

Libraries, clients, printf



Input: `printf("Color %s, Number %d, Float %5.2f", "red", 123456, 3.14;)`

Output: Color red, Number 123456, Float 3.14

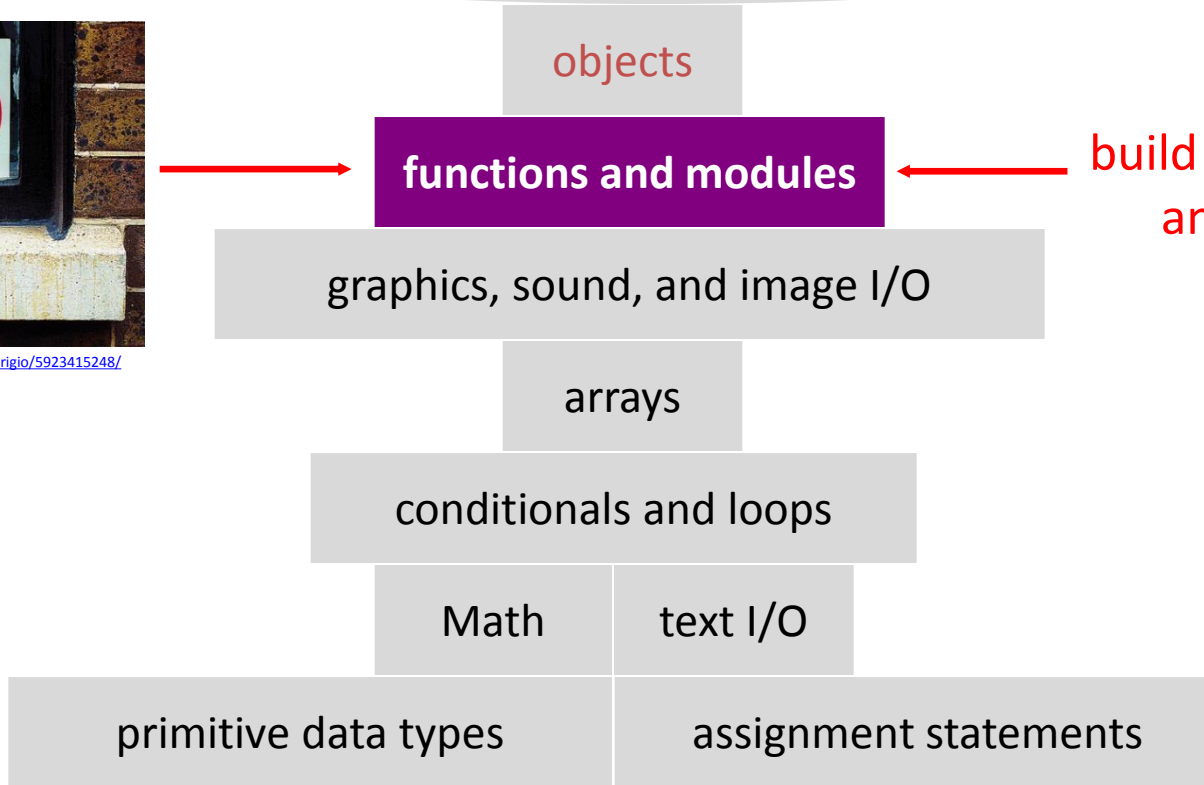
```
graph TD
    subgraph Input
        I1["Color %s, Number %d, Float %5.2f"]
        I2["red"]
        I3["123456"]
        I4["3.14"]
    end
    subgraph Output
        O1["Color red, Number 123456, Float 3.14"]
    end
    I1 --> O1
    I2 --> O1
    I3 --> O1
    I4 --> O1
    I4 --> I1
```

A foundation for programming

any program you might want to write

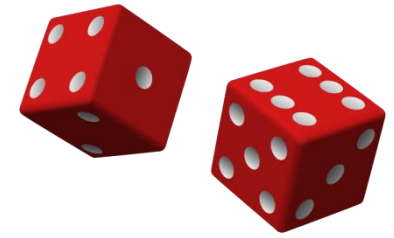


<http://www.flickr.com/photos/vermegrijo/5923415248/>



build bigger programs
and reuse code

Programs thus far



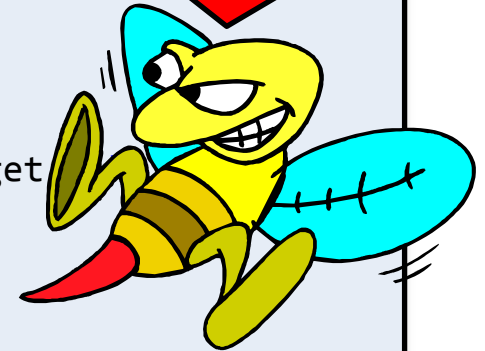
- Problems with one big `main()`:
 - Doesn't scale to complex programs
 - Often find ourselves repeating similar code

```
public class DiceRolling
{
    public static void main(String [] args)
    {
        int rolls = 0;
        int sum = 0;
        int target = (int) (Math.random() * 12) + 2;

        System.out.println("Rolling dice until I get " + target);

        do
        {
            int dice1 = (int) (Math.random() * 6) + 1;
            int dice2 = (int) (Math.random() * 6) + 1;
            sum = dice1 + dice2;
            ...
        }
    }
}
```

"Repeated code is evil!"



Calling our new method

- Use handy new method in DiceRolling
 - Add somewhere inside public class {}'s

```
public class DiceRolling
{
    public static int getRandomNum(int start, int end)
    {
        return (int) (Math.random() *
                    (end - start + 1)) + start;
    }

    public static void main(String [] args)
    {
        int rolls = 0;
        int sum = 0;
        int target = getRandomNum(1, 12);

        System.out.println("Rolling dice until I get " + target + ".");
        do
        {
            int dice1 = getRandomNum(1, 6);
            int dice2 = getRandomNum(1, 6);
            sum = dice1 + dice2;
            ...
        }
    }
}
```

Calling our new method

- **Alternative: put method in new class**
 - Allows us to create a class with a bunch of helper methods (just like `StdIn.java`, `StdDraw.java`)

```
public class RandomUtil
{
    // Return random integer in [start, end] inclusive
    public static int getRandomNum(int start, int end)
    {
        return (int) (Math.random() *
                     (end - start + 1)) + start;
    }

    // Return random integer in [0, end] inclusive
    public static int getRandomNum(int end)
    {
        return (int) (Math.random() * (end + 1));
    }
}
```

`getRandomInt()` is **overloaded**:
Two methods with same name, but different signatures (e.g. different number of parameters)

Using our new class

- Put `RandomUtil.java` in same directory
 - Methods qualified with `RandomUtil.` in front

```
public class DiceRolling
{
    public static void main(String [] args)
    {
        int rolls = 0;
        int sum = 0;
        int target = RandomUtil.getRandomNum(2, 12);

        System.out.println("Rolling dice until I get " + target + ".");
        do
        {
            int dice1 = RandomUtil.getRandomNum(1, 6);
            int dice2 = RandomUtil.getRandomNum(1, 6);
            sum = dice1 + dice2;
            System.out.println(dice1 + " + " + dice2 + " = " + sum);
            rolls++;
        }
        while (sum != target);
        System.out.println("It took " + rolls + " rolls.");
    }
}
```

Terminology

- **Library:** A module whose methods are primarily intended for use by many other programs
- **Client:** Program that calls a library
- **API:**
 - **A**pplication **P**rogramming **I**nterface
 - Contract between client and implementation
- **Implementation:** Program that implements the methods in an API

client

```
Gaussian.Phi(1019)
```

calls methods

API

```
public class Gaussian  
double phi(double x)     $\phi(x)$   
double Phi(double z)    $\Phi(z)$ 
```

*defines signatures
and describes methods*

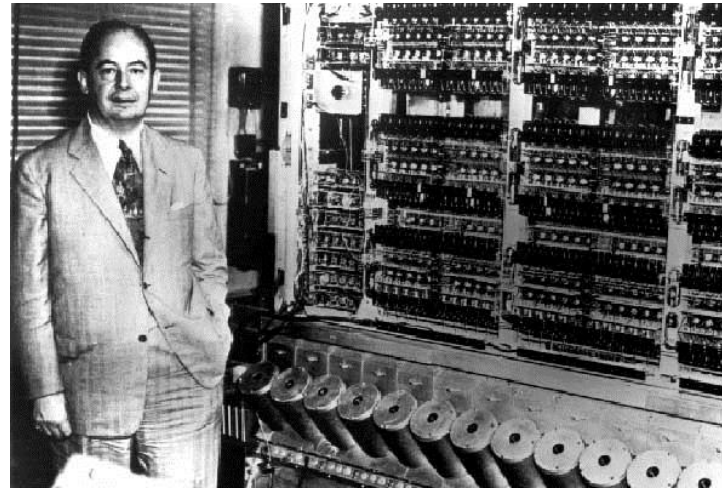
implementation

```
public class Gaussian  
public static double phi(double x)  
  
public static double Phi(double z)
```

*Java code that
implements methods* 7

Random numbers

“ The generation of random numbers is far too important to leave to chance. Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin. ”



Jon von Neumann (left), ENIAC (right)

What's this fuss about *true* randomness?

Perhaps you have wondered how predictable machines like computers can generate randomness. In reality, most random numbers used in computer programs are *pseudo-random*, which means they are generated in a predictable fashion using a mathematical formula. This is fine for many purposes, but it may not be random in the way you expect if you're used to dice rolls and lottery drawings.

RANDOM.ORG offers *true* random numbers to anyone on the Internet. The randomness comes from atmospheric noise, which for many purposes is better than the pseudo-random number algorithms typically used in computer programs. People use RANDOM.ORG for holding drawings, lotteries and sweepstakes, to drive games and gambling sites, for scientific applications and for art and music. The service has existed since 1998 and was built and is being operated by Mads Haahr of the School of Computer Science and Statistics at Trinity College, Dublin in Ireland.

As of today, RANDOM.ORG has generated 1.18 trillion random bits for the Internet community.

True Random Number Generator

Min:

Max:

Result:

50

Powered by [RANDOM.ORG](#)

Coin Flipper

You flipped 3 coins of type American Voting Coin 2004:



Timestamp: 2012-10-06 16:50:22 UTC

"QRBG121 is a fast non-deterministic random bit (number) generator whose randomness relies on intrinsic randomness of the quantum physical process of photonic emission in semiconductors and subsequent detection by photoelectric effect. In this process photons are detected at random, one by one independently of each other. Timing information of detected photons is used to generate random binary digits - bits. The unique feature of this method is that it uses only one photon detector to produce both zeros and ones which results in a very small bias and high immunity to components variation and aging. Furthermore, detection of individual photons is made by a photomultiplier (PMT). Compared to solid state photon detectors the PMT's have drastically superior signal to noise performance and much lower probability of appearing of afterpulses which could be a source of unwanted correlations."



2700€

<http://qrbg.irb.hr>

Standard Random

- Standard random
 - Library to generate pseudo-random numbers
 - Application Programming Interface (API):

```
public class StdRandom
```

<code>int uniform(int N)</code>	<i>integer between 0 and N-1</i>
<code>double uniform(double lo, double hi)</code>	<i>real between lo and hi</i>
<code>boolean bernoulli(double p)</code>	<i>true with probability p</i>
<code>double gaussian()</code>	<i>normal, mean 0, standard deviation 1</i>
<code>double gaussian(double m, double s)</code>	<i>normal, mean m, standard deviation s</i>
<code>int discrete(double[] a)</code>	<i>i with probability a[i]</i>
<code>void shuffle(double[] a)</code>	<i>randomly shuffle the array a[]</i>

Standard random, implementation

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

    // between 0 and N-1
    public static int uniform(int N)
    {
        return (int) (Math.random() * N);
    }

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

    // gaussian with mean = 0, stddev = 1
    public static double gaussian()
        /* see Exercise 1.2.27 */

    // gaussian with given mean and stddev
    public static double gaussian(double mean, double stddev)
    {
        return mean + (stddev * gaussian());
    }
    ...
}
```

Unit testing

- Include `main()` method to test each library

```
public static void main(String[] args)
{
    int N = Integer.parseInt(args[0]);
    if (args.length == 2) StdRandom.setSeed(Long.parseLong(args[1]));
    double[] t = { .5, .3, .1, .1 };

    System.out.println("seed = " + StdRandom.getSeed());
    for (int i = 0; i < N; i++)
    {
        System.out.printf("%2d " , uniform(100));
        System.out.printf("%8.5f " , uniform(10.0, 99.0));
        System.out.printf("%5b " , bernoulli(.5));
        System.out.printf("%7.5f " , gaussian(9.0, .2));
        System.out.printf("%2d " , discrete(t));
        System.out.println();
    }
}
```

These "random" numbers are the same as the first run!

```
% java StdRandom 5
seed = 1349544048443
72 34.23045 false 8.86067
10 47.24745 true 8.83698
65 35.25313 true 9.28941
17 34.37725 false 9.56543
52 90.80849 false 8.84883
```

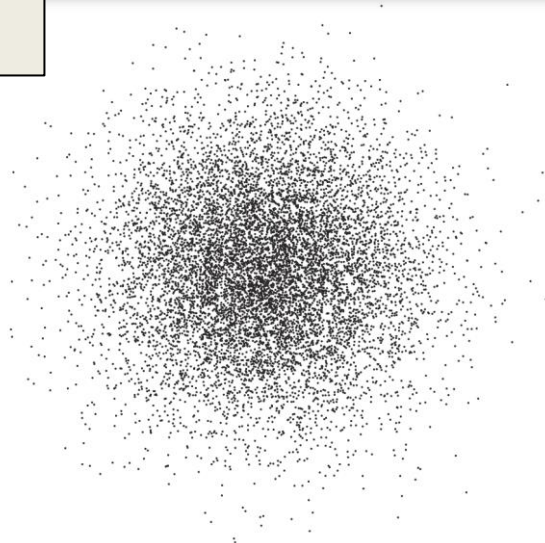
```
% java StdRandom 5
seed = 1349544178256
18 23.68569 false 8.85914
71 70.97195 false 8.71287
17 93.72297 false 9.14421
24 93.54278 false 9.48963
41 52.23556 false 9.10782
```

```
% java StdRandom 5 1349544048443
seed = 1349544048443
72 34.23045 false 8.86067
10 47.24745 true 8.83698
65 35.25313 true 9.28941
17 34.37725 false 9.56543
52 90.80849 false 8.84883
```

Using a library

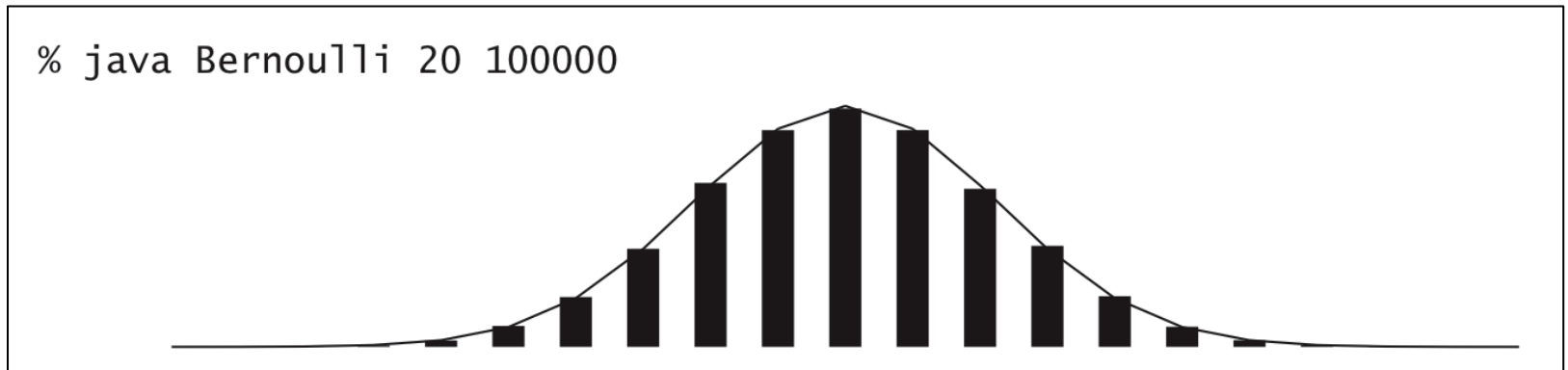
```
public class RandomPoints
{
    public static void main(String args[])
    {
        int N = Integer.parseInt(args[0]);
        for (int i = 0; i < N; i++)
        {
            double x = StdRandom.gaussian(0.5, 0.2);
            double y = StdRandom.gaussian(0.5, 0.2);
            StdDraw.point(x,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

Use library name to
invoke the method



Modular Programming

- **Modular programming**
 - Divide program into self-contained pieces.
 - Test each piece individually.
 - Combine pieces to make program.
- **Example: Flip N coins. How many heads?**
 - Read arguments from user.
 - Flip one fair coin.
 - Flip N fair coins and count number of heads.
 - Repeat simulation, counting number of times each outcome occurs.
 - Plot histogram of empirical results.
 - Compare with theoretical predictions.



Bernoulli Trials

```
public class Bernoulli {
```

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

flip **N** fair coins;
return # heads

```
    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 i = 0; i < T; i++)  
            freq[binomial(N)]++;
```

perform **T** trials
of **N** coin flips each

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

plot histogram
of number of heads

```
        double mean = N / 2.0, 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);
```

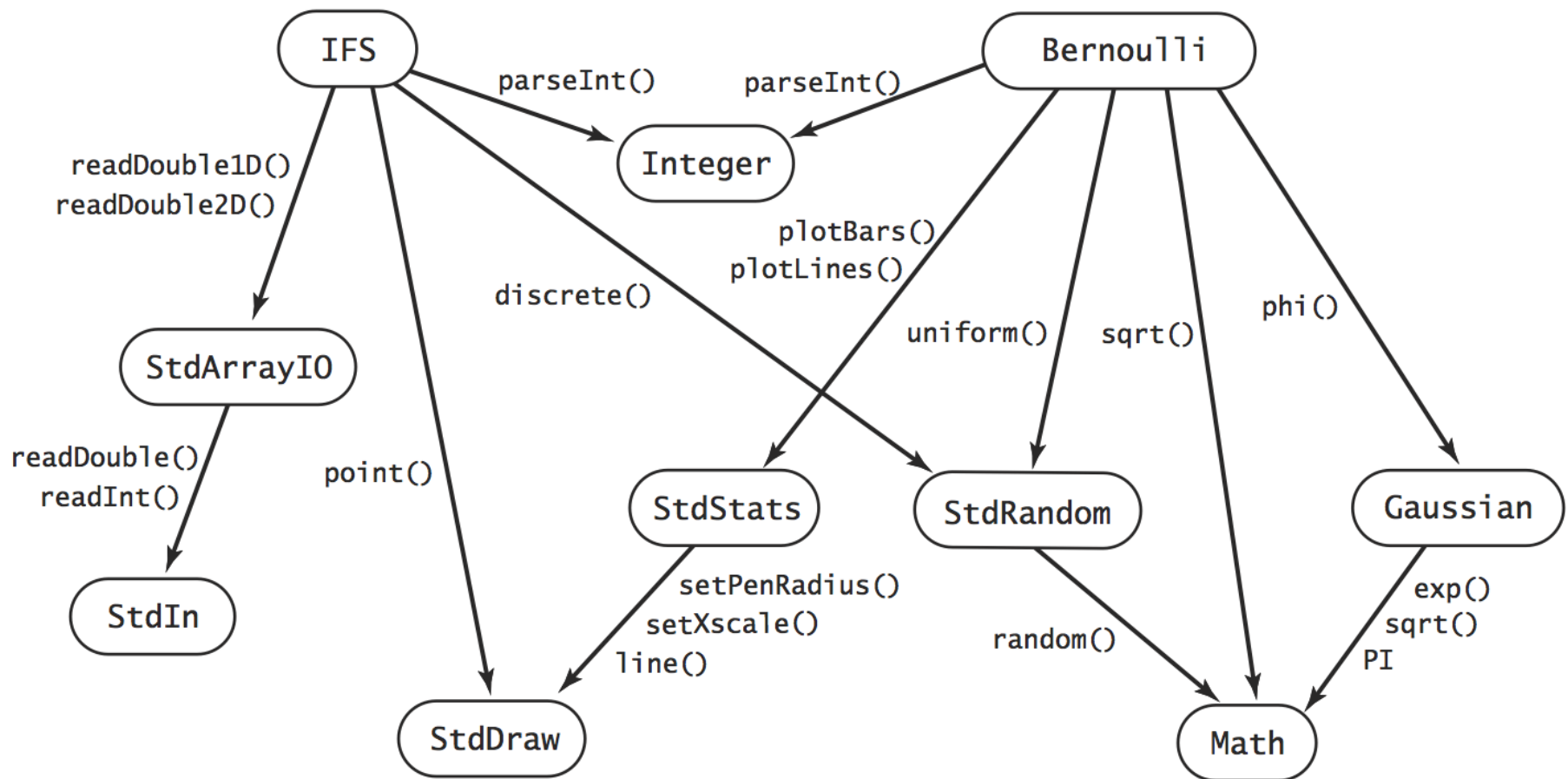
theoretical prediction

```
    }  
}
```


Dependency Graph

- **Modular programming**

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



Pretty text formatting

- **printf-style formatting**
 - Common way to nicely format output
 - Present in many programming languages
 - Java, C++, Perl, PHP, ...
 - Use a special format language:
 - Format string with special codes
 - One or more variables get filled in
- **In Java, used via:**
 - `System.out.printf()` – output to standard out
 - `String.format()` – returns a formatted String

Floating-point formatting

```
double d = 0.123456789;  
float f = 0.123456789f;  
  
// %f code is used with double or float variables  
// %f defaults to rounding to 6 decimal places  
System.out.printf("d is about %f\n", d);  
System.out.printf("f is about %f\n", f);
```

\n means line feed

```
d is about 0.123457  
f is about 0.123457
```

```
// Number of decimal places specified by .X  
// Output is rounded to that number of places  
System.out.printf("PI is about %.1f\n", Math.PI);  
System.out.printf("PI is about %.2f\n", Math.PI);  
System.out.printf("PI is about %.3f\n", Math.PI);  
System.out.printf("PI is about %.4f\n", Math.PI);
```

```
PI is about 3.1  
PI is about 3.14  
PI is about 3.142  
PI is about 3.1416
```

```
// %e code outputs in scientific notation  
// .X specifies number of significant figures  
final double C = 299792458.0;  
System.out.printf("speed of light = %e\n", C);  
System.out.printf("speed of light = %.3e\n", C);
```

```
C = 2.99792e+08  
C = 2.998e+08
```

Integer formatting

```
// %d code is for integer values, int or long
// Multiple % codes can be used in a single printf()
long power = 1;
for (int i = 0; i < 8; i++)
{
    System.out.printf("%d = 2^%d\n", power, i);
    power = power * 2;
}
```

You can have multiple % codes that are filled in by a list of parameters to printf()

```
1 = 2^0
2 = 2^1
4 = 2^2
8 = 2^3
16 = 2^4
32 = 2^5
64 = 2^6
128 = 2^7
```

```
// A number after the % indicates the minimum width
// Good for making a nice looking tables of values
power = 1;
for (int i = 0; i < 8; i++)
{
    System.out.printf("%5d = 2^%d\n", power, i);
    power = power * 2;
}
```

Minimum width of this field in the output. Java will pad with whitespace to reach this width (but can exceed this width if necessary).

```
1 = 2^0
2 = 2^1
4 = 2^2
8 = 2^3
16 = 2^4
32 = 2^5
64 = 2^6
128 = 2^7
```

Flags

```
// Same table, but left justify the first field
power = 1;
for (int i = 0; i < 8; i++)
{
    System.out.printf("%-5d = 2^%d\n", power, i);
    power = power * 2;
}
```

- flag causes this field to be left justified

```
// Use commas when displaying numbers
power = 1;
for (int i = 0; i < 17; i++)
{
    System.out.printf("%,5d = 2^%d\n", power, i);
    power = power * 2;
}
```

, flag causes commas between groups of 3 digits

1	= 2^0
2	= 2^1
4	= 2^2
8	= 2^3
16	= 2^4
32	= 2^5
64	= 2^6
128	= 2^7

1	= 2^0
2	= 2^1
4	= 2^2
8	= 2^3
16	= 2^4
32	= 2^5
64	= 2^6
128	= 2^7
256	= 2^8
512	= 2^9
1,024	= 2^10
2,048	= 2^11
4,096	= 2^12
8,192	= 2^13
16,384	= 2^14
32,768	= 2^15
65,536	= 2^16

Text formatting

```
// Characters can be output with %c, strings using %s
String name = "Bill";
char grade = 'B';
System.out.printf("%s got a %c in the class.\n", name, grade);
```

Bill got a B in the class.

```
// This prints the same thing without using printf
System.out.println(name + " got a " + grade + " in the class.");
```

An equivalent way to print the same thing out using good old println().

Creating formatted strings

- **Formatted String creation**

- You don't always want to immediately print formatted text to standard output
- Save in a String variable for later use

```
// Formatted Strings can be created using format()
String lines = "";
for (int i = 0; i < 4; i++)
    lines += String.format("Random number %d = %.2f\n", i, Math.random());
System.out.print(lines);
```

```
Random number 0 = 0.54
Random number 1 = 0.50
Random number 2 = 0.39
Random number 3 = 0.64
```

The format specifier

Minimum number of character used, but if number is longer it won't get cut off

`% [flags][width][.precision]type`

Special formatting options like inserting commas, making left justified, etc.

Sets the number of decimal places, don't forget the .

Type is the only required part of specifier. "d" for an integer, "f" for a floating-point number

`%[flags][width][.precision]type`

```
System.out.printf("%,6.1f", 42.0);
```


printf gone wild

- Format string specifies:
 - Number of variables to fill in
 - Type of those variables
- With great power comes great responsibility
 - Format must agree with #/types of arguments
 - Runtime error otherwise
 - Compiler / Eclipse won't protect you

```
// Runtime error %f expects a floating-point argument
System.out.printf("crash %f\n", 1);

// Runtime error, %d expects an integer argument
System.out.printf("crash %d\n", 1.23);

// Runtime error, not enough arguments
System.out.printf("crash %d %d\n", 2);
```

printf puzzler

Code	Letter
System.out.printf("%f", 4242.00);	E
System.out.printf("%d", 4242);	A
System.out.printf("%.2f", 4242.0);	B
System.out.printf("%.3e", (double) 4242);	C
System.out.printf("%,d", 4242);	D

Letter	Output
A	4242
B	4242.00
C	4.242e+03
D	4,242
E	4242.000000

Code	#
System.out.printf("%d%d", 42, 42);	2
System.out.printf("%d+%d", 42, 42);	1
System.out.printf("%d %d", 42);	5
System.out.printf("%8d", 42);	3
System.out.printf("%-8d", 42);	4
System.out.printf("%d", 42.0);	5

#	Output
1	42+42
2	4242
3	42
4	42
5	runtime error

Summary

- Libraries

- Collection of useful **methods** for use by **clients**
- Implements a specific **API**
- Why?
 - Makes code easier to understand
 - Makes code easier to debug
 - Makes code easier to maintain and improve
 - Makes code easier to reuse

- Formatting output

- **printf()** method
- First parameter: specifies **output format using % codes**
- Variable number of other parameters
 - Must match number and types of "slots" in the format specifier