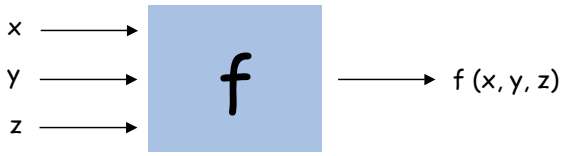
2.1 Functions



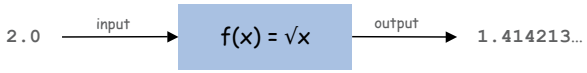
Java function.
- Takes zero or more input arguments.
- Returns one output value.

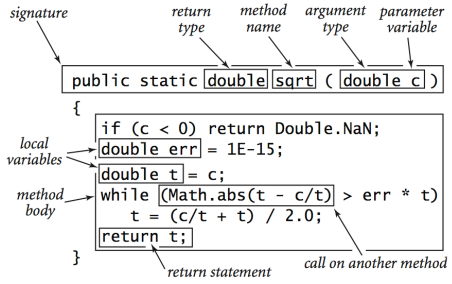
Applications.
- Scientists use mathematical functions to calculate formulas.
- Programmers use functions to build modular programs.
- You use functions for both.

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

Java Functions

Java functions. Easy to write your own.





Flow of Control

Flow of control. Functions provide a new way to control the flow of execution of a program.

```
public class Newton {

   public static double sqrt(double c)
      // see previous slide

   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]));
   }
}
```

"pass-by-value"

```
% java Newton 1 2 3
1.0
1.414213562373095
1.7320508075688772
```

Scope

Scope. Set of statements that can refer to that name.

- Scope of a variable defined within a block is limited to the statements in that block.
- Best practice: declare variables to limit their scope.

including function block

```
public class Scope {

   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));
   }
}
```

two variables named i are independent

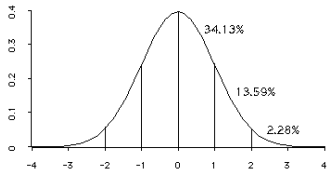
```
% java Scope 3
1 1
2 8
3 27
```


Gaussian Distribution

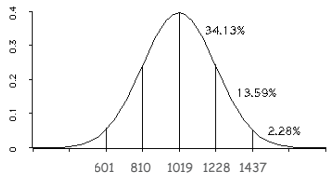
Standard Gaussian distribution.

- "Bell curve."
- Basis of most statistical analysis in social and physical sciences.

Ex. 2000 SAT scores follow a Gaussian distribution with mean μ = 1019, stddev σ = 209.



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



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

Java Function for $\phi(x)$

Mathematical functions. Use built-in functions when possible; build your own when not available.

```
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;
   }
}
```

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

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

Overloading. Functions with different signatures are different.
Multiple arguments. Functions can take any number of arguments.
Calling other functions. Functions can call other functions.

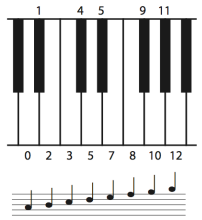
library or user-defined

Digital Audio

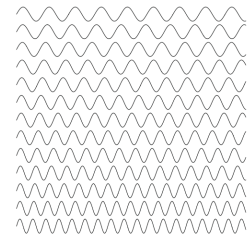
Crash Course in Sound

Sound. Perception of the vibration of molecules in our eardrums.

Concert A. Sine wave, scaled to oscillated at 440Hz.
Other notes. 12 notes on chromatic scale, divided logarithmically.



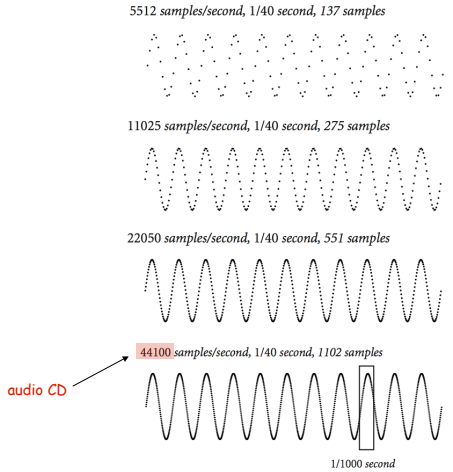
note	i	frequency
A	0	440.00
A♯ or B♭	1	466.16
B	2	493.88
C	3	523.25
C♯ or D♭	4	554.37
D	5	587.33
D♯ or E♭	6	622.25
E	7	659.26
F	8	698.46
F♯ or G♭	9	739.99
G	10	783.99
G♯ or A♭	11	830.61
A	12	880.00



$440 \times (2^{1/12})^i$

Digital Audio

Sampling. Represent curve by sampling it at regular intervals.



$$y(i) = \sin\left(\frac{2\pi \cdot i \cdot 440}{44{,}100}\right)$$

Musical Tone Function

Musical tone. Create a music tone of a given frequency and duration.

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

$$y(i) = \sin\left(\frac{2\pi \cdot i \cdot hz}{44{,}100}\right)$$

Remark. Can use arrays as function return value and/or argument.

Digital Audio in Java

Standard audio. Library for playing digital audio.

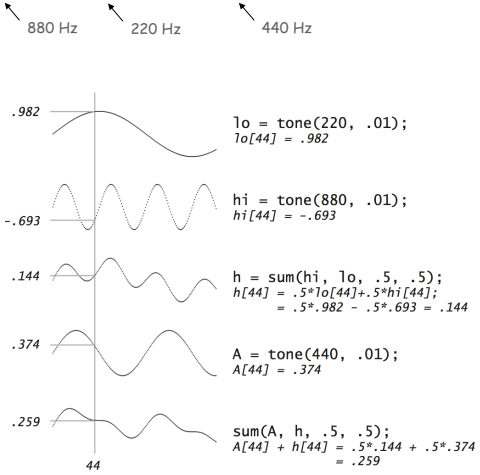
public class StdAudio	
void play(String file)	*play the given .wav file*
void play(double[] a)	*play the given sound wave*
void save(String file, double[] a)	*save to a .wav file*
void double[] read(String file)	*read from a .wav file*

Concert A. Play concert A for `1.5` seconds using `StdAudio`.

```
double[] a = tone(440, 1.5);
StdAudio.play(a);
```

Harmonics

Concert A with harmonics. Obtain richer sound by adding tones one octave above and below concert A.



Harmonics

```
public class PlayThatTune {

   // return weighted sum of two arrays
   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;
   }

   // return a note of given pitch and duration
   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[] h  = sum(hi, lo, .5, .5);
      return sum(a, h, .5, .5);
   }

   public static double[] tone(double hz, double t)
      // see previous slide

   public static void main(String[] args)
      // see next slide
}
```

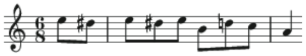

Harmonics

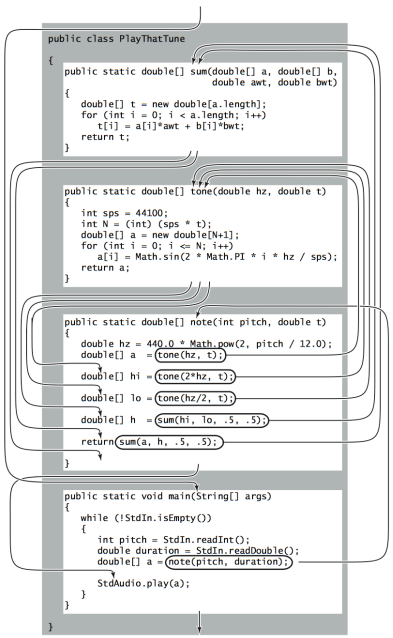
Play that tune. Read in pitches and durations from standard input, and play using standard audio.

```
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);
   }
}
```

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

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





Building Libraries

Standard Random

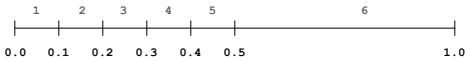
Ex. Library to generate pseudo-random numbers.

```
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()
      // recall Assignment 0

   public static double discrete(double[] p)
      // next slide
}
```


Discrete Distribution

Discrete distribution. Given an array of weights (that sum to 1), choose an index at random with probability equal to its weight.

```
p = { 0, .1, .1, .1, .1, .1, .5 }
```




```
public static int discrete(double[] p) {

   // check that weights are nonnegative and sum to 1

   double r = Math.random();
   double sum = 0.0;
   for (int i = 0; i < p.length; i++) {
      sum = sum + p[i];
      if (sum >= r) return i;
   }
   return -1;
}
```

something went wrong

Using a Library

To use the standard random library:

- Put a copy of `StdRandom.java` in current directory.
- Write a client program that uses it.

```
public class LoadedDie {
   public static void main(String args[]) {
      int N = Integer.parseInt(args[0]);
      double[] p = { 0, .1, .1, .1, .1, .1, .5 };
      for (int i = 0; i < N; i++) {
         int die = StdRandom.discrete(p);
         System.out.print(die + " ");
      }
   }
}
```

50% chance of 6,
10% chance of 1-5

```
% javac LoadedDie.java
% java LoadedDie 10
6 5 1 2 6 6 2 6 6 6
```

automatically compiles `StdRandom.java` if needed

Building Libraries

Functions enable you to build a new layer of abstraction.

- Takes you beyond pre-packaged libraries.
- You build the tools you need: `Gaussian.Phi()`, `StdRandom.uniform()`, …

Process.

- Step 1: identify a useful feature.
- Step 2: implement it.
- Step 3: use it.

- Step 3': re-use it in any of your programs.

Standard Statistics

Ex. Library to compute statistics on an array of real numbers.

```
public class StdStats {

   public static double max(double[] a) {
      double max = Double.NEGATIVE_INFINITY;
      for (int i = 0; i < a.length; i++)
         if (a[i] > max) max = a[i];
      return max;
   }

   public static double mean(double[] a) {
      double sum = 0.0;
      for (int i = 0; i < a.length; i++)
         sum = sum + a[i];
      return sum / a.length;
   }

   public static double stddev(double[] a)
     // see text

}
```


Modular Programming

Modular Programming

Modular programming.

- Divide program into self-contained pieces.
- Test each piece individually.
- Combine pieces to make program.

Ex. Coupon collector.

- Read parameters from user.
- Choose a random card between 0 and N-1.
- Run one coupon collector simulation.
- Repeat simulation many times.
- Tabulate statistics.
- Print results.

Coupon Collector

Coupon collector function. Given N card types, how many cards do you need to collect until you have at least one of each type?

```
public class Coupon {

   public static int collect(int N) {
      int cardcnt = 0;   // number of cards collected
      int valcnt  = 0;   // number of distinct cards

      // found[i] = true if card type i already collected
      boolean[] found = new boolean[N];

      // collect cards until you have one of each type
      while (valcnt < N) {
         int r = StdRandom.uniform(N);
         if (!found[r]) valcnt++;
         found[r] = true;
         cardcnt++;
      }
      return cardcnt;
   }

}
```


Coupon Collector Experiment

Computational experiment.

- For each N, collect coupons until at least one of each type.
- Repeat experiment several times for each value of N.
- Tabulate statistics and analyze results.

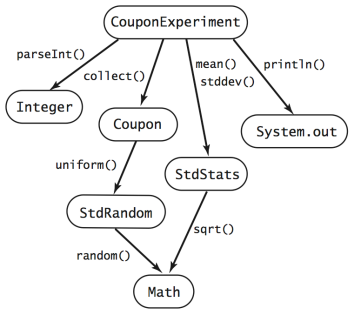
```
public class CouponExperiment {
   public static void main(String[] args) {
      int TRIALS = Integer.parseInt(args[0]);
      double[] results = new double[TRIALS];
      for (int N = 100; N <= 10000; N *= 10) {
         for (int i = 0; i < TRIALS; i++)
            results[i] = Coupon.collect(N);
         double mean   = StdStats.mean(results);
         double stddev = StdStats.stddev(results);
         StdOut.printf("%8d %8.2f %8.2f\n", N, mean, stddev);
      }
   }
}
```

```
% java CouponExperiment 10000
   100   516.07   126.10
  1000  7511.94  1300.04
 10000 97778.80 12795.39
```

mean stddev

Coupon Collector: Dependency Graph

Modular programming. Build relatively complicated program by combining several small, independent, modules.



Dependency graph

Functions

Why use functions?

- Makes code easier to understand.
- Makes code easier to debug.
- Makes code easier to maintain.
- Makes code easier to re-use.