
CS 211
Command-Line Interface
JavaDoc
Number Representations

Command-Line Arguments

Command-Line Interface

Q: Other than typing in input, how can we feed values to our compiled program from "the outside"?

Q: What is that `String[]args` for in main, anyways?

A: Command-line arguments!

CLI – Example

```
public class TestCLI {
    public static void main (String[] args) {
        System.out.println("String[] args contained:");
        for (int i=0; i<args.length; i++) {
            System.out.println("#"+i+":"+args[i]);
        }
    }
}
```

```
demo$ java TestCLI a b c
String[] args contained:
#0:a
#1:b
#2:c
demo$ java TestCLI
String[] args contained:
demo$
```

```
demo$ java TestCLI a 'b c' "d e" f
String[] args contained:
#0:a
#1:b c
#2:d e
#3:f
demo$ java TestCLI
String[] args contained:
demo$
```

CLI - Details

Only String values are possible. (String[] args)

Spaces separate values.

- use single or double quotes to provide a single string value that contains spaces.

```
demo$ java Test one "two words" three "4 4 4"  'f i v e'
```

Getting other types: call conversion methods.

- Integer.java: `public static int parseInt(String s) {…}`
- Double.java: `public static double parseDouble(String s) {…}`
- (others, too)

Practice Problems

Write a program that accepts command line arguments. If there were not exactly three arguments (which we will assume are double values), then print "invalid usage" and quit. If there were three, print "largest value of the three: ", and the actual largest value out of the three doubles that were passed in.

Write a program that accepts an arbitrary number of integers on the command line; print out the sum, average, and maximum of those numbers.

If you want to run these programs with different inputs, do you have to recompile between each run? Why or why not?

Three Versions of Input

We have three distinct ways we can get input for our program:

1. `System.in`: usually keyboard (but we can use < to pipe input from other places, like files)
2. **Command-line arguments**.
3. **reading files** directly (e.g., via Scanner)

You can also use any combination in a single program as desired.

JavaDoc

JavaDoc: Overview

Detailed description of code in a format that can generate spiffy API documentation, likely as HTML (just like what generated our API readings all semester long!)

We can document **individual classes and their members** (directly in source code) **or entire packages** (through extra files).

JavaDoc Example: methods

```
/**
 * Short sentence summarizing: Returns a boolean
 * representing if the single input is prime.
 *
 * @param n    the number to be tested if prime.
 * @return     boolean for prime or not.
*/
public boolean isPrime (int n){
        //boring non-javadoc comment…
        for (int i=2; i<n;i++){
                if (n%i==0){ return false; }
        }
        return true;
}
```

JavaDoc Example: classes

```
/**
 * the Student class represents an individual
 * within the GMU community; each student has a
 * name, age, and studentID.
 *
 * @author George Mason
 * @version 0.10  April 1776
 */
public class Student {
 …
```

JavaDoc Example: packages

file: Foo/Bar/package-info.java

```
/**
 * the bar package provide various foo
 * interactions.
*/
package Foo.Bar;
//empty ever after
```

JavaDoc Notes

- We can write actual HTML in these comments.
- links can be created to other classes' documentation via `{@link ClassName}`
- There are various @tags that can be created:

@param	@return	@exception
@author	@version	@see
@since	@serial	@deprecated …

Generating Documentation: javadoc

```
javadoc -d /Users/me/htmlDir -subpackages Foo
```

-subpackages packageName - please create documentation for this package and recursively through all subpackages as well. (convenient!)

-d some/Location/ - please put the (many) generated html files at some/Location that I've given you.

-classpath what:ever – regardless of where I am, please use this classpath.

-sourcepath some/Location – where can Javadoc look for files? Defaults to the classpath (convenient!)

More options: javadoc

-public, -protected, -package, -private – please show anything as visible as what I've mentioned (use **-private** to show all; **-protected** is the default)

-exclude these:packages - please don't include these packages or their subpackages, even though a **–subpackages** option refers to them.

- many more… perhaps you will learn on-the-job about them ☺ For now, we have enough to keep us occupied!
- Our goal is to be aware of javadoc, not to memorize details.

Links Abound!

Much more info:

http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html

Options for the `javadoc` command:

http://www.java2s.com/Tutorial/Java/0020__Language/ThejavadocTool.htm

http://docs.oracle.com/javase/7/docs/technotes/tools/windows/javadoc.html#runningjavadoc

Doclets can be written to generate other outputs (such as pdfs, personalized/better(?) HTML, etc)

Number Representations

Number Representations

Different bases of interest:

→ 10, 2, 16

Counting up in various bases

Conversions between bases

Representations of Numbers

There is only one set of integers, no matter what you call them.

$5 = V = 101_2 =$ 

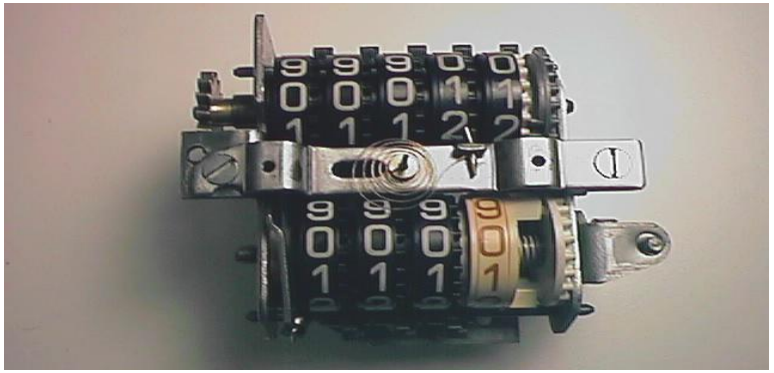
Each base gives a different perspective on how to name these numeric values.

Representations of Numbers

- **Decimal**
  0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,…
- **Binary**
  0,1,10,11,100,101,110,111,1000,1001,1010,1011,1100,1101,1110,1111,10000,…
- **Hexadecimal**
  0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F,10,11,12,13,14,15,16,17,…
- **Roman**
  I,II,III,IV,V,VI,VII,VIII,IX,X,XI,XII,XIII,XIV,XV,XVI,XVII,XVIII,XIX,XX,XXI,...
- **Tallies**: |, ||, |||, ||||, 𝍸, 𝍸 |, 𝍸 ||, 𝍸 |||, 𝍸 ||||, 𝍸 𝍸, …

Different Bases

A base defines how many symbols we have for numbers, and also defines the value of each column.



Those symbols are used "odometer-style" to count upwards.

Different Bases

We count in **decimal** (base 10),
thanks to our ten fingers.

- Base 10: ten symbols: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

Computers store everything in **binary** (base 2),
thanks to the simplicity of having just 2 states.

- Base 2: two symbols: 0, 1.

Different Bases

We use **hexadecimal** (base 16) as a convenient shorthand for longer binary numbers (more on that convenience soon!).

→ Base 16: 16 symbols:
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.

Base N: use any N symbols. Example:
0,1,…, (n-1).

Counting Up

We count up though the symbols of our base, and when we run out, we 'clock over' to the next column, starting over in the earlier columns.

It just happens a lot faster in binary…

Notice that binary clocks over into 4 new columns exactly when hexadecimal clocks over once. Not a coincidence!

reason: $16 = 2^4$.

- Think of counting in each base as using an odometer that has a different number of symbols on its wheels.

Base 10:	Base 2:	Base 16:	Base 8:
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	8	10
9	1001	9	11
10	1010	A	12
11	1011	B	13
12	1100	C	14
13	1101	D	15
14	1110	E	16
15	1111	F	17
16	10000	10	20
17	10001	11	21
18	10010	12	22
19	10011	13	23
…	…	…	…

Column Values

Each column of a number contains one of the digits in the base, and is worth the base raised to the column's number:

Consider 1437, in *decimal* (written 1437_{10}):

Digits:	1	4	3	7
Values:	10^3	10^2	10^1	10^0

total value (written in base ten):

$= (1*10^3) + (4*10^2) + (3*10^1) + (7*10^0)$

$= 1000+400+30+7$

$= 1437$

Column Values

Consider 11010 in binary:

Digits (in base 2): 1 1 0 1 0

Values (in base 10): 2^4 2^3 2^2 2^1 2^0

Total value: (written in base 10):

$= (1*2^4)+(1*2^3)+(0*2^2)+(1*2^1)+(0*2^0)$

$= 16 + 8 + 0 + 2 + 0$

$= 26$

Base 10:	Base 2:	Base 16:	Base 8:
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	8	10
9	1001	9	11
10	1010	A	12
11	1011	B	13
12	1100	C	14
13	1101	D	15
14	1110	E	16
15	1111	F	17
16	10000	10	20
17	10001	11	21
18	10010	12	22
19	10011	13	23
…	…	…	…

Column Values

Consider 2AC in hexadecimal:

Digits (in base 16): 2 A C

Values (in base 10): 16^2 16^1 16^0

Total value: (written in base 10):

= (2*16^2) + (10*16^1) + (12*16^0)

= 2*256 + 10*16 + 12*1

= 512 + 160 + 12

= 684

Base 10:	Base 2:	Base 16:	Base 8:
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	8	10
9	1001	9	11
10	1010	A	12
11	1011	B	13
12	1100	C	14
13	1101	D	15
14	1110	E	16
15	1111	F	17
16	10000	10	20
17	10001	11	21
18	10010	12	22
19	10011	13	23
...	...	...	...

Conversions: 2 → 10, 16 → 10

The previous explanations of column values actually instructed you how to convert from a different base to base 10:

- for each column, **multiply the column's value** (in base 10) **by the column's contents** (in base 10, based on that chart).
- add up these products from each column to get the answer (all work in base 10 now).

→ This actually works for any base to any base! But performing the multiplication and addition in non-base-10 bases can be tricky, so we use a different approach converting out of base 10.

Practice Problems

Convert the following numbers from binary to decimal:

110_2
10101_2
1110_2
100011_2

Convert the following numbers from hexadecimal to decimal:

F_{16}
$C5_{16}$
$3B2_{16}$
100_{16}

Base 10:	Base 2:	Base 16:	Base 8:
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	8	10
9	1001	9	11
10	1010	A	12
11	1011	B	13
12	1100	C	14
13	1101	D	15
14	1110	E	16
15	1111	F	17
16	10000	10	20
17	10001	11	21
18	10010	12	22
19	10011	13	23
…	…	…	…

Conversions: 10 → 2

A binary number either has, (1), or does not have, (0), the value of each column. Our goal is to figure out which columns should have 1's in them.

1. Start in the largest column that isn't larger than your number.
2. Add a 1 to the column if it's ≤ your number; subtract the column's value from the number to track how much value is left.
3. Consider the next column to the right. Again, if it's ≤ the remaining value, add a 1 in the column and subtract the column's value.
4. Keep repeating until you have considered the rightmost column.

→ if you have any value left over, your calculations are wrong.

Example: Convert 26_{10} to binary

Which column do we start with? 32 > **26** > 16.
→ all columns 32 and bigger have implicit 0's in them.

(o)	(o)	?	?	?	?	?
64	32	16	8	4	2	1

16 ≤26. Put a 1 in the 16 column, subtract 16 (26-16 = **10 left**).

0	0	**1**				
64	32	16	8	4	2	1

8≤10. (Remaining: 10-8 = **2 left**).

0	0	1	**1**			
64	32	16	8	4	2	1

4 isn't ≤2; column is **0**.
(Remaining: 2-0=2 left).

0	0	1	1	**0**		
64	32	16	8	4	2	1

- 2≤2. (Remaining: 2-2 = 0).

0	0	1	1	0	**1**	
64	32	16	8	4	2	1

- 1 isn't ≤0. (Remaining: 0-0=0).

0	0	1	1	0	1	**0**
64	32	16	8	4	2	1

- No more columns, remaining value=0: we probably did this correctly.

→ **Result: 11010_2**

Example: Convert 421_{10} to hexadecimal

Columns:
… $16^3=4096$
$16^2=256$
$16^1=16$
$16^0=1$

Which column do we start with? 4096 > 421 ≥ 256.
→ all columns 4096 and bigger have implicit 0's in them.

(0)	(0)			
…	4096	256	16	1

256 ≤421. **How many 256's fit** in 421? Just **one**. Put a **1** in the 256 column, subtract (Remaining: 421 - 256*1 = 165).

0	0	1		
…	4096	256	16	1

16≤165. **How many 16's fit** into 165? **Ten**. Put an **A** in the 16 column.(Remaining: 165-10*16 = 165 – 160 = 5).

0	0	1	A	
…	4096	256	16	1

1≤5. How **many 1's fit**? **Five**. Put a **5** in the 1 column. (Remaining: 5-5*1=0).

0	0	1	A	5
…	4096	256	16	1

→ **Result: $1A5_{16}$**

Practice Problems

Convert the following numbers from decimal to binary:

5
24
127
1000

Convert the following numbers from decimal to hexadecimal:

20
170
4100
5

Base 10:	Base 2:	Base 16:	Base 8:
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	8	10
9	1001	9	11
10	1010	A	12
11	1011	B	13
12	1100	C	14
13	1101	D	15
14	1110	E	16
15	1111	F	17
16	10000	10	20
17	10001	11	21
18	10010	12	22
19	10011	13	23
…	…	…	…

Conversions: 2 → 16 → 2

We could convert from 2→10→16, but there is a faster way.

Group every 4 bits of the binary number (starting at the **right**)

Look up each 4-bit pattern in our original chart of counting up.

Works in reverse, too.

Example: 10 1100 0000 1010 0101:
2 C 0 A 5

Example: 23ACE in binary:
0010 0011 1010 1100 1110

Base2:	Base16:
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

extra 0's shown for padding: each pattern must be 4 bits