENCE**
**SEDGEWICK / WAYNE**



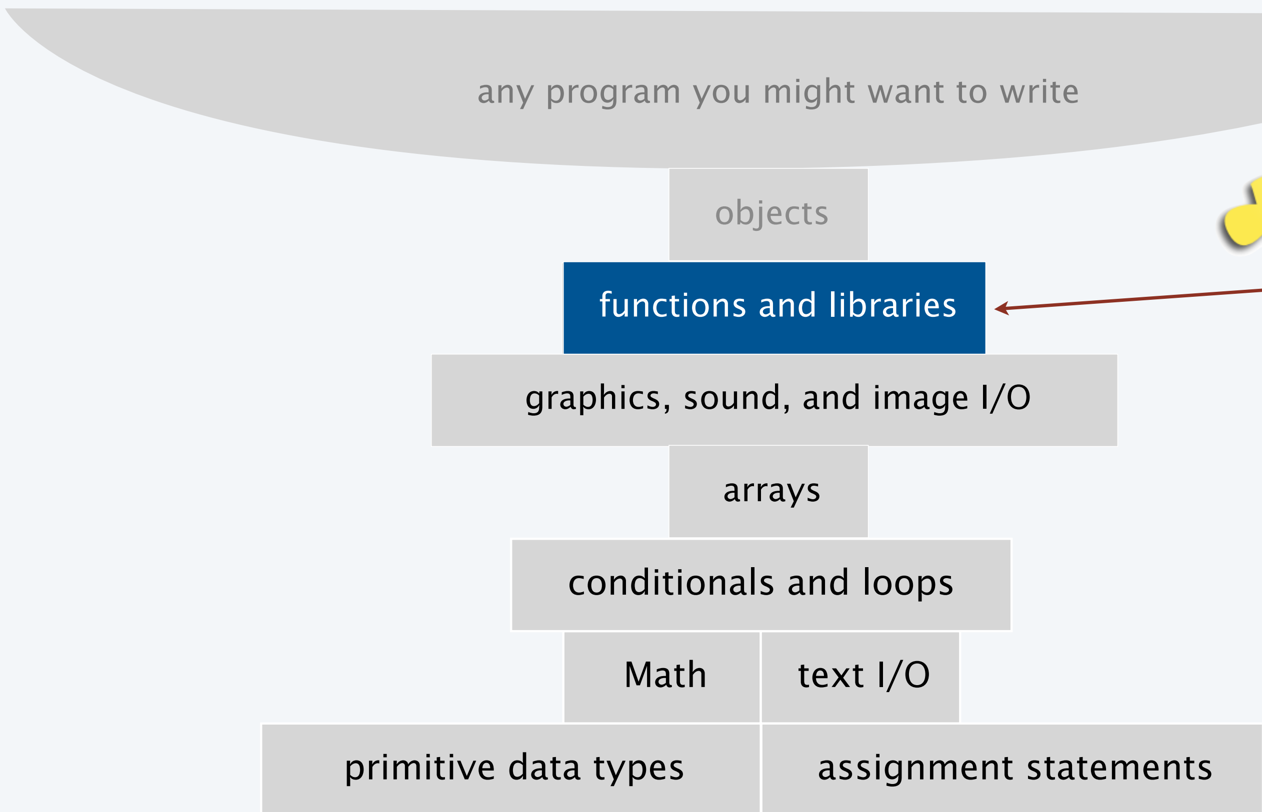

http://introcs.cs.princeton.edu

# Functions and Libraries

# 6. 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.

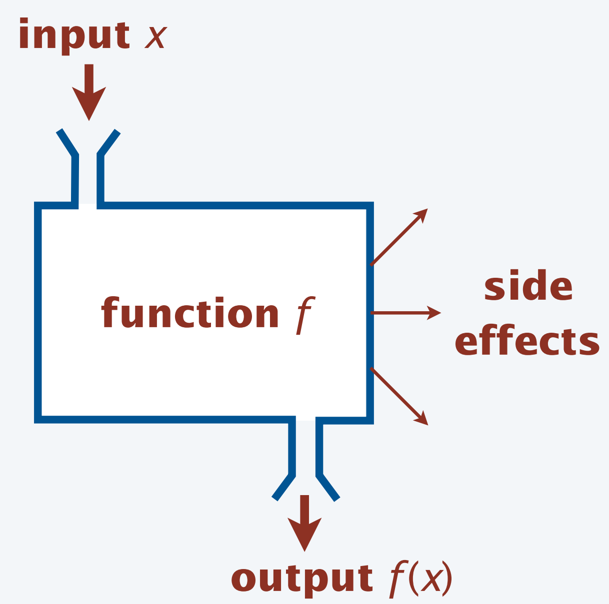
for purposes of this course

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

## 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.

Notes:

We are using our own implementation of `sqrt()` from the previous lecture, to illustrate the basics with a familiar function.

You can use `Math.sqrt()`.

Don't worry about technical details now, just flow of control.

sqrt() method

module named Newton.java

main() method

library/module 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++)
      {
         x = sqrt(a[i], 1e-3);
         StdOut.println(x);
      }
   }
}
```

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++)
      {
         x = sqrt(a[i], 1e-3);
         StdOut.println(x);
      }
   }
}
```

scope of c and eps

scope of t

cannot refer to a or i in this code

scope of a

two *different* variables named i each with two lines of scope

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

A *local variable*'s scope is the code following its declaration, in the same block.

An *argument variable*'s scope is the whole method.

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 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 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 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 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));
   }
}
```

# 6. Functions and Libraries

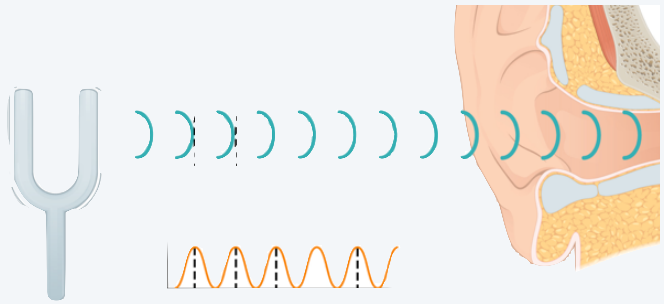
- Basic concepts
- Case study: Digital audio
- Application: Gaussian distribution
- Modular programming

# Crash course in sound

Sound is the perception of the vibration of molecules.

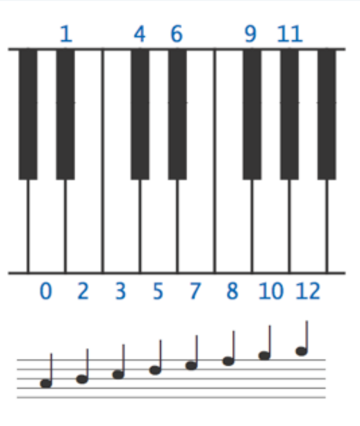
A musical tone is a periodic sound.

A pure tone is a sinusoidal waveform.

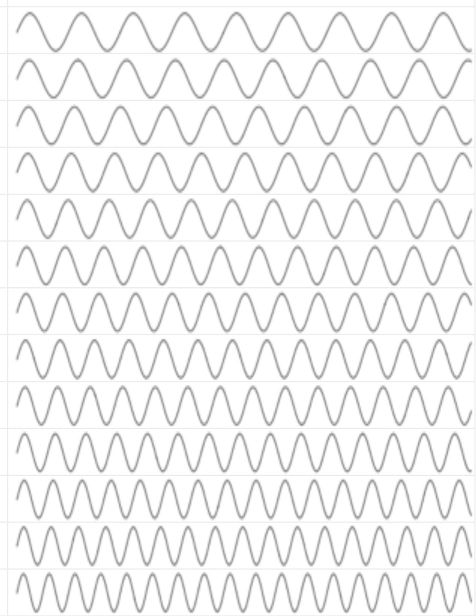


**Western musical scale**

- Concert A is 440 Hz.
- 12 notes, logarithmic scale.




| *pitch* | *i* | *frequency* (440*$2^{i/12}$) | *sinusodial waveform* |
|---|---|---|---|
| A | 0 | 440.00 | |
| A♯ / B♭ | 1 | 466.16 | |
| B | 2 | 493.88 | |
| C | 3 | 523.25 | |
| C♯ / D♭ | 4 | 554.37 | |
| D | 5 | 587.33 | |
| D♯ / E♭ | 6 | 622.25 | |
| E | 7 | 659.26 | |
| F | 8 | 698.46 | |
| F♯ / G♭ | 9 | 739.99 | |
| G | 10 | 783.99 | |
| G♯ / A♭ | 11 | 830.61 | |
| A | 12 | 880.00 | |



# Digital audio

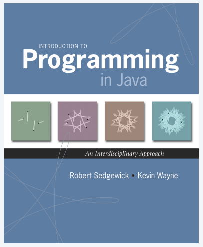
To represent a tone, *sample* a sine wave at regular intervals and save the values in an array.

same as when plotting a function (previous lecture)

| | *samples/sec* | *samples* | *sampled waveform* |
|---|---|---|---|
| **1/40 second of concert A** | 5,512 | 137 |  |
| | 11,025 | 275 |  |
| | 22,050 | 551 |  |
| CD standard → | 44,100 | 1102 |  |

Bottom line. You can *write programs* to manipulate sound (arrays of `double` values).

## StdAudio library




### Developed for this course, also broadly useful

- Play a sound wave (array of double values) on your computer's audio output.
- Convert to and from standard `.wav` file format.

**API**

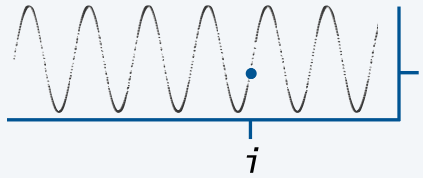
| `public class StdAudio` | |
|---|---|
| `void play(String file)` | *play the given `.wav` file* |
| `void play(double[] a)` | *play the given sound wave* |
| `void play(double x)` | *play the sample for 1/44100 second* |
| `void save(String file, double[] a)` | *save to a `.wav` file* |
| `double[] read(String file)` | *read from a `.wav` file* |

Enables you to *hear the results* of your programs that manipulate sound.

# “Hello, World” for StdAudio

pitch

```
public class PlayThatNote
{
   public static double[] tone(double hz, double duration)
   {
      int N = (int) (44100 * duration);
      double[] a = new double[N+1];
      for (int i = 0; i <= N; i++)
         a[i] = Math.sin(2 * Math.PI * i * hz / 44100);
     return a;
   }

   public static void main(String[] args)
   {
      double hz = Double.parseDouble(args[0]);
      double duration = Double.parseDouble(args[1]);
      double[] a = tone(hz, duration);
      StdAudio.play(a);
   }
}
```




$$\sin\left(\frac{2\pi i \cdot hz}{44100}\right)$$

```
% java PlayThatNote 440.0 3.0
```

```
% java PlayThatNote 880.0 3.0
```

```
% java PlayThatNote 220.0 3.0
```

```
% java PlayThatNote 494.0 3.0
```

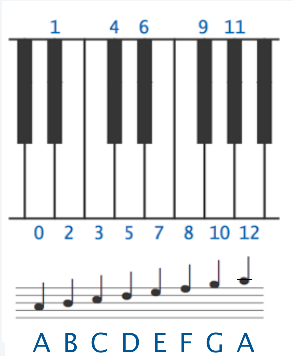
## Play that tune

Read a list of tones and durations from standard input to play a tune.

```
public class PlayThatTune
{
   public static void main(String[] args)
   {
      double tempo = Double.parseDouble(args[0]);
      while (!StdIn.isEmpty())
      {
         int pitch = StdIn.readInt();
         double duration = StdIn.readDouble() * tempo;
         double hz = 440 * Math.pow(2, pitch / 12.0);
         double[] a = PlayThatNote.tone(hz, duration);
         StdAudio.play(a);
      }
      StdAudio.close();
   }
}
```

control tempo from command line

```
% more < elise.txt
7 .125
6 .125
7 .125
6 .125
7 .125
2 .125
5 .125
3 .125
0 .25
...
```




```
% java PlayThatTune 2.0 < elise.txt
```


```
% java PlayThatTune 1.0 < elise.txt
```

## Pop quiz 2 on functions

Q. What sound does the following program produce?

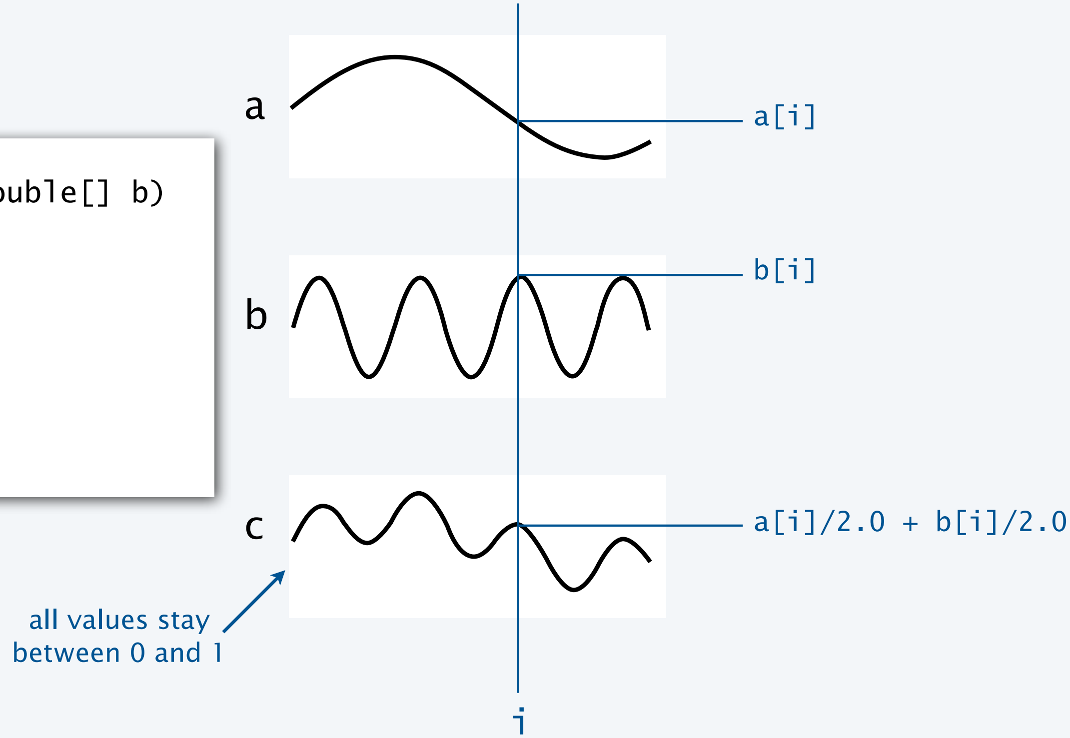
```
public class PQfunctions2
{
    public static void main(String[] args)
    {
        int N = (int) (44100 * 11);
        double[] a = new double[N+1];
        for (int i = 0; i <= N; i++)
            a[i] = Math.random();
        StdAudio.play(a);
    }
}
```

## Play that chord

Produce chords by *adding* waveforms (normalized).

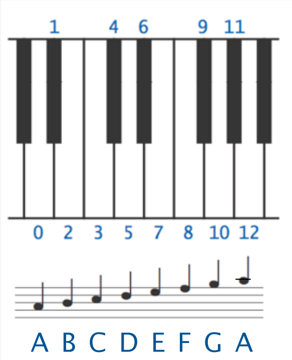
Not really—technically need *three* tones for a chord

```
public static double[] superpose(double[] a, double[] b)
{
   double[] c = new double[a.length];
   for (int i = 0; i < a.length; i++)
      c[i] = a[i]/2.0 + b[i]/2.0;
   return c;
}
```



# Play that chord implementation

```
public class PlayThatChord
{
    public static double[] superpose(double[] a, double[] b)
    {  /* See previous slide. */ }

    public static double[] chord(int pitch1, int pitch2, double duration)
    {
        double hz1 = 440.0 * Math.pow(2, pitch1 / 12.0);
        double hz2 = 440.0 * Math.pow(2, pitch2 / 12.0);
        double[] a = PlayThatNote.tone(hz1, duration);
        double[] b = PlayThatNote.tone(hz2, duration);
        return superpose(a, b);
    }

    public static void main(String[] args)
    {
        int pitch1 = Integer.parseInt(args[0]);
        int pitch2 = Integer.parseInt(args[1]);
        double duration = Double.parseDouble(args[2]);
        double[] a = chord(pitch1, pitch2, duration);
        StdAudio.play(a);
    }
}
```




```
% java PlayThatChord 0 3 5.0
```

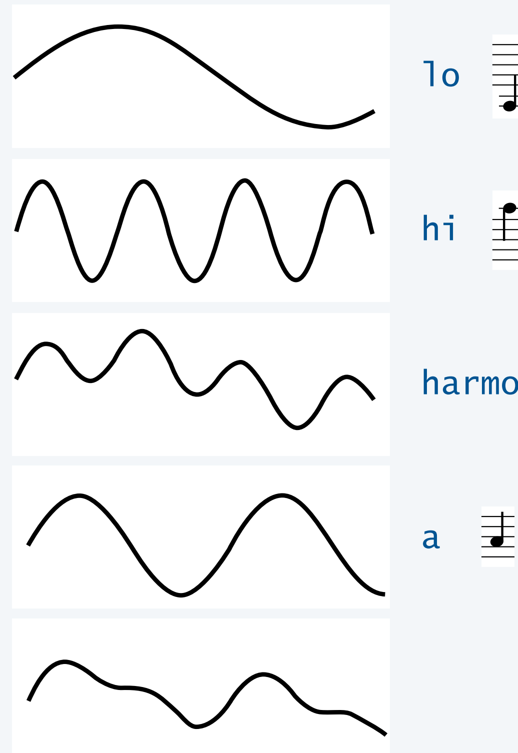
```
% java PlayThatChord 0 12 5.0
```

# Play that tune (deluxe version)

Add harmonics to PlayThatTune to produce a more realistic sound.

**Function to add harmonics to a tone**

```
public static double[] note(int pitch, double duration)
{
      double hz = 440.0 * Math.pow(2, pitch / 12.0);
      double[] a  = tone(1.0 * hz, duration);
      double[] hi = tone(2.0 * hz, duration);
      double[] lo = tone(0.5 * hz, duration);
      double[] harmonic  = superpose(hi, lo);
      return superpose(a, harmonic);
}
```




```
% java PlayThatTune 1.5 < elise.txt
```

Program 2.1.4 in text (with tempo added)

# Digital audio (summary)



```
% java PlayThatTune 1.5 < entertainer.txt
```

```
public class PlayThatTune
{
   public static double[] sum(double[] a, double[] b,
                              double awt, double bwt)
   {
      double[] c = new double[a.length];
      for (int i = 0; i < a.length; i++)
         c[i] = a[i]*awt + b[i]*bwt;
      return c;
   }

   public static double[] tone(double hz, double t)
   {
      int sps = 44100;
      int N = (int) (sps * t);
      double[] a = new double[N+1];
      for (int i = 0; i <= N; i++)
         a[i] = Math.sin(2 * Math.PI * i * hz / sps);
      return a;
   }

   public static double[] note(int pitch, double t)
   {
      double hz = 440.0 * Math.pow(2, pitch / 12.0);
      double[] a  = tone(hz, t);
      double[] hi = tone(2*hz, t);
      double[] lo = tone(hz/2, t);
      double[] h  = sum(hi, lo, .5, .5);
      return sum(a, h, .5, .5);
   }

   public static void main(String[] args)
   {
      while (!StdIn.isEmpty())
      {
         int pitch = StdIn.readInt();
         double duration = StdIn.readDouble();
         double[] a = note(pitch, duration);
         StdAudio.play(a);
      }
   }
}
```

Note: This code (from book) slightly differs from lecture code.

**Bottom line.** *You* can write programs to manipulate sound.

This lecture: Case study of the utility of functions.

Upcoming assignment: Fun with musical tones.

# 6. Functions and Libraries

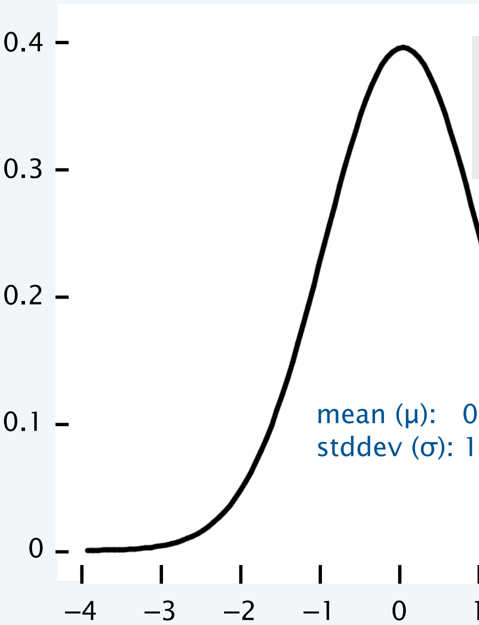
- Basic concepts
- Case study: Digital audio
- **Application: Gaussian distribution**
- Modular programming

# Gaussian distribution

**Gaussian distribution.**

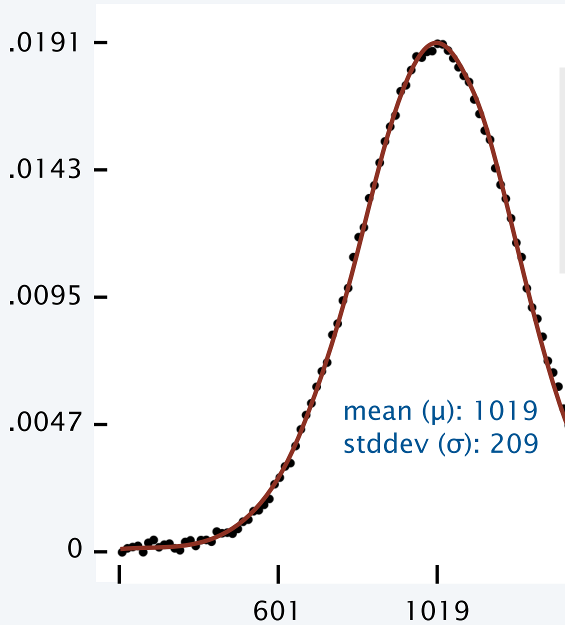
- A mathematical model used successfully for centuries.
- "Bell curve" fits experimental observations in many contexts.

Gaussian distribution function




$$\phi(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}$$

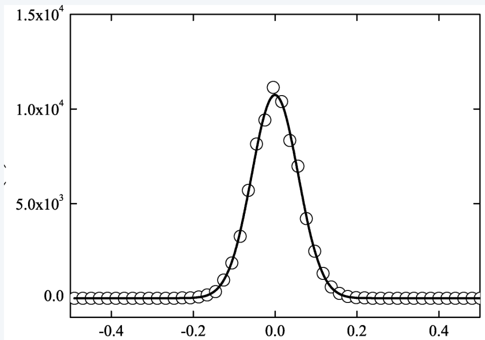
Example: SAT scores in 20xx (verbal + math)




$$\phi(x, \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/(2\sigma^2)} = \frac{1}{\sigma}\phi\left(\frac{x-\mu}{\sigma}\right)$$

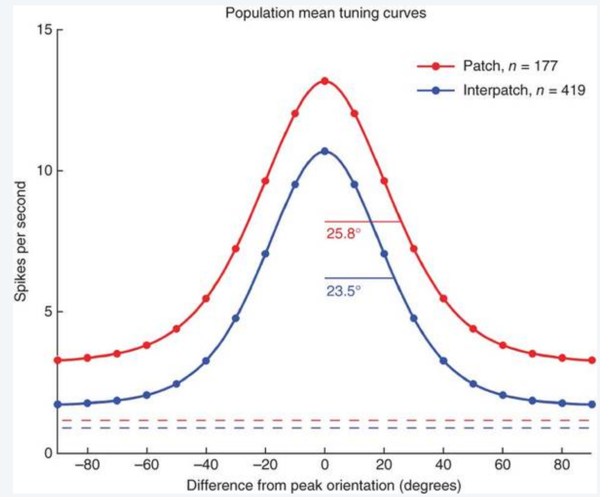
# Gaussian distribution in the wild

Polystyrene particles in glycerol

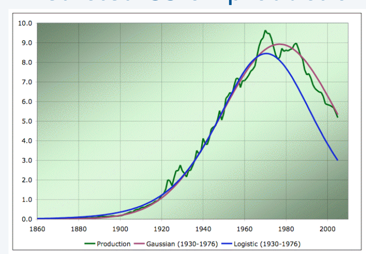



German money

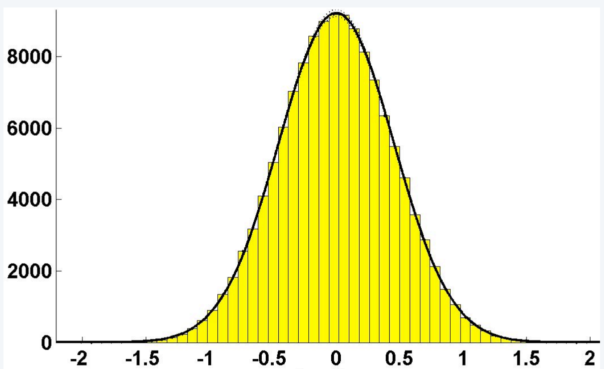

Cytochrome oxidase patches
in macaque primary visual cortex



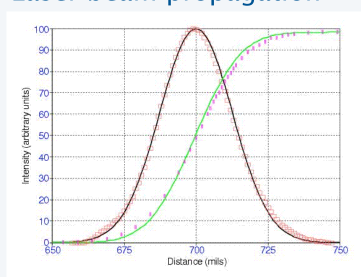

Predicted US oil production



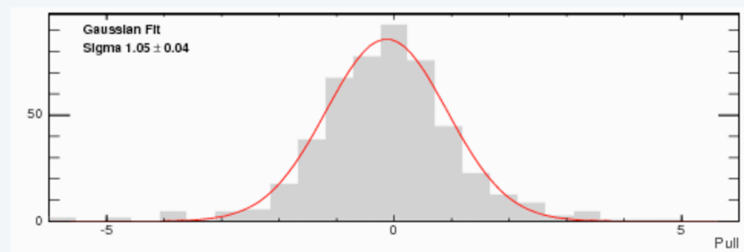

Calibration of optical tweezers




Laser beam propagation




Polarized W bosons from top-quark decay



## Defining a library of functions

Q. Is the Gaussian function implemented in Java's Math library?

A. No.

Q. Why not?

A. Maybe because it is so easy for you to do it yourself.

$$\phi(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}$$

$$\phi(x, \mu, \sigma) = \frac{1}{\sigma}\phi\left(\frac{x-\mu}{\sigma}\right)$$

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

   public static double phi(double x, double mu, double sigma)
   {  return phi((x - mu) / sigma) / sigma; }

   // Stay tuned for more functions.
}
```

call a function in another module

module named Gaussian.java

functions with different signatures are different (even if names match)

Functions and libraries provide an easy way for any user to *extend* the Java system.

## Gaussian cumulative distribution function

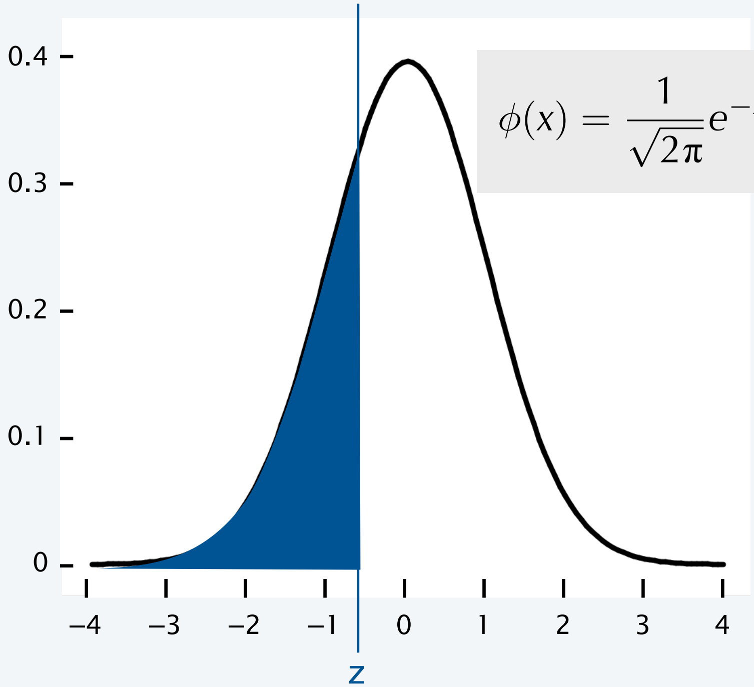
Q. What percentage of the total is less than or equal to *z*?

Q. (equivalent). What is the area under the curve to the left of *z*?

A. Gaussian *cumulative distribution function*.

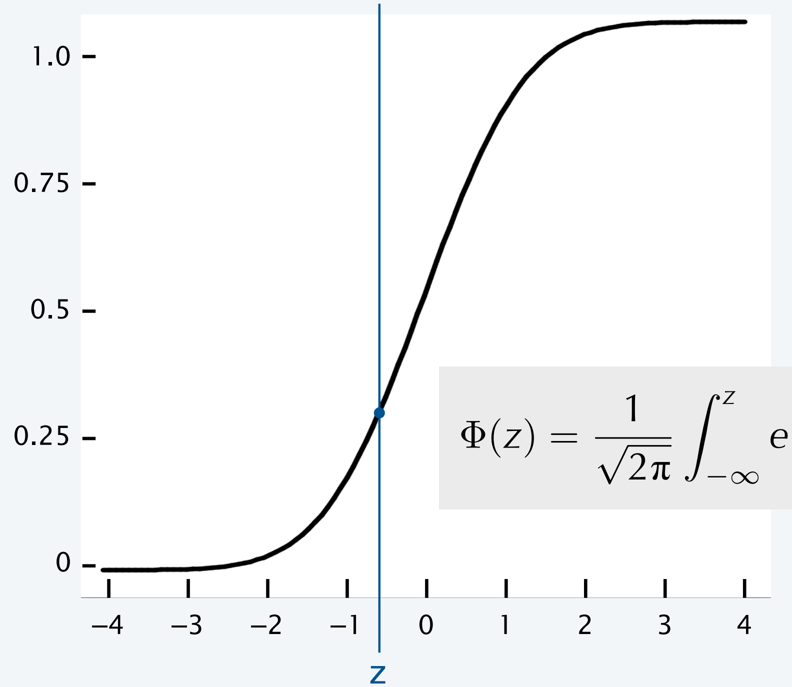
$$\Phi(x, \mu, \sigma) = \Phi\left(\frac{x-\mu}{\sigma}\right)$$

Gaussian distribution function




$$\phi(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}$$

Gaussian cumulative distribution function

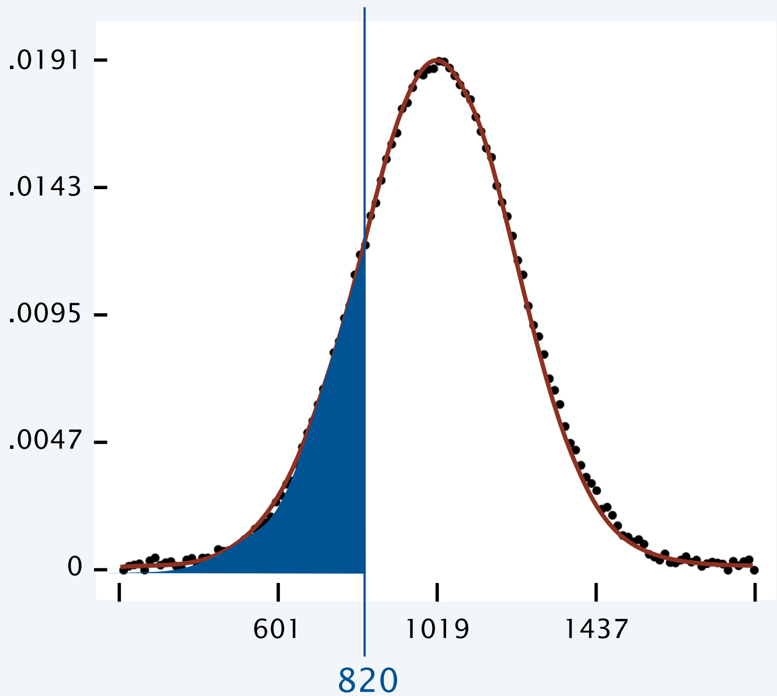


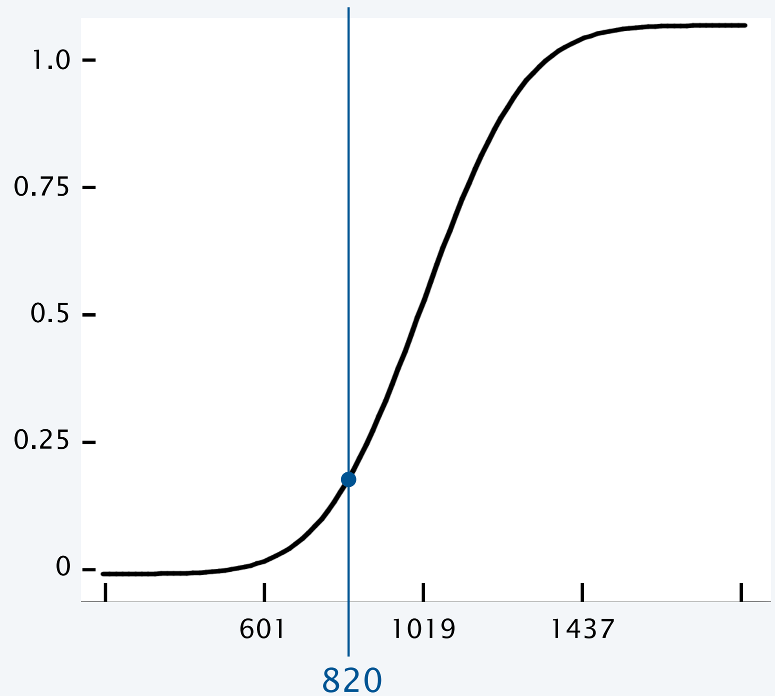

$$\Phi(z) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{z} e^{-x^2/2} dx$$

## Typical application: SAT scores

Q. In 20xx NCAA required at least 820 for Division I athletes.
What fraction of test takers did not qualify??

A. About 17%, since $\Phi(820, 1019, 209) = 0.17050966869132111...$



# Gaussian CDF implementation

Q. No closed form for Gaussian CDF. How to implement?

A. Use Taylor series. $\Phi(z) \equiv \int_{-\infty}^{z} \phi(x)dx = \frac{1}{2} + \phi(z)\left(z + \frac{z^3}{3} + \frac{z^5}{3 \cdot 5} + \frac{z^7}{3 \cdot 5 \cdot 7} + \ldots\right)$

```
public static double Phi(double z)
{
   if (z < -8.0) return 0.0;
   if (z >  8.0) return 1.0;
   double sum = 0.0, term = z;
   for (int i = 3; sum + term != sum; i += 2)
   {
      sum  = sum + term;
      term = term * z * z / i;
   }
   return 0.5 + sum * phi(z);   ← accurate to 15 places
}

public static double Phi(double z, double mu, double sigma)
{  return Phi((z - mu) / sigma);  }
```

Bottom line. 1,000 years of mathematical formulas at your fingertips.

## Summary: a library for Gaussian distribution functions

### Best practice

- Test all code at least once in `main()`.
- Also have it do something useful.

Q. What fraction of SAT test takers did not qualify for NCAA participation in 20xx?

```
% java Gaussian 820 1019 209
0.17050966869132111
```

### Fun fact

We use `Phi()` to evaluate randomness in submitted programs.

### Bottom line

YOU can build a layer of abstraction to use in any future program.

```
public class Gaussian
{
   public static double phi(double x)
   {  return Math.exp(-x*x / 2) / Math.sqrt(2 * Math.PI);  }
   public static double phi(double x, double mu, double sigma)
   {  return phi((x - mu) / sigma) / sigma; }
   public static double Phi(double z)
   {
      if (z < -8.0) return 0.0;
      if (z >  8.0) return 1.0;
      double sum = 0.0, term = z;
      for (int i = 3; sum + term != sum; i += 2)
      {
         sum  = sum + term;
         term = term * z * z / i;
      }
      return 0.5 + sum * phi(z);
   }
   public static double Phi(double z, double mu, double sigma)
   {  return Phi((z - mu) / sigma);  }
   public static void main(String[] args)
   {
      int z = Integer.parseInt(args[0]);
      int mu = Integer.parseInt(args[1]);
      int sigma = Integer.parseInt(args[2]);
      StdOut.println(Phi(z, mu, sigma));
   }
}
```

# Using a library

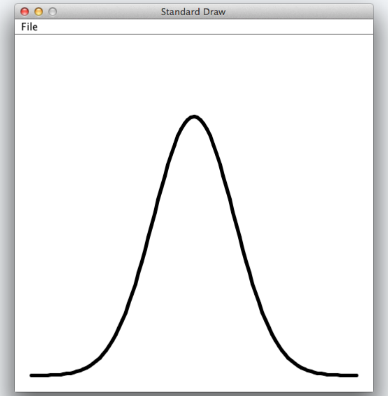
## To use these methods in another program

Learn to use your OS "classpath" mechanism if you find yourself with too many copies.

- Put a copy of `Gaussian.java` in your working directory.
- Call `Gaussian.phi()` or `Gaussian.Phi()` from any other module in that directory.

**Example.** Draw a plot of $\phi(x, 0, 1)$ in (–4, 4)

```
% java GaussianPlot 200
```




```java
public class GaussianPlot
{
    public static void main(String[] args)
    {
        int N = Integer.parseInt(args[0]);
        StdDraw.setXscale(-4.0, +4.0);
        StdDraw.setYscale(0, .5);
        StdDraw.setPenRadius(0.01);
        double[] x = new double[N+1];
        double[] y = new double[N+1];
        for (int i = 0; i <= N; i++)
        {
           x[i] = -4.0 + 8.0 * i / N;
           y[i] = Gaussian.phi(x[i], 0, 1);
        }
        for (int i = 0; i < N; i++)
           StdDraw.line(x[i], y[i], x[i+1], y[i+1]);
    }
}
```

Libraries of functions provide an easy way for *any* user (you) to extend the Java system.

# 6. 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.

**API**
Defines signatures, describes methods.

**Implementation**
Module containing library's Java code.

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

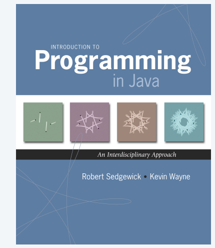
| public class Gaussian | |
|---|---|
| double phi(double x) | *Gaussian distribution* |
| double Phi(double x) | *Gaussian CDF* |

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

# Example: StdRandom library

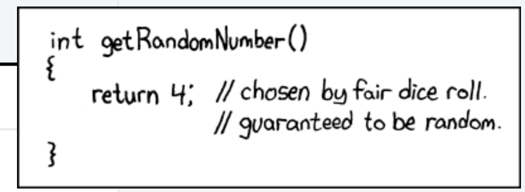
Developed for this course, but broadly useful

- 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**

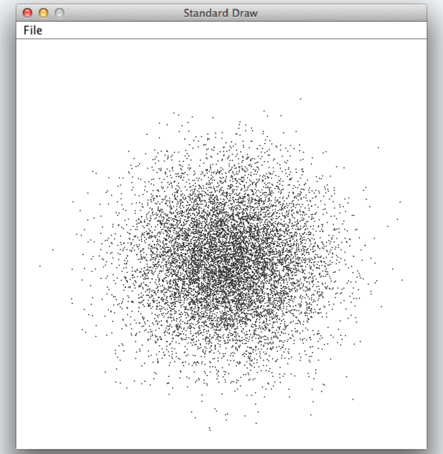
```
public class StdRandom
{
   public static double uniform(double a, double b)
   {  return a + Math.random() * (b-a);  }

   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
```



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

Experiment

- Flip *N* coins.
- How many heads?

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

Prediction: Expect N/2 heads.

Prediction (more detailed)

- Run experiment T times.
- How many occurrences of each possible outcome (number of 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 T = Integer.parseInt(args[1]);

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

      double[] normalized = new double[N+1];
      for (int i = 0; i <= N; i++)
         normalized[i] = (double) freq[i] / T;
      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.phi(i, mean, stddev);
      StdStats.plotLines(phi);
   }
}
```

see text

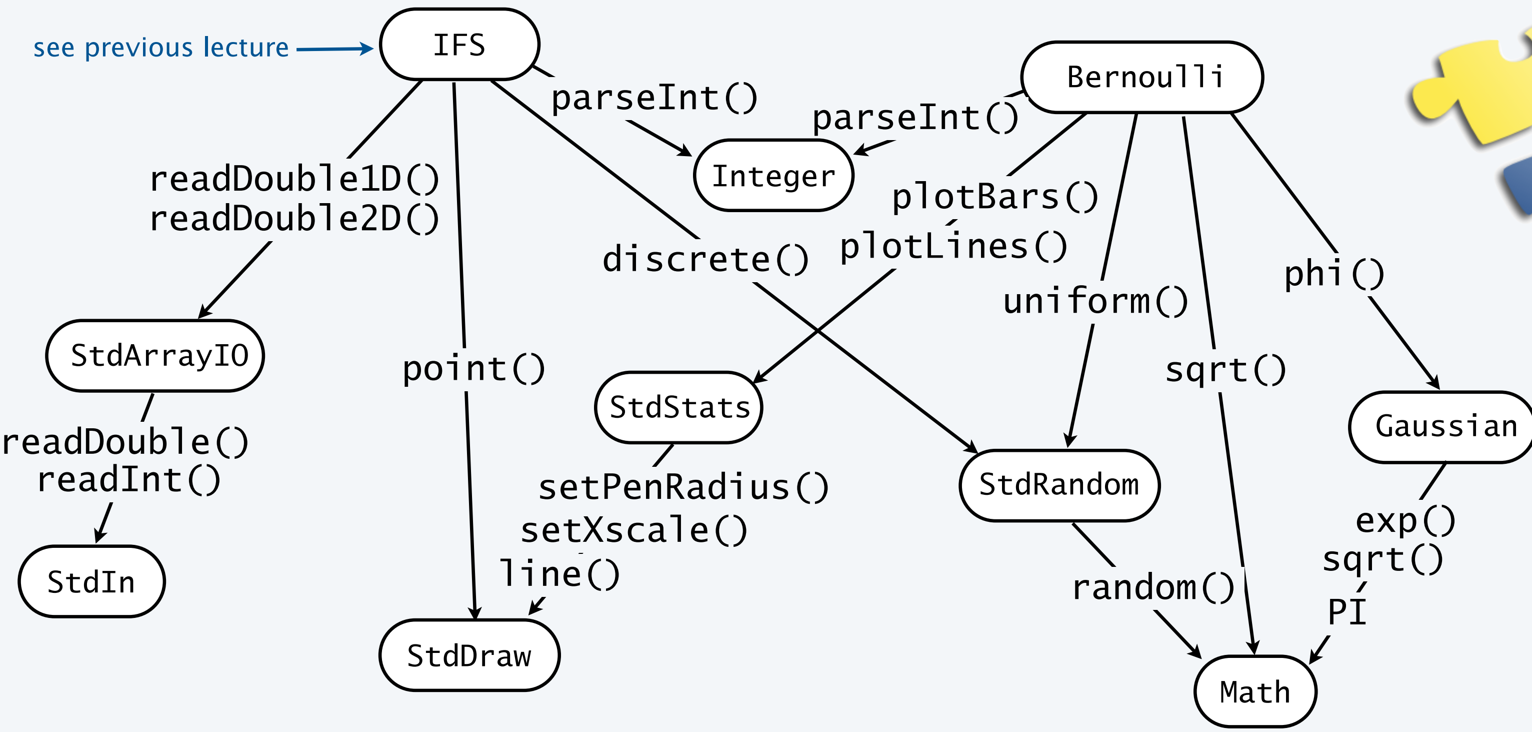
## Bernoulli simulation

- Get command-line arguments ($T$ experiments of $N$ flips).
- Compute frequency of occurrence of each possible experiment outcome.
- Convert frequencies to probabilities and 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

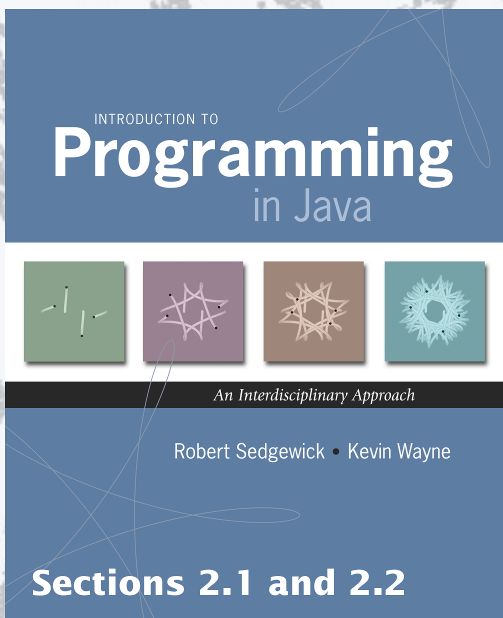
- Separation of client from implementation benefits all *future* clients.
- Contract between implementation and clients (API) benefits all *past* clients.

### Challenges

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

**Client** ⟷ **API** ⟷ **Implementation**

COMPUTER SCIENCE
SEDGEWICK / WAYNE




http://introcs.cs.princeton.edu

# 6. Functions and Libraries