

# CS312 Course Introduction

"Computers are good at following instructions, but not at reading your mind."

-Donald Knuth, *Tex* p. 9

Mike Scott, Gates 6.304  
scottm@cs.utexas.edu  
[www.cs.utexas.edu/~scottm/cs312](http://www.cs.utexas.edu/~scottm/cs312)



# Who Am I

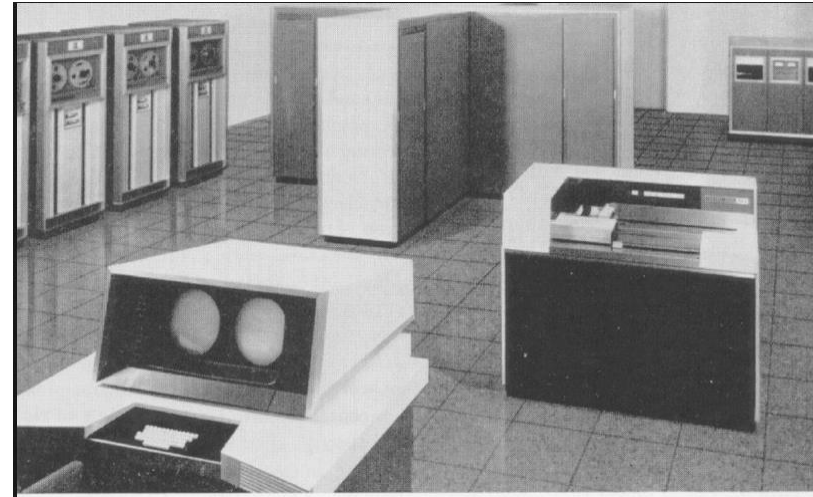
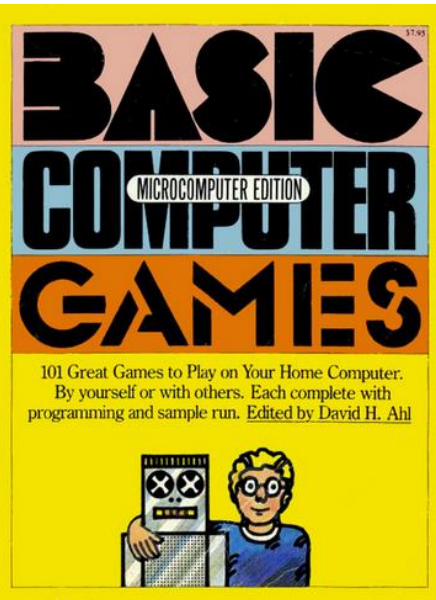
- ▶ Lecturer in CS department since 2000
- ▶ Undergrad Stanford, MSCS RPI
- ▶ US Navy for 8 years, submarines
- ▶ 2 years Round Rock High School



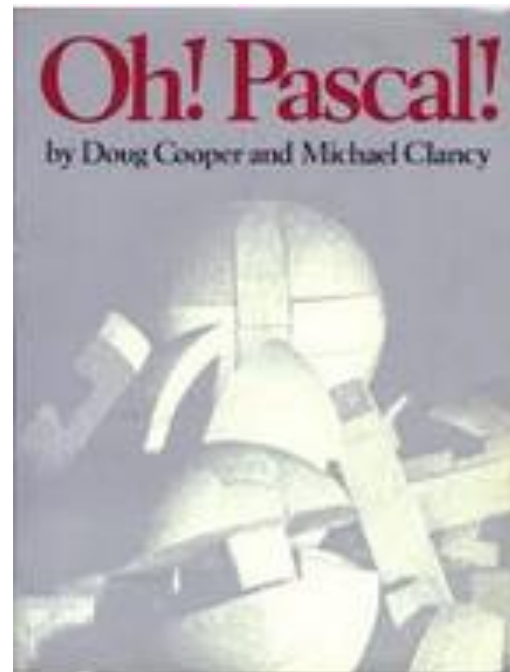
Rensselaer



# My Path to CS



```
10 INPUT "What is your name: "; U$
20 PRINT "Hello "; U$
25 REM
30 INPUT "How many stars do you want: "; N
35 S$ = ""
40 FOR I = 1 TO N
50 S$ = S$ + "*"
55 NEXT I
60 PRINT S$
65 REM
70 INPUT "Do you want more stars? "; A$
80 IF LEN(A$) = 0 THEN GOTO 70
90 A$ = LEFT$(A$, 1)
100 IF (A$ = "Y") OR (A$ = "y") THEN GOTO 30
110 PRINT "Goodbye ";
120 FOR I = 1 TO 200
130 PRINT U$; " ";
140 NEXT I
150 PRINT
```



# What We Will Do Today

- ▶ Introductions and administrative details
- ▶ Start Java Basics



# Intro to Programming

- Learn to design and implement computer programs to solve problems.

- |                                 |                           |                             |
|---------------------------------|---------------------------|-----------------------------|
| 1. course Intro                 | 12. cumulative algorithms | 24. sorting, searching      |
| 2. basic Java                   | 13. Strings               | 25. more array algos        |
| 3. static methods               | 14. while loops           | 26. 2d arrays               |
| 4. expressions & variables      | 15. random numbers        | 27. classes and objects     |
| 5. for loops                    | 16. Boolean logic         | 28. methods                 |
| 6. more loops, constants        | 17. assertions            | 29. constructors            |
| 7. parameters                   | 18. file input 1          | 30. creating classes, Enums |
| 8. 2d graphics                  | 19. file input 2          | 31. inheritance             |
| 9. more graphics                | 20. file input 3          | 32. polymorphism            |
| 10. return values, Math methods | 21. arrays                | 33. ArrayList               |
| 11. conditional statements      | 22. more arrays           | 34. recursion               |
|                                 | 23. tallying algos        |                             |

# Programing and CS

- ▶ A tool for doing the cool stuff in CS
- ▶ You can't create a self driving vehicle without the software to control the vehicle



# The Parable of Aaron D.

- ▶ I assume no prior programming experience.
- ▶ You are limited to what you can use on assignment to what we have covered in the book.
- ▶ I will defer questions that are well past what we are currently covering.
- ▶ Programming is not a spectator sport.
  - The only way to learn to program is to program
- ▶ Aaron D. and the 500 problems.



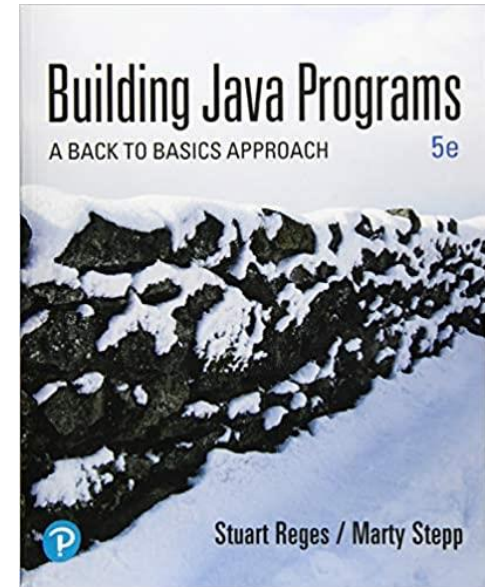
# Startup

- ▶ If you have not already done so ...
- ▶ ... complete the items on the class start-up page
- ▶ <http://www.cs.utexas.edu/~scottm/cs312/handouts/startup.htm>



# Books and software

- book is required - we follow it quite closely
- Chrome and Proctorio extension for exams
- Java for programming
- IDE for programming
- Canvas for turning in assignments, grades, UT Instapoll, section problems, exams



# Clicker 1

Which of these best describes you?

- A. first year at UT and first year college student
- B. first year at UT, transferring from another college or university
- C. in second year at UT
- D. in third year at UT
- E. other

# Graded Course Components

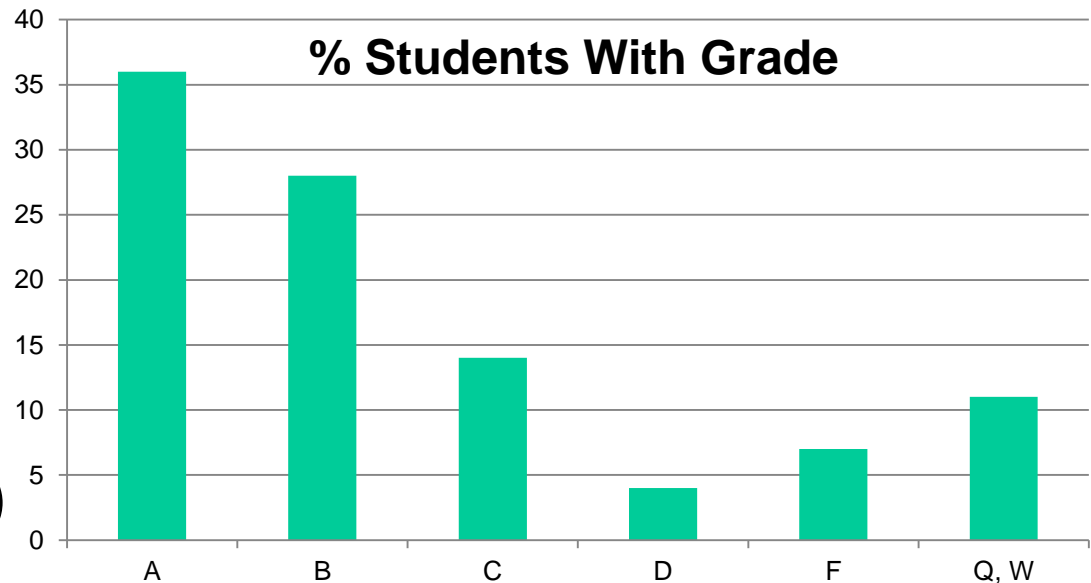
- ▶ UT Instapoll (In class questions)
  - 42 lectures with UT Instapoll, 7 dropped, **35 points**
- ▶ Discussion section problems (Go to your section.  
Do not refer to the course id from Canvas, same for all.)
  - 10 problems, 3 points each, 2 dropped, **24 points**
- ▶ Programming projects
  - 12 projects, 1<sup>st</sup> 10, rest 20 points each: **210 points total**  
(lowest grade of assignments 2 - 12 dropped)
- ▶ Exams: Outside of class
  - Exam 1, Wednesday, 9/30, approx. 6:45 – 9:15 pm, **150 points**
  - Exam 2, Wednesday, 11/11, approx. 6:45 – 9:15 pm, **250 points**
  - Exam 3, Date and time TBD (during finals week), **340 points**

$$35 + 24 + 210 + 150 + 250 + 340 + 5 \text{ (quiz)} = 1014$$

- ▶ clicker, Quizzes, Programming Assignments capped at 260 pts
  - 14 points of “slack” among those 3 components
  - Extra Credit: Computing background survey +3 points,  
practice exam + 3 points, eCIS & TA Survey +6 points

# Grades and Performance

- ▶ No points added! Grades based on 1000 points, not 1014
- ▶ Final grade determined by final point total and a 900 – 800 – 700 – 600 scale
  - plusses and minuses if within 25 points of cutoff:  
875 – 899: B+, 900 – 924: A-
- ▶ historically my CS312 classes (~2000 Students)
- ▶ **80% C- or higher:**
  - 38% A's,**
  - 28% B's**
  - 12% C's**
- ▶ **10% D or F**
- ▶ **10% Q or W (drop)**



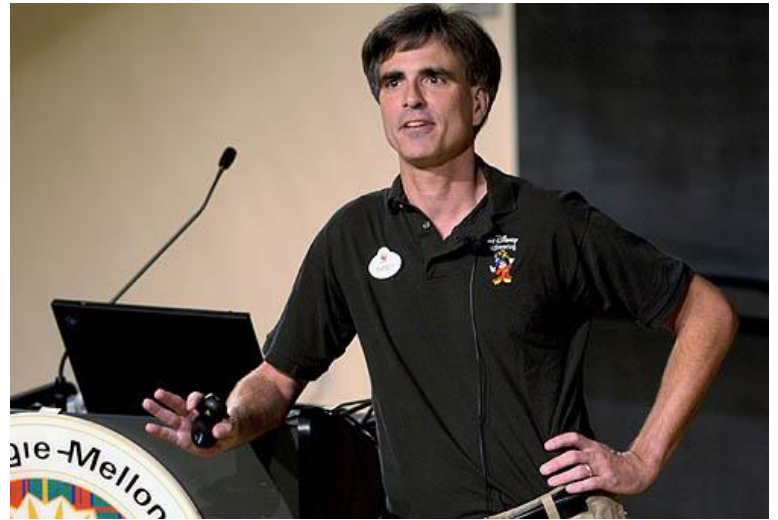


# Assignments

- ▶ Start out easy but get **much, much** harder
- ▶ Individual – do your own work
- ▶ Programs checked automatically with plagiarism detection software
- ▶ Turn in the right thing - correct name, correct format or you will lose points / slip days
- ▶ Slip days
  - 8 for term, max 2 per assignment
  - don't use frivolously

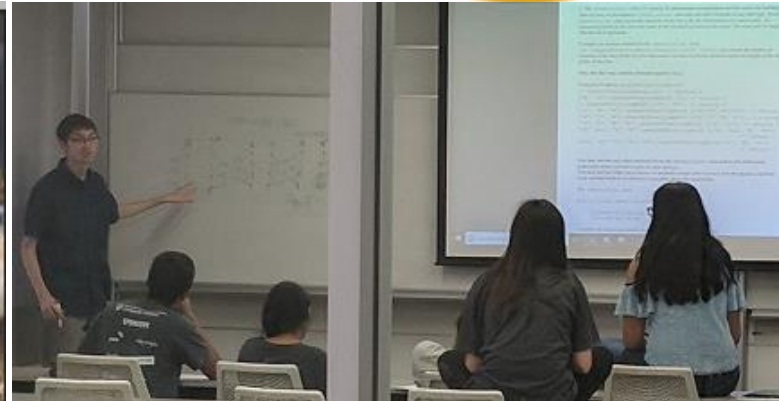
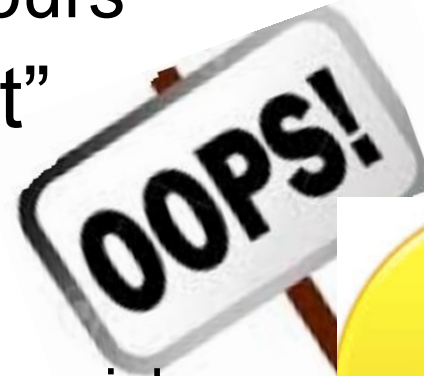
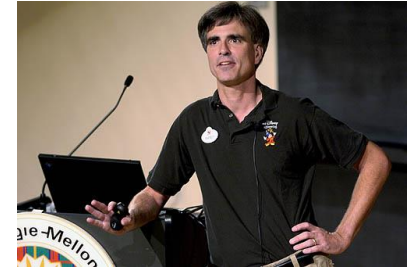
# Succeeding in the Course

- ▶ Randy Pausch, CS Professor at CMU said:
- ▶ *"When I got tenure a year early at Virginia, other Assistant Professors would come up to me and say, 'You got tenure early!?!?! What's your secret?!?!?' and I would tell them, 'Call me in my office at 10pm on Friday night and I'll tell you.' "*
- ▶ *"A lot of people want a shortcut. I find the best shortcut is the long way, which is basically two words: work hard."*



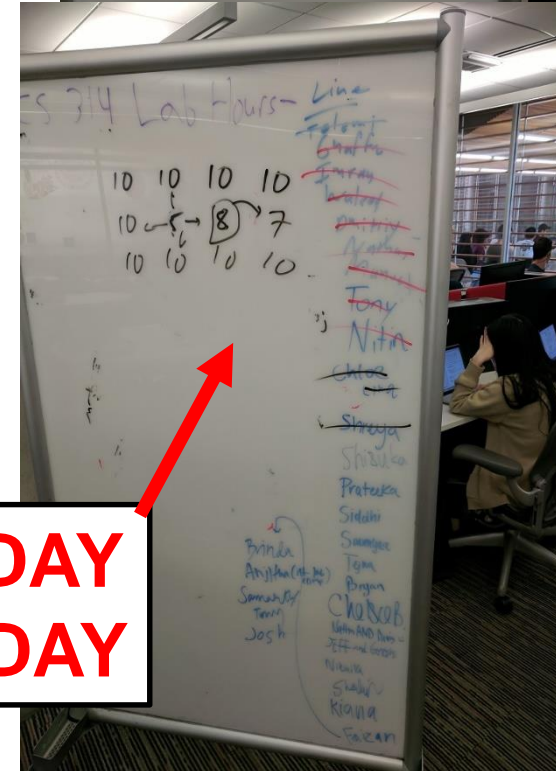
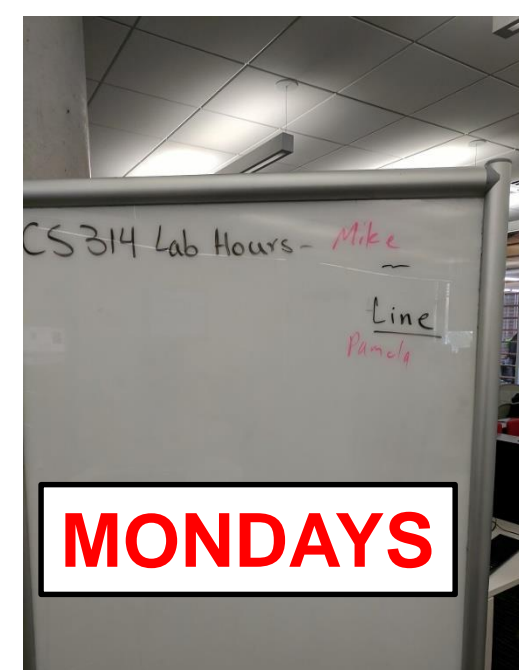
# Succeeding in the Course - Meta

- ▶ “Be the first penguin”
  - Ask questions!!!
  - lecture, section, Piazza, lab hours
- ▶ “It is impossible to be perfect”
  - Mistakes are okay.
  - That is how we learn.
  - Trying to be perfect means not taking risks.
  - no risks, no learning
- ▶ “Find a Pack”
  - Make friends.
  - Study with them!



# How to Get Help

- ▶ Piazza Post
- ▶ [Help Hours](#) via Zoom
- ▶ SI Sessions
- ▶ Email instructor or TAs
  - Prefer Piazza
- ▶ Class examples
- ▶ Examples from book
- ▶ Discuss with other students  
at a *high level*





# Succeeding in the Course - Concrete

- ▶ Whole course is cumulative!
- ▶ Material builds on itself
  - failure to understand a concept leads to bigger problems down the road, so ...
- ▶ do the readings
- ▶ start on assignments early
- ▶ get help from the teaching staff when you get stuck on an assignment
- ▶ attend lecture and discussion sections
- ▶ participate on the class discussion group
- ▶ **do extra problems (Practice It!**  
**<http://practiceit.cs.washington.edu/>**
- ▶ study for tests using the old tests
- ▶ study for tests in groups
- ▶ ask questions and get help when needed

# Succeeding in the Course

- ▶ Cannot succeed via memorization.
- ▶ The things I expect you to do are **not** rote.
- ▶ Learn by doing.
- ▶ If you are brand new to programming or have limited experience I **strongly** recommend you do ***lots and lots of practice problems.***
  - Practice It! web site
  - JavaBat

# Programming is like Legos...











# Legos and Programming

- ▶ With Legos and Programming you have a small number of primitives. (basic tools or pieces)
- ▶ But you build huge, elaborate structures out of those simple pieces.

# A Brief Look at Computer Science

- ▶ This class, like most first classes in Computer Science, focuses solving problems and implementing those solutions as computer programs.
  - you learn how to program
- ▶ ... and yet, computer science and computer programming are not the same thing!
- ▶ So what is Computer Science?

# What is Computer Science?

- ▶ Poorly named in the first place.
- ▶ It is not so much about the computer as it is about *Computation*.
- ▶ *“Computer Science is more the study of managing and processing information than it is the study of computers.”*
  - Owen Astrachan, Duke University
- ▶ learn to program
  - programming a key tool in later courses



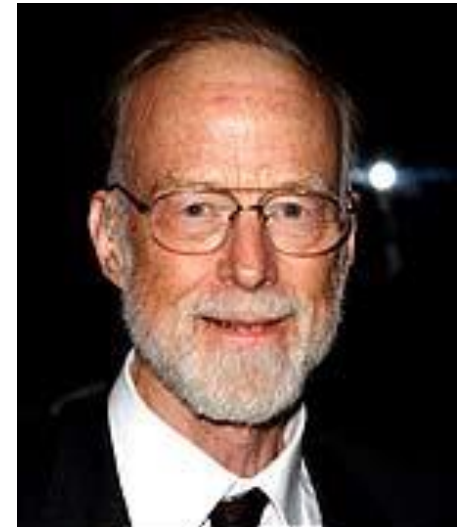
# Computer Programming and Computer Science

- ▶ Generally the first thing that is studied in Chemistry is stoichiometry.
  - Why? It is a skill necessary in order to study more advanced topics in Chemistry
- ▶ The same is true of problems solving / programming and computer science.

- ▶ “What is the linking thread which gathers these disparate branches into a single discipline? ...it is the art of programming a computer. It is the art of designing efficient and elegant methods of getting a computer to solve problems, theoretical or practical, small or large, simple or complex.”

- C. A. R. Hoare

- ▶ Sir Tony Hoare. Turing Award Winner. Inventor of the quicksort algorithm





- ▶ “Programming is unquestionably the central topic of computing.

In addition to being important, programming is an enormously exciting intellectual activity. In its purest form, it is the systematic mastery of complexity. For some problems, the complexity is akin to that associated with designing a fine mechanical watch, i.e., discovering the best way to assemble a relatively small number of pieces into a harmonious and efficient mechanism. For other problems, the complexity is more akin to that associated with putting a man on the moon, i.e, managing a massive amount of detail.

In addition to being important and intellectually challenging, programming is a great deal of fun. Programmers get to build things and see them work.. What could be more satisfying? “

- John V. Guttag, Professor at MIT  
research in AI, medical systems, wireless  
networking



# Computer Programming

- ▶ a skill and tool that are applied to all other areas of computer science
  - artificial intelligence, networks, cpu architecture, graphics, systems (programming languages, operating systems, compilers), security, and on and on ...
- ▶ We will be using solving problems and implementing solutions in a programming language called Java
- ▶ problem solving and computational thinking are key

# What do Computer Scientists do?

- ▶ Computer Scientists solve problems
  - creation of algorithms
- ▶ Some examples
  - you
  - Kurt Dresner, Intersection Control
  - Austin Villa, Robot Soccer
  - Doug and Steve, the TRIPS processor

# You!

- ▶ Encryption and Decryption
- ▶ Ever entered your credit card number to a website? game company?

HABUQXC 3

TLQ BJJ QABQ DCY. GXTTXQ, ALPXIXC, PZQA QAX BYYZYQBTHX LS AXC SZIX  
OBKEAQXCY, HLKJO BYN LT QAX YKGVXHQ, PBY YKSSZHZXTQ QL OCBP SCLD AXC  
AKYGBTO BTW YBQZYSBHQLCW OXYHCZUQZLT LS DC. GZTEJXW. QAXW BQQBHNXO AZD  
ZT IBCZLKY PBWY--PZQA GBCXSBHXXO FKXYQZLTY, ZTEXTZLKY YKUULYZQZLTY, BTO  
OZYQBTQ YKCDZYXY; GKQ AX XJKOXO QAX YNZJJ LS QAXD BJJ, BTO QAXW PXCX BQ  
JBYQ LGJZEXO QL BHHXUQ QAX YXHLTO-ABTO ZTQXJJZEXTHX LS QAXZC TXZEAGLKC,  
JBOW JKHBY. AXC CXULCQ PBY AZEAJW SBILKCBGJX. YZC PZJJZBD ABO GXXT  
OXJZEAQXO PZQA AZD. AX PBY FKZQX WLKTE, PLTOXCSKJJW ABTOYLDX, XMQCXDXJW  
BECXXBGJX, BTO, QL HCLPT QAX PALJX, AX DXBTQ QL GX BQ QAX TXMQ BYYXDGJW  
PZQA B JBCEX UBCQW. TLQAZTE HLKJO GX DLCX OXJZEAQSKJ! QL GX SLTO LS  
OBTHZTE PBY B HXCQBZT YQXU QLPBCOY SBJJZTE ZT JLIH; BTO IXCW JZIXJW  
ALUXY LS DC. GZTEJXW'Y AXBCQ PXCX XTQXCQBZTXO.

# After a Little Computation:

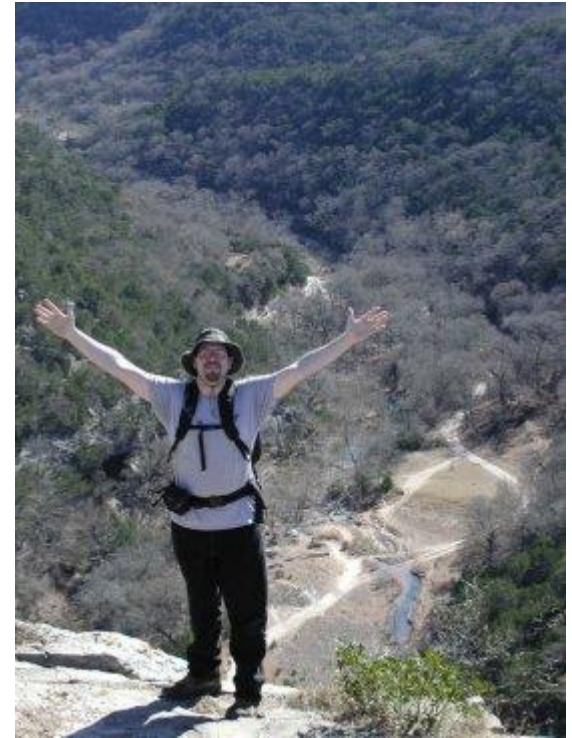
FSABTER 3

ONT ALL TSAT URH. PEOOET, SNMEVER, MITS TSE AHHIHTAOFE NY SER YIVE  
DACGSTERH, FNCLD AHK NO TSE HCPQEFT, MAH HCYIFIEOT TN DRAM YRNU SER  
SCHPAOD AOW HATIHYAFTNRW DEHFRIBTINO NY UR. PIOGLEW. TSEW ATTAFKED SIU  
IO VARINCH MAWH--MITS PAREYAFED JCEHTINOH, IOGEOINCH HCBBNHITINOH, AOD  
DIHTAOT HCRUIHEH; PCT SE ELCDDED TSE HKILL NY TSEU ALL, AOD TSEW MERE AT  
LAHT NPLIGED TN AFFEBT TSE HEFNOD-SAOD IOTELLIGEOFE NY TSEIR OEIGSPNCR,  
LADW LCFAH. SER REBNRT MAH SIGSLW YAVNCRAPLE. HIR MILLIAU SAD PEEO  
DELIGSTED MITS SIU. SE MAH JCITE WNCOG, MNODERYCLLW SAODHNUE, EXTREUELW  
AGREEAPLE, AOD, TN FRNMO TSE MSNLE, SE UEAOT TN PE AT TSE DEXT AHHEUPLW  
MITS A LARGE BARTW. ONTSIOG FNCLD PE UNRE DELIGSTYCL! TN PE YNOD NY  
DAOFIOG MAH A FERTAIO HTEB TNMARDH YALLIOG IO LNVE; AOD VERW LIVELW  
SNBEH NY UR. PIOGLEW'H SEART MERE EOTERTAIOED.

- Apply some human smarts:

# Kurt Dresner – Intersection Control

- ▶ Former PhD student in UTCS department
  - working at Google now
- ▶ area of interest artificial intelligence
- ▶ ***Multiagent Traffic Management: A Reservation-Based Intersection Control Mechanism***
  - how will intersections work if and when cars are autonomous?
  - Simulator





# Austin Villa – Robot Soccer

- ▶ Multiple Autonomous Agents
- ▶ Get a bunch of Sony Aibo robots to play soccer
- ▶ Problems:
  - *vision* (is that the ball?)
  - *localization* (where am I?)
  - *locomotion* (I want to be there!)
  - *coordination* (I am open! pass me the ball!)
- ▶ <http://www.cs.utexas.edu/~AustinVilla/>
- ▶ [Video](#) [Video2](#)

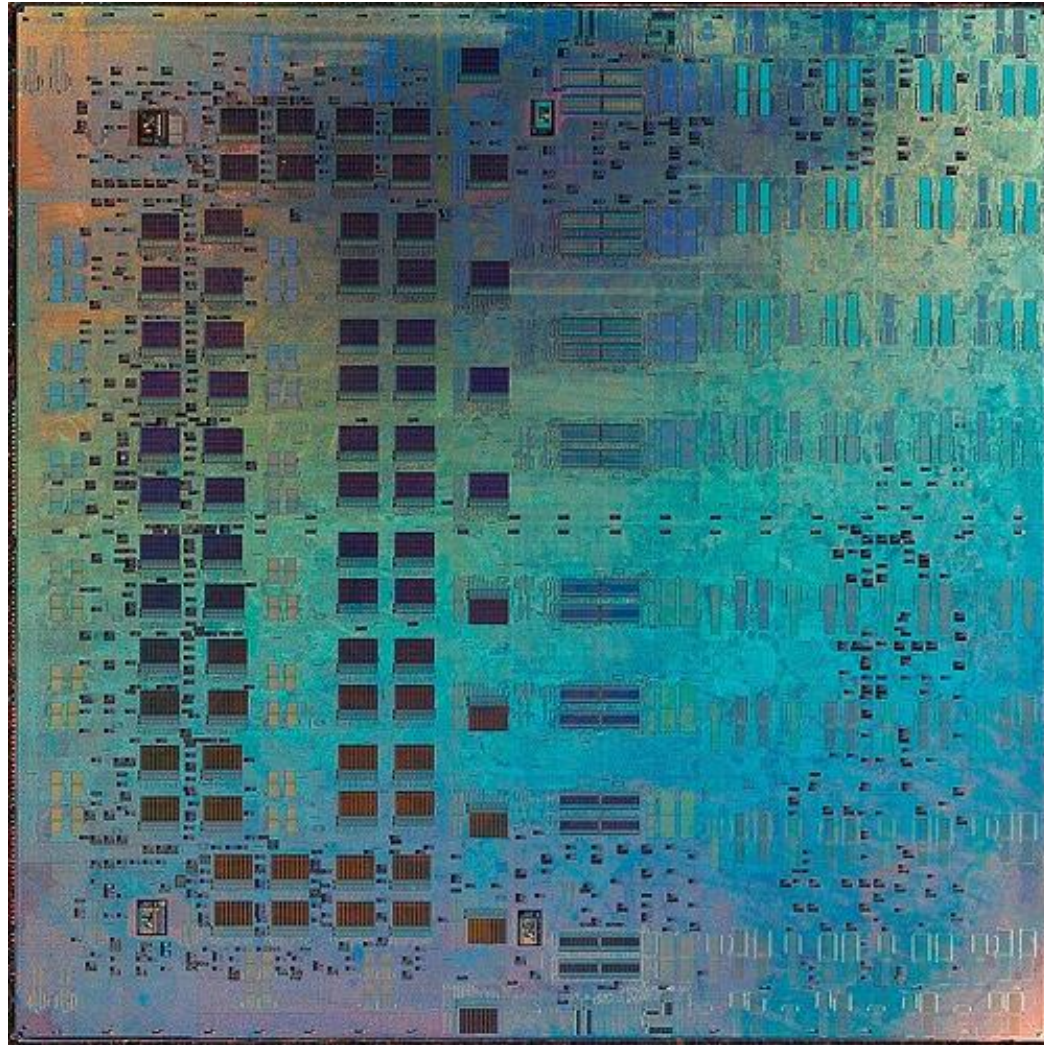


# Doug and Steve

- ▶ Doug Burger and Steve Keckler
  - and many, many others ....
- ▶ TRIPS
  - what has happened to processor speeds the past 5 years?
  - what is a super computer?
  - <http://www.cs.utexas.edu/users/cart/trips/>



# The Trips Chip Prototype



# Google Trends

- ▶ <http://www.google.com/trends>
- ▶ Try these:
  - computer science
  - Mumford and Sons
  - computer science, Mumford and Sons
  - facebook, computer science, Mumford and Sons
  - binary search tree
  - recursion
  - linked lists, binary search tree
  - AP
  - super bowl

# Goolge N Grams

► <http://books.google.com/ngrams>

Google books Ngram Viewer

Graph these [case-sensitive](#) comma-separated phrases: Albert Einstein, Sherlock Holmes, Frankenstein

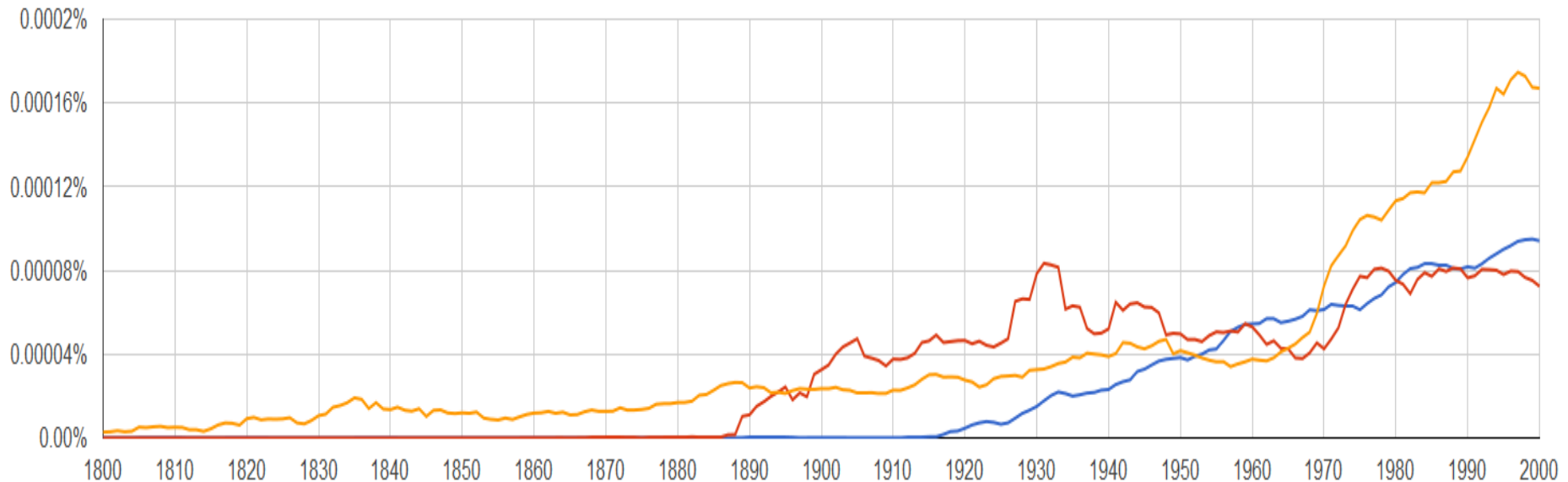
between 1800 and 2000 from the corpus English with smoothing of 3

Search lots of books

Share 1.3k

Tweet 1,228

Albert Einstein Sherlock Holmes Frankenstein



# Topic 2 Java Basics

"To excel in Java, or any computer language, you want to build skill in both the "large" and "small". By "large" I mean the sweeping, strategic issues of algorithms, data structures, ... what we think of basically as a degree in Computer Science. You also need skill in the "small" -- 10 or 20 line methods built of loops, logic, strings, lists etc. to solve each piece of the larger problem. Working with students in my office hours, I see what an advantage it is for students who are practiced and quick with their method code. Skill with the method code allows you to concentrate on the larger parts of the problem. Or put another way, someone who struggles with the loops, logic, etc. does not have time for the larger issues."

- Nick Parlante  
Stanford University, Google





# What We Will Do Today

- ▶ What are computer languages?
- ▶ Writing Java Programs
  - text editor and command line
  - Eclipse
- ▶ First programming concepts
  - output with println statements
  - syntax and errors
- ▶ identifiers, keywords, and comments
- ▶ Strings

# Computers and Computer Languages

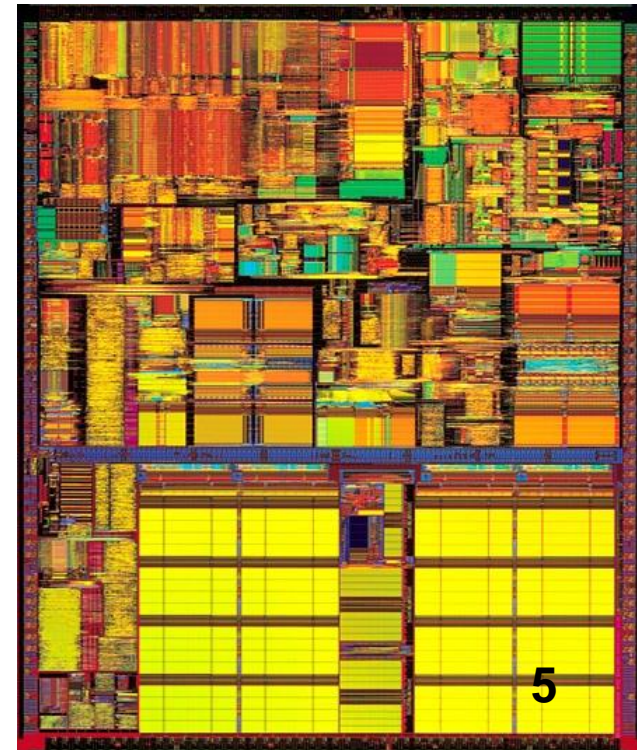
- ▶ Computers are everywhere
  - how many computers do you own?
- ▶ Computers are useful because they run programs
  - program is simply a set of instructions to complete some task
  - how many different programs do you use in a day?

# Definitions

- ▶ **program:** A set of instructions that are to be carried out by a computer.
- ▶ **program execution:** The act of carrying out the instructions contained in a program.
  - this is done by feeding the instructions to the CPU
- ▶ **programming language:** A systematic set of rules used to describe computations, generally in a format that is readable and editable by humans.
  - in this class we use Java

# High Level Languages

- ▶ Computers are fast
  - Intel® Core™ i7-8086K Processor on the order of 2 billion transistors (a switch that is on or off)
  - performs tens of billions of operations per second
- ▶ Computers are simple
  - They can only carry out a very limited set of instructions
    - on the order of 100 or so depending on the computer's processor
    - machine language instructions, aka instruction set architecture (ISA)
    - Add, Branch, Jump, Get Data, Get Instruction, Store, (CS429)



# Machine Code

- ▶ John von Neumann - co-author of paper in 1946 with Arthur W. Burks and Hermann H. Goldstine,
  - "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument"
- ▶ One of the key points
  - program commands and data stored as sequences of bits in the computer's memory
- ▶ A program:  

```
1110001100000000
0101011011100000
0110100001000000
0000100000001000
0001011011000100
0001001001100001
0110100001000000
0000111000000011
```





# Say What?

- ▶ Programming with Strings of bits (1s or 0s) is not the easiest thing to do.
- ▶ Assembly language
  - mnemonics for machine language instructions

.ORIG	x3001
LD	R1, x3100
AND	R3, R3 #0
LD	R4, R1
BRn	x3008
ADD	R3, R3, R4
ADD	R1, R1, #1
LD	R4, R1
BRnzp	x3003

# High Level Languages

- ▶ Assembly language, still not so easy, and lots of commands to accomplish things
- ▶ High Level Computer Languages provide the ability to accomplish a lot with fewer commands than machine or assembly language in a way that is hopefully easier to understand

```
int sum = 0;
int count = 0;
while (list[count] != -1) {
    sum += list[count];
    count = count + 1;
}
```

# Binary Numbers - Base 2

► Base 10, 10 digits

►  $5127_{10} = 5 * 1000 + 1 * 100 + 2 * 10 + 7 * 1$

►  $5127_{10} = 5 * 10^3 + 1 * 10^2 + 2 * 10^1 + 7 * 10^0$

► Base 2, 2 digits

►  $110101_2 = 1 * 32 + 1 * 16 + 0 * 8 + 1 * 4 + 0 * 2 + 1 * 1$

►  $110101_2 = 1 * 2^5 + 1 * 2^4 + 1 * 2^2 + 1 * 2^0$

►  $110101_2 = 53_{10}$



There are  
10 types  
of people  
in the world:  
  
Those who  
understand binary,  
and those  
who don't. |

# Java

- ▶ There are thousands of high level computer languages. Java, C++, C, Basic, Fortran, Cobol, Lisp, Perl, Prolog, Eiffel, Python
- ▶ The capabilities of the languages vary widely, but they all need a way to do
  - declarative statements
  - conditional statements
  - iterative or repetitive statements
- ▶ A compiler is a program that converts commands in high level languages to machine language instructions

# A Simple Java Program

```
public class Hello {  
    public static void main(String[] args) {  
        System.out.println("Hello World!");  
    }  
}
```

This would be in a text file named Hello.java

DEMO of writing and running a program via notepad and the command line

# Running a program

## 1. Write it.

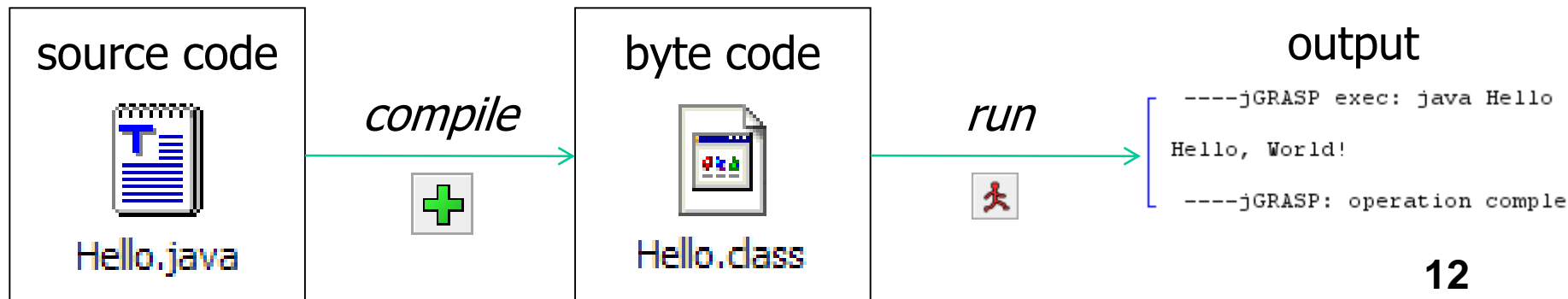
- **code** or **source code**: The set of instructions in a program.

## 2. Compile it.

- **compile**: Translate a program from one language to another.
- **byte code**: The Java compiler converts your code into a format named *byte code* that runs on many computer types.

## 3. Run (execute) it.

- **output**: The messages printed to the user by a program.





# Bigger Java program!

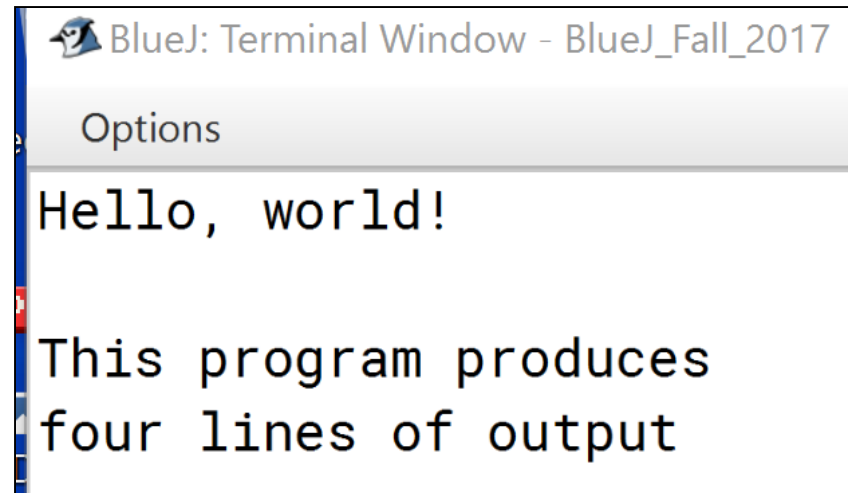
```
public class Hello {  
    public static void main(String[] args) {  
        System.out.println("Hello, world!");  
        System.out.println();  
        System.out.println("This program produces");  
        System.out.println("four lines of output");  
    }  
}
```

## ► Its output:

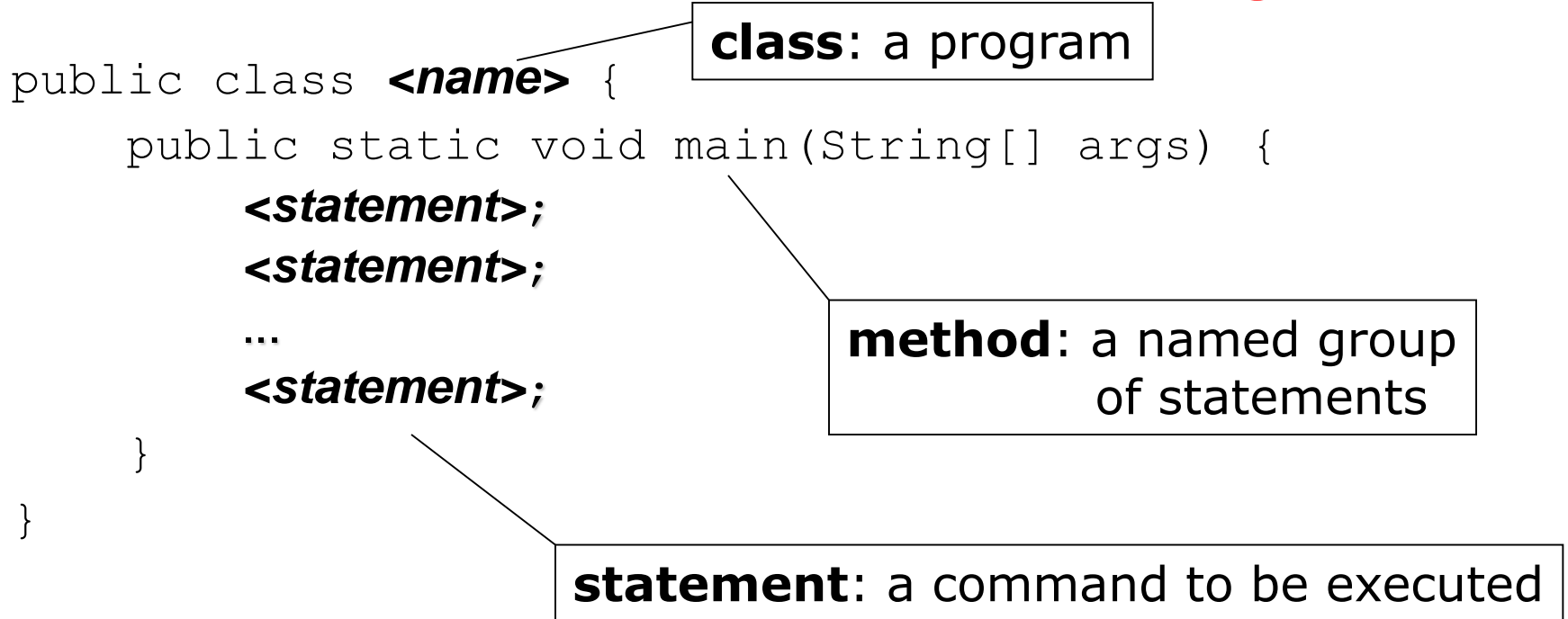
Hello, world!

This program produces  
four lines of output

## ► **console:** Text box into which the program's output is printed.



# Structure of a Java program



► Every executable Java program consists of a **class**,

– that contains a **method** named `main`,

- that contains the **statements** (commands) to be executed.

# System.out.println

- ▶ A statement that prints a line of output on the console.
  - pronounced "print-linn"
- ▶ Two ways to use `System.out.println` :
  - `System.out.println(" <text>") ;`  
Prints the given message as output.
  - `System.out.println() ;`  
Prints a blank line of output.

# Syntax

- ▶ **syntax:** The set of legal structures and commands that can be used in a particular language.
  - Every basic Java statement ends with a semicolon ;
  - The contents of a class or method occur between { and }
- ▶ **syntax error (compiler error):** A problem in the structure of a program that causes the compiler to fail.
  - Missing semicolon
  - Too many or too few { } braces, braces not matching
  - Class and file names do not match
  - ...

# Syntax error example

```
1 public class Hello {  
2     pooblic static void main(String[] args) {  
3         System.owt.println("Hello, world!")_  
4     }  
5 }
```

## ► Compiler output:

```
Hello.java:2: <identifier> expected  
    pooblic static void main(String[] args) {  
      ^  
Hello.java:3: ';' expected  
    }  
    ^  
2 errors
```

- The compiler shows the line number where it found the error.
- The error messages sometimes can be tough to understand:
  - Why can't the computer just say “*You misspelled ‘public’?*”?

# An Important Realization

- ▶ Computers are stupid.
- ▶ Computers can't read minds.
- ▶ Computers ***seldom*** make mistakes.
- ▶ If the computer is not doing what we want, it's because **WE** made a mistake.



# More on syntax errors

## ► Java is case-sensitive

- Hello and hello are not the same

```
1 Public class Hello {  
2     public static void main(String[] args) {  
3         System.out.println("Hello, world!");  
4     }  
5 }
```

compiler output:

```
Hello.java:1: class, interface, or enum expected  
Public class Hello {  
^  
1 error
```

# Names

- ▶ You must give your program a name.

```
public class
```

```
SubstitutionCipherDecoder {
```

- Naming convention: capitalize each word (e.g. `MyClassName`)
- Your program's file must match exactly (`SubstitutionCipherDecoder.java`)
  - includes capitalization (remember, Java is "case-sensitive")

# Identifiers

► **identifier:** A name given to an item in your program.

- must start with a letter, underscore, or \$
- subsequent characters can be any of those or digits 0 through 9

• **legal:**            `_myName`            `TheCure`  
                      `ANSWER_IS_42`       `$bling$`

• **illegal:**           `me+u`            `49ers`            `side-swipe`  
                      `Ph.D's`

# Keywords

- **keyword:** An identifier that you cannot use because it already has a reserved (special) meaning in Java.

abstract	default	if	private	this
boolean	do	implements	protected	throw
break	double	import	<b>public</b>	throws
byte	else	instanceof	return	transient
case	extends	int	short	try
catch	final	interface	<b>static</b>	<b>void</b>
char	finally	long	strictfp	volatile
<b>class</b>	float	native	super	while
const	for	new	switch	
continue	goto	package	synchronized	

- Because Java is case-sensitive, you could technically use `Class` or `cLaSs` as identifiers, but this is very confusing and thus **strongly discouraged**.

# Clicker 1

- ▶ Which of the following is not a syntactically correct Java identifier for the name of a program?
- A. static
- B. Void
- C. FirstProgram
- D. \_My\_program
- E. More than one of A - D is not a syntactically correct Java identifier.

# Strings

- ▶ **string**: A sequence of text characters.
  - Starts and ends with a " (quotation mark character).
    - The quotes do not appear in the output.
  - Examples:
    - `"hello"`
    - `"This is a string. It's very long!"`
- ▶ **Restrictions**:
  - May not span multiple lines.
    - `"This is not  
a legal String."`
  - May not contain a " character.
    - `"This is not a "legal" String either."`
- ▶ This begs the question...



# Escape sequences

- ▶ **escape sequence:** A special sequence of characters used to represent certain special characters in a string.

`\t`      tab character

`\n`      new line character

`\"`      quotation mark character

`\\`      backslash character

- **Example:**

```
System.out.println("\\hello\\nhow\\tare \"you\"?\\\\\\");
```

- **Output:**

```
\\hello
```

```
how      are "you"?\\
```

## Clicker 2

- ▶ How many visible characters does the following println statement produce when run?

```
System.out.println("\t\nn\\t\"\\tt");
```

- A. 0
- B. 1
- C. 2
- D. 3
- E. 4

# Practice Program 1

- ▶ What sequence of println statements will generate the following output?

This program prints the first lines  
of the song "slots".

"She lives in a trailer"

"On the outskirts 'a Reno"

"She plays quarter slots in the local's casino."

# Practice Program 2

- ▶ What sequence of println statements will generate the following output?

```
A "quoted" String is  
'much' better if you learn  
the rules of "escape sequences."
```

```
Also, "" represents an empty String.  
Don't forget to use \" instead of " !  
' ' is not the same as "
```

# Practice Program 3

- ▶ What is the output of the following `println` statements?

```
System.out.println("\ta\tb\tc");  
System.out.println("\\\\");  
System.out.println("'");  
System.out.println("\"\"");  
System.out.println("C:\nin\the downward spiral");
```

# Answer to Practice Program 3

Output of each println statement:

```
        a      b      c
\\
'
""
C:
i
the downward spiral
```



# Practice Program 4

- ▶ Write a `println` statement to produce this output:

/ \ // \ \ /// \ \ \

# Answer to Practice Program 4

println statement to produce the line of output:

```
System.out.println("/  \  //  \\\  ///  \\\\");
```

# Topic 3

## static Methods and Structured Programming

"The cleaner and nicer the program,  
the faster it's going to run.  
And if it doesn't, it'll be easy  
to make it fast."

-Joshua Bloch

Based on slides by Marty Stepp and Stuart Reges  
from <http://www.buildingjavaprograms.com/>



# Clicker 1

► What is the name of the method that is called when a Java program starts?

A. main

B. static

C. void

D. println

E. class

# Comments

- ▶ **comment:** A note written in source code by the programmer to describe or clarify the code.
  - Comments are not executed when your program runs.

- ▶ **Syntax:**

`// comment text, on one line`

or,

`/* comment text; may span multiple lines */`

- ▶ **Examples:**

`// This is a one-line comment.`

`/* This is a very long  
multi-line comment. */`

# Using comments

- ▶ Where to place comments:
  - at the top of each file (a "comment header")
  - at the start of every method (seen later)
  - to explain complex pieces of code
- ▶ Comments are useful for:
  - Understanding larger, more complex programs.
  - Multiple programmers working together, who must understand each other's code.

# Comments example

```
/* Suzy Student, CS 101, Fall 2019
   This program prints lyrics about ... something. */

public class BaWitDaBa {

    public static void main(String[] args) {
        // first verse
        System.out.println("Bawitdaba");
        System.out.println("da bang a dang diggy diggy");
        System.out.println();


        // second verse
        System.out.println("diggy said the boogy");
        System.out.println("said up jump the boogy");
    }
}
```



# Program Hygiene - a.k.a. Style

- ▶ Provide a structure to the program
- ▶ Eliminate redundant code
- ▶ Use spaces judiciously and **consistently**
- ▶ Indent properly
- ▶ Follow the naming conventions
- ▶ Use comments to describe code behavior
- ▶ Follow a brace style
- ▶ Good software follows a style guide
  - See links on assignment page

# Google C++ Style Guide


 **google-styleguide**  
Style guides for Google-originated open-source projects

Search projects

Project Home [Source](#)

Summary [People](#)

**Project Information**


 +603 Recommend this on Google+

[Project feeds](#)

**Code license**  
[Artistic License/GPL](#)

**Content license**  
[Creative Commons 3.0 BY](#)

**Labels**  
[Google](#), [Documentation](#),  
[CPlusPlus](#), [Objective-C](#), [XML](#),  
[Python](#), [JavaScript](#)

 **Members**  
[mmento...@gmail.com](#),  
[mark@chromium.org](#)

Every major open-source project has its own style guide: a set of conventions (sometimes arbitrary) about how to write code for that project. It is much easier to understand a large codebase when all the code in it is in a consistent style.

"Style" covers a wide range of things, from "use camelCase for variable names" to "never use global variables" to "never use exceptions." This project holds the style guidelines we use for Google code. If you are modifying a project that originated at Google, you may be pointed to this page to see the style guides that apply to that project.

Our [C++ Style Guide](#), [Objective-C Style Guide](#), [Python Style Guide](#), [Shell Style Guide](#), [HTML/CSS Style Guide](#), [JavaScript Style Guide](#), and [Common Lisp Style Guide](#) are now available. We have also released [cpplint](#), a tool to assist with style guide compliance, and [google-c-style.el](#), an Emacs settings file for Google style.

If your project requires that you create a new XML document format, our [XML Document Format Style Guide](#) may be helpful. In addition to actual style rules, it also contains advice on designing your own vs. adapting an existing format, on XML instance document formatting, and on elements vs. attributes.

<http://code.google.com/p/google-styleguide/>

7

# Google C++ Style Guide

## Local Variables

- ▽ Place a function's variables in the narrowest scope possible, and initialize variables in the declaration.

C++ allows you to declare variables anywhere in a function. We encourage you to declare them in as local a scope as possible, and as close to the first use as possible. This makes it easier for the reader to find the declaration and see what type the variable is and what it was initialized to. In particular, initialization should be used instead of declaration and assignment, e.g.

```
int i;  
i = f();      // Bad -- initialization separate from declaration.
```

```
int j = g();  // Good -- declaration has initialization.
```

```
vector<int> v;  
v.push_back(1); // Prefer initializing using brace initialization.  
v.push_back(2);
```

```
vector<int> v = {1, 2}; // Good -- v starts initialized.
```

# Why Worry About Program Hygiene ?

- ▶ Programmers build on top of other's code all the time.
  - Computer Scientists and Software developers spend as much time maintaining code as they do creating new code
  - You shouldn't waste time deciphering what a method does.
- ▶ You should spend time on thinking and coding.  
You should **NOT** be wasting time looking for that missing closing brace.
- ▶ "Code is read more often than it is written."
  - *Guido Van Rossum* (Creator of the Python Language)

# Algorithms

- ▶ **algorithm:** A list of steps for solving a problem.
- ▶ Example algorithm: "Bake sugar cookies"
  - Mix the dry ingredients.
  - Cream the butter and sugar.
  - Beat in the eggs.
  - Stir in the dry ingredients.
  - Set the oven temperature.
  - Set the timer.
  - Place the cookies into the oven.
  - Allow the cookies to bake.
  - Mix ingredients for frosting.
  - ...



# Problems with algorithms

- ▶ *lack of structure*: Many small steps; tough to remember.
- ▶ *redundancy*: Consider making a double batch...
  - Mix the dry ingredients.
  - Cream the butter and sugar.
  - Beat in the eggs.
  - Stir in the dry ingredients.
  - Set the oven temperature.
  - Set the timer.
  - Place the first batch of cookies into the oven.
  - Allow the cookies to bake.
  - Set the oven temperature.
  - Set the timer.
  - Place the second batch of cookies into the oven.
  - Allow the cookies to bake.
  - Mix ingredients for frosting.
  - ...

# Structured algorithms

## ► **structured algorithm:**

Split solution into coherent tasks.

### 1 Make the cookie batter.

- Mix the dry ingredients.
- Cream the butter and sugar.
- Beat in the eggs.
- Stir in the dry ingredients.

### 2 Bake the cookies.

- Set the oven temperature.
- Set the timer.
- Place the cookies into the oven.
- Allow the cookies to bake.

### 3 Add frosting and sprinkles.

- Mix the ingredients for the frosting.
- Spread frosting and sprinkles onto the cookies.
- ...



# Removing redundancy

- ▶ A well-structured algorithm can describe repeated tasks with less redundancy.

## 1 Make the cookie batter.

- Mix the dry ingredients.
- ...

## 2a Bake the cookies (first batch).

- Set the oven temperature.
- Set the timer.
- ...

## 2b Bake the cookies (second batch).

## 3 Decorate the cookies.

- ...

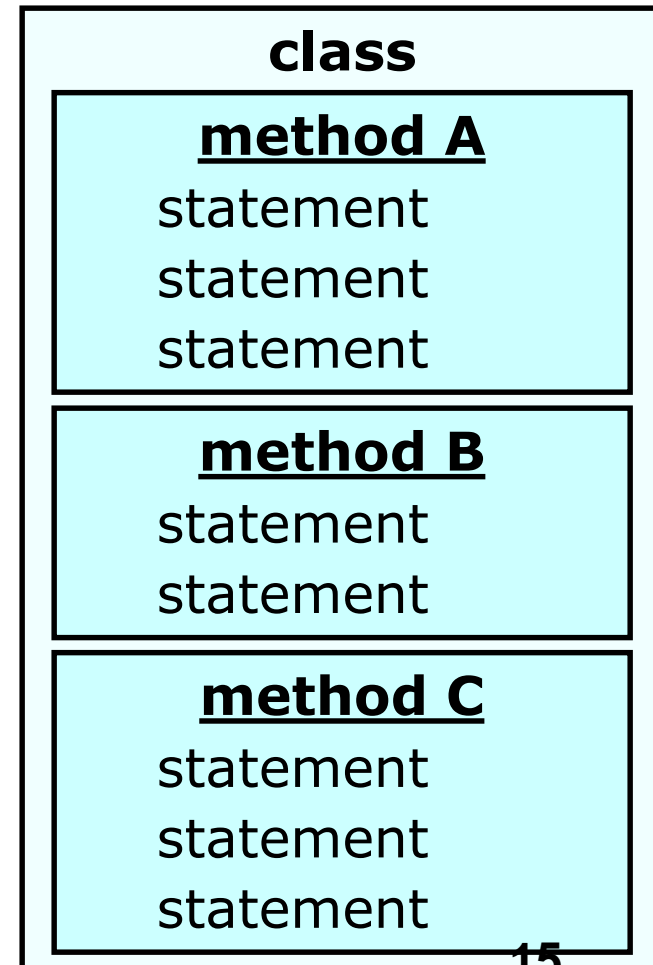
# A program with redundancy

// This program displays a delicious recipe for baking cookies.

```
public class BakeCookies {  
  
    public static void main(String[] args) {  
        System.out.println("Mix the dry ingredients.");  
        System.out.println("Cream the butter and sugar.");  
        System.out.println("Beat in the eggs.");  
        System.out.println("Stir in the dry ingredients.");  
        System.out.println("Set the oven temperature.");  
        System.out.println("Set the timer.");  
        System.out.println("Place a batch of cookies into the oven.");  
        System.out.println("Allow the cookies to bake.");  
        System.out.println("Set the oven temperature.");  
        System.out.println("Set the timer.");  
        System.out.println("Place a batch of cookies into the oven.");  
        System.out.println("Allow the cookies to bake.");  
        System.out.println("Mix ingredients for frosting.");  
        System.out.println("Spread frosting and sprinkles.");  
    }  
}
```

# Static methods

- ▶ **static method:** A named group of statements.
  - denotes the *structure* of a program
  - eliminates *redundancy* by code reuse
- ▶ **procedural decomposition:**  
dividing a problem into methods
  - a way to *manage complexity*
- ▶ Writing a static method is like adding a new command to Java.



- Building complex systems is hard
- Some of the most complex systems are software systems

YEAR	COMPANY	OUTCOME (COSTS IN US \$)
2005	Hudson Bay Co. [Canada]	Problems with inventory system contribute to \$33.3 million* loss.
2004-05	UK Inland Revenue	Software errors contribute to \$3.45 billion* tax-credit overpayment.
2004	Avis Europe PLC [UK]	Enterprise resource planning (ERP) system canceled after \$54.5 million <sup>†</sup> is spent.
2004	Ford Motor Co.	Purchasing system abandoned after deployment costing approximately \$400 million.
2004	J Sainsbury PLC [UK]	Supply-chain management system abandoned after deployment costing \$527 million. <sup>†</sup>
2004	Hewlett-Packard Co.	Problems with ERP system contribute to \$160 million loss.
2003-04	AT&T Wireless	Customer relations management (CRM) upgrade problems lead to revenue loss of \$100 million.
2002	McDonald's Corp.	The Innovate information-purchasing system canceled after \$170 million is spent.
2002	Sydney Water Corp. [Australia]	Billing system canceled after \$33.2 million <sup>†</sup> is spent.
2002	CIGNA Corp.	Problems with CRM system contribute to \$445 million loss.
2001	Nike Inc.	Problems with supply-chain management system contribute to \$100 million loss.
2001	Kmart Corp.	Supply-chain management system canceled after \$130 million is spent.

# Using static methods

1. **Design** the algorithm.
  - Look at the structure, and which commands are repeated.
  - Decide what are the important overall tasks.
  - Good programmers do this BEFORE writing any code
2. **Declare** (write down) the methods.
  - Arrange statements into groups and give each group a name.
3. **Call** (run) the methods.
  - The program's `main` method executes the other methods to perform the overall task.

# Design of an algorithm

```
// This program displays a delicious recipe for baking cookies.
public class BakeCookies2 {

    public static void main(String[] args) {
        // Step 1: Make the cake batter.
        System.out.println("Mix the dry ingredients.");
        System.out.println("Cream the butter and sugar.");
        System.out.println("Beat in the eggs.");
        System.out.println("Stir in the dry ingredients.");

        // Step 2a: Bake cookies (first batch).
        System.out.println("Set the oven temperature.");
        System.out.println("Set the timer.");
        System.out.println("Place a batch of cookies into the oven.");
        System.out.println("Allow the cookies to bake.");

        // Step 2b: Bake cookies (second batch).
        System.out.println("Set the oven temperature.");
        System.out.println("Set the timer.");
        System.out.println("Place a batch of cookies into the oven.");
        System.out.println("Allow the cookies to bake.");

        // Step 3: Decorate the cookies.
        System.out.println("Mix ingredients for frosting.");
        System.out.println("Spread frosting and sprinkles.");
    }
}
```

# Declaring a method

*Gives your method a name so it can be executed*

## ► Syntax:

```
public static void <name>() {  
    <statement>;  
    <statement>;  
    ...  
    <statement>;  
}
```

## ► Example:

```
public static void printWarning() {  
    System.out.println("This product causes cancer");  
    System.out.println("in lab rats and humans.");  
}
```

# Calling a method

*Executes the method's code*

## ► Syntax:

***<name>*** ( ) ;

- You can call the same method many times if you like.

## ► Example:

```
printWarning ( ) ;
```

- Output:

```
This product causes cancer  
in lab rats and humans.
```



# Program with static method

```
public class FreshPrince {  
  
    public static void main(String[] args) {  
        rap();                // Calling (running) the rap method  
        System.out.println();  
        rap();                // Calling the rap method again  
    }  
  
    // This method prints the lyrics to my favorite song.  
    public static void rap() {  
        System.out.println("Now this is the story all about how");  
        System.out.println("My life got flipped turned upside-down");  
    }  
}
```

## Output:

```
Now this is the story all about how  
My life got flipped turned upside-down
```

```
Now this is the story all about how  
My life got flipped turned upside-down
```

# Final cookie program

```
// This program displays a delicious recipe for baking cookies.
public class BakeCookies3 {

    public static void main(String[] args) {
        makeBatter();
        bake();           // 1st batch
        bake();           // 2nd batch
        decorate();
    }



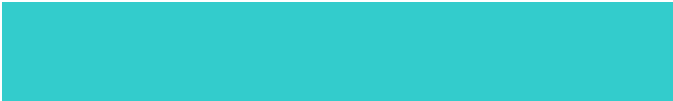
    // Step 1: Make the cookie batter.
    public static void makeBatter() {
        System.out.println("Mix the dry ingredients.");
        System.out.println("Cream the butter and sugar.");
        System.out.println("Beat in the eggs.");
        System.out.println("Stir in the dry ingredients.");
    }

    // Step 2: Bake a batch of cookies.
    public static void bake() {
        System.out.println("Set the oven temperature.");
        System.out.println("Set the timer.");
        System.out.println("Place a batch of cookies into the oven.");
        System.out.println("Allow the cookies to bake.");
    }








    // Step 3: Decorate the cookies.
    public static void decorate() {
        System.out.println("Mix ingredients for frosting.");
        System.out.println("Spread frosting and sprinkles.");
    }
}
```

# Summary: Why methods?

- Makes code easier to read by capturing the structure of the program
  - `main` should be a good summary of the program

```
public static void main(String[] args) {  
      
      
      
}
```

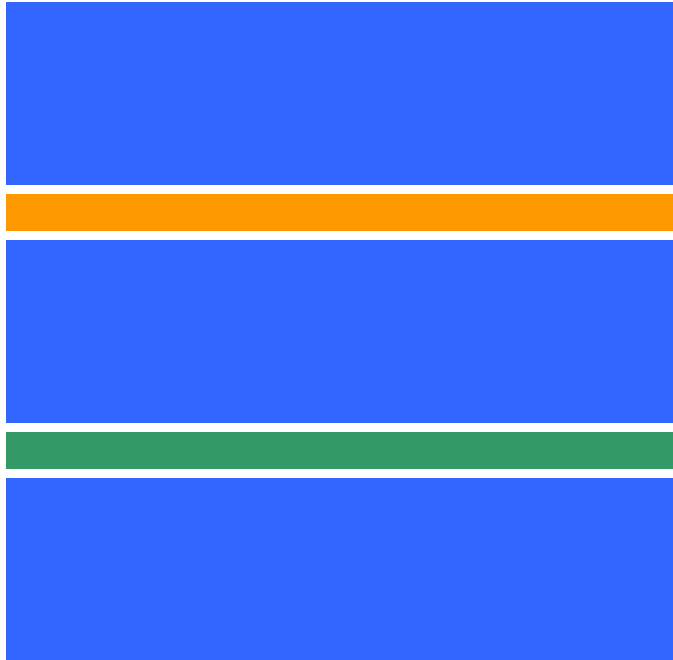
**Note:** Longer code doesn't necessarily mean worse code!!!

```
public static void main(String[] args) {  
      
      
      
}  
  
public static ...  (...) {  
      
}  
  
public static ...  (...) {  
      
}
```

# Summary: Why methods?

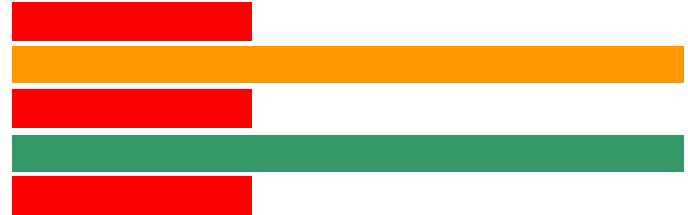
## ► Eliminate redundancy

```
public static void main(String[] args) {
```



```
}
```

```
public static void main(String[] args) {
```



```
}
```

```
public static ...          (...) {
```



```
}
```

# Methods calling methods

```
public class MethodsExample {  
  
    public static void main(String[] args) {  
        message1();  
        message2();  
        System.out.println("Done with main.");  
    }  
  
    public static void message1() {  
        System.out.println("This is message1.");  
    }  
  
    public static void message2() {  
        System.out.println("This is message2.");  
        message1();  
        System.out.println("Done with message2.");  
    }  
}
```

## ► Output:

```
This is message1.  
This is message2.  
This is message1.  
Done with message2.  
Done with main.
```

# Control flow

- ▶ When a method is called, the program's execution...
  - "jumps" into that method, executing its statements, then
  - "jumps" back to the point where the method was called.

```
public class MethodsExample {  
    public static void main(String[] args) {  
        message1() ;  
        message2() ;  
        System.out.println("Done with message2.");  
    }  
    ...  
}
```

```
public static void message1() {  
    System.out.println("This is message1.");  
}
```

```
public static void message2() {  
    System.out.println("This is message2.");  
    message1() ;  
    System.out.println("Done with message2.");  
}
```

```
public static void message1() {  
    System.out.println("This is message1.");  
}
```

► **Clicker 2** - How many lines of output with visible characters does the following program produce?

```
public class MethodCalls {  
    public static void main(String[] args ) {  
        a();  
        b();  
        c();  
        c();  
    }  
  
    public static void a() {      System.out.println("A");      }  
  
    public static void b() {  
        System.out.println("B");  
        a();  
        System.out.println("B");  
    }  
  
    public static void c() {  
        a();  
        b();  
        System.out.println("C");  
        b();  
    }  
}
```

A. 3      B. 4      C. 8      D. 12      E. 20

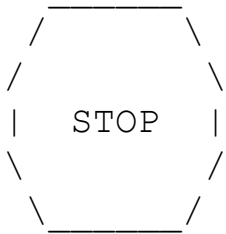
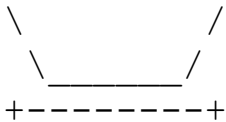
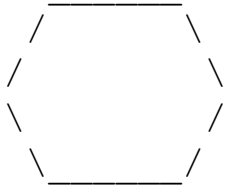
# Drawing complex figures with static methods

**reading: 1.5**  
(Ch. 1 Case Study:  
`DrawFigures`)

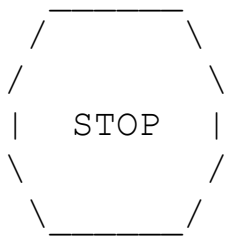
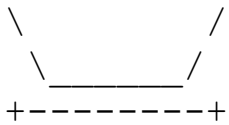
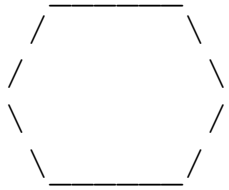


# Static methods question

- Write a program to print these figures.



# Development strategy



First version (unstructured):

Create an empty program and `main` method.

Copy the expected output into it, surrounding each line with `System.out.println` syntax.

Run it to verify the output.

**Clicker 3 - Are there repeated sections of output for this program?**

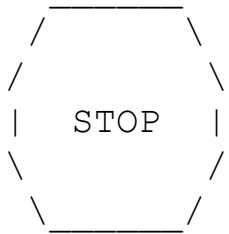
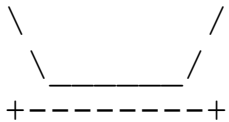
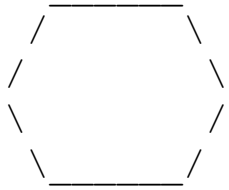
**A. No**

**B. Yes**

# Program version 1

```
public class Figures1 {  
  
    public static void main(String[] args) {  
        System.out.println("      ");  
        System.out.println(" /_____\\");  
        System.out.println("/           \\");  
        System.out.println("\\           /");  
        System.out.println(" \\_____ /");  
        System.out.println();  
        System.out.println("\\           /");  
        System.out.println(" \\_____ /");  
        System.out.println("+-----+");  
        System.out.println();  
        System.out.println("      ");  
        System.out.println(" /_____\\");  
        System.out.println("/           \\");  
        System.out.println("|   STOP   |");  
        System.out.println("\\           /");  
        System.out.println(" \\_____ /");  
        System.out.println();  
        System.out.println("      ");  
        System.out.println(" /_____\\");  
        System.out.println("/           \\");  
        System.out.println("+-----+");  
    }  
}
```

# Development strategy 2

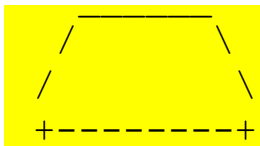
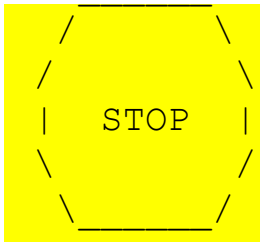
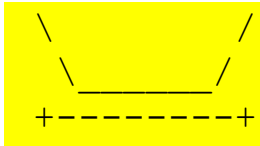
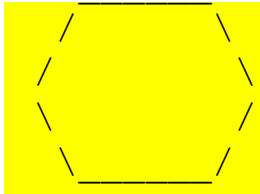


Second version (structured, with redundancy):

Identify the structure of the output.

Divide the `main` method into static methods based on this structure.

# Output structure



The structure of the output:

initial "egg" figure

second "teacup" figure

third "stop sign" figure

fourth "hat" figure

This structure can be represented by methods:

`egg`

`teaCup`

`stopSign`

`hat`

# Program version 2

```
public class Figures2 {

    public static void main(String[] args) {
        egg();
        teaCup();
        stopSign();
        hat();
    }

    public static void egg() {
        System.out.println("          ");
        System.out.println(" /_____\\");
        System.out.println("/          \\");
        System.out.println("\\          /");
        System.out.println(" \\_____ /");
        System.out.println();
    }

    public static void teaCup() {
        System.out.println("\\          /");
        System.out.println(" \\_____ /");
        System.out.println("+-----+");
        System.out.println();
    }

    ...
}
```

# Program version 2, cont'd.

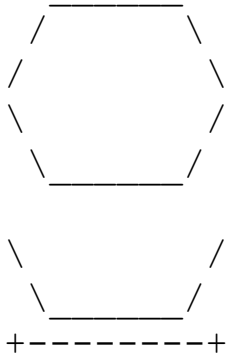
...

```
public static void stopSign() {  
    System.out.println("      ");  
    System.out.println(" /_____\\");  
    System.out.println("/           \\");  
    System.out.println("|   STOP   |");  
    System.out.println("\\           /");  
    System.out.println(" \\_____ /");  
    System.out.println();  
}
```

```
public static void hat() {  
    System.out.println("      ");  
    System.out.println(" /_____\\");  
    System.out.println("/           \\");  
    System.out.println("+-----+");  
}
```

```
}
```

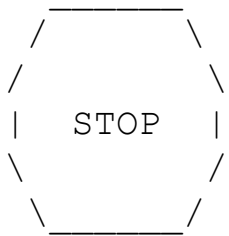
# Development strategy 3



Third version (structured, without redundancy):

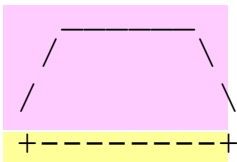
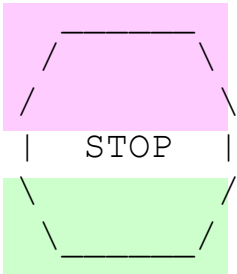
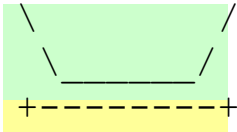
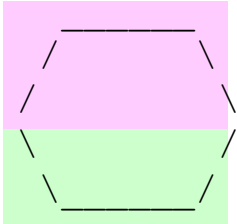
Identify redundancy in the output, and create methods to eliminate as much as possible.

Add comments to the program.





# Output redundancy



The redundancy in the output:

egg top:	reused on stop sign, hat
egg bottom:	reused on teacup, stop sign
divider line:	used on teacup, hat

This redundancy can be fixed by methods:

eggTop  
eggBottom  
line

# Program version 3

```
// Suzy Student, CSE 138, Spring 2094
// Prints several figures, with methods for structure and redundancy.
public class Figures3 {

    public static void main(String[] args) {
        egg();
        teaCup();
        stopSign();
        hat();
    }

    // Draws the top half of an an egg figure.
    public static void eggTop() {
        System.out.println("      ");
        System.out.println(" /-----\\");
        System.out.println("/           \\");
    }

    // Draws the bottom half of an egg figure.
    public static void eggBottom() {
        System.out.println("\\           /");
        System.out.println("\\-----/");
    }

    // Draws a complete egg figure.
    public static void egg() {
        eggTop();
        eggBottom();
        System.out.println();
    }

    ...
}
```

# Program version 3, cont'd.

```
...  
// Draws a teacup figure.  
public static void teaCup() {  
    eggBottom();  
    line();  
    System.out.println();  
}  
  
// Draws a stop sign figure.  
public static void stopSign() {  
    eggTop();  
    System.out.println("|  STOP  |");  
    eggBottom();  
    System.out.println();  
}  
  
// Draws a figure that looks sort of like a hat.  
public static void hat() {  
    eggTop();  
    line();  
}  
  
// Draws a line of dashes.  
public static void line() {  
    System.out.println("+-----+");  
}  
}
```

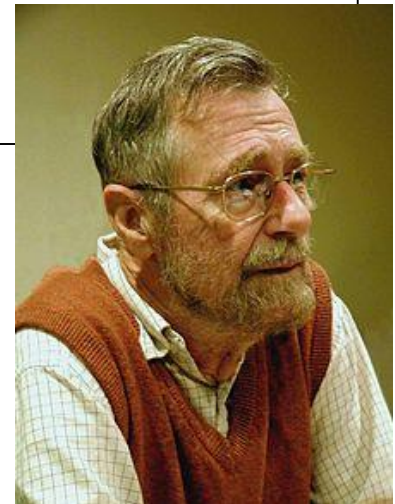
# Topic 4

## Expressions and Variables

"Once a person has understood the way variables are used in programming, they have understood the quintessence of programming."

*-Professor Edsger W. Dijkstra*

Based on slides by Marty Stepp and Stuart Reges  
from <http://www.buildingjavaprograms.com/>



# Data and expressions

**reading: 2.1**

# The computer's view

- ▶ Internally, most computers store everything as 1's and 0's
  - Example:
    - h → 01101000
    - "hi" → 0110100001101001
    - 104 → 01101000
- ▶ How can the computer tell the difference between an `h` and `104`?
- ▶ **type**: A category or set of data values.
  - Constrains the operations that can be performed on data
  - Many languages ask the programmer to specify types
  - Examples: integer, real number, string
- ▶ Binary Numbers

# Java's primitive types

- ▶ **primitive types**: 8 simple types for numbers, characters, etc.
  - Java also has **object types**, which we'll talk about later

Name	Description	Examples
<code>int</code>	integers (up to $2^{31} - 1$ )	<code>42</code> , <code>-3</code> , <code>0</code> , <code>926394</code>
<code>double</code>	real numbers (up to $10^{308}$ )	<code>3.1</code> , <code>-0.25</code> , <code>9.4e3</code>
<code>char</code>	single text characters	<code>'a'</code> , <code>'X'</code> , <code>'?'</code> , <code>'\n'</code>
<code>boolean</code>	logical values	<code>true</code> , <code>false</code>

- Why does Java distinguish integers vs. real numbers?

# Integer or real number?

- ▶ Which category is more appropriate?

integer ( <code>int</code> )	real number ( <code>double</code> )

1. Temperature in degrees Celsius
2. The population of lemmings
3. Your grade point average
4. A person's age in years
5. A person's weight in pounds
6. A person's height in meters
7. Number of miles traveled
8. Number of dry days in the past month
9. Your locker number
10. Number of seconds left in a game
11. The sum of a group of integers
12. The average of a group of integers

- ▶ credit: Kate Deibel, <http://www.cs.washington.edu/homes/deibel/CATs/>



# Clicker 1

► What is best choice for data type?

CHOICE	Number of days it rained in year	Sum of group of integers	Average of group of integers
A	<code>int</code>	<code>int</code>	<code>double</code>
B	<code>int</code>	<code>int</code>	<code>int</code>
C	<code>double</code>	<code>int</code>	<code>int</code>
D	<code>double</code>	<code>int</code>	<code>double</code>
E	<code>int</code>	<code>double</code>	<code>double</code>

# Expressions

- ▶ **expression:** A combination of values and / or operations that results (via computation) in a value.

- Examples: `1 + 4 * 5`

`(7 + 2) * 6 / 3`

`42`

`"Hello, world!"`

- The simplest expression is a *literal value*.
- A complex expression uses operators and parentheses.

# Arithmetic operators

- ▶ **operator:** Combines multiple values or expressions.

- + addition
  - subtraction (or negation)
  - \* multiplication
  - / division
  - % remainder (sometimes called modulus)

- ▶ As a program runs, its expressions are *evaluated*.

`1 + 1` evaluates to 2

`System.out.println(3 * 4);` prints 12

How would we print the text `3 * 4` ?

# Integer division with /

- ▶ When we divide integers, the quotient is also an integer.
- ▶ **Euclidean division a.k.a. division with remainder.**

14 / 4 is 3, not 3.5

$$\begin{array}{r} 3 \\ 4 \overline{) 14} \\ \underline{12} \\ 2 \end{array}$$

$$\begin{array}{r} 4 \\ 10 \overline{) 45} \\ \underline{40} \\ 5 \end{array}$$

$$\begin{array}{r} 52 \\ 27 \overline{) 1425} \\ \underline{135} \\ 75 \\ \underline{54} \\ 21 \end{array}$$

- ▶ More examples:

– 32 / 5 is 6

– 84 / 10 is 8

– 156 / 100 is 1

- Dividing by 0 causes an error when your program runs with integer division. Try floating point division by 0.

# Integer remainder with %

- ▶ The % operator computes the remainder from integer division.

14 % 4      is 2

218 % 5      is 3

$$\begin{array}{r} 3 \\ 4 \overline{) 14} \\ \underline{12} \\ 2 \end{array}$$

$$\begin{array}{r} 43 \\ 5 \overline{) 218} \\ \underline{20} \\ 18 \\ \underline{15} \\ 3 \end{array}$$

What is the result?

45 % 6

2 % 2

8 % 20

11 % 0

- ▶ Applications of % operator:

- Obtain last digit of a number:  $230857 \% 10$  is 7
- Obtain last 4 digits:  $658236489 \% 10000$  is 6489
- See whether a number is odd:  $7 \% 2$  is 1,  $42 \% 2$  is 0

# Clicker 2

► What does each expression evaluate to?

CHOICE	$13 \% 5$	$5 \% 13$	$30 \% 5$
A	3	3	0
B	3	5	0
C	2	5	5
D	2	13	6
E	$2.4$	13	6

# Clicker 3

- ▶ What does the following expression evaluate to?

$$1017 \% 100 + 12 \% 100$$

- A. 10
- B. 17
- C. 12
- D. 22
- E. 29

# Remember PEMDAS?

► **precedence:** Order in which operators are evaluated.

– Generally operators evaluate left-to-right.

$1 - 2 - 3$  is  $(1 - 2) - 3$  which is  $-4$

– But  $*$   $/$   $\%$  have a higher level of precedence than  $+$   $-$

$1 + 3 * 4$  is  $13$

$6 + 8 / 2 * 3$

$6 + 4 * 3$

$6 + 12$  is  $18$

– Parentheses can force a certain order of evaluation:

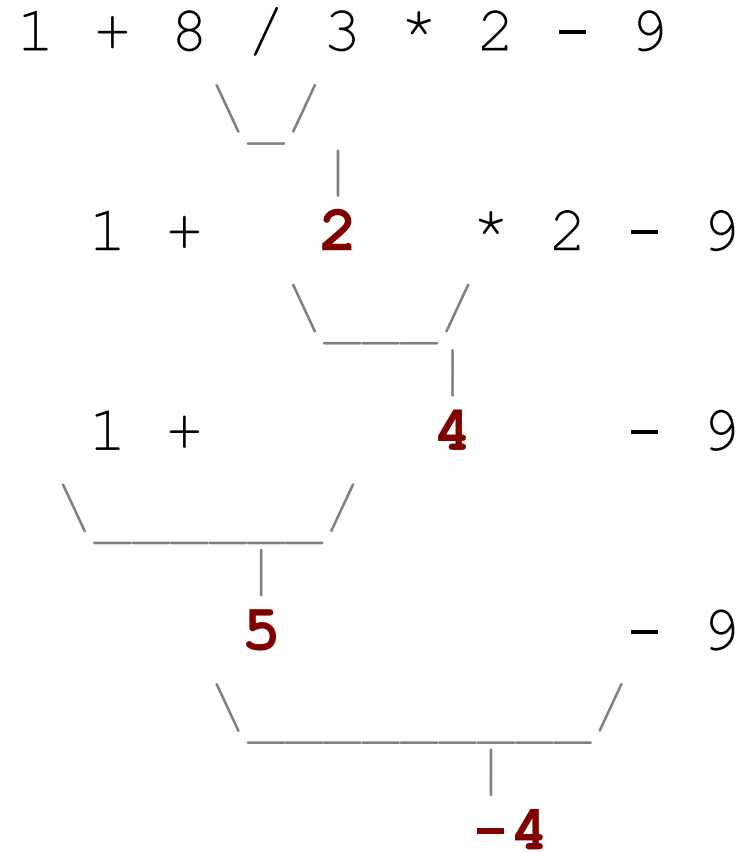
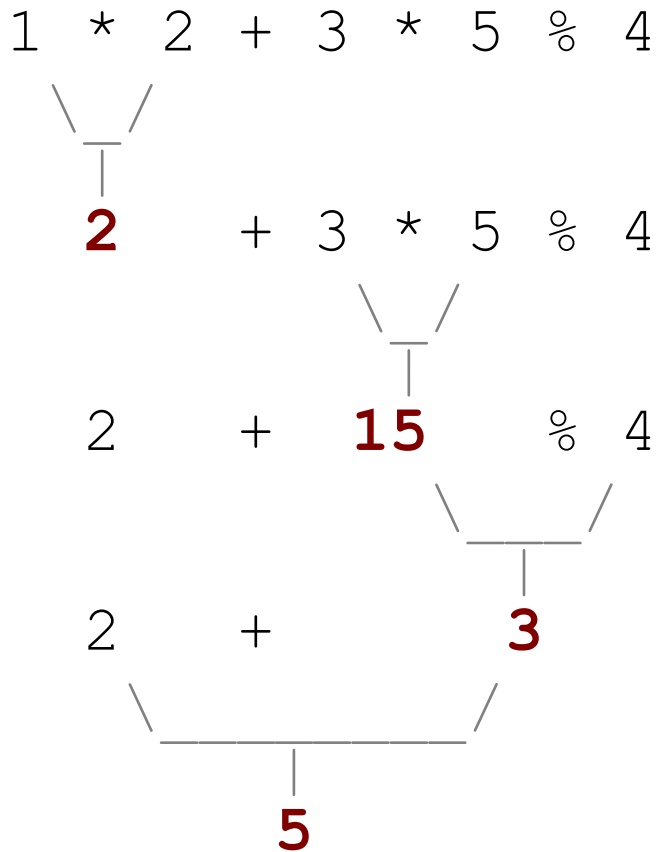
$(1 + 3) * 4$  is  $16$

– Spacing does not affect order of evaluation

$1+3 * 4-2$  is  $11$



# Precedence examples



# Precedence questions

- ▶ What values result from the following expressions?

$$9 / 5$$

$$695 \% 20$$

$$7 + 6 * 5$$

$$7 * 6 + 5$$

$$248 \% 100 / 5$$

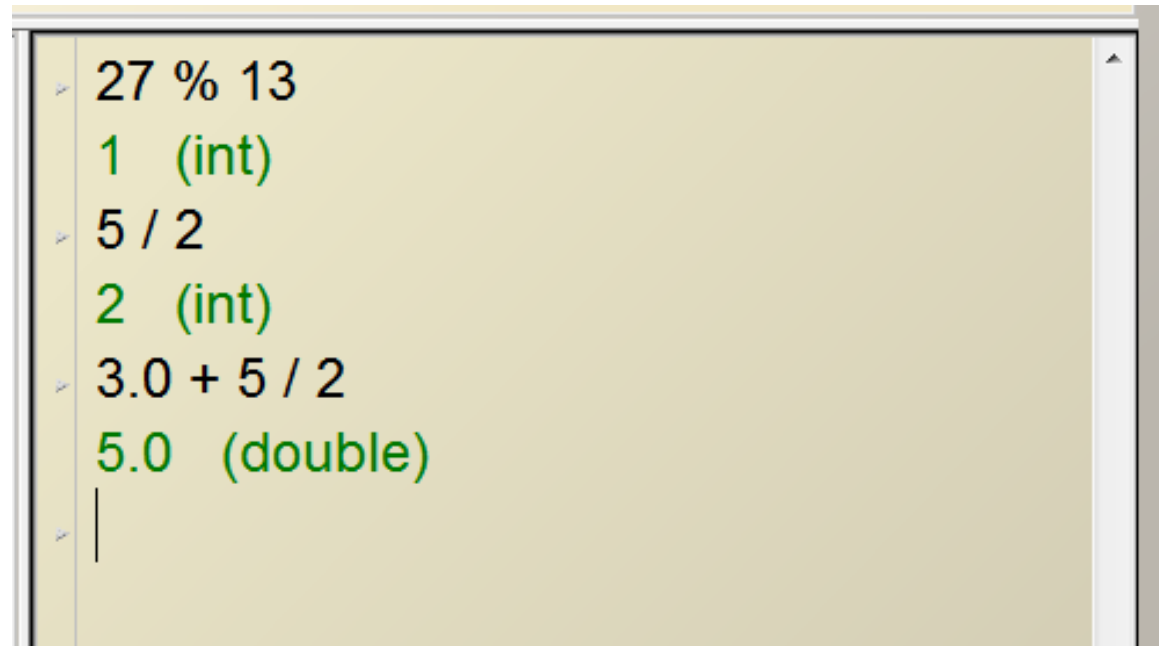
$$6 * 3 - 9 / 4$$

$$(5 - 7) * 4$$

$$6 + (18 \% (17 - 12))$$

# Practice!!

- ▶ BlueJ includes a *Code Pad*
  - *View -> Show Code Pad*
- ▶ *read - eval - print* loop
  - Alternative is JShell
- ▶ Useful to try various expressions

A screenshot of the BlueJ Code Pad window. The window has a light yellow background and a dark border. It contains a list of expressions being evaluated, each preceded by a small blue arrow icon. The expressions and their results are: '27 % 13' (no result), '1 (int)' (result in green), '5 / 2' (no result), '2 (int)' (result in green), '3.0 + 5 / 2' (no result), and '5.0 (double)' (result in green). A vertical cursor is visible at the bottom of the list.

```
> 27 % 13
> 1 (int)
> 5 / 2
> 2 (int)
> 3.0 + 5 / 2
> 5.0 (double)
> |
```

# Real numbers (type double)


- ▶ Examples: `6.022` , `-42.0` , `2.143e17`
  - Placing `.0` or `.` after an integer makes it a `double`.
- ▶ The operators `+` `-` `*` `/` `%` `()` all still work with `double`.
  - `/` produces an exact answer: `15.0 / 2.0` is `7.5`
  - Precedence is the same: `()` before `*` `/` `%` before `+` `-`
  - `%` works with doubles too: `1.25 % 0.75` is `0.5`


# Real number example

2.0 \* 2.4 + 2.25 \* 4.0 / 2.0

  
4.8

+ 2.25 \* 4.0 / 2.0

4.8 +  9.0 / 2.0

4.8 +  4.5

  
9.3

# Precision in real numbers

- ▶ The computer internally represents real numbers in an imprecise way.

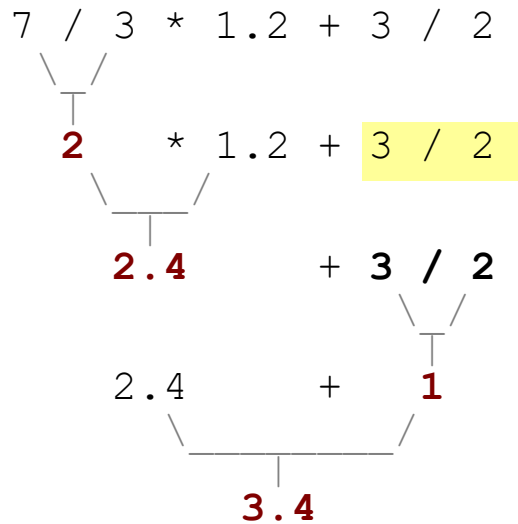
- ▶ Example:

```
System.out.println(0.1 + 0.2);
```

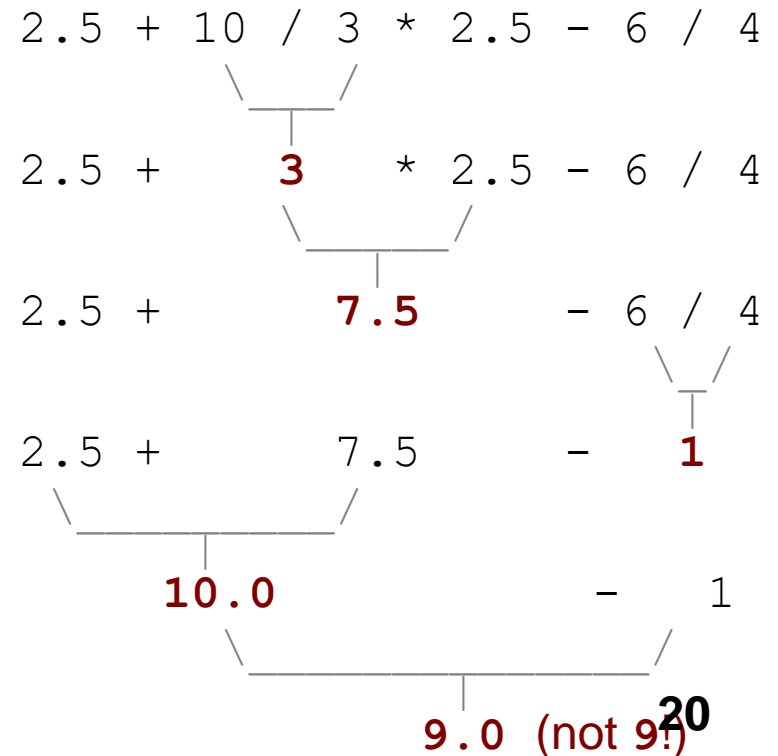
– The output is 0.300000000000000000000004!

# Mixing types

- ▶ When `int` and `double` are mixed, the result is a `double`.
  - `4.2 * 3` is `12.6`
- ▶ The conversion is per-operator, affecting only its operands.



`3 / 2` is `1` above, not `1.5`.



# String concatenation

- ▶ **string concatenation:** Using + between a string and another value to make a longer string.

`"hello" + 42` is `"hello42"`

`1 + "abc" + 2` is `"1abc2"`

`"abc" + 1 + 2` is `"abc12"`

`1 + 2 + "abc"` is `"3abc"`

`"abc" + 9 * 3` is `"abc27"`

`"1" + 1` is `"11"`

`4 - 1 + "abc"` is `"3abc"`

- ▶ Use + to print a string and an expression's value together.

```
System.out.println("Grade: " + (95.1 + 71.9) / 2);
```

- Output: Grade: 83.5



## Clicker 4

What does the following expression evaluate to?

$1.25 + 7 / 4 + \text{"CS"} + 3 + 4$

- A. "3.0CS34"
- B. "2.25CS7"
- C. "2CS7"
- D. "2.25CS34"
- E. Something other than A - D

# Variables

**reading: 2.2**

# Receipt example

What's bad about the following code?

```
public class Receipt {  
  
    public static void main(String[] args) {  
        // Calculate total owed, assuming 8% tax / 15% tip  
        System.out.println("Subtotal:");  
        System.out.println(38 + 40 + 30);  
  
        System.out.println("Tax:");  
        System.out.println((38 + 40 + 30) * .08);  
        System.out.println("Tip:");  
        System.out.println((38 + 40 + 30) * .15);  
        System.out.println("Total:");  
        System.out.println(38 + 40 + 30 +  
                            (38 + 40 + 30) * .08 +  
                            (38 + 40 + 30) * .15);  
    }  
}
```

- The subtotal expression `(38 + 40 + 30)` is repeated
- So many `println` statements

# Variables

- ▶ **variable:** A piece of the computer's memory that is given a name and type, and can store a value.
  - Like preset stations on a car stereo, or cell phone speed dial:



- Steps for using a variable:
  - *Declare* it - state its name and type
  - *Initialize* it - store a value into it
  - *Use* it - print it or use it as part of an expression<sub>25</sub>

# Declaration

- ▶ **variable declaration:** Sets aside memory for storing a value.
  - Variables must be declared before they can be used.

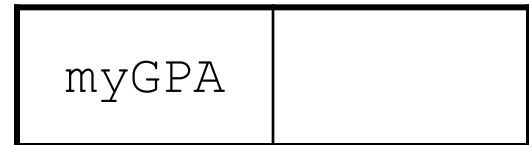
- ▶ Syntax:

***<type>*** ***<name>***;

– `int x;`



– `double myGPA;`



# Assignment

- ▶ **assignment:** Stores a value in a variable.
  - The value is the result of an expression;
  - the variable stores its result.

- ▶ Syntax:

***<name> = <expression>;***

x	3
---	---

```
int x;
```

```
x = 3; // or int x = 3;
```

myGPA	3.25
-------	------

```
double myGPA;
```

```
myGPA = 1.0 + 2.25; // or double myGPA = 3.25
```

# Declaration/initialization

- ▶ A variable can be declared/initialized in one statement.

- ▶ Syntax:

***<type> <name> = <expression>;***

x	14
---	----

```
int x = (11 % 3) + 12;
```

myGPA	3.95
-------	------

```
double myGPA = 3.95;
```

# Using variables

- Once given a value, a variable can be used in expressions:

```
int x = 3;  
System.out.println("x is " + x);           // x is 3  
System.out.println(5 * x - 1);             // 14
```

- You can assign a value more than once:

```
int x = 3;  
System.out.println(x + " here");           // 3 here
```

```
x = 4 + 7;  
System.out.println("now x is " + x);      // now x is 11
```

x	11
---	----



# Assignment vs. algebra

- ▶ Assignment uses `=`, but it is not an algebraic equation.

`=` means, *"store the value at right in variable at left"*

`x = 3;` means, *"x becomes 3" or "x should now store 3"*

- ▶ **ERROR:** `3 = 1 + 2;` is an illegal statement, because 3 is not a variable.

- ▶ What happens here?

```
int x = 3;
```

```
x = x + 2;    // ???
```

x	5
---	---

# Clicker 5

- ▶ What is the output of the following Java code?

```
int x = 3;
```

```
int y = x; // y stores 3
```

```
x = 5; // x now stores 5
```

```
y = y + x;
```

```
System.out.println( x + " " + y );
```

A: "5 8"                      B: 5 10                      C: 10 10

D: 5 + 10                      E: 5 8

# Swapping the Contents of Two Variables

► Output of this code?

```
int x = 12;  
int y = 32;  
x = y;  
y = x;  
System.out.println(x + " " + y);
```

► Output of this code?

```
int x = 12;  
int y = 32;  
int t = x;  
x = y;  
y = t;  
System.out.println(x + " " + y + " " + t);
```

# Assignment and types

- ▶ A variable can only store a value of its own type.

```
int x = 2.5;    // ERROR: incompatible types
```

- ▶ An `int` value can be stored in a `double` variable.
  - The value is converted into the equivalent real number.

```
double myGPA = 4;
```

myGPA	4.0
-------	-----

```
double avg = 11 / 2;
```

avg	5.0
-----	-----

Why does `avg` store 5.0 and not 5.5 ?

# Compiler errors

- ▶ A variable can't be used until it is assigned a value.

```
int x;
```

```
System.out.println(x) ;// ERROR: x has no value
```

- ▶ You may not declare the same variable twice (in the same block of code. methods for now.)

```
int x;
```

```
int x;           // ERROR: x already exists
```

```
int x = 3;
```

```
int x = 5;       // ERROR: x already exists
```

- ▶ How can this code be fixed?

# Printing a variable's value

- ▶ Use + to print a string and a variable's value on one line.

```
double grade = (95.1 + 71.9 + 82.6) / 3.0;  
System.out.println("Your grade was " + grade);
```

```
int students = 11 + 17 + 4 + 19 + 14;  
System.out.println("There are " + students +  
    " students in the course.");
```

- Output:

Your grade was 83.2

There are 65 students in the course.

# Example Problem - BMI

- ▶ **Body Mass Index** or **BMI** is a quick calculation based on height and mass (weight) used by medical professionals to broadly categorize people .

- ▶ Formula:

$$\text{BMI} = \frac{\text{mass}_{\text{kg}}}{\text{height}_{\text{m}}^2} = \frac{\text{mass}_{\text{lb}}}{\text{height}_{\text{in}}^2} \times 703$$

- ▶ Quick tool to get a rough estimate if someone is underweight, normal weight, overweight, or obese
- ▶ Write a program to calculate BMI for a given height and mass.

# Example Problem 2

## - Day of Week

- ▶ For the Gregorian Calendar
- ▶ Given month, day, and year, calculate day of week
- ▶ months, 1 = January, 2 = February, ... 12 = December

$$y = \text{year} - (14 - \text{month}) / 12$$

$$x = y + y / 4 - y / 100 + y / 400$$

$$m = \text{month} + 12 * ((14 - \text{month}) / 12) - 2$$

$$d = (\text{day} + x + (31 * m) / 12) \% 7$$

$$0 = \text{Sunday}, 1 = \text{Monday}, 2 = \text{Tuesday}$$



# Receipt question

Improve the receipt program using variables.

```
public class Receipt {  
  
    public static void main(String[] args) {  
        // Calculate total owed, assuming 8% tax / 15% tip  
        System.out.println("Subtotal:");  
        System.out.println(38 + 40 + 30);  
  
        System.out.println("Tax:");  
        System.out.println((38 + 40 + 30) * .08);  
  
        System.out.println("Tip:");  
        System.out.println((38 + 40 + 30) * .15);  
  
        System.out.println("Total:");  
        System.out.println(38 + 40 + 30 +  
                            (38 + 40 + 30) * .15 +  
                            (38 + 40 + 30) * .08);  
    }  
}
```

# Receipt answer

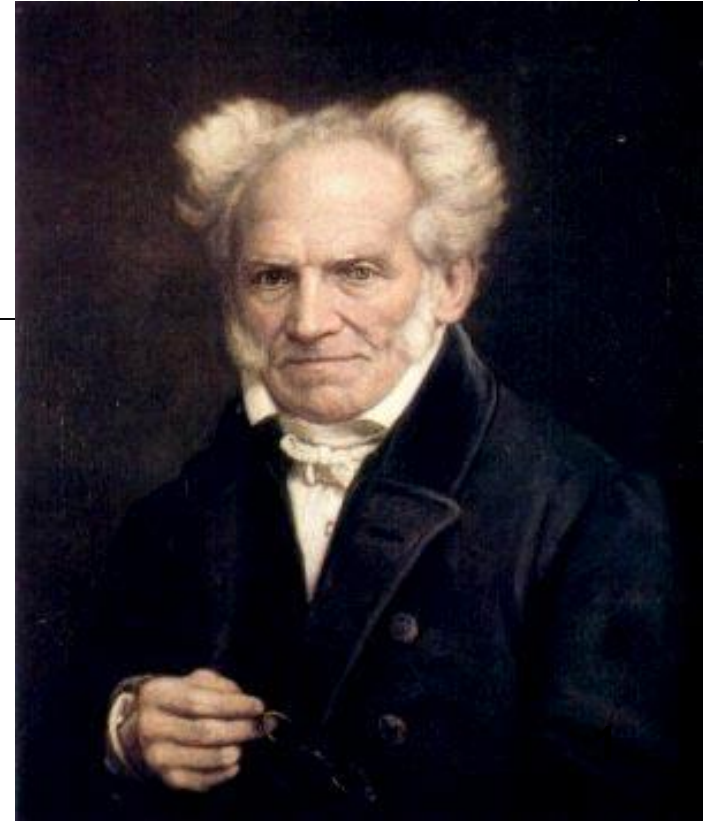
```
public class Receipt {  
  
    public static void main(String[] args) {  
        // Calculate total owed, assuming 8% tax / 15% tip  
        int subtotal = 38 + 40 + 30;  
        double tax = subtotal * .08;  
        double tip = subtotal * .15;  
        double total = subtotal + tax + tip;  
  
        System.out.println("Subtotal: " + subtotal);  
        System.out.println("Tax: " + tax);  
        System.out.println("Tip: " + tip);  
        System.out.println("Total: " + total);  
    }  
}
```

# Topic 5

## for loops and nested loops

“Always to see the general in the particular is the very foundation of genius.”

-Arthur Schopenhauer



Based on slides by Marty Stepp and Stuart Reges  
from <http://www.buildingjavaprograms.com/>

# Repetition with `for` loops

- ▶ So far, repeating a statement is redundant:

```
System.out.println("Mike says:");  
System.out.println("Do Practice-It problems!");  
System.out.println("Do Practice-It problems!");  
System.out.println("Do Practice-It problems!");  
System.out.println("Do Practice-It problems!");  
System.out.println("Do Practice-It problems!");  
System.out.println("It makes a HUGE difference.");
```

- ▶ Java's **`for loop`** statement performs a task many times.

```
System.out.println("Mike says:");  
for (int i = 1; i <= 5; i++) {    // repeat 5 times  
    System.out.println("Do Practice-It problems!");  
}  
System.out.println("It makes a HUGE difference.");2
```

# for loop syntax

```
for ( <initialization>; <test>; <update> ) {  
    <statement>;  
    <statement>;  
    ...  
    <statement>;  
}
```

header

body

- Perform **<initialization>** once.
- Repeat the following:
  - Check if the **<test>** is true. If not, stop.
  - Execute the **<statement>**s.
  - Perform the **<update>**.

# Initialization

```
for (int i = 1; i <= 5; i++) {  
    System.out.println("Do Practice-It!");  
}
```

- ▶ Tells Java compiler what variable to use in the loop
  - Performed once as the loop begins
  - The variable is called a *loop counter* or *loop control variable*
    - can use any name, not just `i`
    - can start at any value, not just `1`

# Test

```
for (int i = 1; i <= 5; i++) {  
    System.out.println("Do Practice-It!");  
}
```

- ▶ Tests the loop counter variable against a limit
  - Uses comparison operators:
    - < less than
    - <= less than or equal to
    - > greater than
    - >= greater than or equal to
    - == equality != not equals

# Body

```
for (int i = 1; i <= 5; i++) {  
    System.out.println("Do Practice-It!");  
}
```

- ▶ If the test is true, the statements in the body of the loop execute in sequential order one time
- ▶ The body of the loop is between the curly braces
- ▶ If the body is one statement the curly braces are not required, but by convention we still add them
- ▶ After the body of the loop completes the update statement is executed.



# Update

```
for (int i = 1; i <= 5; i++) {  
    System.out.println("Do Practice-It!");  
}
```

## ► Perform update step

- Generally adding one to loop control variable
- Could be other operations such as subtracting one, multiplying

# Aside: Increment and Decrement Operators

*shortcuts to increase or decrease a variable's value by 1*

## Shorthand

**<variable>++;**

**<variable>--;**

## Equivalent longer version

**<variable> = <variable> + 1;**

**<variable> = <variable> - 1;**

```
int x = 2;
```

```
x++;
```

```
// x = x + 1;
```

```
// x now stores 3
```

```
double gpa = 2.5;
```

```
gpa--;
```

```
// gpa = gpa - 1;
```

```
// gpa now stores 1.5
```

# Aside: Modify-and-assign operators

*shortcuts to modify a variable's value*

## Shorthand

**<variable> += <exp>;**

**<variable> -= <exp>;**

**<variable> \*= <exp>;**

**<variable> /= <exp>;**

**<variable> %= <exp>;**

## Equivalent longer version

**<variable> = <variable> + (<exp>);**

**<variable> = <variable> - (<exp>);**

**<variable> = <variable> \* (<exp>);**

**<variable> = <variable> / (<exp>);**

**<variable> = <variable> % (<exp>);**

**x += 3;**

**// x = x + 3;**

**gpa -= 0.5;**

**// gpa = gpa - 0.5;**

**number \*= 2 + 1;    // number = number \* (2 + 1);**

# Clicker 1

► What is output by the following code?

```
int x = 2;  
int y = 5;  
x *= 3 + y + x;  
System.out.println(x + " " + y);
```

**A.** 20 5

**B.** 2 5

**C.** 13 5

**D.** 20 10

**E.** Something other than A - D

# `for` loop is **NOT** a method

- ▶ The `for` loop is a ***control structure***
  - a syntactic structure that *controls* the execution of other statements.
- ▶ Example:
  - “Shampoo hair. Rinse. **Repeat.**”

# Repetition over a range

```
System.out.println("1 squared = " + 1 * 1);  
System.out.println("2 squared = " + 2 * 2);  
System.out.println("3 squared = " + 3 * 3);  
System.out.println("4 squared = " + 4 * 4);  
System.out.println("5 squared = " + 5 * 5);  
System.out.println("6 squared = " + 6 * 6);
```

– Intuition: "I want to print a line for each number from 1 to 6"

► The `for` loop does exactly that!

```
for (int i = 1; i <= 6; i++) {  
    System.out.println(i + " squared = " + (i * i));  
}
```

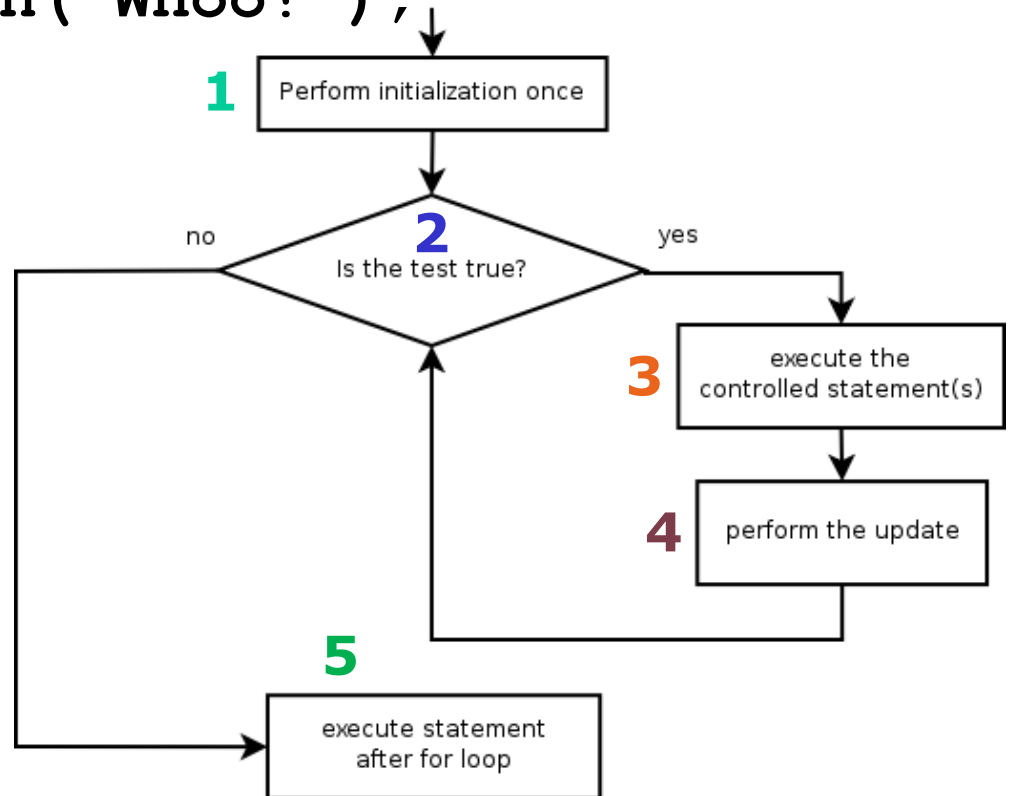
– "For each integer `i` from 1 through 6, print ..."

# Loop walkthrough

```
for (int i = 1; i <= 4; i++) {  
    System.out.println(i + " squared = " + (i * i));  
}  
System.out.println("Whoo!");
```

Output:

```
1 squared = 1  
2 squared = 4  
3 squared = 9  
4 squared = 16  
Whoo!
```



# Simple Loop Example

- ▶ Write a program to calculate and print out the values of  $N!$  from 1 to 50 using a for loop
- ▶  $0! = 1$
- ▶  $1! = 1 * 0! = 1 * 1 = 1$
- ▶  $2! = 2 * 1! = 2 * 1 * 1 = 2$
- ▶  $3! = 3 * 2! = 3 * 2 * 1 * 1 = 6$
- ▶  $4! = 4 * 3! = 4 * 3 * 2 * 1 * 1 = 24$



# Multi-line loop body

```
System.out.println("+-----+");  
for (int i = 1; i <= 3; i++) {  
    System.out.println("\\      /");  
    System.out.println("/      \\");  
}  
System.out.println("+-----+");
```

Output:

```
+-----+  
\\      /  
/      \  
\\      /  
/      \  
\\      /  
/      \  
+-----+
```

# Expressions for counter

```
int highTemp = 5;  
for (int i = -3; i <= highTemp / 2; i++) {  
    System.out.println(i * 1.8 + 32);  
}
```

- This computes the Fahrenheit equivalents for -3 degrees Celsius to 2 degrees Celsius.

Output:

26.6  
28.4  
30.2  
32.0  
33.8  
35.6

# System.out.print

- ▶ Prints without moving to a new line
  - allows you to print partial messages on the same line

```
int highestTemp = 5;  
for (int i = -3; i <= highestTemp / 2; i++) {  
    System.out.print((i * 1.8 + 32) + " ");  
}
```

- Output:

26.6    28.4    30.2    32.0    33.8    35.6

- Concatenate " " to separate the numbers

## Clicker 2

- ▶ How many asterisks are output by the following code?

```
for (int i = -2; i <= 13; i++) {  
    System.out.print("*");  
    System.out.print("**");  
}
```

A. 0

B. 15

C. 45

D. 48

E. 68

# Counting down

- ▶ The **<update>** can use -- to make the loop count down.

- The **<test>** must say > instead of < (or logic error)

```
System.out.print("T-minus ");  
for (int i = 10; i >= 1; i--) {  
    System.out.print(i + ", ");  
}  
System.out.println("blastoff!");  
System.out.println("The end.");
```

Output:

```
T-minus 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, blastoff!  
The end.
```

# Practice Problem

- ▶ Newton's method for approximating square roots adapted from the Dr. Math website

The goal is to find the square root of a number. Let's call it num

1. Choose a rough approximation of the square root of num, call it approx.

How to choose?

2. Divide num by approx and then average the quotient with approx,

in other words we want to evaluate the

expression  $((\text{num}/\text{approx}) + \text{approx}) / 2$

3. How close are we? In programming we would store the result of the expression back into the variable approx.

4. How do you know if you have the right answer?

# Sample of Newton's Method

num	approx	$((\text{num}/\text{approx}) + \text{approx})/2$	$\text{approx} * \text{approx}$
12	6	$(12 / 6 + 6) / 2 = 4$	16
12	4	$(12 / 4 + 4) / 2 = 3.5$	12.25
12	3.5	$(12 / 3.5 + 3.5) / 2 = 3.4642857...$	12.0012..
12	3.4642857	$= 3.46410162...$	12.00000003
12	3.46410162	$= 3.46410161...$	11.999999999

3.4641016151377544      after 5 steps

3.4641016151377545870548926830117 (from calculator)

# Nested loops

**reading: 2.3**



# Nested loops

- ▶ **nested loop:** A loop placed inside another loop.

```
for (int i = 1; i <= 5; i++) {  
    for (int j = 1; j <= 10; j++) {  
        System.out.print("*");  
    }  
    System.out.println();    // to end the line  
}
```

- ▶ **Output:**

```
*****  
*****  
*****  
*****  
*****
```

- ▶ The outer loop repeats 5 times; the inner one 10 times.
  - "sets and reps" exercise analogy

# Nested for loop exercise

- What is the output of the following nested for loops?

```
for (int i = 1; i <= 5; i++) {  
    for (int j = 1; j <= i; j++) {  
        System.out.print("*");  
    }  
    System.out.println();  
}
```

- Output:

```
*  
**  
***  
****  
*****
```

# Nested for loop exercise

- What is the output of the following nested for loops?

```
for (int i = 1; i <= 5; i++) {  
    for (int j = 1; j <= i; j++) {  
        System.out.print(i);  
    }  
    System.out.println();  
}
```

- Output:

```
1  
22  
333  
4444  
55555
```

# Clicker 3

► What is output by the following code?

```
int total = 0;
for(int i = 1; i <= 4; i++) {
    for(int j = 1; j <= i; j++) {
        total += i;
    }
}
System.out.println(total);
```

A. 4

B. 10

C. 16

D. 24

E. 30

# Common errors

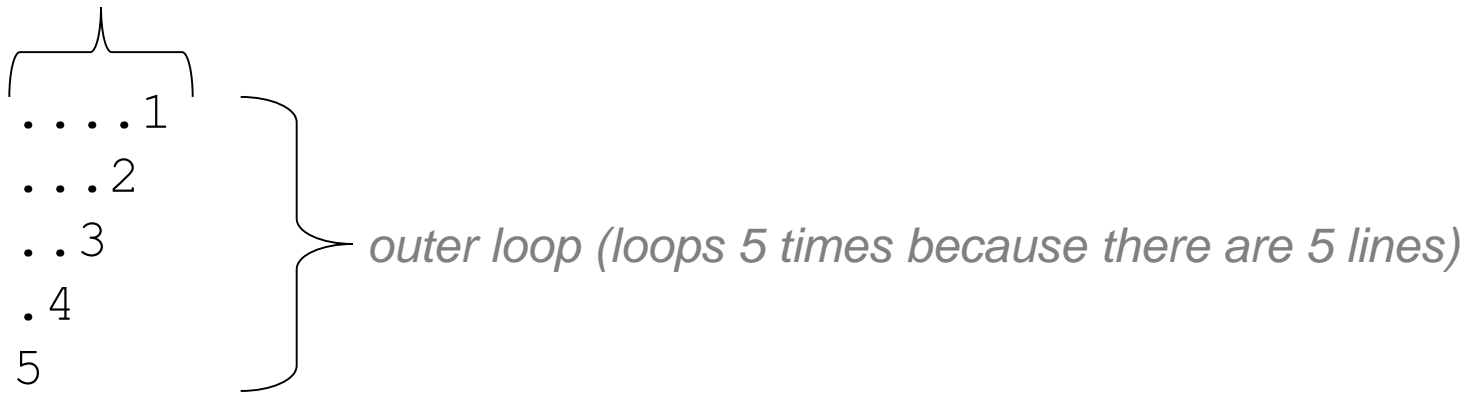
- ▶ Both of the following sets of code produce *infinite loops*:

```
for (int i = 1; i <= 5; i++) {  
    for (int j = 1; i <= 10; j++) {  
        System.out.print("*");  
    }  
    System.out.println();  
}
```

```
for (int i = 1; i <= 5; i++) {  
    for (int j = 1; j <= 10; i++) {  
        System.out.print("*");  
    }  
    System.out.println();  
}
```

# Complex output

- Write a nested `for` loop to produce the following output.  
*inner loop (repeated characters on each line)*



The diagram illustrates the output pattern with annotations. On the left, the output is shown as five lines of dots followed by a number: `.....1`, `....2`, `...3`, `..4`, and `.5`. A horizontal curly brace above the first line indicates the inner loop, which repeats characters on each line. A vertical curly brace to the right of the five lines indicates the outer loop, which loops 5 times because there are 5 lines.

- We must build multiple complex lines of output using:
  - an *outer "vertical" loop* for each of the lines
  - *inner "horizontal" loop(s)* for the patterns within each line

# Outer and inner loop

- ▶ First write the outer loop, from 1 to the number of lines.

```
for (int line = 1; line <= 5; line++) {  
    ...  
}
```

- ▶ Now look at the line contents. Each line has a pattern:
  - some dots (0 dots on the last line), then a number

....1

...2

..3

.4

5

- Observation: the number of dots is related to the line number.

# Mapping loops to numbers

```
for (int count = 1; count <= 5;
    count++) {
    System.out.print( ... );
}
```

- What statement in the body would cause the loop to print:

**4 7 10 13 16**

```
for (int count = 1; count <= 5; count++) {
    System.out.print(3 * count + 1 + " ");
}
```



# Loop tables

- ▶ What statement in the body would cause the loop to print:

2 7 12 17 22

- ▶ To see patterns, make a table of `count` and the numbers.
  - Each time `count` goes up by 1, the number should go up by 5.
  - But `count * 5` is too great by 3, so we subtract 3.

<code>count</code>	number to print	<code>5 * count</code>	<code>5 * count - 3</code>
1	2	5	2
2	7	10	7
3	12	15	12
4	17	20	17
5	22	25	22

# Loop tables question

- ▶ What statement in the body would cause the loop to print:

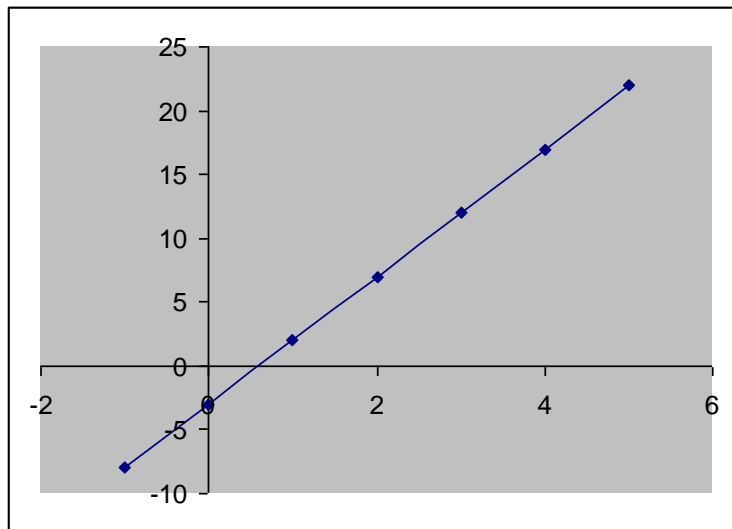
17 13 9 5 1

- Let's create the loop table together.
  - Each time `count` goes up 1, the number printed should ...
  - But this multiple is off by a margin of ...

count	number to print	<code>-4 * count</code>	<code>-4 * count + 21</code>
1	17	-4	17
2	13	-8	13
3	9	-12	9
4	5	-16	5
5	1	-20	1

# Another view: Slope-intercept

- ▶ The next three slides present the mathematical basis for the loop tables.



count (x)	number to print (y)
1	2
2	7
3	12
4	17
5	22

# Another view: Slope-intercept

- ▶ *Caution:* This is algebra, not assignment!
- ▶ Recall: slope-intercept form ( $y = mx + b$ )
- ▶ Slope is defined as “rise over run” (i.e. rise / run). Since the “run” is always 1 (we increment along  $x$  by 1), we just need to look at the “rise”. The rise is the difference between the  $y$  values. Thus, the slope ( $m$ ) is the difference between  $y$  values; in this case, it is +5.
- ▶ To compute the  $y$ -intercept ( $b$ ), plug in the value of  $y$  at  $x = 1$  and solve for  $b$ . In this case,  $y = 2$ .

$$y = m * x + b$$

$$2 = 5 * 1 + b$$

$$\text{Then } b = -3$$

- ▶ So the equation is

$$y = m * x + b$$

$$y = 5 * x - 3$$

$$y = 5 * \text{count} - 3$$

count (x)	number to print (y)
1	2
2	7
3	12
4	17
5	22

# Another view: Slope-intercept

- ▶ Algebraically, if we always take the value of  $y$  at  $x = 1$ , then we can solve for  $b$  as follows:
$$y = m * x + b$$
$$y_1 = m * 1 + b$$
$$y_1 = m + b$$
$$b = y_1 - m$$
- ▶ In other words, to get the  $y$ -intercept, just subtract the slope from the first  $y$  value ( $b = 2 - 5 = -3$ )
  - This gets us the equation
$$y = m * x + b$$
$$y = 5 * x - 3$$
$$y = 5 * \text{count} - 3$$
(which is exactly the equation from the previous slides)

# Nested for loop exercise

- ▶ Make a table to represent any patterns on each line.

.....1

...2

..3

.4

5

line	# of dots	$-1 * line$	$-1 * line + 5$
1	4	-1	4
2	3	-2	3
3	2	-3	2
4	1	-4	1
5	0	-5	0

- ▶ To print a character multiple times, use a `for` loop.

```
for (int j = 1; j <= 4; j++) {  
    System.out.print(".");           // 4 dots  
}
```

# Nested for loop solution

## ► Answer:

```
for (int line = 1; line <= 5; line++) {  
    for (int j = 1; j <= (-1 * line + 5); j++) {  
        System.out.print(".");  
    }  
    System.out.println(line);  
}
```

## ► Output:

```
.....1  
....2  
...3  
..4  
.5  
5
```

# Nested for loop exercise

- What is the output of the following nested for loops?

```
for (int line = 1; line <= 5; line++) {  
    for (int j = 1; j <= (-1 * line + 5); j++) {  
        System.out.print(".");  
    }  
    for (int k = 1; k <= line; k++) {  
        System.out.print(line);  
    }  
    System.out.println();  
}
```

- Answer:

```
....1  
...22  
..333  
.4444  
55555
```



# Nested for loop exercise

- Modify the previous code to produce this output:

```
....1
...2.
..3..
.4...
5....
```

```
for (int line = 1; line <= 5; line++) {  
    for (int j = 1; j <= (-1 * line + 5); j++) {  
        System.out.print(".");  
    }  
    System.out.print(line);  
    for (int j = 1; j <= (line - 1); j++) {  
        System.out.print(".");  
    }  
    System.out.println();  
}
```

# Topic 6

## loops, figures, constants

"Complexity has and will maintain a strong fascination for many people. It is true that we live in a complex world and strive to solve inherently complex problems, which often do require complex mechanisms. However, this should not diminish our desire for elegant solutions, which convince by their clarity and effectiveness. Simple, elegant solutions are more effective, but they are harder to find than complex ones, and they require more time, which we too often believe to be unaffordable."

-Niklaus Wirth

Based on slides by Marty Stepp and Stuart Reges  
from <http://www.buildingjavaprograms.com/>



# Clicker 1

► What is the base 10 equivalent of the base 2 number 1011001

**A. 27**

**B. 89**

**C. 93**

**D. 127**

**E. 1011001**

# Clicker 2

► What does  $5!$  equal?

A. 5

B. 32

C. 120

D. 3125

E. a lot

# Clicker 3

► Which of the following is closest to the value that overflows the Java int data type when calculating  $N!$

A. 1

B. 15

C. 60

D. 100

E. 1000

# Drawing complex figures

- ▶ Use nested `for` loops to produce the following output.
- ▶ Why draw ASCII art?
  - Real graphics require more finesse
  - ASCII art has complex patterns
  - Can focus on the algorithms

```
#=====#
|          <><>          |
|        <>...<>        |
|      <>.....<>      |
| <>.....<>          |
| <>.....<>          |
|      <>.....<>      |
|        <>...<>        |
|          <><>          |
|                                     5|
#=====#
```

# Development strategy

- Recommendations for managing complexity:
  1. Design the program (think about steps or methods needed).
    - write an English description of steps required
    - use this description to decide the methods

## 2. Create a table for patterns of characters

- use tables to write your `for` loops

#=====										#
								<><>		
								<> . . . . <>		
								<> . . . . . <>		
								<> . . . . . <>		
								<> . . . . . <>		
								<> . . . . . <>		
								<> . . . . . <>		
								<> . . . . . <>		
								<><>		
#=====										#

# 1. Pseudo-code

- ▶ **pseudo-code:** An English description of an algorithm.
- ▶ Example: Drawing a 12 wide by 7 tall box of stars

```
print 12 stars.  
for (each of 5 lines) {  
    print a star.  
    print 10 spaces.  
    print a star.  
}  
print 12 stars.
```

```
* * * * * * * * * * * *  
*                               *  
*                               *  
*                               *  
*                               *  
*                               *  
* * * * * * * * * * * *
```



# Pseudo-code algorithm

## 1. Line

- # , 16 =, #

## 2. Top half

- |
- spaces (decreasing)
- <>
- dots (increasing)
- <>
- spaces (same as above)
- |

## 3. Bottom half (top half upside-down)

## 4. Line

- # , 16 =, #

```
#=====#
|           <><>           |
|         <>...<>         |
|      <>.....<>      |
| <>.....<> |
| <>.....<> |
|   <>.....<>   |
|     <>...<>     |
|           <><>           |
#=====#
```

# Methods from pseudocode

```
public class Mirror {
    public static void main(String[] args) {
        line();
        topHalf();
        bottomHalf();
        line();
    }

    public static void topHalf() {
        for (int line = 1; line <= 4; line++) {
            // contents of each line
        }
    }

    public static void bottomHalf() {
        for (int line = 1; line <= 4; line++) {
            // contents of each line
        }
    }

    public static void line() {
        // ...
    }
}
```

## 2. Tables

- ▶ A table for the top half:
  - Compute spaces and dots expressions from line number

line	spaces	$-2 * \text{line} + 8$	dots	$4 * \text{line} - 4$
1	6	6	0	0
2	4	4	4	4
3	2	2	8	8
4	0	0	12	12

# 3. Writing the code

- ▶ Useful questions about the top half:
  - What methods? (think structure and redundancy)
  - Number of (nested) loops per line?

```
#=====#  
|          <><>          |  
|        <>...<>        |  
|      <>.....<>      |  
| <>.....<> |  
| <>.....<> |  
|  <>.....<>  |  
|   <>...<>   |  
|    <><>    |  
#=====#
```

# Partial solution

```
// Prints the expanding pattern of <> for
// the top half of the figure.
public static void topHalf() {
    for (int line = 1; line <= 4; line++) {
        System.out.print("|");

        for (int space = 1; space <= (line * -2 + 8); space++) {
            System.out.print(" ");
        }

        System.out.print("<>");

        for (int dot = 1; dot <= (line * 4 - 4); dot++) {
            System.out.print(".");
        }

        System.out.print("<>");

        for (int space = 1; space <= (line * -2 + 8); space++) {
            System.out.print(" ");
        }

        System.out.println("|");
    }
}
```

# Class constants and scope

**reading: 2.4**

# Scaling the mirror

- ▶ Modify the Mirror program so that it can scale.
  - The current mirror (left) is at size 4; the right is at size 3.
- ▶ We'd like to structure the code so we can scale the figure by changing the code in just one place.

```
#=====#  
|          <><>          |  
|      <>...<>      |  
|  <>.....<>  |  
|<>.....<>|  
|<>.....<>|  
|  <>.....<>  |  
|      <>...<>      |  
|          <><>          |  
#=====#
```

```
#=====#  
|          <><>          |  
|      <>...<>      |  
|<>.....<>|  
|<>.....<>|  
|  <>...<>  |  
|          <><>          |  
#=====#
```

# Limitations of variables

- ▶ Idea: Make a variable to represent the size.
  - Use the variable's value in the methods.
- ▶ Problem: A variable in one method can't be seen in others.

```
public static void main(String[] args) {  
    int size = 4;  
    topHalf();  
    printBottom();  
}  
  
public static void topHalf() {  
    for (int i = 1; i <= size; i++) { // ERROR: size not found  
        ...  
    }  
}  
  
public static void bottomHalf() {  
    for (int i = size; i >= 1; i--) { // ERROR: size not found  
        ...  
    }  
}
```



# Scope

- ▶ **scope**: The part of a program where a variable exists.
  - From its declaration to the end of the { } braces
    - A variable declared in a `for` loop exists only in that loop.
    - A variable declared in a method exists only in that method.

```
public static void example() {  
    int x = 3;  
    for (int i = 1; i <= 10; i++) {  
        System.out.println(x);  
    }  
    // i no longer exists here  
} // x ceases to exist here
```

*i's scope* { } *x's scope* { }

# Scope implications

- ▶ Variables whose scope does NOT overlap can have same name.

```
for (int i = 1; i <= 100; i++) {  
    System.out.print("/");  
}  
for (int i = 1; i <= 100; i++) {    // OK  
    System.out.print("\\");  
}  
int i = 5;                        // OK: outside of loop's scope
```

- ▶ A variable can't be declared twice or used out of its scope.

```
for (int i = 1; i <= 100 * line; i++) {  
    int i = 2;                        // ERROR: overlapping scope  
    System.out.print("/");  
}  
i = 4;                                // ERROR: outside scope
```

# Class constants

- ▶ **class constant:** A fixed value visible to the whole program.
  - value can be set only at declaration; cannot be reassigned, hence the name: *constant*

- ▶ **Syntax:**

```
public static final <type> <name> = <exp>;
```

- name in ALL\_UPPER\_CASE by convention

- **Examples:**

```
public static final int DAYS_IN_WEEK = 7;
```

```
public static final double INTEREST_RATE = 0.5;
```

```
public static final int SSN = 658234569;
```

# Constants and figures

- ▶ Consider the task of drawing the following scalable figure:

Multiples of 5 occur many times

$$\begin{array}{ccccccc} + & / & \backslash & / & \backslash & / & \backslash & + \\ | & & & & & & & | \\ | & & & & & & & | \\ + & / & \backslash & / & \backslash & / & \backslash & + \end{array}$$

## The same figure at size 2

# Repetitive figure code

```
public class Sign {  
  
    public static void main(String[] args) {  
        drawLine();  
        drawBody();  
        drawLine();  
    }  
  
    public static void drawLine() {  
        System.out.print("+");  
        for (int i = 1; i <= 10; i++) {  
            System.out.print("/\\");  
        }  
        System.out.println("+");  
    }  
  
    public static void drawBody() {  
        for (int line = 1; line <= 5; line++) {  
            System.out.print("|");  
            for (int spaces = 1; spaces <= 20; spaces++) {  
                System.out.print(" ");  
            }  
            System.out.println("|");  
        }  
    }  
}
```

# Adding a constant

```
public class Sign {
    public static final int HEIGHT = 5;

    public static void main(String[] args) {
        drawLine();
        drawBody();
        drawLine();
    }

    public static void drawLine() {
        System.out.print("+");
        for (int i = 1; i <= HEIGHT * 2; i++) {
            System.out.print("/\\");
        }
        System.out.println("+");
    }

    public static void drawBody() {
        for (int line = 1; line <= HEIGHT; line++) {
            System.out.print("|");
            for (int spaces = 1; spaces <= HEIGHT * 4; spaces++) {
                System.out.print(" ");
            }
            System.out.println("|");
        }
    }
}
```

# Complex figure w/ constant

- Modify the Mirror code to be resizable using a constant.

A mirror of size 4:

```
#=====#  
|           <><>           |  
|       <> . . . . <>       |  
|   <> . . . . . . . . <>   |  
| <> . . . . . . . . . . <> |  
| <> . . . . . . . . . . <> |  
|   <> . . . . . . . . <>   |  
|       <> . . . . <>       |  
|           <><>           |  
#=====#
```

A mirror of size 3:

```
#=====#  
|           <><>           |  
|       <> . . . . <>       |  
| <> . . . . . . . . <> |  
| <> . . . . . . . . <> |  
|       <> . . . . <>       |  
|           <><>           |  
#=====#
```

# Clicker 4

► Should every instance of the literal int 4 and multiples of 4 in the program be replaced with the class constant SIZE?

A. No

B. Yes



# Using a constant

- Constant allows many methods to refer to same value:

```
public static final int SIZE = 4;
```

```
public static void main(String[] args) {  
    topHalf();  
    printBottom();  
}
```

```
public static void topHalf() {  
    for (int i = 1; i <= SIZE; i++) {           // OK  
        ...  
    }  
}
```

```
public static void bottomHalf() {  
    for (int i = SIZE; i >= 1; i--) {           // OK  
        ...  
    }  
}
```

# Loop tables and constant

- ▶ Let's modify our loop table to use `SIZE`
  - This can change the amount added in the loop expression

SIZE	line	spaces	$-2*\text{line} + (2*SIZE)$	dots	$4*\text{line} - 4$
4	1,2,3,4	6,4,2,0	$-2*\text{line} + \mathbf{8}$	0,4,8,12	$4*\text{line} - 4$
3	1,2,3	4,2,0	$-2*\text{line} + \mathbf{6}$	0,4,8	$4*\text{line} - 4$

```
#=====#
|           |
|      <><>  |
|    <>....<>  |
|  <>.....<>  |
|<>.....<>  |
|<>.....<>  |
|  <>.....<>  |
|    <>....<>  |
|      <><>  |
|           |
#=====#
```

```
#=====#
|           |
|      <><>  |
|    <>....<>  |
|  <>.....<>  |
|  <>.....<>  |
|    <>....<>  |
|      <><>  |
#=====#
```

# Partial solution

```
public static final int SIZE = 4;

// Prints the expanding pattern of <> for the top half of the figure.
public static void topHalf() {
    for (int line = 1; line <= SIZE; line++) {
        System.out.print("|");

        for (int space = 1; space <= (line * -2 + (2*SIZE)); space++)
        {
            System.out.print(" ");
        }

        System.out.print("<>");

        for (int dot = 1; dot <= (line * 4 - 4); dot++) {
            System.out.print(".");
        }

        System.out.print("<>");

        for (int space = 1; space <= (line * -2 + (2*SIZE)); space++)
        {
            System.out.print(" ");
        }

        System.out.println("|");
    }
}
```

# Observations about constant

- ▶ The constant can change the "intercept" in an expression.

- Usually the "slope" is unchanged.

```
public static final int SIZE = 4;

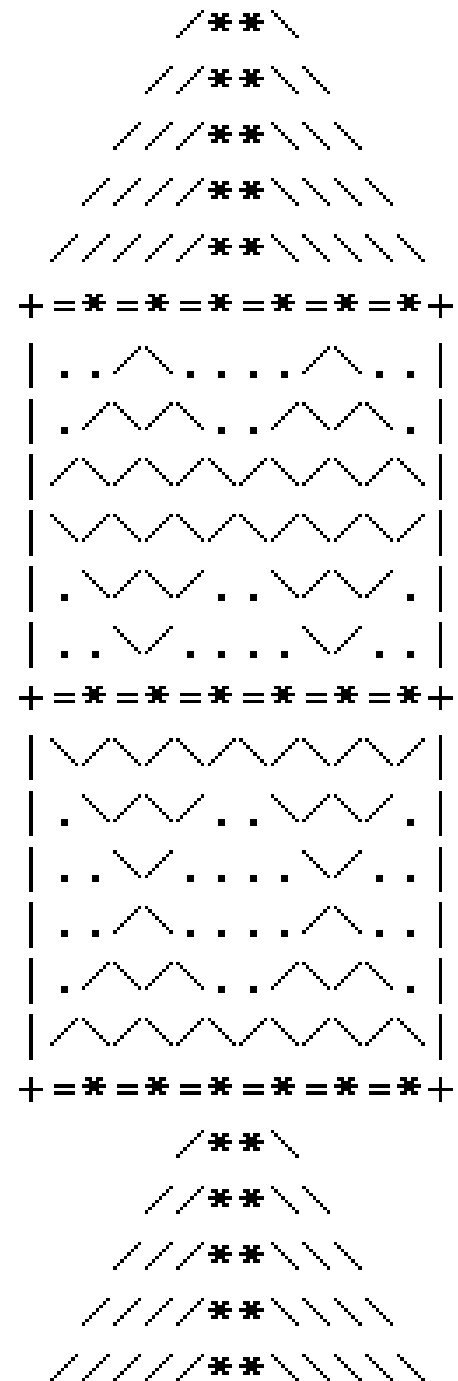
for (int space = 1; space <= (line * -2 + (2 * SIZE));
    space++) {
    System.out.print(" ");
}
```

- ▶ It doesn't replace *every* occurrence of the original value.

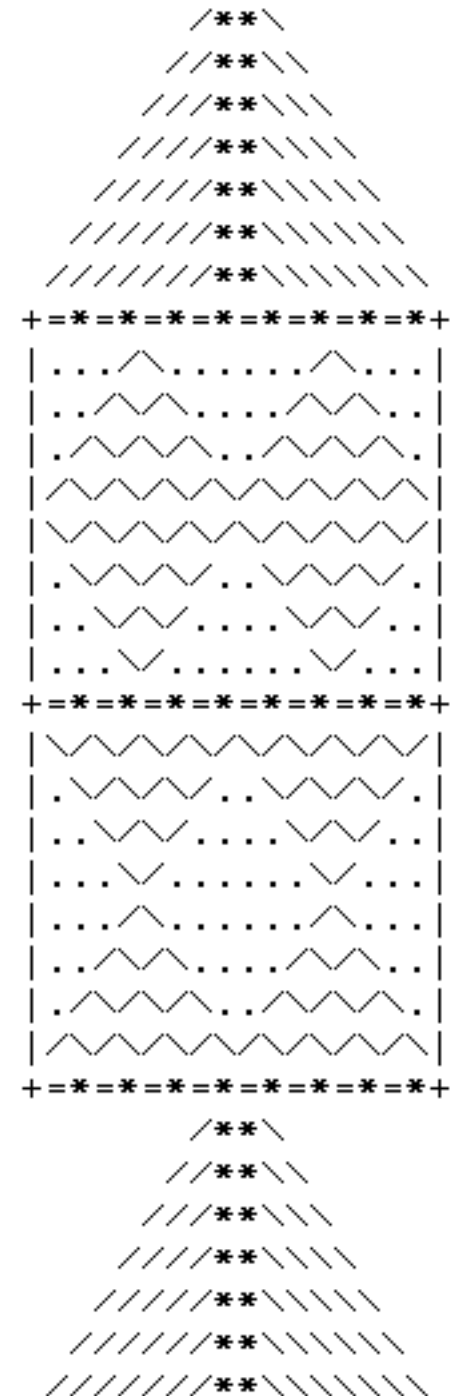
```
for (int dot = 1; dot <= (line * 4 - 4); dot++) {
    System.out.print(".");
}
```

# Another Example

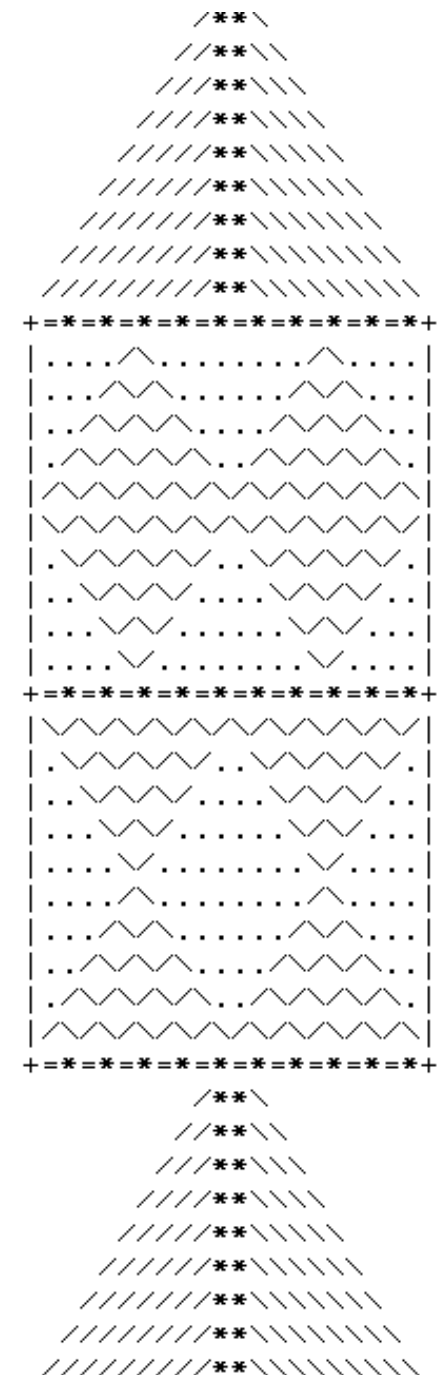
- ▶ Create a program to produce the following ASCII art rocket
- ▶ SIZE = 3 for the rocket to the right



SIZE = 4 Rocket



SIZE = 5 Rocket



# Assignment 2: ASCII Art





# Topic 7

## parameters

"We're flooding people with information. We need to feed it through a processor. A human must turn information into intelligence or knowledge. ***We've tended to forget that no computer will ever ask a new question.***"

— Rear Admiral Grace Murray Hopper "



# Redundant recipes

- ▶ Recipe for baking **20** cookies:
  - Mix the following ingredients in a bowl:
    - **4** cups flour
    - **1** cup butter
    - **1** cup sugar
    - **2** eggs
    - **40** oz. chocolate chips ...
  - Place on sheet and Bake for about **10** minutes.
  
- ▶ Recipe for baking **40** cookies:
  - Mix the following ingredients in a bowl:
    - **8** cups flour
    - **2** cups butter
    - **2** cups sugar
    - **4** eggs
    - **80** oz. chocolate chips ...
  - Place on sheet and Bake for about **10** minutes.

# Parameterized recipe

- ▶ Recipe for baking **20** cookies:
  - Mix the following ingredients in a bowl:
    - 4 cups flour
    - 1 cup sugar
    - 2 eggs
    - ...
- ▶ Recipe for baking **N** cookies:
  - Mix the following ingredients in a bowl:
    - **N/5** cups flour
    - **N/20** cups butter
    - **N/20** cups sugar
    - **N/10** eggs
    - **2N** oz. chocolate chips ...
  - Place on sheet and Bake for about 10 minutes.
- ▶ **parameter**: A value that distinguishes similar tasks.

# Redundant figures

- Consider the task of printing the following lines/boxes:

\* \* \* \* \*

\* \* \* \* \*

\* \* \* \* \*

\* \* \* \* \*

\* \* \*

\* \* \* \* \*

\* \* \* \* \*

\* \*

\* \*

\* \* \* \* \*

# A redundant solution

```
public class Stars1 {
    public static void main(String[] args) {
        lineOf13();
        lineOf7();
        lineOf35();
        box10x3();
        box5x4();
    }

    public static void lineOf13() {
        for (int i = 1; i <= 13; i++) {
            System.out.print("*");
        }
        System.out.println();
    }

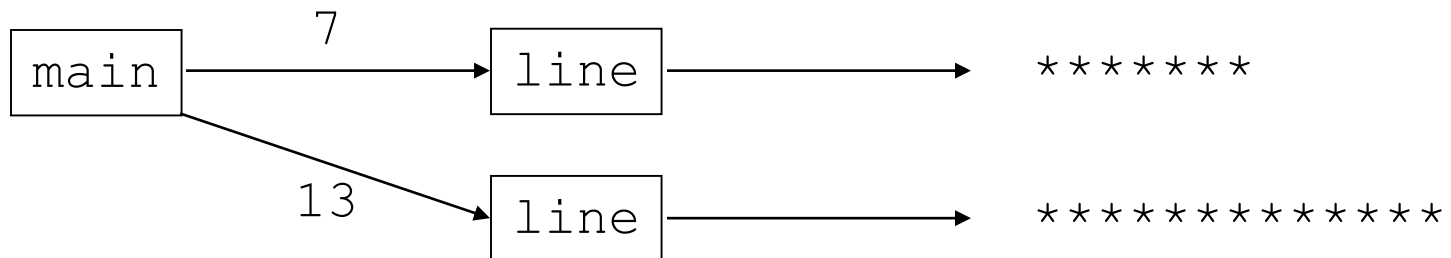
    public static void lineOf7() {
        for (int i = 1; i <= 7; i++) {
            System.out.print("*");
        }
        System.out.println();
    }

    public static void lineOf35() {
        for (int i = 1; i <= 35; i++) {
            System.out.print("*");
        }
        System.out.println();
    }
    ...
}
```

- This code is redundant.
- Would variables help?  
Would constants help?
- What is a better solution?
- `line` - A method to draw a line of any number of stars.
- `box` - A method to draw a box of any size.

# Parameterization

- ▶ **parameter:** A value passed to a method by its caller.
  - Instead of `lineOf7`, `lineOf13`, write `line` to draw any length.
    - When *declaring* the method, we will state that it requires a parameter for the number of stars.
    - When *calling* the method, we will specify how many stars to draw.



# Declaring a parameter

*Stating that a method requires a parameter in order to run*

```
public static void <name> (<type> <name>) {  
    <statement>(s);  
}
```

## ► Example:

```
public static void sayPassword(int code) {  
    System.out.println("The password is: " + code);  
}
```

- When `sayPassword` is called, the caller must specify the integer code to print.

# Passing a parameter

*Calling a method and specifying values for its parameters*

**<name> (<expression>) ;**

## ► Example:

```
public static void main(String[] args) {  
    sayPassword(42) ;  
    sayPassword(12345) ;  
}
```

Output:

The password is 42

The password is 12345



# Parameters and loops

- ▶ A parameter can guide the number of repetitions of a loop.

```
public static void main(String[] args) {  
    chant(3);  
}  
  
public static void chant(int times) {  
    for (int i = 1; i <= times; i++) {  
        System.out.println("Just a salad...");  
    }  
}
```

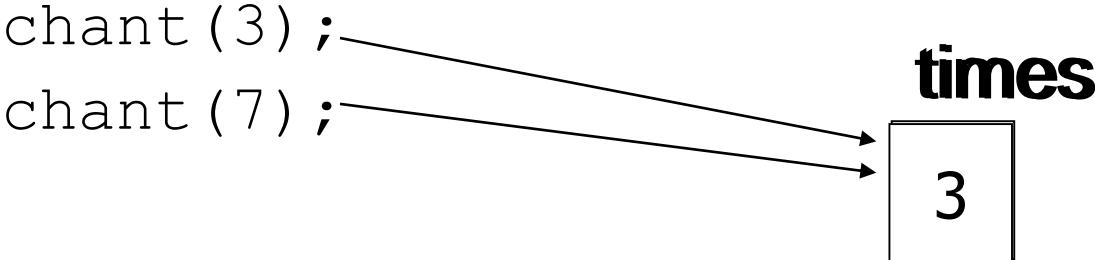
Output:

```
Just a salad...  
Just a salad...  
Just a salad...
```

# How parameters are passed

- ▶ When the method is called:
  - The value is stored into the parameter variable.
  - The method's code executes using that value.

```
public static void main(String[] args) {  
    chant(3);  
    chant(7);  
}
```



The diagram illustrates how the parameter **times** is passed. Two arrows originate from the arguments `3` and `7` in the `chant` method calls within the `main` method. Both arrows point to a box labeled **times** which contains the value `3`, indicating that the parameter `times` receives the value `3` for the first call.

```
public static void chant(int times) {  
    for (int i = 1; i <= times; i++) {  
        System.out.println("Just a salad...");  
    }  
}
```

# Common errors

- ▶ If a method accepts a parameter, it is illegal to call it without passing any value for that parameter.

```
chant(); // ERROR: parameter value required
```

- ▶ The value passed to a method must be of the correct type.

```
chant(3.7); // ERROR: must be of type int
```

- ▶ Exercise: Change the `Stars` program to use a parameterized method for drawing lines of stars.

# Stars solution

```
// Prints several lines of stars.  
// Uses a parameterized method to remove redundancy.  
public class Stars2 {  
    public static void main(String[] args) {  
        line(13);  
        line(7);  
        line(35);  
    }  
  
    // Prints the given number of stars plus a line break.  
    public static void line(int count) {  
        for (int i = 1; i <= count; i++) {  
            System.out.print("*");  
        }  
        System.out.println();  
    }  
}
```

# Multiple parameters

- ▶ A method can accept multiple parameters.  
(separate with , )
  - When calling it, you **must** pass values for each parameter.

- ▶ Declaration:

```
public static void <name>(<type> <name>, ..., <type> <name>) {  
    <statement>(s);  
}
```

- ▶ Call:

```
<name> (<exp>, <exp>, ..., <exp>) ;
```

# Multiple parameters example

```
public static void main(String[] args) {  
    printNumber(4, 9);  
    printNumber(17, 6);  
    printNumber(8, 0);  
    printNumber(0, 8);  
}  
  
public static void printNumber(int number, int count) {  
    for (int i = 1; i <= count; i++) {  
        System.out.print(number);  
    }  
    System.out.println();  
}
```

Output:

```
444444444  
171717171717
```

```
00000000
```

- Modify the `Stars` program to draw boxes with parameters.

# Stars solution

```
// Prints several lines and boxes made of stars.
// Third version with multiple parameterized methods.

public class Stars3 {
    public static void main(String[] args) {
        line(13);
        line(7);
        line(35);
        System.out.println();
        box(10, 3);
        box(5, 4);
        box(20, 7);
    }

    // Prints the given number of stars plus a line break.
    public static void line(int count) {
        for (int i = 1; i <= count; i++) {
            System.out.print("*");
        }
        System.out.println();
    }

    ...
}
```

# Stars solution, cont'd.

...

```
// Prints a box of stars of the given size.
public static void box(int width, int height) {
    line(width);

    for (int line = 1; line <= height - 2; line++) {
        System.out.print("*");
        for (int space = 1; space <= width - 2; space++) {
            System.out.print(" ");
        }
        System.out.println("*");
    }

    line(width);
}
}
```



# Value semantics

- ▶ **value semantics:** When primitive variables (`int`, `double`) are passed as parameters, their values are copied.
  - Modifying the parameter will not affect the variable passed in.

```
public static void strange(int x) {  
    x = x + 1;  
    System.out.println("1. x = " + x);  
}
```

```
public static void main(String[] args) {  
    int x = 23;  
    strange(x);  
    System.out.println("2. x = " + x);  
    ...  
}
```

Output:

```
1. x = 24  
2. x = 23
```

# Clicker 1 -

## Output of "Parameter Mystery"

```
public class ParameterMystery {  
    public static void main(String[] args) {  
        int x = 9;  
        int y = 2;  
        int z = 5;  
  
        mystery(z, y, x);  
  
        mystery(y, x, z);  
    }  
  
    public static void mystery(int x, int z, int y) {  
        System.out.print(z + " " + (y - x) + " ");  
    }  
}
```

**A.** 5 -7 5 -7

**B.** 9 -3 5 7

**C.** 2 4 9 3

**D.** 9 -3 5 12

**E.** None of A through D

## Clicker 2 - What is output by the following code?

```
int x = 2;
int y = 5;
mystery2(x, y);
System.out.print(x + " " + y + " ");

public static void mystery2(int x, int y) {
    System.out.print(x + " " + y + " ");
    x *= y + 3;
    y--;
    x++;
    System.out.print(x + " " + y + " ");
}
```

**A.** 2 5 17 4 2 5

**B.** 2 5 17 4 17 4

**C.** 17 4 2 5 17 4

**D.** 2 5 2 5 17 4

**E.** None of A through D

# Recall: Strings

- ▶ **string**: A sequence of text characters.

```
String <name> = "<text>" ;
```

```
String <name> = <expression resulting in  
String>;
```

- Examples:

```
String name = "Marla Singer" ;
```

```
int x = 3;
```

```
int y = 5;
```

```
String point = "(" + x + ", " + y + ")";
```

# Clicker 3

► Are Strings a primitive data type just like `int` and `double`?

A. No

B. Yes

# Strings as parameters

```
public class StringParameters {  
    public static void main(String[] args) {  
        sayHello("Marty") ;  
        String teacher = "Bictolia";  
        sayHello(teacher) ;  
    }  
    public static void sayHello(String name) {  
        System.out.println("Welcome, " + name);  
    }  
}
```

## Output:

```
Welcome, Marty  
Welcome, Bictolia
```

- Modify the `Stars` program to use string parameters. Use a method named `repeat` that prints a string many times.

# Stars solution

```
// Prints several lines and boxes made of stars.  
// Fourth version with String parameters.
```

```
public class Stars4 {  
    public static void main(String[] args) {  
        line(13);  
        line(7);  
        line(35);  
        System.out.println();  
        box(10, 3);  
        box(5, 4);  
        box(20, 7);  
    }  
  
    // Prints the given number of stars plus a line break.  
    public static void line(int count) {  
        repeat(" ", count);  
        System.out.println();  
    }  
  
    ...  
}
```

# Stars solution, cont'd.

...

```
// Prints a box of stars of the given size.
```

```
public static void box(int width, int height) {  
    line(width);  
    for (int line = 1; line <= height - 2; line++) {  
        System.out.print("*");  
        repeat(" ", width - 2);  
        System.out.println("*");  
    }  
    line(width);  
}
```

```
// Prints the given String the given number of times.
```

```
public static void repeat(String s, int times) {  
    for (int i = 1; i <= times; i++) {  
        System.out.print(s);  
    }  
}
```

```
}
```



# Topic 8

## graphics

"What makes the situation worse is that the highest level CS course I've ever taken is cs4, and quotes from the graphics group startup readme like '*these paths are abstracted as being the result of a topological sort on the graph of ordering dependencies for the entries*' make me lose consciousness in my chair and bleed from the nose."

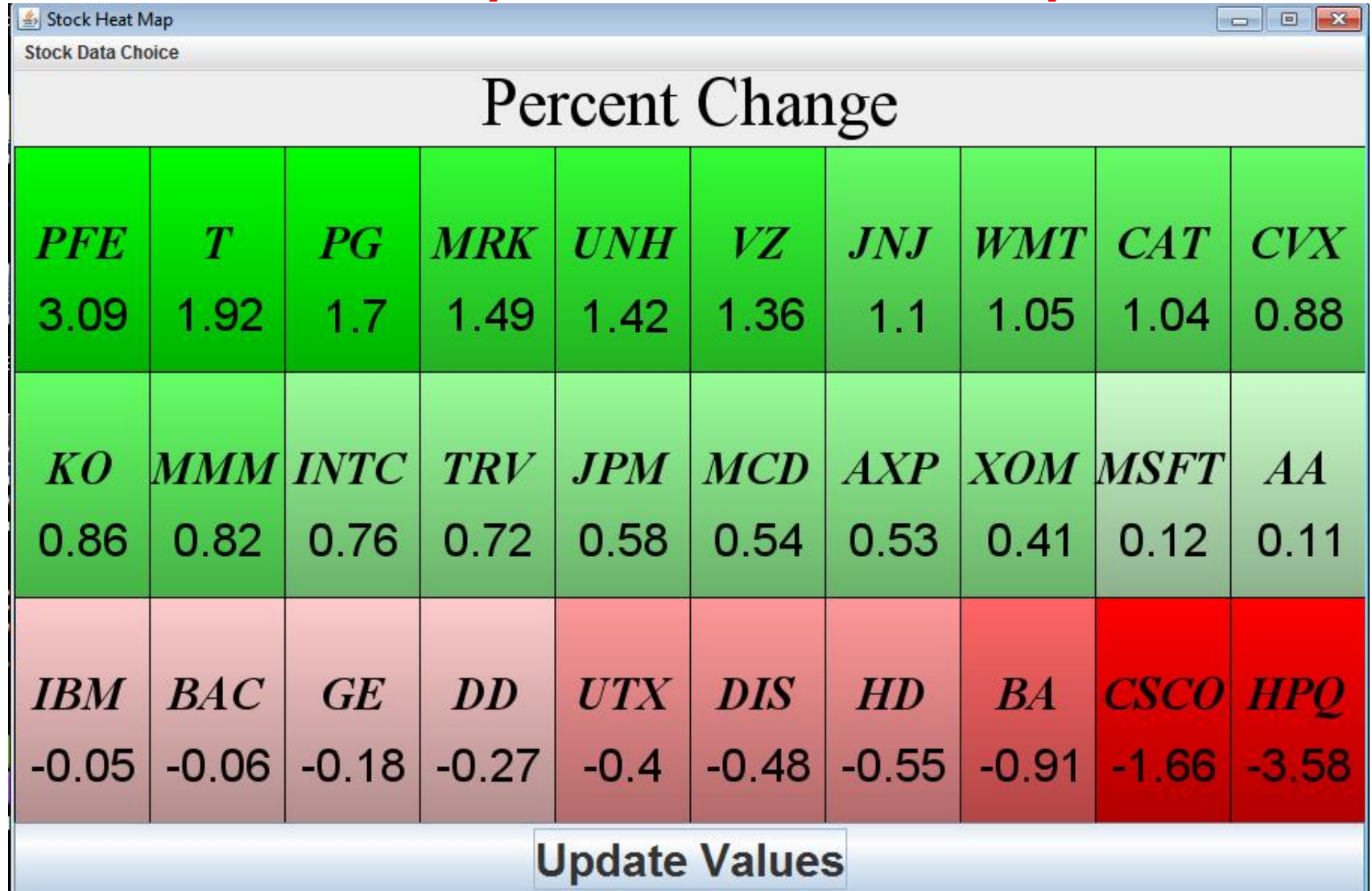
-mgrimes, Graphics problem report 134



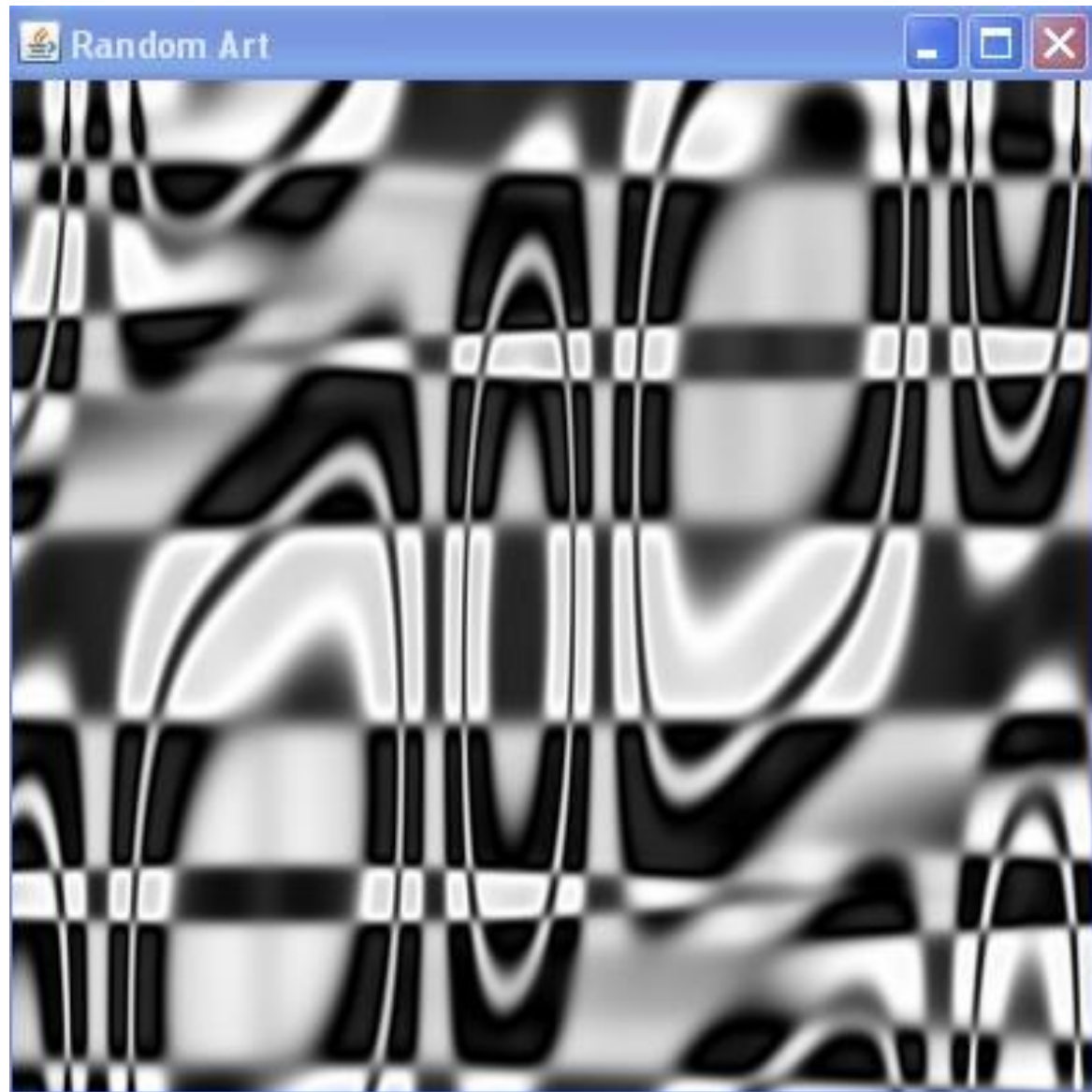
**Andries van Dam**  
Head of the Brown  
Graphics Group

# CS324E, Graphics and Visualization

## Examples - Heat Map

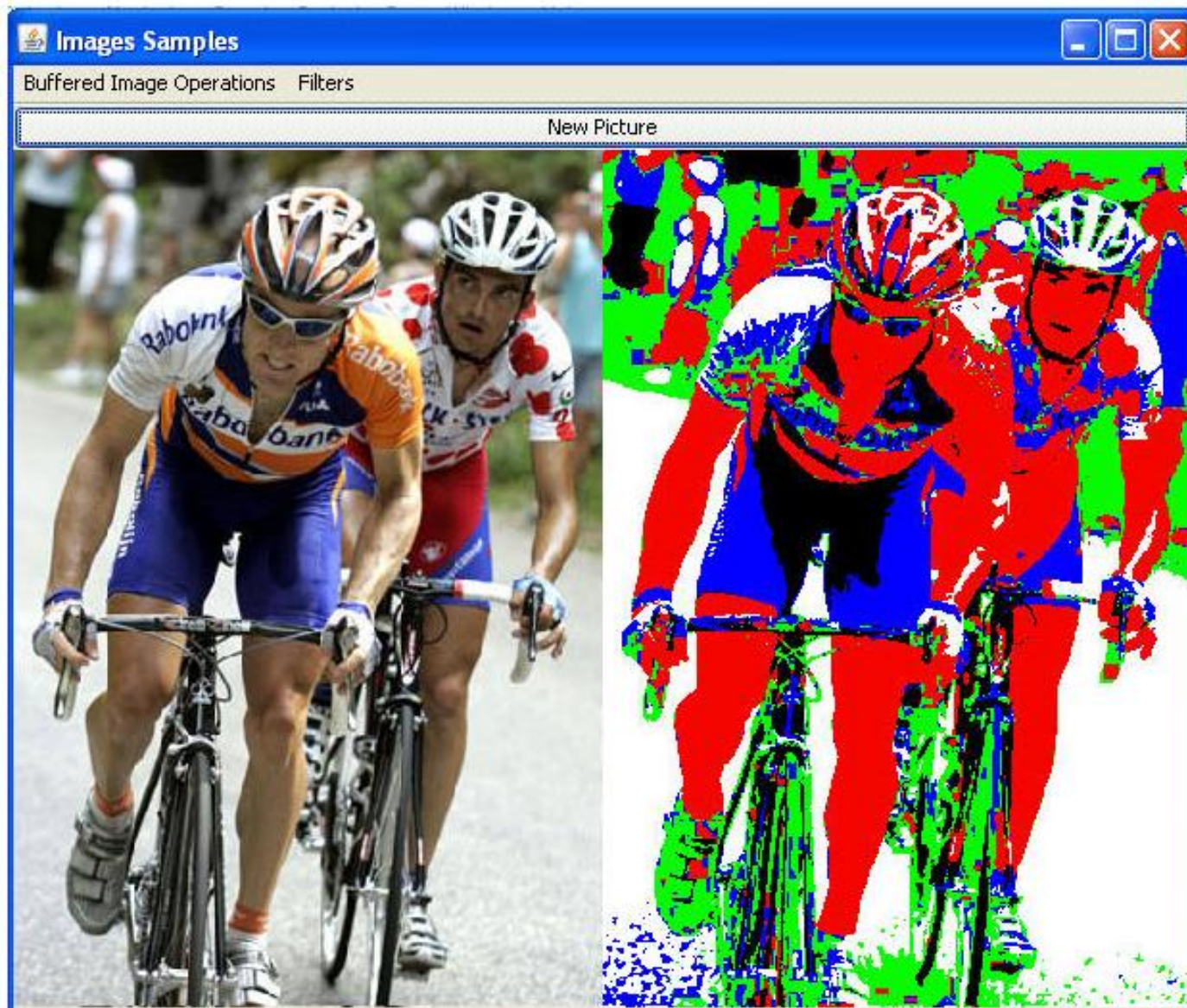


# Random Art

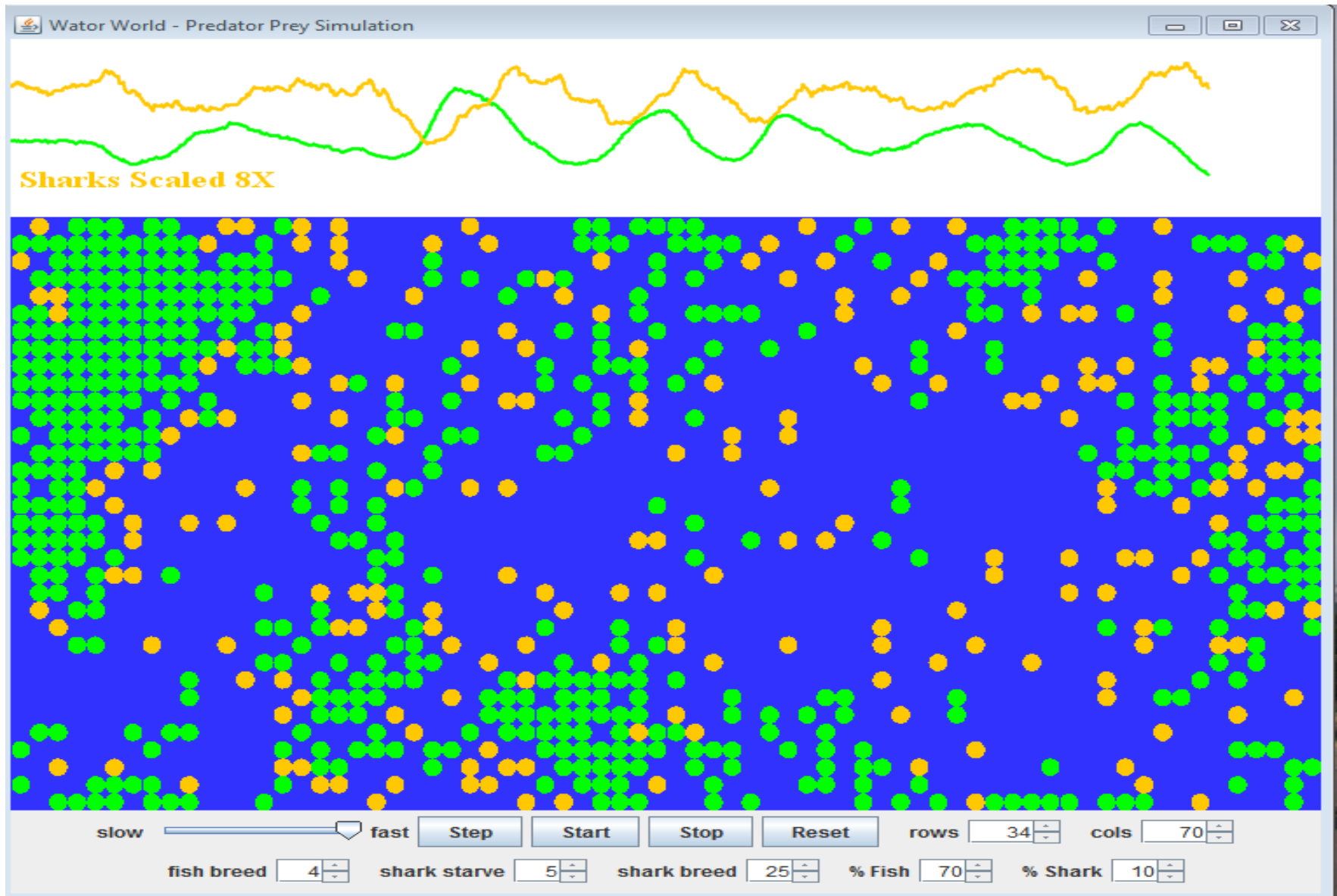




# Image Manipulation

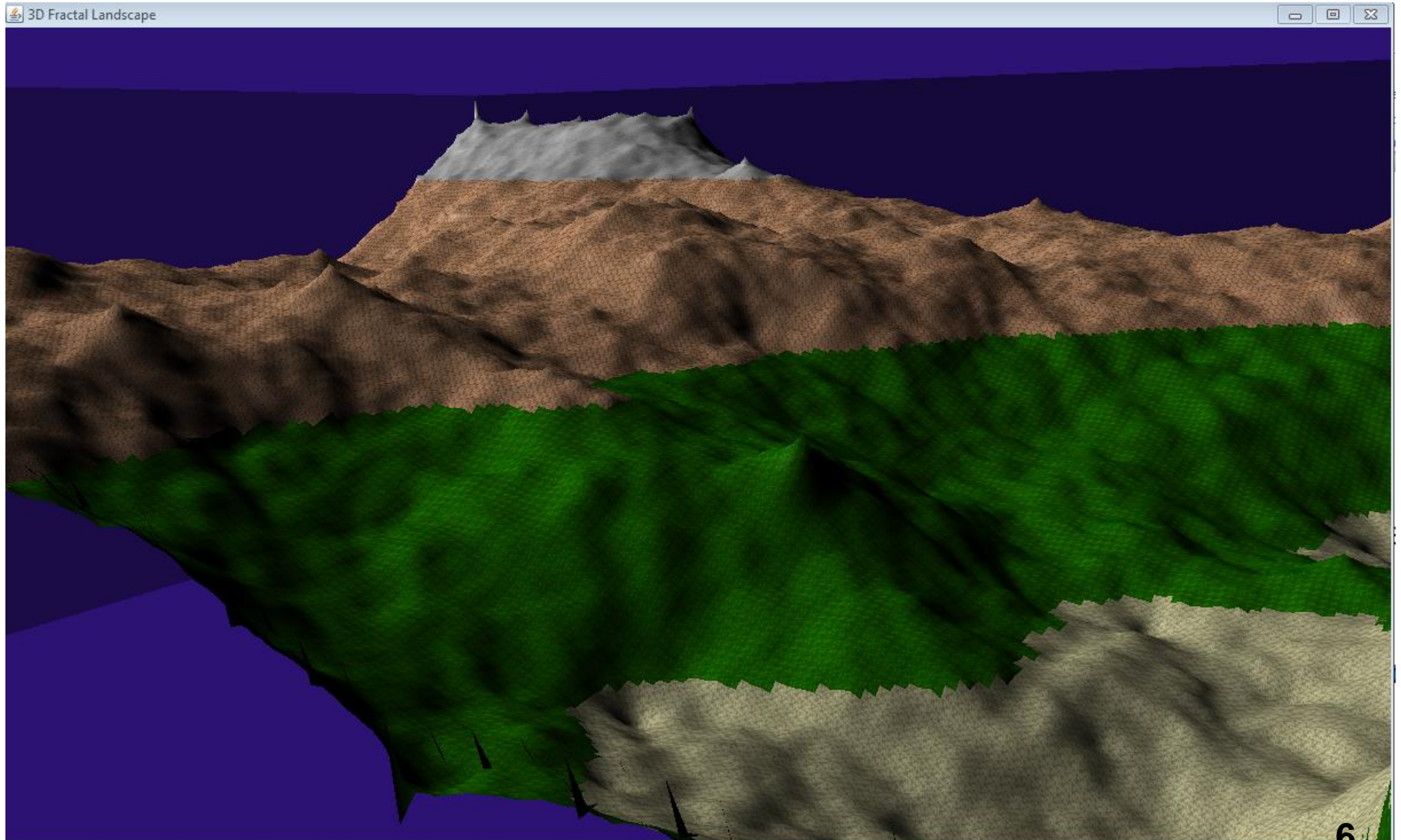


# Simulation and Visualization WaterWorld



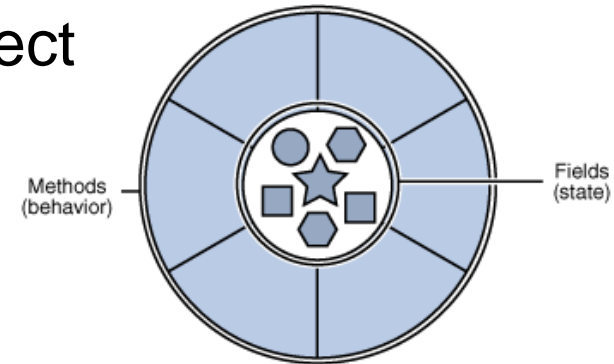


# Fractal 3D Landscape



# Objects (briefly)

- ▶ **object:** An entity that contains data and behavior.
  - *data:* variables inside the object
  - *behavior:* methods called on object
    - You interact with the methods; the data is hidden in the object.
    - A **class** is a data type.

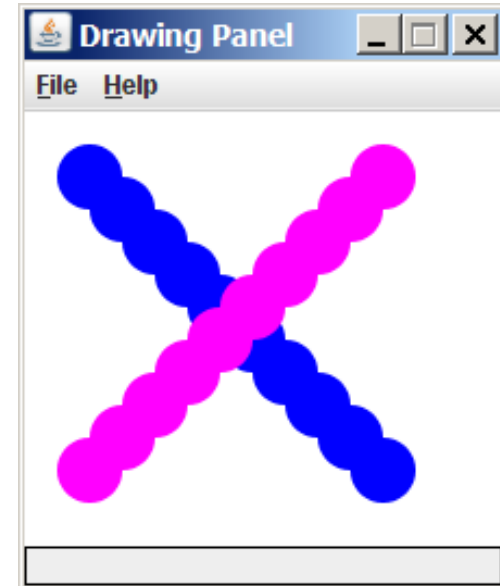


- ▶ Constructing (creating) an object:  
**Type** **objectName** = new **Type** (**parameters**) ;
- ▶ Calling an object's method:  
**objectName** . **methodName** (**parameters**) ;

# Graphical objects

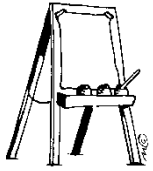
We will draw graphics in Java using 3 kinds of objects:

- ▶ `DrawingPanel`: A window on the screen.
  - Not part of standard Java; provided by the authors.  
See class web site.
- ▶ `Graphics`: A "pen" to draw shapes and lines on a window.
- ▶ `Color`: Colors in which to draw shapes.





# DrawingPanel



*"Canvas" objects that represents windows/drawing surfaces*

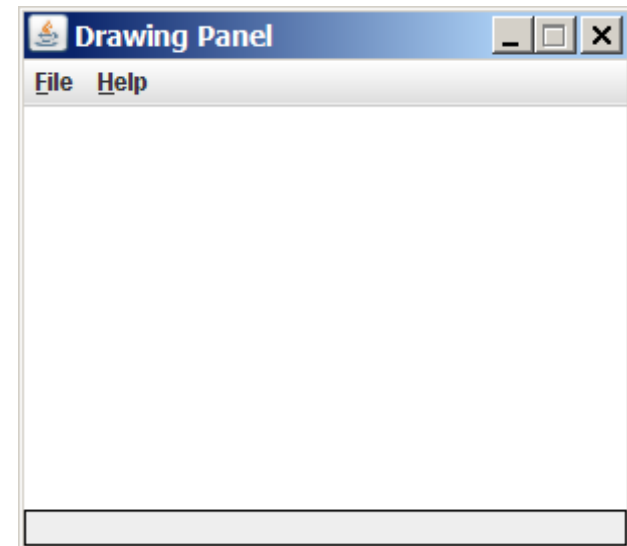
- ▶ To create a window:

```
DrawingPanel name = new DrawingPanel(width, height) ;
```

Example:

```
DrawingPanel panel = new DrawingPanel(300, 200) ;
```

- ▶ The window has nothing on it.
  - We draw shapes / lines on it with another object of type `Graphics`.



# Graphics



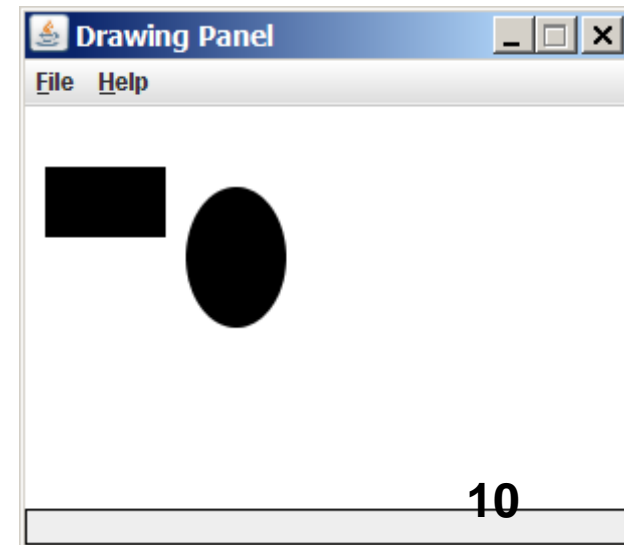
*"Pen" or "paint brush" objects to draw lines and shapes*

- Access it by calling `getGraphics` on your `DrawingPanel`.

```
Graphics g = panel.getGraphics();
```

- ▶ Draw shapes by calling methods on the `Graphics` object.

```
g.fillRect(10, 30, 60, 35);  
g.fillOval(80, 40, 50, 70);
```



# Java class libraries, import

- ▶ **Java class libraries**: Classes included with Java's JDK.
  - organized into groups named *packages*
  - To use a package, put an *import declaration* in your program:  

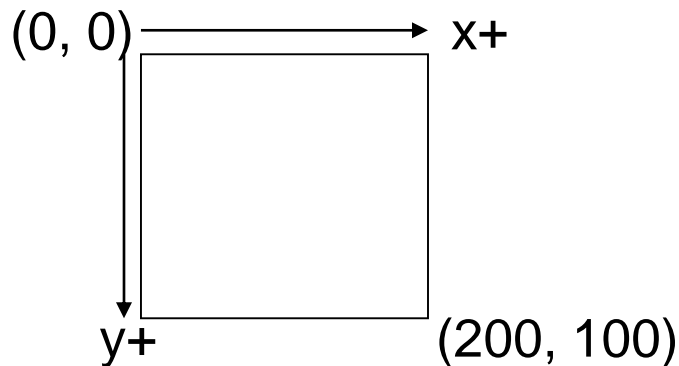
```
// put this at the very top of your program  
import packageName.ClassName;
```
- ▶ `Graphics` belongs to a package named `java.awt`  

```
import java.awt.Graphics;
```

  - To use `Graphics`, you must place the above line at the very top of your program, before the `public class` header.

# Coordinate system

- ▶ Each  $(x, y)$  position is a *pixel* ("picture element").
- ▶ Position  $(0, 0)$  is at the window's top-left corner.
  - $x$  increases rightward and the  $y$  increases downward.
- ▶ The rectangle from  $(0, 0)$  to  $(200, 100)$  looks like this:



# Graphics methods

Method name	Description
<code>g.drawLine(<b>x1</b>, <b>y1</b>, <b>x2</b>, <b>y2</b>) ;</code>	line between points $(x1, y1)$ , $(x2, y2)$
<code>g.drawOval(<b>x</b>, <b>y</b>, <b>width</b>, <b>height</b>) ;</code>	outline largest oval that fits in a box of size $width * height$ with top-left at $(x, y)$
<code>g.drawRect(<b>x</b>, <b>y</b>, <b>width</b>, <b>height</b>) ;</code>	outline of rectangle of size $width * height$ with top-left at $(x, y)$
<code>g.drawString(<b>text</b>, <b>x</b>, <b>y</b>) ;</code>	text with bottom-left at $(x, y)$
<code>g.fillOval(<b>x</b>, <b>y</b>, <b>width</b>, <b>height</b>) ;</code>	fill largest oval that fits in a box of size $width * height$ with top-left at $(x, y)$
<code>g.fillRect(<b>x</b>, <b>y</b>, <b>width</b>, <b>height</b>) ;</code>	fill rectangle of size $width * height$ with top-left at $(x, y)$
<code>g.setColor(<b>Color</b>) ;</code>	set Graphics to paint any following shapes in the given color

# Color



- Specified as predefined `Color` class constants.

`Color.CONSTANT_NAME`

where **CONSTANT\_NAME** is one of:

BLACK,	BLUE,	CYAN,	DARK_GRAY,	GRAY,
GREEN,	LIGHT_GRAY,	MAGENTA,	ORANGE,	
PINK,	RED,	WHITE,	YELLOW	

- Or create one using Red-Green-Blue (RGB) values of 0-255

```
Color name = new Color(red, green, blue);
```

– Example:

```
Color brown = new Color(192, 128, 64);  
Color burntOrange = new Color(191, 87, 0);
```

## List of Colors

# Clicker 1

- ▶ How many rectangles appear on the DrawingPanel when the following code is run?

```
DrawingPanel p1 = new DrawingPanel(200, 200);  
Graphics gr = new Graphics();  
for(int i = 0; i < 5; i++) {  
    gr.drawRect(i * 25, i * 20, 20, 50);  
}
```

- A. 5
- B. 6
- C. 20
- D. None due to syntax error
- E. None due to runtime error

## Clicker 2

- ▶ What named color is closest to the Color object created by this code?

```
Color mc = new Color(255, 255, 255);
```

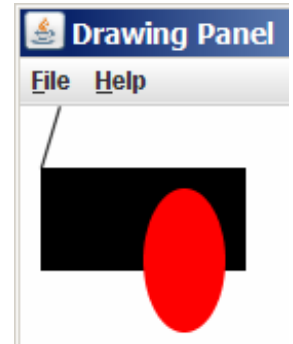
- A. Black
- B. Brown
- C. Gray
- D. Orange
- E. White



# Using colors

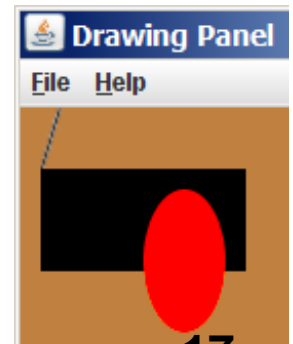
- ▶ Pass a `Color` to Graphics object's `setColor` method
  - Subsequent shapes will be drawn in the new color.

```
g.setColor(Color.BLACK) ;  
g.fillRect(10, 30, 100, 50) ;  
g.drawLine(20, 0, 10, 30) ;  
g.setColor(Color.RED) ;  
g.fillOval(60, 40, 40, 70) ;
```



- ▶ Pass a color to `DrawingPanel`'s `setBackground` method
  - The overall window background color will change.

```
Color brown = new Color(192, 128, 64) ;  
panel.setBackground(brown) ;
```



# Outlined shapes

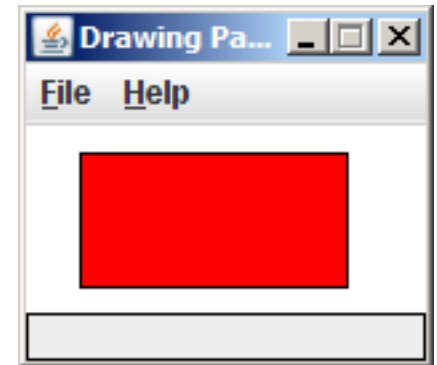
- ▶ To draw a colored shape with an outline, first *fill* it, then *draw* the same shape in the outline color.

```
import java.awt.Graphics;    // so I can use Graphics
import java.awt.Color;

public class OutlineExample {
    public static void main(String[] args) {
        DrawingPanel panel = new DrawingPanel(150, 70);
        Graphics g = panel.getGraphics();

        // inner red fill
        g.setColor(Color.RED);
        g.fillRect(20, 10, 100, 50);

        // black outline
        g.setColor(Color.BLACK);
        g.drawRect(20, 10, 100, 50);
    }
}
```



# Superimposing shapes

- ▶ When  $\geq 2$  shapes occupy the same pixels, the last drawn "wins."

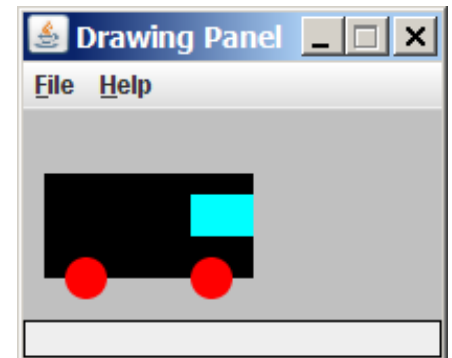
```
import java.awt.Graphics;
import java.awt.Color;

public class Car {
    public static void main(String[] args) {
        DrawingPanel panel = new DrawingPanel(200, 100);
        panel.setBackground(Color.LIGHT_GRAY);
        Graphics g = panel.getGraphics();

        g.setColor(Color.BLACK);
        g.fillRect(10, 30, 100, 50);

        g.setColor(Color.RED);
        g.fillOval(20, 70, 20, 20);
        g.fillOval(80, 70, 20, 20);

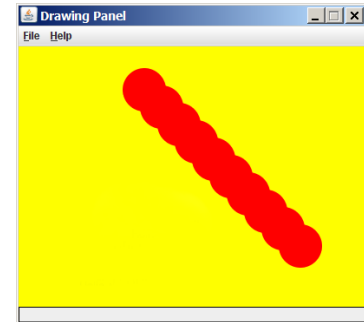
        g.setColor(Color.CYAN);
        g.fillRect(80, 40, 30, 20);
    }
}
```



# Drawing with loops

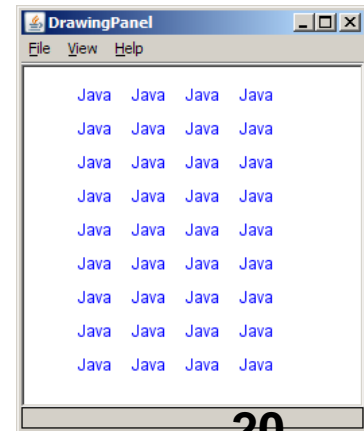
- ▶ The  $x, y, w, h$  expressions can use the loop counter variable:

```
panel.setBackground(Color.YELLOW);
g.setColor(Color.RED);
for (int i = 1; i <= 10; i++) {
    //
    g.fillOval(100 + 20 * i, 5 + 20 * i, 50, 50);
}
```



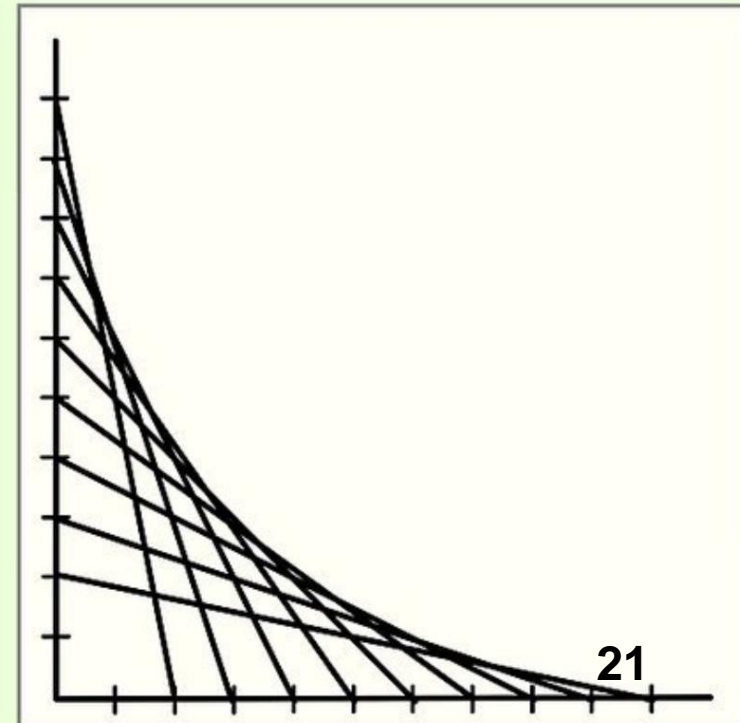
- ▶ Nested loops can be used with graphics:

```
g.setColor(Color.BLUE);
for (int x = 1; x <= 4; x++) {
    for (int y = 1; y <= 9; y++) {
        g.drawString("Java", x * 40, y * 25);
    }
}
```



# Graphics Example

- ▶ Write a method that draws straight lines to create a shape with a curved appearance
- ▶ Specify the x and y location of the upper left corner of the drawing and the size of the square to contain the drawing

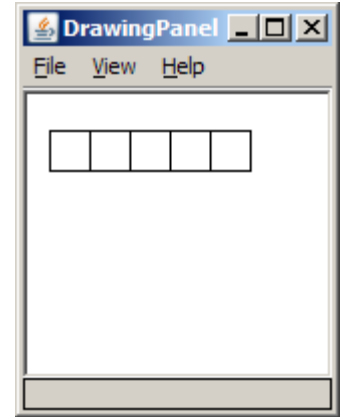


# Zero-based loops

- ▶ Beginning at 0 and using < can make calculating coordinates easier.

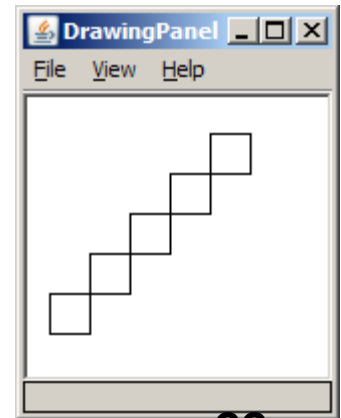
```
DrawingPanel panel = new DrawingPanel(150, 140);  
Graphics g = panel.getGraphics();
```

```
// horizontal line of 5 20x20 rectangles starting  
// at (11, 18); x increases by 20 each time  
for (int i = 0; i < 5; i++) {  
    g.drawRect(11 + 20 * i, 18, 20, 20);  
}
```



- ▶ Exercise: Write a variation of the above program that draws the output at right.
  - The bottom-left rectangle is at (11, 98).

```
for (int i = 0; i < 5; i++) {  
    g.drawRect(11 + 20 * i, 98 - 20 * i, 20, 20);  
}
```



# Topic 9

## More Graphics

# Clicker 1

- ▶ What happens if a graphics object is used to draw a shape that exceeds the boundaries of the `DrawingPanel`?

```
DrawingPanel p3 = new DrawingPanel(100, 100);  
Graphics g2 = p3.getGraphics();  
g2.fillRect(50, 50, 200, 200);
```

- A. Only the visible portion is shown
- B. The `DrawingPanel` expands to show whole rectangle
- C. Syntax error
- D. Runtime error
- E. None of A - D are correct



# Graphics exercise

- Modify the following program to draw a generalized truck.

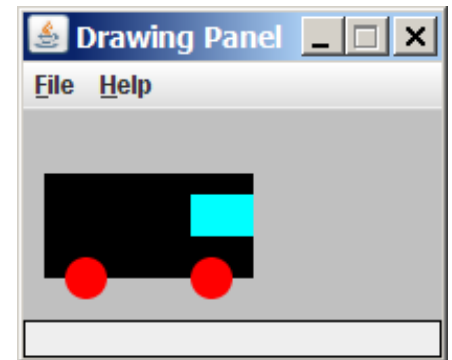
```
import java.awt.Graphics;
import java.awt.Color;

public class Car {
    public static void main(String[] args) {
        DrawingPanel panel = new DrawingPanel(200, 100);
        panel.setBackground(Color.LIGHT_GRAY);
        Graphics g = panel.getGraphics();

        g.setColor(Color.BLACK);
        g.fillRect(10, 30, 100, 50);

        g.setColor(Color.RED);
        g.fillOval(20, 70, 20, 20);
        g.fillOval(80, 70, 20, 20);

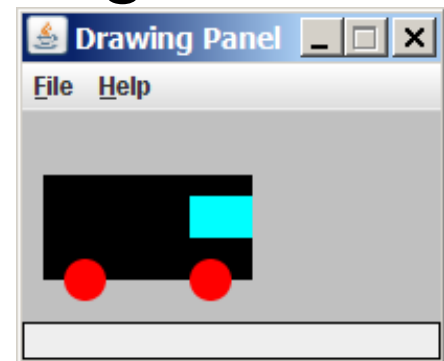
        g.setColor(Color.CYAN);
        g.fillRect(80, 40, 30, 20);
    }
}
```



# Clicker 2

► What dimension should we use as a parameter to draw the truck?

- A. Wheel diameter (width)
- B. Large rectangle (body) width
- C. Large rectangle (body) height
- D. Small rectangle (windshield) width
- E. Small rectangle (windshield) height



# Parameterized Drawing

- ▶ drawTruck0 -> hard coded location and size
- ▶ drawTruck1 -> parameterized location, hard coded size
- ▶ drawTruck2 -> parameterized location and size
- ▶ animate the truck using the sleep method from drawing panel

# Any Mistakes?

- ▶ Typically easy to spot significant logic errors in graphical output.
- ▶ Does the truck scale or do we have an abstract, deconstruction of a truck?
- ▶ "Truck, by CS312"



## Richard Tuttle

⊕ Follow

*Light Pink Octagon, 1967*

Canvas dyed with Tintex

56 4/5 x 53 in

144.2 x 134.7 cm

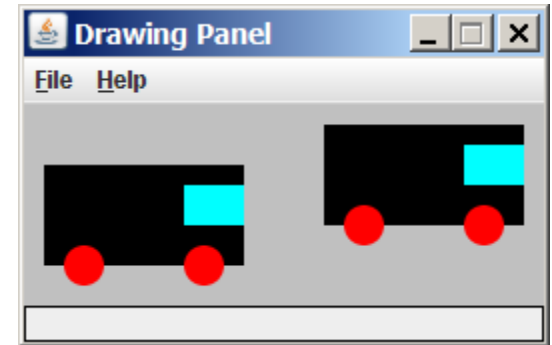
## Blanton Museum of Art

📍 Austin

Want to sell a work by this artist? [Consign with Artex](#)

# Parameterized figures

- ▶ Modify the car-drawing method so that it can draw cars at different positions, as in the following image.
  - Top-left corners: (10, 30), (150, 10)
  - Increase the drawing panel's size to 260x100 to fit.



# Drawing with parameters

- ▶ To draw in a method, you must pass `Graphics g` to it.
  - Otherwise, `g` is out of scope and cannot be used.

- ▶ syntax (declaration):

```
public static void <name> (Graphics g, <parameters>) {  
    <statement(s)> ;  
}
```

- ▶ syntax (call):

```
<name> (g, <values>) ;
```

# Parameterized answer

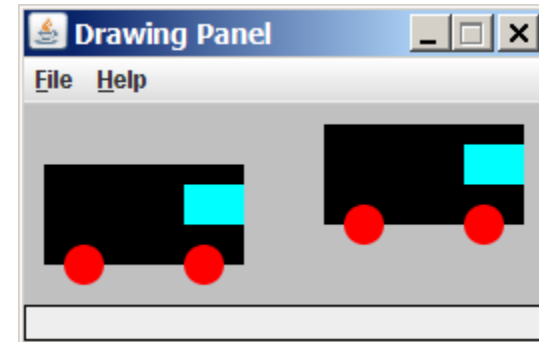
```
import java.awt.*;

public class Car3 {
    public static void main(String[] args) {
        DrawingPanel panel = new DrawingPanel(260, 100);
        panel.setBackground(Color.LIGHT_GRAY);
        Graphics g = panel.getGraphics();
        drawCar(g, 10, 30);
        drawCar(g, 150, 10);
    }

    public static void drawCar(Graphics g, int x, int y) {
        g.setColor(Color.BLACK);
        g.fillRect(x, y, 100, 50);

        g.setColor(Color.RED);
        g.fillOval(x + 10, y + 40, 20, 20);
        g.fillOval(x + 70, y + 40, 20, 20);

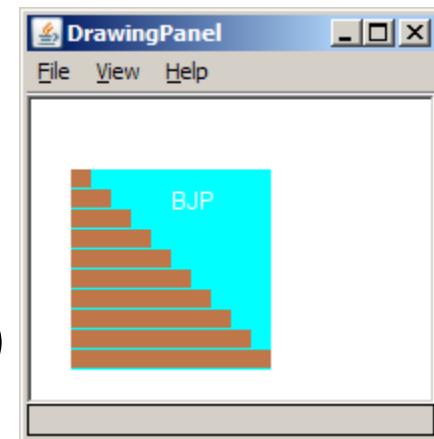
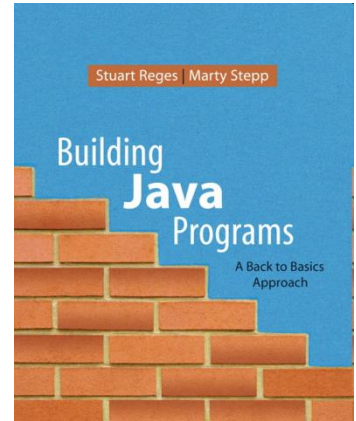
        g.setColor(Color.CYAN);
        g.fillRect(x + 70, y + 10, 30, 20);
    }
}
```



# Java book figure

► Write a program that draws the following figure:

- drawing panel is size 200x150
- book is at (20, 35), size 100x100
- cyan background
- white "BJP" text at position (70, 55)
- stairs are (red=191, green=118, blue=73)
- each stair is 9px tall
  - 1st stair is 10px wide
  - 2nd stair is 20px wide ...
- stairs are 10px apart (1 blank pixel between)





# Java book solution

```
// Draws a Building Java Programs textbook with DrawingPanel.
import java.awt.*;

public class Book {
    public static void main(String[] args) {
        DrawingPanel panel = new DrawingPanel(200, 150);
        panel.setBackground(Color.WHITE);
        Graphics g = panel.getGraphics();

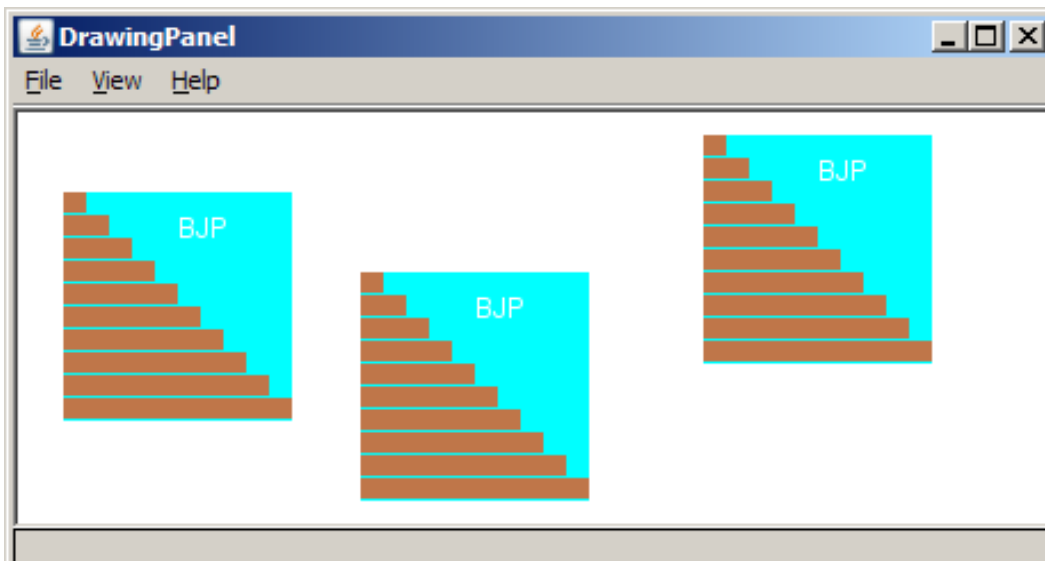
        g.setColor(Color.CYAN);           // cyan background
        g.fillRect(20, 35, 100, 100);

        g.setColor(Color.WHITE);          // white "bjp" text
        g.drawString("BJP", 70, 55);

        g.setColor(new Color(191, 118, 73));
        for (int i = 0; i < 10; i++) {    // orange "bricks"
            g.fillRect(20, 35 + 10 * i, 10 + 10 * i, 9);
        }
    }
}
```

# Multiple Java books

- ▶ Modify the Java book program so that it can draw books at different *positions* as shown below.
  - book top/left positions: (20, 35), (150, 70), (300, 10)
  - drawing panel's new size: 450x180



# Multiple books solution

```
// Draws many BJP textbooks using parameters.
import java.awt.*;

public class Book2 {
    public static void main(String[] args) {
        DrawingPanel panel = new DrawingPanel(450, 180);
        panel.setBackground(Color.WHITE);
        Graphics g = panel.getGraphics();

        // draw three books at different locations
        drawBook(g, 20, 35);
        drawBook(g, 150, 70);
        drawBook(g, 300, 10);
    }

    ...
}
```

# Multiple books, cont'd.

...

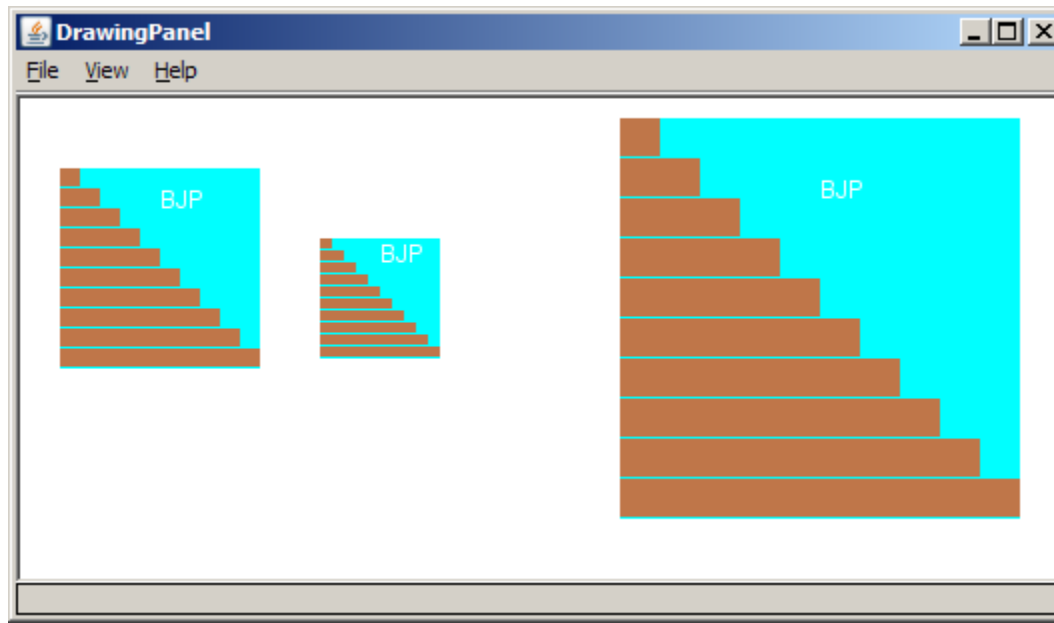
```
// Draws a BJP textbook at the given x/y position.
public static void drawBook(Graphics g, int x, int y) {
    g.setColor(Color.CYAN);                // cyan background
    g.fillRect(x, y, 100, 100);

    g.setColor(Color.WHITE);                // white "bjp" text
    g.drawString("BJP", x + 50, y + 20);

    g.setColor(new Color(191, 118, 73));
    for (int i = 0; i < 10; i++) {          // orange "bricks"
        g.fillRect(x, y + 10 * i, 10 * (i + 1), 9);
    }
}
```

# Resizable Java books

- ▶ Modify the Java book program so that it can draw books at different *sizes* as shown below.
  - book sizes: 100x100, 60x60, 200x200
  - drawing panel's new size: 520x240



# Resizable books solution

```
// Draws many sized BJP textbooks using parameters.
import java.awt.*;

public class Book3 {
    public static void main(String[] args) {
        DrawingPanel panel = new DrawingPanel(520, 240);
        panel.setBackground(Color.WHITE);
        Graphics g = panel.getGraphics();

        // draw three books at different locations/sizes
        drawBook(g, 20, 35, 100);
        drawBook(g, 150, 70, 60);
        drawBook(g, 300, 10, 200);
    }

    ...
}
```

# Resizable solution, cont'd.

...

```
// Draws a book of the given size at the given position.
public static void drawBook(Graphics g, int x, int y, int size) {
    g.setColor(Color.CYAN);           // cyan background
    g.fillRect(x, y, size, size);

    g.setColor(Color.WHITE);           // white "bjp" text
    g.drawString("BJP", x + size/2, y + size/5);

    g.setColor(new Color(191, 118, 73));
    for (int i = 0; i < 10; i++) {     // orange "bricks"
        g.fillRect(x,                  // x
                   y + size/10 * i,    // y
                   size/10 * (i + 1),  // width
                   size/10 - 1);       // height
    }
}
```

# Polygon

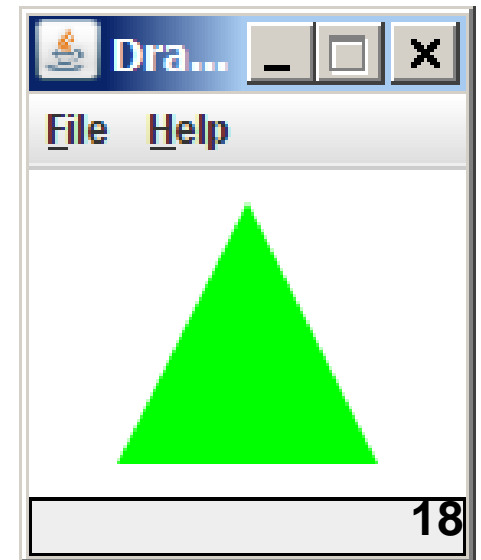
*Objects that represent arbitrary shapes*

- ▶ Add points to a `Polygon` using its `addPoint(x, y)` method.

- ▶ Example:

```
DrawingPanel p = new DrawingPanel(100, 100);  
Graphics g = p.getGraphics();  
g.setColor(Color.GREEN);
```

```
Polygon poly = new Polygon();  
poly.addPoint(10, 90);  
poly.addPoint(50, 10);  
poly.addPoint(90, 90);  
g.fillPolygon(poly);
```





# DrawingPanel methods

- ▶ **panel.save(filename) ;**

Saves the image on the panel to the given file (String).

- ▶ **panel.sleep(ms) ;**

Pauses the drawing for the given number of milliseconds.

# Animation with `sleep`

- ▶ `DrawingPanel`'s `sleep` method pauses your program for a given number of milliseconds.
- ▶ You can use `sleep` to create simple animations.

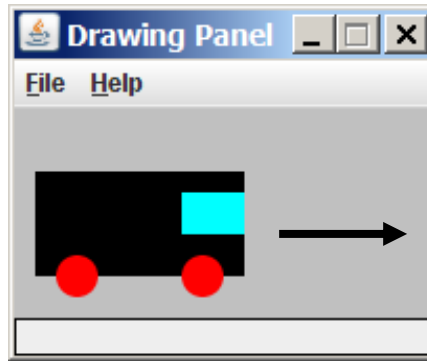
```
DrawingPanel panel = new DrawingPanel(250, 200);  
Graphics g = panel.getGraphics();
```

```
g.setColor(Color.BLUE);  
for (int i = 1; i <= 10; i++) {  
    g.fillOval(15 * i, 15 * i, 30, 30);  
    panel.sleep(500);  
}
```

- Try adding `sleep` commands to loops in past exercises in this chapter and watch the panel draw itself piece by piece.

# Animation exercise

- ▶ Modify the previous program to draw a "moving" animated car.



# Topic 10

## return values, Math methods

"Thinking like a computer scientist means more than being able to program a computer. It requires thinking at multiple levels of abstraction."

— Jeannette M. Wing



# static methods in other Classes

- ▶ Java includes 8 primitive data types
  - byte, short, **int**, long, float, **double**, char, boolean
- ▶ The Java Standard Library includes thousands of other data types, classes
  - System, String, Graphics, Color, ...
- ▶ The Math class contains static methods for common mathematical operations (for which an operator does not exist in Java)
- ▶ Call those methods: `Math.<MethodName>`  
`Math.pow(2, 5);`

# Java's Math class

Method name	Description		
<code>Math.abs(<i>value</i>)</code>	absolute value		
<code>Math.ceil(<i>value</i>)</code>	moves up to ceiling		
<code>Math.floor(<i>value</i>)</code>	moves down to floor		
<code>Math.log10(<i>value</i>)</code>	logarithm, base 10		
<code>Math.max(<i>value1</i>, <i>value2</i>)</code>	larger of two values		
<code>Math.min(<i>value1</i>, <i>value2</i>)</code>	smaller of two values		
<code>Math.pow(<i>base</i>, <i>exp</i>)</code>	<i>base</i> to the <i>exp</i> power		
<code>Math.random()</code>	random double between 0 and 1		
<code>Math rint(<i>value</i>)</code>	Round int, nearest whole number		
<code>Math.sqrt(<i>value</i>)</code>	square root		
<code>Math.sin(<i>value</i>)</code> <code>Math.cos(<i>value</i>)</code> <code>Math.tan(<i>value</i>)</code>	sine/cosine/tangent of an angle in radians		
<code>Math.toDegrees(<i>value</i>)</code> <code>Math.toRadians(<i>value</i>)</code>	convert degrees to radians and back	<b>Constant</b>	<b>Description</b>
		<code>Math.E</code>	2.7182818...
		<code>Math.PI</code>	3.1415926...

# No output?

- Simply calling these methods produces no visible result.

```
Math.pow(3, 4);    // no output
```

- Math method calls use a Java feature called *return values* that cause them to be treated as expressions.
- The program runs the method, computes the answer, and then "replaces" the call with its computed result value.

```
Math.pow(3, 4);    // no output  
81.0;            // no output
```

- To see the result, we must print it or store it in a variable.

```
double result = Math.pow(3, 4);
```

```
System.out.println(result);    // 81.0    4
```

# Calling Math methods

`Math.methodName (parameters)`

## ► Examples:

```
double squareRoot = Math.sqrt(121.0);  
System.out.println(squareRoot);           // 11.0
```

```
int absoluteValue = Math.abs(-50);  
System.out.println(absoluteValue);        // 50
```

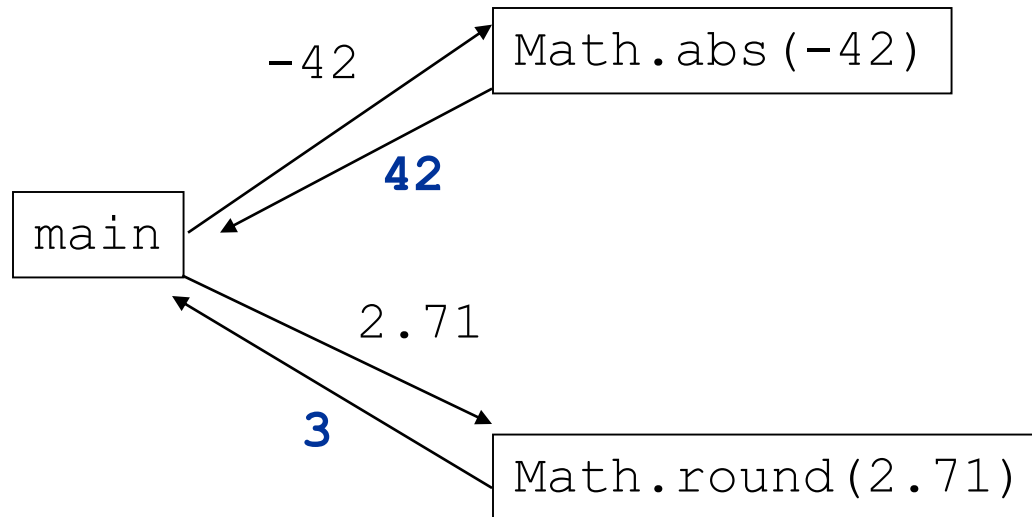
```
System.out.println(Math.min(3, 7) + 2);    // 5
```

- The `Math` methods do not print to the console.
  - Each method produces ("returns") a numeric result.
  - The results are used as expressions (printed, stored, etc.).



# Return

- ▶ **return:** To send out a value as the result of a method.
  - The opposite of a parameter:
    - Parameters send information *in* from the caller to the method.
    - Return values send information *out* from a method to its caller.
      - A call to the method can be used as part of an expression.



# Why return and not print?

- ▶ It might seem more useful for the `Math` methods to print their results rather than returning them. Why don't they?
- ▶ Answer: Returning is more flexible than printing.
  - We can compute several things before printing:

```
double pow1 = Math.pow(3, 4);  
double pow2 = Math.pow(10, 6);  
System.out.println("Powers are " + pow1 + " and " + pow2);
```

- We can combine the results of many computations:

```
double k = 13 * Math.pow(3, 4) + 5 - Math.sqrt(17.8);
```

# Clicker 1

► What is output by the following code?

```
double a = -1.9;  
double b = 2.25;  
System.out.print( Math.floor(a) +  
    " " + Math.ceil(b) + " " + a );
```

- A. 3.0
- B. -2.0 3.0 -2.0
- C. -1.0 3.0 -1.0
- D. -1 3 -1.9
- E. -2.0 3.0 -1.9

# Math questions

- ▶ Evaluate the following expressions:

```
Math.abs(-1.23)
```

```
Math.pow(3, 2)
```

```
Math.pow(10, -2)
```

```
Math.sqrt(121.0) - Math.sqrt(256.0)
```

```
Math.round(Math.PI) + Math.round(Math.E)
```

```
Math.ceil(6.022) + Math.floor(15.9994)
```

```
Math.abs(Math.min(-3, -5))
```

- ▶ `Math.max` and `Math.min` can be used to bound numbers.

Consider an `int` variable named `age`.

What statement would replace negative ages with 0?

What statement would cap the maximum age to 40?

# Quirks of real numbers

- ▶ Some `Math` methods return `double` or other `non-int` types.

```
int x = Math.pow(10, 3); // ERROR: incompat. types
```

- ▶ Some `double` values print poorly (too many digits).

```
double result = 1.0 / 3.0;  
System.out.println(result); // 0.3333333333333333
```

- ▶ The computer represents `doubles` in an imprecise way.

```
System.out.println(0.1 + 0.2);
```

– Instead of 0.3, the output is 0.30000000000000004 **10**

# Type casting

- ▶ **type cast:** A conversion from one type to another.
  - To promote an `int` into a `double` for floating point division
  - To truncate a `double` from a real number to an integer
- ▶ **Syntax:**

**(type) expression**

Examples:

```
double result = (double) 19 / 5;           // 3.8
int result2 = (int) result;                // 3
int x = (int) Math.pow(10, 3);             // 1000
```

# More about type casting

- ▶ Type casting has high precedence and only casts the item immediately next to it.

```
double x = (double) 1 + 1 / 2;           // 1.0
double y = 1 + (double) 1 / 2;           // 1.5
```

- ▶ You can use parentheses to force evaluation order.

```
double average = (double) (a + b + c) / 3;
```

- ▶ A conversion to `double` can be achieved in other ways.

```
double average = 1.0 * (a + b + c) / 3;
```

# Returning a value from a method

```
public static type name(parameters) {  
    statements;  
    ...  
    return expression;  
}
```

## ► Example:

```
// Returns the slope of the line between the given points.  
public static double slope(int x1, int y1, int x2, int y2) {  
    double dy = y2 - y1;  
    double dx = x2 - x1;  
    return dy / dx;  
}
```

slope(5, 11, 1, 3) returns 2.0



# Clicker 2

► Have we (in CS312, before today) used a method that returns a value in lecture?

A. No

B. Yes, a few times

C. Yes, hundreds of time

D. Lecture?? What lecture?

E. Maybe?

# Return examples

**// Converts degrees Fahrenheit to Celsius.**

```
public static double fToC(double degreesF) {  
    double degreesC = 5.0 / 9.0 * (degreesF - 32);  
    return degreesC;  
}
```

**// Computes triangle hypotenuse length given its side lengths.**

```
public static double hypotenuse(int a, int b) {  
    double c = Math.sqrt(a * a + b * b);  
    return c;  
}
```

- You can shorten the examples by returning an expression:

```
public static double fToC(double degreesF) {  
    return 5.0 / 9.0 * (degreesF - 32);  
}
```

# Common error: Not storing

- ▶ a `return` statement DOES NOT send a variable's name back to the calling method.

```
public static void main(String[] args) {  
    slope(0, 0, 6, 3);  
    System.out.println("The slope is " + result);  
    // ERROR: result not defined  
  
}  
  
public static double slope(int x1, int x2, int y1, int y2) {  
    double dy = y2 - y1;  
    double dx = x2 - x1;  
    double result = dy / dx;  
    return result;  
}
```

# Fixing the common error

- ▶ Instead, returning sends the variable's *value* back.
  - The returned value must be stored into a variable or used in an expression to be useful to the caller.

```
public static void main(String[] args) {  
    double s = slope(0, 0, 6, 3);  
    System.out.println("The slope is " + s);  
}
```

```
public static double slope(int x1, int x2, int y1, int y2) {  
    double dy = y2 - y1;  
    double dx = x2 - x1;  
    double result = dy / dx;  
    return result;  
}
```

# Clicker 3

► What is the output of the following code?

```
int x = 5;
int y = 7;
System.out.print(m(x, y) + " " + x + " " + m(y, x));

public static int m(int x, int y) {
    x += 2;
    System.out.print(x + " ");
    y -= 2;
    return x * y;
}
```

**A. 7 9 35 5 27**

**B. 7 7 35 7 27**

**C. 7 5 9 27 35**

**D. 35 7 5 9 27**

**E. None of A - D are correct**

# Exercise

- ▶ In physics, the *displacement* of a moving body represents its change in position over time while accelerating.
  - Given initial velocity  $v_0$  in m/s, acceleration  $a$  in  $\text{m/s}^2$ , and elapsed time  $t$  in s, the displacement of the body is:
    - Displacement =  $v_0 t + \frac{1}{2} a t^2$
- ▶ Write a method `displacement` that accepts  $v_0$ ,  $a$ , and  $t$  and computes and returns the change in position.
  - example: `displacement(3.0, 4.0, 5.0)`  
returns `65.0`

# Exercise solution

```
public static double displacement(  
    double v0, double a,  
    double t) {  
  
    double d = v0 * t + 0.5  
                * a * Math.pow(t, 2);  
    return d;  
}
```

# Exercises

- ▶ write a method to
  - return the int average of 3 ints
  - return the double average of 3 ints
  - return the average of a given number of rolls of 2 six sided dice
  - calculate and return N factorial (N!).
  - return the number of seconds in a given number of years.
  - return the Nth digit of a given integer.
  - return the distance between two points.





# Exercise

- ▶ If you drop two balls, which will hit the ground first?
  - Ball 1: height of 600m, initial velocity = 25 m/sec downward
  - Ball 2: height of 500m, initial velocity = 15 m/sec downward
- ▶ Write a program that determines how long each ball takes to hit the ground (and draws each ball falling).
- ▶ Total time is based on the force of gravity on each ball.
  - Acceleration due to gravity  $\cong 9.81 \text{ m/s}^2$ , downward
  - Displacement =  $v_0 t + \frac{1}{2} a t^2$

# Ball solution

```
// Simulates the dropping of two balls from various heights.
import java.awt.*;

public class Balls {
    public static void main(String[] args) {
        DrawingPanel panel = new DrawingPanel(600, 600);
        Graphics g = panel.getGraphics();

        int ball1x = 100, ball1y = 0, v01 = 25;
        int ball2x = 200, ball2y = 100, v02 = 15;

        // draw the balls at each time increment
        for (double t = 0; t <= 10.0; t = t + 0.1) {
            g.setColor(Color.GRAY);
            panel.fillRect(0, 0, 600, 600);
            g.setColor(Color.RED);
            double disp1 = displacement(v01, t, 9.81);
            g.fillOval(ball1x, ball1y + (int) disp1, 10, 10);
            double disp2 = displacement(v02, t, 9.81);
            g.fillOval(ball2x, ball2y + (int) disp2, 10, 10);

            panel.sleep(50);    // pause for 50 ms
        }
    }
    ...
}
```

# Topic 11

## Scanner objects, conditional execution

" There are only two kinds of languages:  
the ones people complain about  
and the ones nobody uses."

— Bjarne Stroustrup, creator of C++



# Input and `System.in`

- ▶ **interactive program:** Reads input from the console.
  - While the program runs, it asks the user to type input.
  - The input typed by the user is stored in variables in the code.
  - Can be tricky; users are unpredictable and misbehave.
  - But interactive programs have more interesting behavior.
- ▶ **Scanner:** An object that can read input from many sources.
  - Communicates with `System.in`
  - Can also read from files (Ch. 6), web sites, databases, ...

# Scanner syntax

- ▶ The `Scanner` class is found in the `java.util` package.

```
import java.util.Scanner;
```

- ▶ Constructing a `Scanner` object to read console input:

```
Scanner name = new Scanner(System.in);
```

- Example:

```
Scanner console = new Scanner(System.in);
```

# Scanner methods

Method	Description
<code>nextInt()</code>	reads an <code>int</code> from the user and returns it
<code>nextDouble()</code>	reads a <code>double</code> from the user
<code>nextLine()</code>	reads a <i>one-line</i> <code>String</code> from the user
<code>next()</code>	reads a one-word <code>String</code> from the user Avoid when Scanner connected to <code>System.in</code>

- Each method waits until the user presses Enter.
- The value typed by the user is returned.
- **prompt:** A message telling the user what input to type.

```
System.out.print("How old are you? "); // prompt
int age = console.nextInt();
System.out.println("You typed " + age);
```

# Scanner example

```
import java.util.Scanner;
```

```
public class UserInputExample {  
    public static void main(String[] args) {  
        Scanner console = new Scanner(System.in);  
  
        → System.out.print("How old are you? ");  
        → int age = console.nextInt();  
  
        → int years = 65 - age;  
        System.out.println(years + " years until retirement!");  
    }  
}
```

age

years



## ▶ Console (user input underlined):

How old are you? 29  
36 years until retirement!



# Scanner example 2

- ▶ The `Scanner` can read multiple values from one line.

```
import java.util.Scanner;
public class ScannerMultiply {
    public static void main(String[] args) {
        Scanner console = new Scanner(System.in);

        System.out.print("Please type two numbers: ");
        int num1 = console.nextInt();
        int num2 = console.nextInt();

        int product = num1 * num2;
        System.out.println("The product is " + product);
    }
}
```

- ▶ Output (user input underlined):

```
Please type two numbers: 8 6
The product is 48
```



# Clicker 1 - Input tokens

- ▶ **token**: A unit of user input, as read by the `Scanner`.
  - Tokens are separated by *whitespace* (spaces, tabs, new lines).
  - How many tokens appear on the following line of **input**?

23   John Smith   42.0   "Hello world"   \$2.50   "   19"

- A. 2                  B. 6                  C. 7
- D. 8                  E. 9

# input tokens

- ▶ When a token is the wrong type, the program crashes. (runtime error)

```
System.out.print("What is your age? ");  
int age = console.nextInt();
```

Output:

What is your age? Timmy

**java.util.InputMismatchException**

at java.util.Scanner.next(Unknown Source)

at java.util.Scanner.nextInt(Unknown Source)

...

# The `if/else` statement

**reading: 4.1**

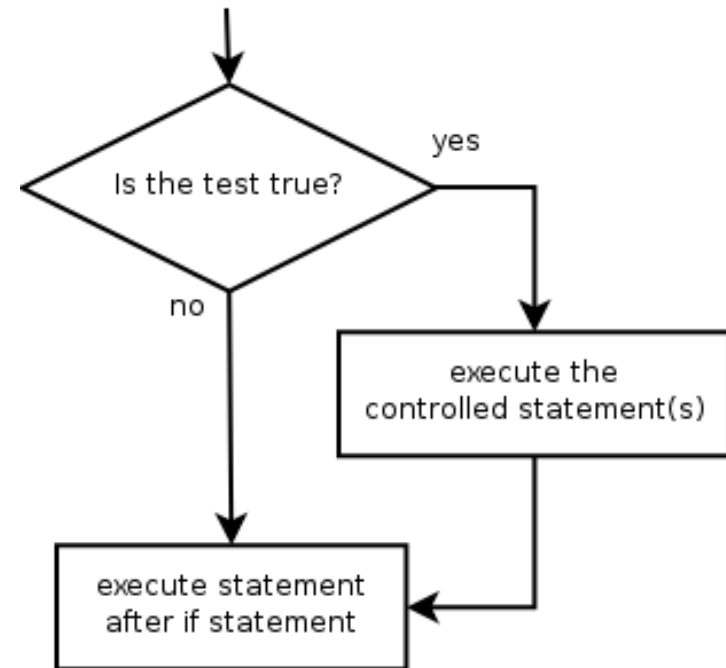
# The `if` statement

*Executes a block of statements only if a test is true*

```
if (test) {  
    statement;  
    ...  
    statement;  
}
```

## ► Example:

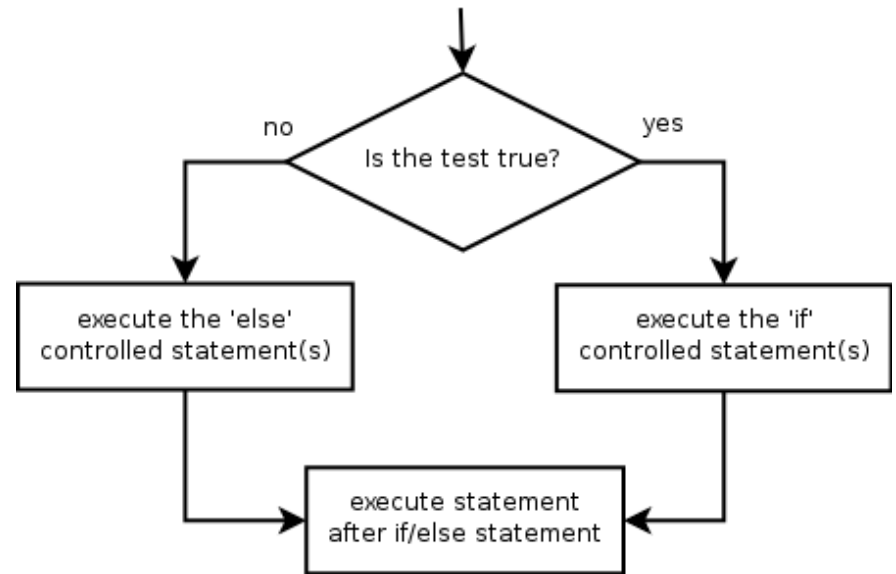
```
double gpa = console.nextDouble();  
if (gpa >= 2.0) {  
    System.out.println("Application accepted.");  
}
```



# The if/else statement

*Executes one block if a test is true, another if false*

```
if (test) {  
    statement(s);  
} else {  
    statement(s);  
}
```



## ► Example:

```
double gpa = console.nextDouble();  
if (gpa >= 2.0) {  
    System.out.println("Welcome to Mars University!");  
} else {  
    System.out.println("Application denied.");  
}
```

# Relational expressions

- ▶ `if` statements and `for` loops both use logical tests.

```
for (int i = 1; i <= 10; i++) { ...  
  if (i <= 10) { ...
```

– These are `boolean` expressions, seen in Ch. 5.

- ▶ Tests use *relational operators*:

Operator	Meaning	Example	Value
<code>==</code>	equals	<code>1 + 1 == 2</code>	true
<code>!=</code>	does not equal	<code>3.2 != 2.5</code>	true
<code>&lt;</code>	less than	<code>10 &lt; 5</code>	false
<code>&gt;</code>	greater than	<code>10 &gt; 5</code>	true
<code>&lt;=</code>	less than or equal to	<code>126 &lt;= 100</code>	false
<code>&gt;=</code>	greater than or equal to	<code>5.0 &gt;= 5.0</code>	true

# Logical operators

- Tests can be combined using *logical operators*:

Operator	Description	Example	Result
&&	and	(2 == 3) && (-1 < 5)	false
	or	(2 == 3)    (-1 < 5)	true
!	not	!(2 == 3)	true

- "Truth tables" for each, used with logical values  $p$  and  $q$ :

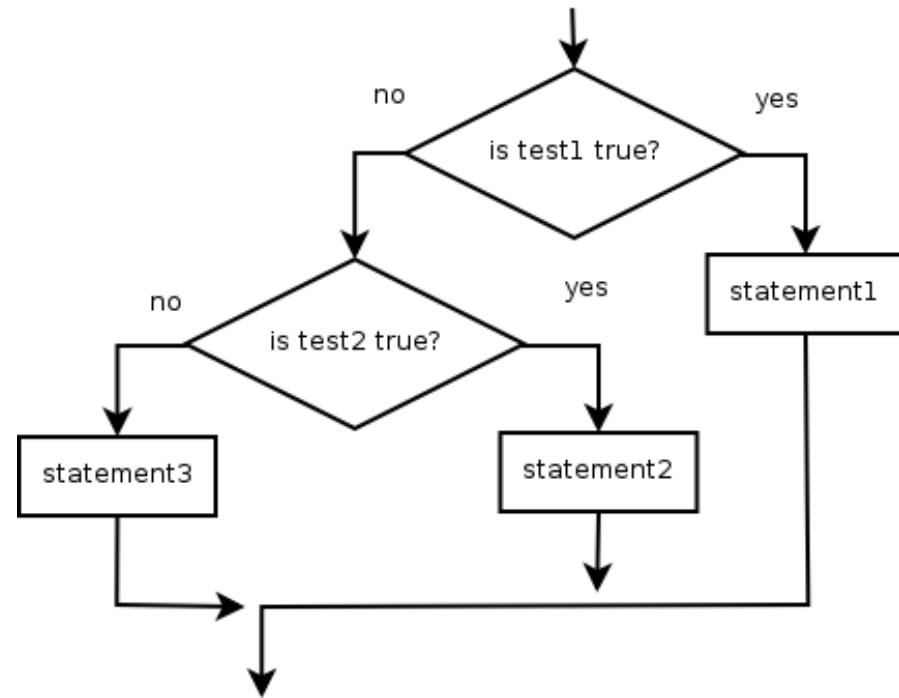
<b>p</b>	<b>q</b>	<b>p &amp;&amp; q</b>	<b>p    q</b>
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

<b>p</b>	<b>!p</b>
true	false
false	true

# Nested if/else

*Chooses between outcomes using many tests*

```
if (test) {  
    statement(s);  
} else if (test) {  
    statement(s);  
} else {  
    statement(s);  
}
```



► Example:

```
if (x > 0) {  
    System.out.println("Positive");  
} else if (x < 0) {  
    System.out.println("Negative");  
} else {  
    System.out.println("Zero");  
}
```



# Exercises

- ▶ Write a method that prints out if it is good weather to go for a bike ride. The weather is good if the temperature is between 40 degrees and 100 degrees inclusive unless it is raining, in which case the temperature must be between 70 degrees and 110 degrees inclusive
- ▶ Write a method that returns the largest of three numbers using if statements
- ▶ Write a method that determines if one day is before another day (given month and day) 15

# Exercise

- ▶ Prompt the user to enter two people's heights in inches.
  - Each person should be classified as one of the following:
    - short (under 5'3")
    - medium(5'3" to 5'11")
    - tall (6' or over)
  - The program should end by printing which person is taller.

Height in feet and inches: 5 7  
You are medium.

Height in feet and inches: 6 1  
You are tall.

Person #2 is taller than person #1.

# Exercises

- ▶ Write a method that simulates rolling 2 six sided dice a given number of time and returns the number of times a given value is the sum of the two dice when rolled.
- ▶ Write a method that determines if a number is a perfect number. A perfect number equals the sum of its integer divisors, excluding itself
  - $6 = 1 + 2 + 3$ , perfect
  - $8 > 1 + 2 + 4$ , deficient
  - $12 < 1 + 2 + 3 + 4 + 6$ , excessive

# Exercises

- ▶ Write a method that determines if we have time to go out for lunch. Inputs are distance to restaurant, average walking speed, time required to finish meal, time available, expected cost of meal, and money available
- ▶ times are expressed as whole number of minutes
- ▶ money is expressed as a double

# Topic 12

## more if/else, cumulative algorithms, printf

"We flew down weekly to meet with IBM, but they thought the way to measure software was the amount of code we wrote, when really the better the software, the fewer lines of code."

-Bill Gates



# Clicker 1

```
int a = 6;
if (a < 6)
    a = a + 1;
    System.out.println("a incremented.");
if (a > 6) {
    System.out.println("a is too high.");
} else {
    System.out.println("a is correctly set.");
}
```

What is output by the code above when it is run?

- A. a incremented.
- B. a is too high.
- C. a is correctly set.
- D. syntax error
- E. Something other than the answers listed here

# Clicker 2

```
int x = 4;
int y = 5;
x = mystery(x, y);
System.out.print(x + " " + y);
y = mystery(x, x);
System.out.print(" " + x + " " + y);

public static int mystery(int x, int y) {
    x *= 3;
    y = x / y;
    return x + y;
}
```

What is output by the code above when it is run?

- A. 4 5 4 5
- B. 14 5 14 45
- C. 14 5 14 5
- D. 14 5 50 5
- E. 14 5 14 50

# Java's Math class

Method name	Description		
<code>Math.abs(<i>value</i>)</code> double	absolute value		
<code>Math.ceil(<i>value</i>)</code> double	rounds up		
<code>Math.floor(<i>value</i>)</code> double	rounds down		
<code>Math.log10(<i>value</i>)</code>	logarithm, base 10		
<code>Math.max(<i>value1</i>, <i>value2</i>)</code>	larger of two values		
<code>Math.min(<i>value1</i>, <i>value2</i>)</code>	smaller of two values		
<code>Math.pow(<i>base</i>, <i>exp</i>)</code>	<i>base</i> to the <i>exp</i> power		
<code>Math.random()</code>	random double between 0 and 1		
<code>Math.round(<i>value</i>)</code> int	nearest whole number		
<code>Math.sqrt(<i>value</i>)</code>	square root		
<code>Math.sin(<i>value</i>)</code> <code>Math.cos(<i>value</i>)</code> <code>Math.tan(<i>value</i>)</code>	sine/cosine/tangent of an angle in radians	<b>Constant</b>	<b>Description</b>
<code>Math.toDegrees(<i>value</i>)</code>	convert degrees to radians and back	<code>Math.E</code>	2.7182818...
<code>Math.toRadians(<i>value</i>)</code>		<code>Math.PI</code>	3.1415926...



# Clicker Question 3

- ▶ What values do the following statements return?

`Math.round(-10.2)`

`Math.round(-10.8)`

`Math.ceil(-10.2)`

`Math.ceil(-10.8)`

`Math.floor(-10.2)`

`Math.floor(-10.8)`

**A: -10, -11, -10.0, -11.0, -11.0, -11.0**

**B: -10, -11, -11.0, -11.0, -10.0, -10.0**

**C: -10, -11, -10.0, -10.0, -11.0, -11.0**

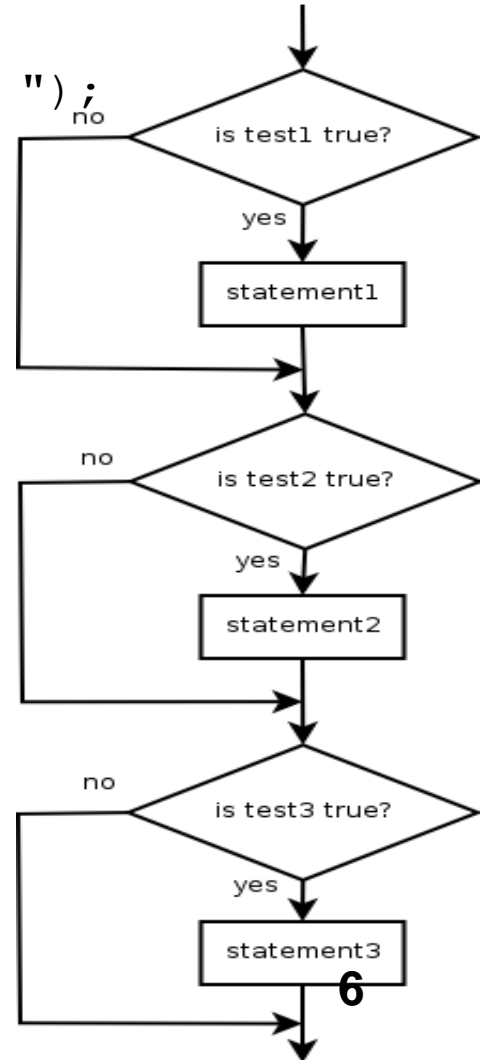
**D: -10, -10, -10.0, -11.0, -11.0, -11.0**

**E: Something else**

# Misuse of `if`

## ► What's wrong with the following code?

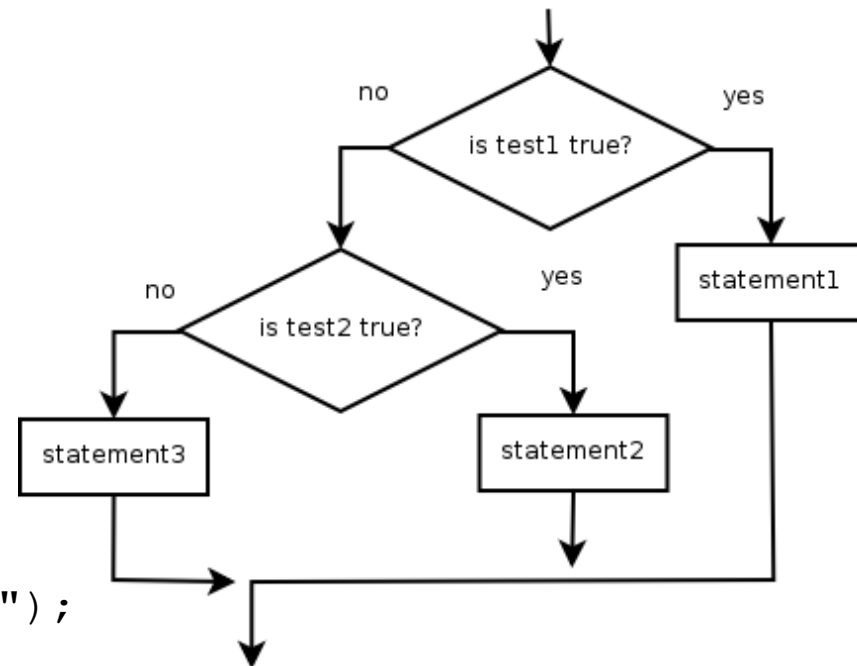
```
Scanner console = new Scanner(System.in);
System.out.print("What percentage did you earn? ");
int percent = console.nextInt();
if (percent >= 90) {
    System.out.println("You got an A!");
}
if (percent >= 80) {
    System.out.println("You got a B!");
}
if (percent >= 70) {
    System.out.println("You got a C!");
}
if (percent >= 60) {
    System.out.println("You got a D!");
}
if (percent < 60) {
    System.out.println("You got an F!");
}
...
```



# Nested if/else

*Chooses between outcomes using many tests*

```
if (test) {  
    statement(s);  
} else if (test) {  
    statement(s);  
} else {  
    statement(s);  
}
```



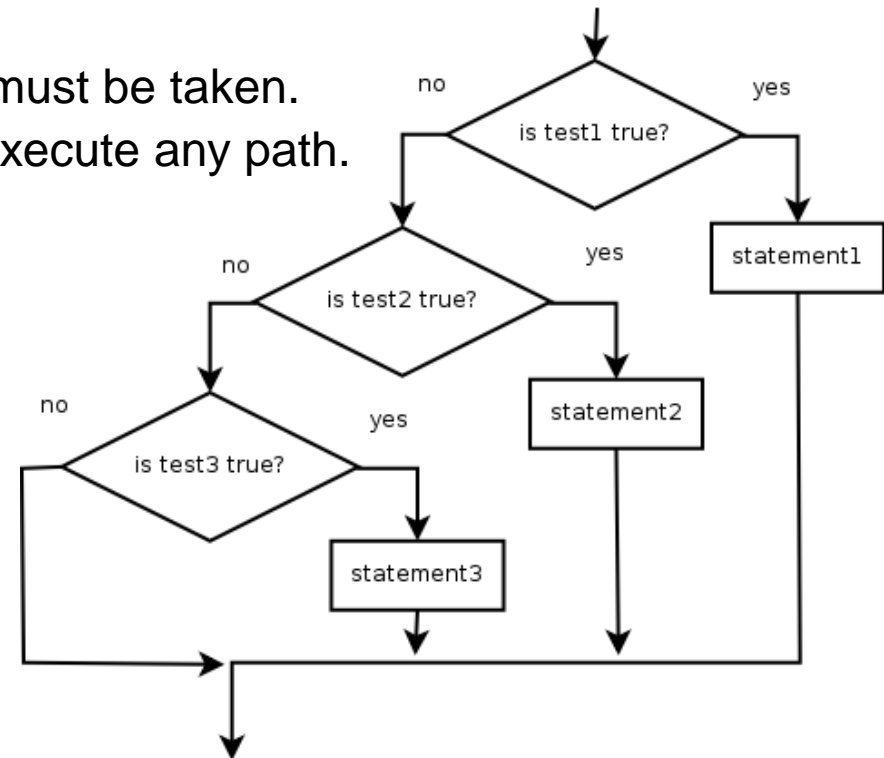
► Example:

```
if (x > 0) {  
    System.out.println("Positive");  
} else if (x < 0) {  
    System.out.println("Negative");  
} else {  
    System.out.println("Zero");  
}
```

# Nested if/else/if

- If it ends with `else`, exactly one path must be taken.
- If it ends with `if`, the code might not execute any path.

```
if (test 1) {  
    statement(s);  
} else if (test 2) {  
    statement(s);  
} else if (test 3) {  
    statement(s);  
}
```



► Example:

```
if (place == 1) {  
    System.out.println("Gold medal!");  
} else if (place == 2) {  
    System.out.println("Silver medal!");  
} else if (place == 3) {  
    System.out.println("Bronze medal.");  
}
```

# Nested `if` structures

- exactly 1 path (*mutually exclusive*)

```
if (test) {  
    statement(s);  
} else if (test) {  
    statement(s);  
} else {  
    statement(s);  
}
```

- 0 or 1 path (*mutually exclusive*)

```
if (test) {  
    statement(s);  
} else if (test) {  
    statement(s);  
} else if (test) {  
    statement(s);  
}
```

- 0, 1, or many paths (*independent tests; not exclusive*)

```
if (test) {  
    statement(s);  
}  
if (test) {  
    statement(s);  
}  
if (test) {  
    statement(s);  
}
```

# Which nested if/else?

## ▸ (1) if/if/if (2) nested if/else (3) nested if/else/if

- Whether a user is lower, middle, or upper-class based on income.
  - **(2)** nested `if / else if / else`
- Whether you made the dean's list ( $\text{GPA} \geq 3.8$ ) or honor roll (3.5-3.8).
  - **(3)** nested `if / else if`
- Whether a number is divisible by 2, 3, and/or 5.
  - **(1)** sequential `if / if / if`
- Computing a grade of A, B, C, D, or F based on a percentage.
  - **(2)** nested `if / else if / else if /  
else if / else`

# if/else with return

- ▶ The following two versions of a `max` method don't compile:

```
public static int max(int a, int b) {  
    if (a > b) {  
        return a;  
    }  
    // Error: not all paths return a value  
}
```

```
public static int max(int a, int b) {  
    if (a > b) {  
        return a;  
    } else if (b >= a) {  
        return b;  
    }  
}
```

- The compiler thinks `if/else/if` code might skip all paths, even though mathematically it must choose one or the other.

# All paths must return

- ▶ This version of `max` does compile and works:

// Returns the larger of the two given integers.

```
public static int max(int a, int b) {  
    if (a > b) {  
        return a;  
    } else {  
        return b;  
    }  
}
```

- ▶ Methods can return different values using `if/else`
  - Whichever path the code enters, it will return that value.
  - Returning a value causes a method to immediately exit.
  - All paths through the code must reach a `return` statement.



# FORMATTING WITH PRINTF

# Formatting text with `printf`

```
System.out.printf("format string", parameters);
```

- A format string can contain *placeholders* to insert parameters:

`%d`            integer

`%f`            real number

`%s`            string

- these placeholders are used instead of concatenation (+)

## –Example:

```
int x = 3;
int y = -17;
System.out.printf("x is %d and y is %d!\n", x, y);
// x is 3 and y is -17!
```

- `printf` does not insert a newline unless you add `\n`<sup>14</sup>

# printf width

`%Wd` integer, **W** characters wide, right-aligned  
`%-Wd` integer, **W** characters wide, *left*-aligned  
`%Wf` real number, **W** characters wide, right-aligned  
...

```
for (int i = 1; i <= 3; i++) {  
    for (int j = 1; j <= 10; j++) {  
        System.out.printf("%4d", (i * j));  
    }  
    System.out.println();    // to end the line  
}
```

Output:

1	2	3	4	5	6	7	8	9	10	
2	4	6	8	10	12	14	16	18	20	
3	6	9	12	15	18	21	24	27	30	<b>15</b>

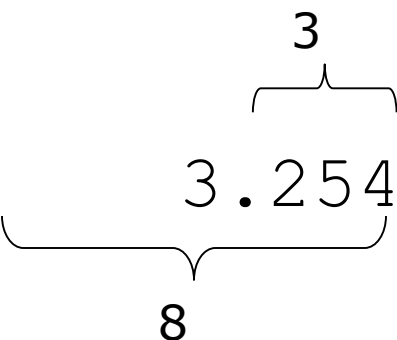
# printf precision

- `% .Df` real number, rounded to **D** digits after decimal
- `%W.Df` real number, **W** chars wide, **D** digits after decimal
- `%-W.Df` real number, **W** wide (left-align), **D** after decimal

```
double gpa = 3.253764;  
System.out.printf("your GPA is %.1f\n", gpa);  
System.out.printf("more precisely: %8.3f\n",  
gpa);
```

Output:

```
your GPA is 3.3  
more precisely: 3.254
```



# Cumulative algorithms

**reading: 4.2**

# Adding many numbers

- ▶ How would you find the sum of all integers from 1-1000?

```
// This may require a lot of typing  
int sum = 1 + 2 + 3 + 4 + ... + 999 + 1000;  
System.out.println("The sum is " + sum);
```

- ▶ What if we want the sum from 1 - 1,000,000?  
Or the sum up to any maximum?
  - How can we generalize the above code?

# A failed attempt

- ▶ An incorrect solution for summing 1-1000:

```
for (int i = 1; i <= 1000; i++) {  
    int sum = 0;  
    sum = sum + i;  
}  
  
// error: sum is undefined here  
System.out.println("The sum is " + sum);
```

- `sum`'s scope is in the `for` loop, so the code does not compile.
- ▶ **cumulative sum**: A variable that keeps a sum in progress and is updated repeatedly until summing is finished.
  - The `sum` above is an incorrect attempt at a cumulative sum.

# Corrected cumulative sum

```
int sum = 0;  
for (int i = 1; i <= 1000; i++) {  
    sum = sum + i;  
}  
System.out.println("The sum is " + sum);
```

- Cumulative sum variables must be declared *outside* the loops that update them, so that they will still exist after the loop.



# Cumulative product

- ▶ This cumulative idea can be used with other operators:

```
int product = 1;  
for (int i = 1; i <= 20; i++) {  
    product = product * 2;  
}  
System.out.println("2 ^ 20 = " + product);
```

- How would we make the base and exponent adjustable?

# Cumulative sum question

- ▶ Modify the `Receipt` program from Ch 2 (tax 8%, tip 15%).
  - Prompt for how many people, and each person's dinner cost.
  - Use static methods to structure the solution.
- ▶ Example log of execution:

```
How many people ate? 4
Person #1: How much did your dinner cost? 20.00
Person #2: How much did your dinner cost? 15
Person #3: How much did your dinner cost? 30.0
Person #4: How much did your dinner cost? 10.00
```

```
Subtotal: $ 75.00
Tax:      $  6.00
Tip:      $ 11.25
Total:    $ 92.25
```

# Cumulative sum answer

```
// This program enhances our Receipt program using a cumulative sum.
import java.util.*;

public class Receipt2 {
    public static void main(String[] args) {
        Scanner console = new Scanner(System.in);
        double subtotal = meals(console);
        results(subtotal);
    }

    // Prompts for number of people and returns total meal subtotal.
    public static double meals(Scanner console) {
        System.out.print("How many people ate? ");
        int people = console.nextInt();
        double subtotal = 0.0;           // cumulative sum

        for (int i = 1; i <= people; i++) {
            System.out.print("Person #" + i +
                             ": How much did your dinner cost? ");
            double personCost = console.nextDouble();
            subtotal = subtotal + personCost; // add to sum
        }
        return subtotal;
    }
    ...
}
```

# printf answer (partial)

...

```
// Calculates total owed, assuming 8% tax and 15% tip
public static void results(double subtotal) {
    double tax = subtotal * .08;
    double tip = subtotal * .15;
    double total = subtotal + tax + tip;

    System.out.printf("Subtotal:  $%6.2f\n", subtotal);
    System.out.printf("Tax:       $%6.2f\n", tax);
    System.out.printf("Tip:        $%6.2f\n", tip);
    System.out.printf("Total:     $%6.2f\n", total);
}
}
```

# Case Study

- ▶ Write program that prompts for exam and homework grades and prints out the letter grade for the student.
- ▶ Each exam has a weight
- ▶ Homeworks, as a whole, have a weight.
- ▶ The sum of the weights shall equal 100.
- ▶ Calculates numeric grade and prints out letter grade: A, B, C, D, F
- ▶ Program does not perform any error checking

# Sample Output

Enter grades to calculate letter grade.

Number of midterms? **2**

Midterm 1:

Weight (1 - 100)? **15**

Score? **82**

Scores bumped? (1=yes, 2=no)? **1**

Bump amount? **5**

Raw points: 87 / 100

Weighted points: 13.1 / 15

Midterm 2:

Weight (1 – 100)? **20**

Score? **93**

Scores bumped? (1=yes, 2=no)? **2**

Raw points: 93 / 100

Weighted points: 18.6 / 20

Final Exam:

Weight (1 – 100)? **30**

Score? **97**

Scores bumped? (1=yes, 2=no)? **1**

Bump amount? **10**

Raw points: 100 / 100

Weighted points: 30.0 / 30

Homeworks:

Weight is 35

Number of homeworks? **4**

Homework 1 score? **13**

Homework 2 score? **19**

Homework 3 score? **18**

Homework 4 score? **17**

Raw points: 68 / 80

Weighted points: 29.3 / 35

Total weighted points: 91.4 / 100

Final grade: A



# Topic 13

## procedural design and Strings

“Ugly programs are like ugly suspension bridges: they're much more liable to collapse than pretty ones, because the way humans (especially engineer-humans) perceive beauty is intimately related to our ability to process and understand complexity.”

- Eric S. Raymond,

Author of *The Cathedral and the Bazaar*

# Nested if/else question

Formula for body mass index (BMI):

$$BMI = \frac{weight}{height^2} \times 703$$

- Write a program that produces output like the following:

This program reads data for two people and computes their body mass index (BMI) and weight status.

Enter next person's information:

height (in inches)? 73.5

weight (in pounds)? 230

BMI = 29.93

overweight

Enter next person's information:

height (in inches)? 71

weight (in pounds)? 220.5

BMI = 30.75

obese

Difference = 0.82

BMI	Weight class
below 18.5	underweight
18.5 - 24.9	normal
25.0 - 29.9	overweight
30.0 and up	obese

# One-person, no methods

```
import java.util.*;

public class BMI {
    public static void main(String[] args) {
        System.out.println("This program reads ... (etc.)");
        Scanner console = new Scanner(System.in);

        System.out.println("Enter next person's information:");
        System.out.print("height (in inches)? ");
        double height = console.nextDouble();

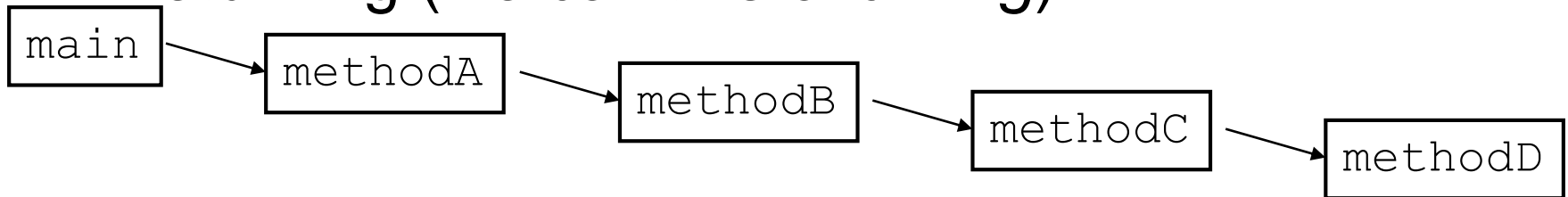
        System.out.print("weight (in pounds)? ");
        double weight = console.nextDouble();

        double bmi = weight * 703 / height / height;

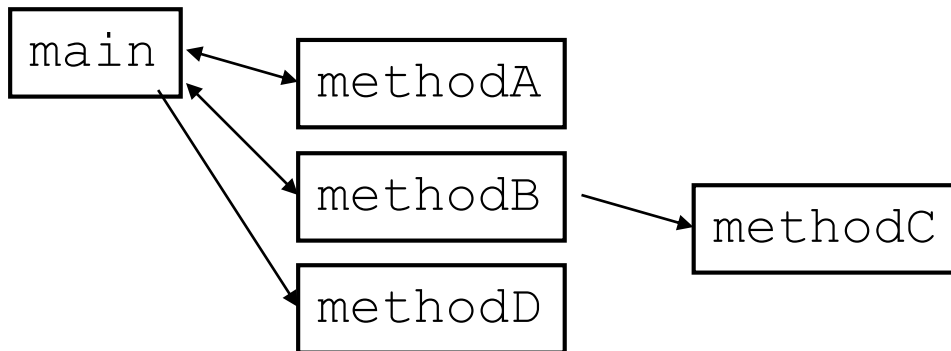
        System.out.printf("BMI = %.2f\n", bmi);
        if (bmi < 18.5) {
            System.out.println("underweight");
        } else if (bmi < 25) {
            System.out.println("normal");
        } else if (bmi < 30) {
            System.out.println("overweight");
        } else {
            System.out.println("obese");
        }
    }
}
```

# "Chaining"

- ▶ `main` should be a concise summary of your program.
  - It is bad if each method calls the next without ever returning (we call this *chaining*):



- ▶ A better structure has `main` make most of the calls.
  - Methods must return values to `main` to be passed on later.



# Bad "chain" code

```
public class BMI {
    public static void main(String[] args) {
        System.out.println("This program reads ... (etc.)");
        Scanner console = new Scanner(System.in);
        person(console);
    }

    public static void person(Scanner console) {
        System.out.println("Enter next person's information:");
        System.out.print("height (in inches)? ");
        double height = console.nextDouble();
        getWeight(console, height);
    }

    public static void getWeight(Scanner console, double height) {
        System.out.print("weight (in pounds)? ");
        double weight = console.nextDouble();
        computeBMI(console, height, weight);
    }

    public static void computeBMI(Scanner s, double h, double w) {
        ...
    }
}
```

# Procedural heuristics

1. Each method should have a clear responsibility.
2. No method should do too large a share of the overall task.
3. Minimize coupling and dependencies between methods.
4. The main method should read as a concise summary of the overall set of tasks performed by the program.
5. Variables should be declared/used at the lowest level possible.

# Better solution

```
// This program computes two people's body mass index (BMI) and
// compares them.
// The code uses Scanner for input, and parameters/returns.
```

```
import java.util.Scanner;

public class BMI {
    public static void main(String[] args) {
        introduction();
        Scanner console = new Scanner(System.in);
        double bmi1 = person(console);
        double bmi2 = person(console);

        // report overall results
        report(1, bmi1);
        report(2, bmi2);
        System.out.println("Difference      = "
                           + Math.abs(bmi1 - bmi2));
    }

    // prints a welcome message explaining the program
    public static void introduction() {
        System.out.println("This program reads ...");
        // ...
    }
}
```

...

# Better solution, cont'd.

```
// reads information for one person, computes their BMI, and returns it
public static double person(Scanner console) {
    System.out.println("Enter next person's information:");
    System.out.print("height (in inches)? ");
    double height = console.nextDouble();

    System.out.print("weight (in pounds)? ");
    double weight = console.nextDouble();
    System.out.println();

    return bmi(height, weight);
}

// Computes/returns a person's BMI based on their height and weight.
public static double bmi(double height, double weight) {
    return weight * 703 / (height * height);
}

// Outputs information about a person's BMI and weight status.
public static void report(int number, double bmi) {
    System.out.printf("Subject%5dBMI = %.2f\n", number, bmi);
    if (bmi < 18.5) {
        System.out.println("underweight");
    } else if (bmi < 25) {
        System.out.println("normal");
    } else if (bmi < 30) {
        System.out.println("overweight");
    } else {
        System.out.println("obese");
    }
}

}
```



# Strings

- ▶ **string**: An object storing a sequence of text characters.
  - Unlike most other objects, a `String` is not always created with `new`.

```
String name = "text";
```

```
String name = expression;
```

- Examples:

```
String name = "Marla Singer";
```

```
int x = 3;
```

```
int y = 5;
```

```
String point = "(" + x + ", " + y + ")";
```

# Indexes

- ▶ Characters of a string are numbered with 0-based *indexes*:

```
String name = "K. Scott";
```

index	0	1	2	3	4	5	6	7
character	K	.		S	c	o	t	t

- First character's index : 0 (zero based indexing)
- Last character's index : 1 less than the string's length
- The individual characters are values of type `char` (another primitive data type)

# String methods

Method name	Description
<code>indexOf(<b>str</b>)</code>	index where the start of the given string appears in this string (-1 if not found)
<code>length()</code>	number of characters in this string
<code>substring(<b>index1</b>, <b>index2</b>)</code> or <code>substring(<b>index1</b>)</code>	the characters in this string from <i>index1</i> (inclusive) to <i>index2</i> ( <u>exclusive</u> ); if <i>index2</i> is omitted, grabs till end of string
<code>toLowerCase()</code>	a new string with all lowercase letters
<code>toUpperCase()</code>	a new string with all uppercase letters

- These methods are called using the dot notation:

```
String student = "Olivia Scott";  
System.out.println(student.length());    // 12
```

# String method examples

```
// index      012345678901
String s1 = "Olivia Scott";
String s2 = "Isabelle Scott";
System.out.println(s2.length());    // 14
System.out.println(s1.indexOf("e")); // -1
System.out.println(s2.indexOf("e")); // 4
System.out.println(s1.substring(7, 10)); // "Sco"
String s3 = s2.substring(4, 10);
System.out.println(s3.toLowerCase()); // "elle s"
```

- ▶ Given the following string:

```
// index      0123456789012345678901
String book = "Building Java Programs";
```

- How would you extract the word "Building" ?  
(Write code that can extract the first word from any string.)

# Clicker 1

► What is output by the following code?

```
String s1 = "Football";  
String s2 = s1.substring(4, 8);  
s2.substring(1);  
System.out.print(s2);
```

- A. Football
- B. ball
- C. all
- D. No output due to syntax error.
- E. No output due to runtime error.

# Modifying strings

- ▶ Methods like `substring` and `toLowerCase` build and return a new string, rather than modifying the current string.

```
String s = "ut Longhorns";  
s.toUpperCase();  
System.out.println(s);    // ut Longhorns
```

- ▶ To modify a variable's value, you must reassign it:

```
String s = "ut Longhorns";  
s = s.toUpperCase();  
System.out.println(s);    // UT LONGHORNS
```

# Strings as user input

- ▶ Scanner's `nextLine` method reads a word of input as a `String`.

```
Scanner console = new Scanner(System.in);
System.out.print("What is your first name? ");
String name = console.nextLine();
System.out.println(name + " has " + name.length() +
    " letters and starts with " + name.substring(0, 1));
```

Output:

```
What is your first name? Chamillionaire
Chamillionaire has 14 letters and starts with C
```

- ▶ The `nextLine` method reads a line of input as a `String`.

```
System.out.print("What is your address? ");
String address = console.nextLine();
```

# Clicker 2

► What is output by the following code?

```
String s1 = "taxicab";  
String s2 = "acables";  
String s3 = s1.substring(4);  
String s4 = s2.substring(1, 4);  
if (s3.length() == s4.length())  
    System.out.print("1");  
else  
    System.out.print("2");  
if (s3 == s4)  
    System.out.print("1");  
else  
    System.out.print("2");
```

- A. 11
- B. 12
- C. 21
- D. 22
- E. No output due to syntax error



# Comparing Strings

- ▶ Relational operators such as `<` and `<=` are undefined on objects in Java.
- ▶ `==` is defined but normally doesn't work as intended

```
Scanner console = new Scanner(System.in);
System.out.print("What is your name? ");
String name = console.next();
if (name == "Barney") {
    System.out.println("I love you, you love me,");
    System.out.println("We're a happy family!");
}
```

- This code will compile, but it will not print the song.
- `==` compares objects by *references* (seen later), so it often gives `false` even when two `Strings` have the same letters.

# The equals method

- Objects are compared using a method named `equals`.

```
Scanner console = new Scanner(System.in);
System.out.print("What is your name? ");
String name = console.next();
if (name.equals("Barney")) {
    System.out.println("Fred's Friend.");
    System.out.println("Purple Dinasuar.");
    System.out.println("In trouble.");
}
```

- The `equals` method returns a value of type `boolean`, the type used in logical tests.

# String test methods

Method	Description
<code>equals(str)</code>	whether two strings contain the same characters
<code>equalsIgnoreCase(str)</code>	whether two strings contain the same characters, ignoring upper vs. lower case
<code>startsWith(str)</code>	whether one contains other's characters at start
<code>endsWith(str)</code>	whether one contains other's characters at end
<code>contains(str)</code>	whether the given string is found within this one

```
String name = console.next();  
if (name.startsWith("Prof")) {  
    System.out.println("When are your office hours?");  
} else if (name.endsWith("OBE")) {  
    System.out.println("Yes Sir!");  
}
```

# Strings questions

- ▶ Write a method to determine if a String is a possible representation of a DNA strand
  - contains only A, C, T, and G
- ▶ Write a method to create a *Watson-Crick complement* given a String that represents a strand of DNA
  - replace A with T, C with G, and vice versa
- ▶ Given a String that represents a strand of DNA return the first substring that exists between "ATG" and either "TAG" or "TGA"
  - no overlap allowed

# String Questions

- ▶ Write a method that returns the number of times a given character occurs in a String
- ▶ Write a method that returns the number of times the punctuation marks . ? ! , : " ; ' occur in a String

# Topic 14

## while loops and loop patterns

"Given enough eyeballs, all bugs are shallow (e.g., given a large enough beta-tester and co-developer base, almost every problem will be characterized quickly and the fix obvious to someone)."

**-Linus's Law, by Eric Raymond**



# A deceptive problem...

- ▶ Write a method `printNumbers` that prints each number from 1 to a given maximum, separated by commas.

For example, the call:

```
printNumbers(5)
```

should print:

```
1, 2, 3, 4, 5
```

# Flawed solutions

```
public static void printNumbers(int max) {  
    for (int i = 1; i <= max; i++) {  
        System.out.print(i + ", ");  
    }  
    System.out.println(); // to end the line of output  
}
```

– Output from `printNumbers(5)`: 1, 2, 3, 4, 5,

```
public static void printNumbers(int max) {  
    for (int i = 1; i <= max; i++) {  
        System.out.print(", " + i);  
    }  
    System.out.println(); // to end the line of output  
}
```

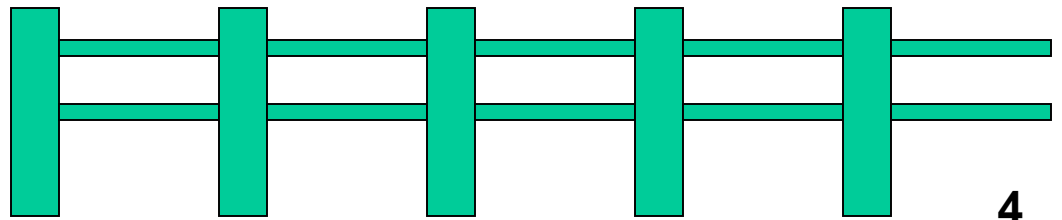
– Output from `printNumbers(5)`: , 1, 2, 3, 4, 5



# Fence post analogy

- ▶ We print  $n$  numbers but need only  $n - 1$  commas.
- ▶ Similar to building a fence with wires separated by posts:
  - If we use a flawed algorithm that repeatedly places a post + wire, the last post will have an extra dangling wire.

```
for (length of fence) {  
    place a post.  
    place some wire.  
}
```



# Fencepost loop

- ▶ Add a statement outside the loop to place the initial "post."
  - Also called a *fencepost loop* or a "loop-and-a-half" solution.

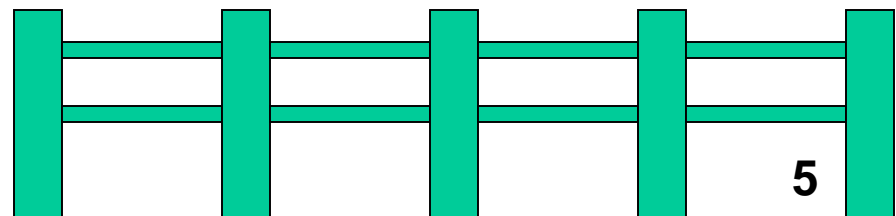
***place a post.***

***for (length of fence - 1) {***

***place some wire.***

***place a post.***

***}***



# Fencepost method solution

```
public static void printNumbers(int max) {  
    System.out.print(1);  
    for (int i = 2; i <= max; i++) {  
        System.out.print(", " + i);  
    }  
    System.out.println(); // to end the line  
}
```

- ▶ Alternate solution: Either first or last "post" can be taken out:

```
public static void printNumbers(int max) {  
    for (int i = 1; i <= max - 1; i++) {  
        System.out.print(i + ", ");  
    }  
    System.out.println(max); // to end the line  
}
```

# Fencepost question

- ▶ Modify your method `printNumbers` into a new method `printPrimes` that prints all *prime* numbers up to a max.
  - Example: `printPrimes(50)` prints  
2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43,  
47
  - If the maximum is less than 2, print no output.
- ▶ To help you, write a method `countFactors` which returns the number of factors of a given integer.
  - `countFactors(20)` returns 6 due to factors 1, 2, 4, 5, 10, 20.

# Fencepost answer

**// Prints all prime numbers up to the given max.**

```
public static void printPrimes(int max) {  
    if (max >= 2) {  
        System.out.print("2");  
        for (int i = 3; i <= max; i++) {  
            if (countFactors(i) == 2) {  
                System.out.print(", " + i);  
            }  
        }  
        System.out.println();  
    }  
}
```

**// Returns how many factors the given number has.**

```
public static int countFactors(int number) {  
    int count = 0;  
    for (int i = 1; i <= number; i++) {  
        if (number % i == 0) {  
            count++;    // i is a factor of number  
        }  
    }  
    return count;  
}
```

`while loops`

**reading: 5.1**

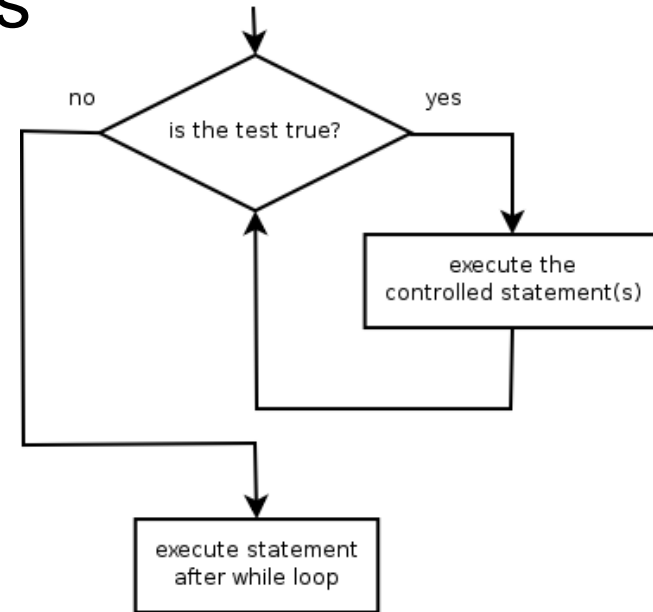
# Categories of loops

- ▶ **definite loop:** Executes a known number of times.
  - The `for` loops we have seen are definite loops.
    - Print "hello" 10 times.
    - Find all the prime numbers up to an integer  $n$ .
    - Print each odd number between 5 and 127.
- ▶ **indefinite loop:** One where the number of times its body repeats is not known in advance.
  - Prompt the user until they type a non-negative number.
  - Print random numbers until a prime number is printed.
  - Repeat until the user has typed "q" to quit.

# The while loop

- ▶ **while loop:** Repeatedly executes its body as long as a logical test is true.

```
while (<test>) {  
    <statement(s)>;  
}
```



- ▶ **Example:**

```
int num = 1;                // initialization  
while (num <= 200) {        // test  
    System.out.print(num + " ");  
    num = num * 2;           // update  
}
```

// output: 1 2 4 8 16 32 64 128



# Example while loop

```
// finds the first factor of 91, other than 1
int n = 91;
int factor = 2;
while (n % factor != 0) {
    factor++;
}
System.out.println("First factor is " +
                    factor);

// output: First factor is 7
```

- `while` is better than `for` because we don't know how many times we will need to increment to find the factor.

# Clicker 1

► What is output by the following code?

```
int x = 1;
int limit = 60;
int val = 1;
while(val < limit) {
    x *= 2;
}
System.out.print(x);
```

A. 1

B. 32

C. 64

D. No output due to syntax error

E. No output due to some other reason

# Sentinel values

- ▶ **sentinel:** A value that signals the end of user input.
  - **sentinel loop:** Repeats until a sentinel value is seen.
- ▶ **Example:** Write a program that prompts the user for text until the user types nothing, then output the total number of characters typed.
  - (In this case, the *empty* string is the sentinel value.)

```
Type a line (or nothing to exit): hello  
Type a line (or nothing to exit): this is a line  
Type a line (or nothing to exit):  
You typed a total of 19 characters.
```

# Solution?

```
Scanner console = new Scanner(System.in);
int sum = 0;
String response = "dummy"; // "dummy" value, anything but ""

while (!response.equals("")) {
    System.out.print("Type a line (or nothing to exit): ");
    response = console.nextLine();
    sum += response.length();
}

System.out.println("You typed a total of " + sum + "
    characters.");
```

# Changing the sentinel value

- ▶ Modify your program to use "quit" as the sentinel value.

– Example log of execution:

Type a line (or "quit" to exit): hello|

Type a line (or "quit" to exit): this is a line

Type a line (or "quit" to exit): quit

You typed a total of 19 characters.

# Changing the sentinel value

- ▶ Changing the sentinel's value to "quit" does not work!

```
Scanner console = new Scanner(System.in);
int sum = 0;
String response = "dummy"; // "dummy" value, anything but "quit"

while (!response.equals("quit")) {
    System.out.print("Type a line (or \"quit\" to exit): ");
    response = console.nextLine();
    sum += response.length();
}
System.out.println("You typed a total of " + sum + "
    characters.");
```

- ▶ This solution produces the wrong output.  
Why?

You typed a total of 23 characters. <sup>17</sup>

# The problem with the code

- ▶ The code uses a pattern like this:

*sum = 0.*

*while (input is not the sentinel) {  
    prompt for input; read input.  
    add input length to the sum.  
}*

# problem with code

- ▶ On the last pass, the sentinel's length (4) is added to the sum:

*prompt for input; read input ("quit").*

*add input length (4) to the sum.*

- ▶ This is a fencepost problem.
  - Must read  $N$  lines, but only sum the lengths of the first  $N-1$ .



# A fencepost solution

*sum = 0.*

*prompt for input; read input.      // place a "post"*

*while (input is not the sentinel) {*

*add input length to the sum.    // place a "wire"*

*prompt for input; read input.    // place a "post"*

*}*

- ▶ Sentinel loops often utilize a fencepost "loop-and-a-half" style solution by pulling some code out of the loop.

# Correct code

```
Scanner console = new Scanner(System.in);
int sum = 0;

// pull one prompt/read ("post") out of the loop
System.out.print("Type a line (or \"quit\" to exit): ");
String response = console.nextLine();

while (!response.equals("quit")) {
    sum += response.length();    // moved to top of loop
    System.out.print("Type a line (or \"quit\" to exit): ");
    response = console.nextLine();
}

System.out.println("You typed a total of " + sum + "
    characters.");
```

# Sentinel as a constant

```
public static final String SENTINEL = "quit";
...

Scanner console = new Scanner(System.in);
int sum = 0;

// pull one prompt/read ("post") out of the loop
System.out.print("Type a line (or \" + SENTINEL + "\" to exit): ");
String response = console.nextLine();

while (!response.equals(SENTINEL)) {
    sum += response.length();    // moved to top of loop
    System.out.print("Type a line (or \" + SENTINEL + "\" to exit): ");
    response = console.nextLine();
}

System.out.println("You typed a total of " + sum + " characters.");
```

# examples

- ▶ write a method to improve checking if a number is prime or not
  - when can we stop?
- ▶ Write a program that flips a coin until there is a run of 10 flips of the same side in a row
  - how many flips were there before 10 in a row?
  - repeat the experiment 1000 times, what is the average number of flips

# Topic 15

## boolean methods and random numbers

"It is a profoundly erroneous truism, repeated by all the copybooks, and by eminent people when they are making speeches, that we should cultivate the habit of thinking of what we are doing. The precise opposite is the case.

**Civilization advances by extending the number of operations which we can perform without thinking about them.** Operations of thought are like cavalry charges in a battle:

- they are strictly limited in number,
- they require fresh horses, and
- must only be made at decisive moments."

**-Alfred North Whitehead**



# Programming Terminology

- ▶ Binary digit, 1 or 0
- ▶ Byte, 8 bits
  - Nibble, half a byte, 4 bits
- ▶ kilobyte, 1024 bytes ( $2^{10} = 1024$ )
- ▶ megabyte,  $2^{20}$  bytes, 1,048,576
  - 1,000,000 bytes in some contexts
- ▶ gigabyte,  $2^{30}$  bytes, 1,073,741,824
  - 1,000,000,000 bytes in some contexts

# Programming Terminology

- ▶ compile
- ▶ syntax error, compile error, runtime error, logic error
- ▶ high level language
- ▶ class
- ▶ object

# Clicker 1

► What is the base 2 representation of  $67_{10}$ ?

A. 100011

B. 111111

C. 1000000

D. 2111

E. None of A-D are correct



► Write a method to convert a base 10 int to a base 2 String



# The keyword list thus far:

► Complete list of Java keywords:

abstract	default	<b>if</b>	private	this
<b>boolean</b>	do	implements	protected	throw
break	<b>double</b>	<b>import</b>	<b>public</b>	throws
byte	<b>else</b>	instanceof	<b>return</b>	transient
case	extends	<b>int</b>	short	try
catch	<b>final</b>	interface	<b>static</b>	<b>void</b>
<b>char</b>	finally	long	strictfp	volatile
<b>class</b>	float	native	super	<b>while</b>
const	<b>for</b>	<b>new</b>	switch	
continue	goto	package	synchronized	
assert	enum			

# Methods that are tests

- ▶ Some methods return logical values (`true` or `false`).
  - A call to such a method is used as a **<test>** in a loop or `if`.

```
Scanner console = new Scanner(System.in);  
System.out.print("Type your first name: ");  
String name = console.next();
```

```
if (name.startsWith("Dr.")) {  
    System.out.println("Med school or PhD?");  
} else if (name.endsWith("Esq.")) {  
    System.out.println("And I am Ted 'Theodore' Logan!");  
}
```

# String test methods

Method	Description
<code>equals(&lt;str&gt;)</code>	whether two strings contain the same characters
<code>equalsIgnoreCase(&lt;str&gt;)</code>	whether two strings contain the same characters, ignoring upper vs. lower case
<code>startsWith(&lt;str&gt;)</code>	whether one contains other's characters at start
<code>endsWith(&lt;str&gt;)</code>	whether one contains other's characters at end
<code>contains(&lt;str&gt;)</code>	whether the given string is found within this one

```
String name = console.next();  
if (name.contains("Prof")) {  
    System.out.println("When are your office hours?");  
} else if (name.equalsIgnoreCase("mavEriCk")) {  
    System.out.println("You're grounded, young man!");  
}
```

# Strings question

► Prompt the user for two words and report whether they:

- *"rhyme"* (end with the same last two letters)
- *alliterate* (begin with the same letter)

– Example output: (run #1)

```
Type two words: car STAR  
They rhyme!
```

(run #2)

```
Type two words: bare bear  
They alliterate!
```

(run #3)

```
Type two words: sell shell  
They alliterate!  
They rhyme!
```

(run #4)

```
Type two words: extra strawberry
```

# Strings answer

**// Determines whether two words rhyme and/or alliterate.**

```
import java.util.*;
```

```
public class Rhyme {
```

```
    public static void main(String[] args) {
```

```
        Scanner console = new Scanner(System.in);
```

```
        System.out.print("Type two words: ");
```

```
        String word1 = console.next().toLowerCase();
```

```
        String word2 = console.next().toLowerCase();
```

**// check whether they end with the same two letters**

```
if (word2.length() >= 2 &&
```

```
    word1.endsWith(word2.substring(word2.length() - 2))) {
```

```
    System.out.println("They rhyme!");
```

```
}
```

**// check whether they alliterate**

```
if (word1.startsWith(word2.substring(0, 1))) {
```

```
    System.out.println("They alliterate!");
```

```
}
```

```
}
```

```
}
```

# Random numbers

**reading: 5.1**

# The Random class

- ▶ A Random object generates pseudo-random numbers.
  - Class Random is found in the `java.util` package.

```
import java.util.Random;
```

Method name	Description
<code>nextInt()</code>	returns a random integer
<code>nextInt(&lt;<b>max</b>&gt;)</code>	returns a random integer in the range $[0, max)$ in other words, 0 to $max-1$ inclusive
<code>nextDouble()</code>	returns a random real number in the range $[0.0, 1.0)$

- Example:

```
Random rand = new Random();  
int randomNumber = rand.nextInt(10); // 0-9
```

# Generating random numbers

- ▶ Common usage: to get a random number from 1 to  $N$

```
int n = rand.nextInt(20) + 1;  
// 1-20 inclusive
```

- ▶ To get a number in arbitrary range  $[min, max]$  inclusive:

**$\langle name \rangle$** .nextInt( **$\langle size\ of\ range \rangle$** ) +  **$\langle min \rangle$**

- Where  **$\langle size\ of\ range \rangle$**  is ( **$\langle max \rangle$**  -  **$\langle min \rangle$**  + 1)
- Example: A random integer between 4 and 10 inclusive:

```
int n = rand.nextInt(7) + 4; 12
```



# Random questions

- ▶ Given the following declaration, how would you get:

```
Random rand = new Random();
```

- A random number between 1 and 47 inclusive?

```
int random1 = rand.nextInt(47) + 1;
```

- A random number between 23 and 30 inclusive?

```
int random2 = rand.nextInt(8) + 23;
```

- A random even number between 4 and 12 inclusive?

```
int random3 = rand.nextInt(5) * 2 + 4;
```

# Random and other types

- ▶ `nextDouble` method returns a double between [0.0 - 1.0)

- Example: Get a random GPA value between 1.5 and 4.0:

```
double randomGpa  
    = rand.nextDouble() * 2.5 + 1.5;
```

- ▶ Any set of possible values can be mapped to integers

- code to randomly play Rock-Paper-Scissors:

```
int r = rand.nextInt(3);  
if (r == 0) {  
    System.out.println("Rock");  
} else if (r == 1) {  
    System.out.println("Paper");  
} else { // r == 2  
    System.out.println("Scissors");  
}
```

# Random question

- ▶ Write a program that simulates rolling of two 6-sided dice until their combined result comes up as 7.

$$2 + 4 = 6$$

$$3 + 5 = 8$$

$$5 + 6 = 11$$

$$1 + 1 = 2$$

$$4 + 3 = 7$$

You won after 5 tries!

# Random answer

```
// Rolls two dice until a sum of 7 is reached.
```

```
import java.util.*;
```

```
public class Dice {
```

```
    public static void main(String[] args) {
```

```
        Random rand = new Random();
```

```
        int tries = 0;
```

```
        int sum = 0;
```

```
        while (sum != 7) {
```

```
            // roll the dice once
```

```
            int roll1 = rand.nextInt(6) + 1;
```

```
            int roll2 = rand.nextInt(6) + 1;
```

```
            sum = roll1 + roll2;
```

```
            System.out.println(roll1 + " + " + roll2 + " = " + sum);
```

```
            tries++;
```

```
        }
```

```
        System.out.println("You won after " + tries + " tries!");
```

```
    }
```

```
}
```

# Random question

- ▶ Write a program that plays an adding game.
  - Ask user to solve random adding problems with 2-5 numbers.
  - The user gets 1 point for a correct answer, 0 for incorrect.
  - The program stops after 3 incorrect answers.

$$4 + 10 + 3 + 10 = \underline{27}$$

$$9 + 2 = \underline{11}$$

$$8 + 6 + 7 + 9 = \underline{25}$$

Wrong! The answer was 30

$$5 + 9 = \underline{13}$$

Wrong! The answer was 14

$$4 + 9 + 9 = \underline{22}$$

$$3 + 1 + 7 + 2 = \underline{13}$$

$$4 + 2 + 10 + 9 + 7 = \underline{42}$$

Wrong! The answer was 32

You earned 4 total points.

# Random answer

```
// Asks the user to do adding problems and scores them.
import java.util.*;

public class AddingGame {
    public static void main(String[] args) {
        Scanner console = new Scanner(System.in);
        Random rand = new Random();

        // play until user gets 3 wrong
        int points = 0;
        int wrong = 0;
        while (wrong < 3) {
            int result = play(console, rand);    // play one game
            if (result == 0) {
                wrong++;
            } else {
                points++;
            }
        }

        System.out.println("You earned " + points + " total points.");
    }
}
```

# Random answer 2

...

```
// Builds one addition problem and presents it to the user.
// Returns 1 point if you get it right, 0 if wrong.
public static int play(Scanner console, Random rand) {
    // print the operands being added, and sum them
    int operands = rand.nextInt(4) + 2;
    int sum = rand.nextInt(10) + 1;
    System.out.print(sum);

    for (int i = 2; i <= operands; i++) {
        int n = rand.nextInt(10) + 1;
        sum += n;
        System.out.print(" + " + n);
    }
    System.out.print(" = ");

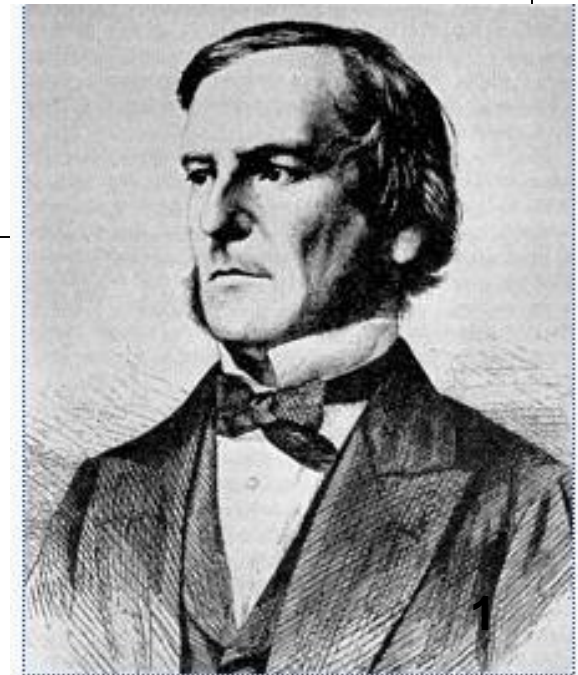
    // read user's guess and report whether it was correct
    int guess = console.nextInt();
    if (guess == sum) {
        return 1;
    } else {
        System.out.println("Wrong! The answer was " + total);
        return 0;
    }
}
```

# Topic 16

## boolean logic

"No matter how correct a mathematical theorem may appear to be, one ought never to be satisfied that there was not something imperfect about it until it also gives the impression of being beautiful."

- George Boole





# Type boolean

- ▶ **boolean**: A logical type whose values are `true` and `false`.
  - A logical **<test>** is actually a `boolean` expression.
  - Like other types, it is legal to:
    - create a `boolean` variable
    - pass a `boolean` value as a parameter
    - return a `boolean` value from methods
    - call a method that returns a `boolean` and use it as a test

```
boolean minor      = age < 21;  
boolean isProf     = name.contains("Prof");  
boolean lovesCS    = true;
```

```
// allow only CS-loving students over 21  
if (minor || isProf || !lovesCS) {  
    System.out.println("Can't enter the club!");  
}
```

# Using boolean

## ► Why is type `boolean` useful?

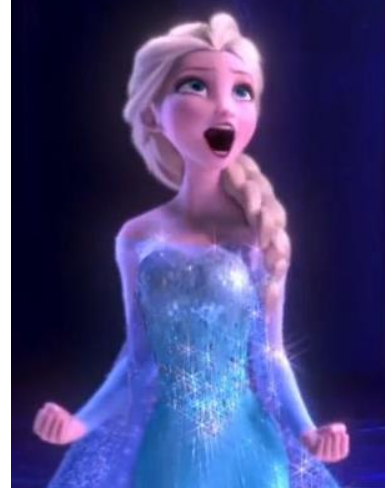
- Can capture a complex logical test result and use it later
- Can write a method that does a complex test and returns it
- Makes code more readable
- Can pass around the result of a logical test (as param/return)

```
boolean goodTemp      = 40 <= temp && temp <= 90;  
boolean goodHumidity = 20 <= humidity && humidity <= 70;  
boolean hasTime       = time >= 90; // minutes  
  
if ((goodTemp && goodHumidity) || hasTime) {  
    System.out.println("Let's RIDE BIKES!!!!");  
} else {  
    System.out.println("Maybe tomorrow");  
}
```

# Returning boolean

```
public static boolean isPrime(int n) {  
    int factors = 0;  
    for (int i = 1; i <= n; i++) {  
        if (n % i == 0) {  
            factors++;  
        }  
    }  
    // NOTE: GACKY STYLE AHEAD!! GACKY == BAD!!  
    if (factors == 2) {  
        return true;  
    } else {  
        return false;  
    }  
}
```

I CAN'T !!!!!!!



- Calls to methods returning `boolean` can be used as tests:

```
if (isPrime(57)) {  
    ...  
}
```

# Boolean question

- ▶ Improve our "rhyme" / "alliterate" program to use boolean methods to test for rhyming and alliteration.

Type two words: **Bare blare**

They rhyme!

They alliterate!

# Boolean answer

```
if (rhyme(word1, word2)) {  
    System.out.println("They rhyme!");  
}  
if (alliterate(word1, word2)) {  
    System.out.println("They alliterate!");  
}  
...
```

// Returns true if s1 and s2 end with the same two letters.

// NOTE: GACKY STYLE AHEAD!!

```
public static boolean rhyme(String s1, String s2) {  
    if (s2.length() >= 2 && s1.endsWith(s2.substring(s2.length() - 2))) {  
        return true;  
    } else {  
        return false;  
    }  
}
```

// Returns true if s1 and s2 start with the same letter.

// NOTE: GACKY STYLE AHEAD!!

```
public static boolean alliterate(String s1, String s2) {  
    if (s1.startsWith(s2.substring(0, 1))) {  
        return true;  
    } else {  
        return false;  
    }  
}
```

# "Boolean Zen", part 2

- ▶ Students new to `boolean` often test if a result is `true`:

```
if (isPrime(57) == true) {    // inelegant
    ...
}
```

- ▶ But this is unnecessary and redundant. Preferred:

```
if (isPrime(57)) {          // elegant, zen
    ...
}
```

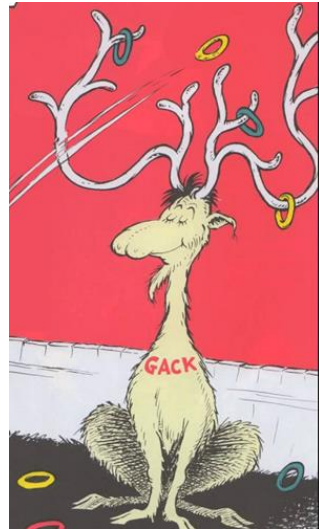
- ▶ A similar pattern can be used for a `false` test:

```
if (isPrime(57) == false) {    // inelegant
if (!isPrime(57)) {            // elegant, zen
```

# "Boolean Zen", part 2

- Programmers often write methods that return boolean often have an if/else that returns true or false:

```
// NOTE: GACKY STYLE AHEAD!!  
public static boolean bothOdd(int n1, int n2)  
{  
    if (n1 % 2 != 0 && n2 % 2 != 0) {  
        return true;  
    } else {  
        return false;  
    }  
}
```



– But the code above is unnecessarily verbose.

# Solution w/ boolean variable

- ▶ We could store the result of the logical test.

```
public static boolean bothOdd(int n1, int n2) {  
    boolean test = (n1 % 2 != 0 && n2 % 2 != 0);  
    // NOTE: BAD STYLE AHEAD!!  
    if (test) {    // test == true  
        return true;  
    } else {      // test == false  
        return false;  
    }  
}
```

- Notice: Whatever `test` is, we want to return that.
  - If `test` is `true` , we want to return `true`.
  - If `test` is `false`, we want to return `false`.



# Solution w/ "Boolean Zen"

► Observation: The `if/else` is unnecessary.

- The variable `test` stores a boolean value; its value is exactly what you want to return. So return that!

```
public static boolean bothOdd(int n1, int n2) {  
    boolean test = (n1 % 2 != 0 && n2 % 2 != 0);  
    return test;  
}
```

► An even shorter version:

- We don't even need the variable `test`.  
We can just perform the test and return its result in one step.

```
public static boolean bothOdd(int n1, int n2) {  
    return (n1 % 2 != 0 && n2 % 2 != 0);  
}
```

# "Boolean Zen" template

## ► Replace

```
public static boolean <name>(<parameters>) {  
    if (<test>) {  
        return true;  
    } else {  
        return false;  
    }  
}
```

- with

```
public static boolean <name>(<parameters>) {  
    return <test>;  
}
```

# Improved isPrime method

- ▶ The following version utilizes Boolean Zen:

```
public static boolean isPrime(int n) {  
    int factors = 0;  
    for (int i = 1; i <= n; i++) {  
        if (n % i == 0) {  
            factors++;  
        }  
    }  
    return factors == 2; // if n has 2 factors -> true  
}
```

- ▶ Modify the Rhyme program to use Boolean Zen.

# Boolean Zen answer

```
public static void main(String[] args) {
    Scanner console = new Scanner(System.in);
    System.out.print("Type two words: ");
    String word1 = console.next().toLowerCase();
    String word2 = console.next().toLowerCase();

    if (rhyme(word1, word2)) {
        System.out.println("They rhyme!");
    }
    if (alliterate(word1, word2)) {
        System.out.println("They alliterate!");
    }
}

// Returns true if s1 and s2 end with the same two letters.
public static boolean rhyme(String s1, String s2) {
    return s2.length() >= 2 && s1.endsWith(s2.substring(s2.length() - 2));
}

// Returns true if s1 and s2 start with the same letter.
public static boolean alliterate(String s1, String s2) {
    return s1.startsWith(s2.substring(0, 1));
}
```

# De Morgan's Law

- ▶ **De Morgan's Law:** Rules used to negate boolean tests.
- ▶  $!(a \ \&\& \ b) == !a \ || \ !b$
- ▶  $!(a \ || \ b) == !a \ \&\& \ !b$ 
  - Useful when you want the opposite of an existing test.

Original Expression	Negated Expression	Alternative
<code>a &amp;&amp; b</code>	<code>!a    !b</code>	<code>! (a &amp;&amp; b)</code>
<code>a    b</code>	<code>!a &amp;&amp; !b</code>	<code>! (a    b)</code>

– Example:

Original Code	Negated Code
<pre>if (x == 7 &amp;&amp; y &gt; 3) {     ... }</pre>	<pre>if (x <b>!=</b> 7 <b>  </b> y <b>&lt;=</b> 3) {     ... }</pre>

# Clicker 1

► Which of the following is equivalent to the boolean expression? x, y, and z's are ints

$! ( (x \geq y) \ || \ (z \neq x) )$

A.  $! (x \geq y) \ \&\& \ ! (z \neq x)$

B.  $! (x \geq y) \ || \ ! (z \neq x)$

C.  $(x == y) \ \&\& \ (z \geq x)$

D.  $(x < y) \ \&\& \ (z == x)$

E. More than one of A - D is correct

# Boolean practice questions

- ▶ Write a method named `isVowel` that returns whether a `String` is a vowel (a, e, i, o, or u), case-insensitively.
  - `isVowel("q")` returns `false`
  - `isVowel("A")` returns `true`
  - `isVowel("e")` returns `true`
- ▶ Write a method `isNonVowel` that returns whether a `String` is any character except a vowel.
  - `isNonVowel("q")` returns `true`
  - `isNonVowel("A")` returns `false`
  - `isNonVowel("e")` returns `false`

# Boolean practice answers

```
// Enlightened version. I have seen the true way (and false way)
public static boolean isVowel(String s) {
    return s.equalsIgnoreCase("a") || s.equalsIgnoreCase("e")
        || s.equalsIgnoreCase("i")
        || s.equalsIgnoreCase("o")
        || s.equalsIgnoreCase("u");
}

// Enlightened "Boolean Zen" version
public static boolean isNonVowel(String s) {
    return !s.equalsIgnoreCase("a") && !s.equalsIgnoreCase("e")
        && !s.equalsIgnoreCase("i")
        && !s.equalsIgnoreCase("o")
        && !s.equalsIgnoreCase("u");

    // or, return !isVowel(s);
}
```



# When to return?

- ▶ Methods with loops and return values can be tricky.
  - When and where should the method return its result?
- ▶ Write a method `seven` that accepts a `Random` parameter and uses it to draw up to ten lotto numbers from 1-30.
  - If any of the numbers is a lucky 7, the method should stop and return `true`. If none of the ten are 7 it should return `false`.
  - The method should print each number as it is drawn.

15 29 18 29 11 3 30 17 19 22 (first call)

29 5 29 4 **7** (second call)

# Flawed solution

```
// Draws 10 lotto numbers; returns true if one is 7.
public static boolean seven(Random rand) {
    for (int i = 1; i <= 10; i++) {
        int num = rand.nextInt(30) + 1;
        System.out.print(num + " ");

        if (num == 7) {
            return true;
        } else {
            return false;
        }
    }
}
```

- The method always returns immediately after the first roll.
- This is wrong if that draw isn't a 7; we need to keep drawing.

# Returning at the right time

```
// Draws 10 lotto numbers; returns true if one is 7.
public static boolean seven(Random rand) {
    for (int i = 1; i <= 10; i++) {
        int num = rand.nextInt(30) + 1;
        System.out.print(num + " ");

        if (num == 7) {    // found lucky 7; can exit now
            return true;
        }
    }

    return false;    // if we get here, there was no 7
}
```

- ▶ Returns `true` immediately if 7 is found.
- ▶ If 7 isn't found, the loop continues drawing lotto numbers.
- ▶ If all ten aren't 7, the loop ends and we return `false`.

# Boolean return questions

- `hasAnOddDigit` : **returns** `true` if any digit of an integer is odd.
  - `hasAnOddDigit(4822116)` **returns** `true`
  - `hasAnOddDigit(2448)` **returns** `false`
- `allDigitsOdd` : **returns** `true` if every digit of an integer is odd.
  - `allDigitsOdd(135319)` **returns** `true`
  - `allDigitsOdd(9174529)` **returns** `false`
- `isAllVowels` : **returns** `true` if every char in a `String` is a vowel.
  - `isAllVowels("eIeIo")` **returns** `true`
  - `isAllVowels("oink")` **returns** `false`
    - These problems are available in our Practice-It! system under **5.x**.

# Boolean return answers

```
public static boolean hasAnOddDigit(int n) {
    while (n != 0) {
        if (n % 2 != 0) {    // check whether last digit is odd
            return true;
        }
        n = n / 10;
    }
    return false;
}

public static boolean allDigitsOdd(int n) {
    while (n != 0) {
        if (n % 2 == 0) {    // check whether last digit is even
            return false;
        }
        n = n / 10;
    }
    return true;
}

public static boolean isAllVowels(String s) {
    for (int i = 0; i < s.length(); i++) {
        String letter = s.substring(i, i + 1);
        if (!isVowel(letter)) {
            return false;
        }
    }
    return true;
}
```

# while loop question

- ▶ Write a method `digitSum` that accepts an integer parameter and returns the sum of its digits.
  - Assume that the number is non-negative.
  - Example: `digitSum(29107)` returns `2+9+1+0+7` or `19`
  - Hint: Use the `%` operator to extract a digit from a number.

# while loop answer

```
public static int digitSum(int n) {  
    n = Math.abs(n); // handle negatives  
  
    int sum = 0;  
    while (n > 0) {  
        // add last digit  
        sum = sum + (n % 10);  
        // remove last digit  
        n = n / 10;  
    }  
    return sum;  
}
```

# Topic 17

## assertions and program logic

"As soon as we started programming, we found out to our surprise that it wasn't as easy to get programs right as we had thought. Debugging had to be discovered. **I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs.**"

**Maurice V Wilkes**





# Clicker 1

► What is output by the following method?

```
public static void mysteryB(boolean b) {  
    System.out.print(b + " ");  
    b = (b == false);  
    System.out.print(b);  
}
```

- A. no output due to syntax error
- B. no output due to runtime error
- C. not possible to predict output
- D. always outputs true false OR false true
- E. always outputs true true OR false false

# Assertions

- ▶ **Assertion:** A declarative sentence that is either true or false
- ▶ **Examples:**
  - $2 + 2$  equals 4**
  - The St. Louis Cardinals played in the 2011 world series**
  - $x > 45$**
  - It is raining.**
  - UT football beat OU last year.**
  - UT volleyball qualified for the NCAA tourney last year.**
- ▶ **Not assertions**
  - How old are you?**
  - Take me to H.E.B.**

# Assertions

- ▶ Some assertions are true or false depending on context. Which of these depend on the context?

**$2 + 2$  equals 4**

**The St. Louis Cardinals played in the world series this year**

**$x > 45$**

**It is raining.**

**UT football beat OU last year.**

**UT volleyball qualified for the NCAA tourney last year.**

# Assertions

- ▶ Assertions that depend on context can be evaluated if the context is provided.
  - when x is 13,  $x > 45$**
  - It was raining in Round Rock, at 8 am on, October 10, 2006.**
- ▶ Many skills required to be a programmer or computer scientists
- ▶ Just a few we have seen so far
  - ability to generalize
  - create structured solutions
  - trace code
  - manage lots of details

# Assertions

- ▶ Another important skill in programming and computer science is the ability "to make assertions about your programs and to understand the contexts in which those assertions will be true."

```
Scanner console = new Scanner(System.in);  
System.out.print("Enter Y or N: ");  
String result = console.nextLine();  
// result is equal to "Y" or "N" here.
```

# Checking Input

```
Scanner console = new Scanner(System.in);
System.out.print("Enter Y or N: ");
String result = console.nextLine();
while(!result.equals("Y") && !result.equals("N")) {
    System.out.print("That wasn't a Y or N. ");
    System.out.print("Enter Y or N: ");
    result = console.nextLine();
}
// result is equal to "Y" or "N" here.
```

# Assertions

- ▶ **Provable Assertion:** An assertion that can be proven to be true at a particular point in program execution.
- ▶ **Program Verification:** A field of computer science that involves reasoning about the formal properties of programs and hardware to prove the correctness of the program or hardware.
  - Instead of testing.
  - A number of UTCS faculty are involved in verification and formal methods research:  
Emerson, Hunt, Lam, Moore, Young

# Reasoning about assertions

- ▶ Suppose you have the following code:

```
if (x > 3) {  
    // Point A  
    x--;  
} else {  
    // Point B  
    x++;  
    // Point C  
}  
// Point D
```

- ▶ What do you know about  $x$ 's value at the three points?
  - Is  $x > 3$ ? Always? Sometimes? Never?



# Assertions in code

- ▶ We can make assertions about our code and ask whether they are true at various points in the code.
  - Valid answers are ALWAYS, NEVER, or SOMETIMES.

```
System.out.print("Type a nonnegative number: ");  
double number = console.nextDouble();  
// Point A: is number < 0.0 here? (SOMETIMES)
```

```
while (number < 0.0) {  
    // Point B: is number < 0.0 here? (ALWAYS)  
    System.out.print("Negative; try again: ");  
  
    number = console.nextDouble();  
    // Point C: is number < 0.0 here? (SOMETIMES)  
}
```

```
// Point D: is number < 0.0 here? (NEVER)
```

# Reasoning about programs

- ▶ Right after a variable is initialized, its value is known:

```
int x = 3;  
// is x > 0?  ALWAYS
```

- ▶ In general you know nothing about parameters' values:

```
public static void mystery(int a, int b) {  
// is a == 10?  SOMETIMES
```

# Reasoning about programs

- ▶ But inside an `if`, `while`, etc., you may know something:

```
public static void mystery(int a, int b) {  
    if (a < 0) {  
        // is a == 10?  NEVER  
        ...  
    }  
}
```

# Assertions and loops

- ▶ At the start of a loop's body, the loop's test must be `true`:

```
while (y < 10) {  
    // is y < 10?  ALWAYS  
    ...  
}
```

- ▶ After a loop, the loop's test must be `false`:

```
while (y < 10) {  
    ...  
}  
// is y < 10?  NEVER
```

- ▶ Inside a loop's body, the loop's test may become `false`:

```
while (y < 10) {  
    y++;  
    // is y < 10?  SOMETIMES  
}
```

# "Sometimes"

- ▶ Things that cause a variable's value to be unknown  
(often leads to "sometimes" answers):
  - reading from a `Scanner`
  - reading a number from a `Random` object
  - initial value of a parameter in a method
- ▶ If you can reach a part of the program both with the answer being "yes" and the answer being "no", then the correct answer is "sometimes".

# Assertion example 1

```
public static void mystery(int x, int y) {  
    int z = 0;
```

```
    // Point A
```

```
    while (x >= y) {
```

```
        // Point B
```

```
        x = x - y;
```

```
        z++;
```

```
        if (x != y) {
```

```
            // Point C
```

```
            z = z * 2;
```

```
        }
```

```
    // Point D
```

```
}
```

```
// Point E
```

```
System.out.println(z);
```

```
}
```

For each assertion state if it is ALWAYS, NEVER, or SOMETIMES true at the specified points in the code.

•

	$x < y$	$x == y$	$z == 0$
Point A			
Point B			
Point C			
Point D			
Point E			

# Assertion example 1

```
public static void mystery(int x, int y) {  
    int z = 0;
```

```
    // Point A
```

```
    while (x >= y) {
```

```
        // Point B
```

```
        x = x - y;
```

```
        z++;
```

```
        if (x != y) {
```

```
            // Point C
```

```
            z = z * 2;
```

```
        }
```

```
    // Point D
```

```
}
```

```
// Point E
```

```
    System.out.println(z);
```

```
}
```

For each assertion state if it is ALWAYS, NEVER, or SOMETIMES true at the specified points in the code.

•

	$x < y$	$x == y$	$z == 0$
Point A	SOMETIMES	SOMETIMES	ALWAYS
Point B	NEVER	SOMETIMES	SOMETIMES
Point C	SOMETIMES	NEVER	NEVER
Point D	SOMETIMES	SOMETIMES	NEVER
Point E	ALWAYS	NEVER	SOMETIMES

# Assertion example 2

```
public static int mystery(Scanner console) {  
    int prev = 0;  
    int count = 0;  
    int next = console.nextInt();  
  
    // Point A  
  
    while (next != 0) {  
        // Point B  
        if (next == prev) {  
            // Point C  
            count++;  
        }  
  
        prev = next;  
        next = console.nextInt();  
        // Point D  
    }  
  
    // Point E  
    return count;  
}
```

For each assertion state if it is ALWAYS, NEVER, or SOMETIMES true at the specified points in the code.

	next == 0	prev == 0	next == prev
Point A			
Point B			
Point C			
Point D			
Point E			



# Assertion example 2

```
public static int mystery(Scanner console) {
    int prev = 0;
    int count = 0;
    int next = console.nextInt();

    // Point A
    while (next != 0) {
        // Point B
        if (next == prev) {
            // Point C
            count++;
        }

        prev = next;
        next = console.nextInt();
        // Point D
    }

    // Point E
    return count;
}
```

For each assertion state if it is ALWAYS, NEVER, or SOMETIMES true at the specified points in the code.

	next == 0	prev == 0	next == prev
Point A	SOMETIMES	ALWAYS	SOMETIMES
Point B	NEVER	SOMETIMES	SOMETIMES
Point C	NEVER	NEVER	ALWAYS
Point D	SOMETIMES	NEVER	SOMETIMES
Point E	ALWAYS	SOMETIMES	SOMETIMES

# Assertion example 3

```
public static int pow(int x, int y) {  
    int prod = 1;  
    if (y >= 0) {
```

```
        // Point A
```

```
        while (y > 0) {
```

```
            // Point B
```

```
            if (y % 2 == 0) {
```

```
                // Point C
```

```
                x = x * x;
```

```
                y = y / 2;
```

```
                // Point D
```

```
            } else {
```

```
                // Point E
```

```
                prod = prod * x;
```

```
                y--;
```

```
                // Point F
```

```
            }
```

```
        }
```

```
        // Point G
```

```
    }  
    return prod;  
}
```

For each assertion state if it is ALWAYS, NEVER, or SOMETIMES true at the specified points in the code.

	$y > 0$	$y \% 2 == 0$
Point A		
Point B		
Point C		
Point D		
Point E		
Point F		
Point G		

# Assertion example 3

```
public static int pow(int x, int y) {  
    int prod = 1;  
    if (y >= 0) {
```

```
        // Point A
```

```
        while (y > 0) {
```

```
            // Point B
```

```
            if (y % 2 == 0) {
```

```
                // Point C
```

```
                x = x * x;
```

```
                y = y / 2;
```

```
                // Point D
```

```
            } else {
```

```
                // Point E
```

```
                prod = prod * x;
```

```
                y--;
```

```
                // Point F
```

```
            }
```

```
        }
```

```
        // Point G
```

```
    }  
    return prod;  
}
```

For each assertion state if it is ALWAYS, NEVER, or SOMETIMES true at the specified points in the code.

	$y > 0$	$y \% 2 == 0$
Point A	SOMETIMES	SOMETIMES
Point B	ALWAYS	SOMETIMES
Point C	ALWAYS	ALWAYS
Point D	ALWAYS	SOMETIMES
Point E	ALWAYS	NEVER
Point F	SOMETIMES	ALWAYS
Point G	NEVER	ALWAYS

# Topic 18

## file input, tokens

"We have also obtained a glimpse of another crucial idea about languages and program design. This is the approach of stratified design, the notion that a complex system should be structured as a sequence of levels that are described using a sequence of languages. Each level is constructed by combining parts that are regarded as primitive at that level, and the parts constructed at each level are used as primitives at the next level. The language used at each level of a stratified design has primitives, means of combination, and means of abstraction appropriate to that level of detail. "

- Hal Abelson  
and Gerald Sussman



# File Input/output (I/O)

```
import java.io.File;
```

- Create a `File` object to get info about a file on your drive.

- (This doesn't actually create a new file on the hard disk.)

```
File f = new File("example.txt");  
if (f.exists() && f.length() > 1000) {  
    f.delete();  
}
```

Method name	Description
<code>canRead()</code>	returns whether file is able to be read
<code>delete()</code>	removes file from disk
<code>exists()</code>	whether this file exists on disk
<code>getName()</code>	returns file's name
<code>length()</code>	returns number of bytes in file

# Reading files

- ▶ To read a file, pass a `File` object to the `Scanner` Constructor (instead of `System.in`).

```
Scanner <name> = new Scanner(new File("<filename>"));
```

- Example:

```
File file = new File("mydata.txt");  
Scanner input = new Scanner(file);
```

- or (shorter):

```
Scanner input  
    = new Scanner(new File("mydata.txt"));
```

# File paths

- ▶ **absolute path:** specifies a drive or a top "/" folder

`C:/Documents/smith/hw6/input/data.csv`

- Windows can also use backslashes to separate folders.

- ▶ **relative path:** does not specify any top-level folder

`names.dat`

`input/kinglear.txt`

- Assumed to be relative to the *current directory*:

```
Scanner input = new Scanner(new File("data/readme.txt"));
```

If our program is in `H:/hw6`, the `Scanner` will look for `H:/hw6/data/readme.txt`

# Working Directory

- ▶ New programmers are often not sure what their current directory is.
- ▶ Easy to print out:

```
public static void  
printWorkingDirectory() {  
    System.out.println("Working Directory = " +  
        System.getProperty("user.dir"));  
}
```

Working Directory = C:\Users\scottm\Documents\312\BlueJ

<http://docs.oracle.com/javase/tutorial/essential/environment/sysprop.html>



# Compiler error w/ files

```
import java.io.File;
import java.util.Scanner;

public class ReadFile {
    public static void main(String[] args) {
        Scanner input = new Scanner(new File("data.txt"));
        String text = input.next();
        System.out.println(text);
    }
}
```

- ▶ The program fails to compile with the following error:

```
ReadFile.java:6: unreported exception java.io.FileNotFoundException;
must be caught or declared to be thrown
```

```
    Scanner input = new Scanner(new File("data.txt"));
```

^

# Exceptions



- ▶ **exception:** An object representing a runtime error.
  - dividing an integer by 0
  - calling `substring` on a `String` and passing too large an index
  - trying to read the wrong type of value from a `Scanner`
  - trying to read a file that does not exist
  - We say that a program with a runtime error "*throws*" an exception.
  - It is also possible to "*catch*" (handle or fix) an exception.
- ▶ **checked exception:** An error that must be handled by our program (otherwise it will not compile).
  - We must specify how our program will handle file I/O failures.

# Clicker 1

► Can a programmer prevent a divide by zero error from occurring in all cases?

A. no

B. yes

C. maybe

## Clicker 2

► Can a programmer prevent a file not found error from occurring in all cases?

A. no

B. yes

C. maybe

# The throws clause

- ▶ **throws clause:** Keywords on a method's header that state that it may generate an exception (and will not handle it).

- ▶ Syntax:

```
public static <type> <name> (...) throws <type> {
```

- Example:

```
public class ReadFile {  
    public static void main(String[] args)  
        throws FileNotFoundException {
```

- Like saying, *"I hereby announce that this method might throw an exception, and I accept the consequences if this happens. OR I am passing the buck to someone else."*<sup>10</sup>

# Input tokens

- ▶ **token:** A unit of user input, separated by whitespace.
  - A `Scanner` splits a file's contents into tokens.
- ▶ If an input file contains the following:

23	3.14
"John Smith"	

The `Scanner` can interpret the tokens as the following types:

<u>Token</u>	<u>Type(s)</u>
23	int, double, String
3.14	double, String
"John	String
Smith"	String

# Files and input cursor

- ▶ Consider a file `weather.txt` that contains this text:

```
16.2    23.5
      19.1 7.4    22.8

18.5    -1.8 14.9
```

- ▶ A `Scanner` **views** all input as a stream of characters:

```
16.2    23.5\n19.1 7.4    22.8\n\n18.5    -1.8 14.9\n
```

^

- ▶ **input cursor**: The current position of the `Scanner`.

# Consuming tokens

- ▶ **consuming input:** Reading input and advancing the cursor.
  - Calling `nextInt` etc. moves the cursor past the current token

```
16.2    23.5\n19.1  7.4    22.8\n\n18.5    -1.8  14.9\n
```

^

```
double d = input.nextDouble();    // 16.2
```

```
16.2    23.5\n19.1  7.4    22.8\n\n18.5    -1.8  14.9\n
```

^

```
String s = input.next();          // "23.5"
```

```
16.2    23.5\n19.1  7.4    22.8\n\n18.5    -1.8  14.9\n
```

^



# File input question

- Recall the input file `weather.txt`:

```
16.2    23.5  
      19.1  7.4   22.8  
  
18.5    -1.8  14.9
```

- Write a program that prints the change in temperature between each pair of neighboring days.

```
16.2 to 23.5, change = 7.3  
23.5 to 19.1, change = -4.4  
19.1 to 7.4, change = -11.7  
7.4 to 22.8, change = 15.4  
22.8 to 18.5, change = -4.3  
18.5 to -1.8, change = -20.3  
-1.8 to 14.9, change = 16.7
```

# File input answer

```
// Displays changes in temperature from data in an input file.

import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class Temperatures {
    public static void main(String[] args)
        throws FileNotFoundException {
        Scanner input = new Scanner(new File("weather.txt"));
        double prev = input.nextDouble();    // fencepost
        for (int i = 1; i <= 7; i++) {
            double next = input.nextDouble();
            System.out.println(prev + " to " + next +
                               ", change = " + (next - prev));
            prev = next;
        }
    }
}
```

# Reading an entire file

- ▶ Suppose we want our program to work no matter how many numbers are in the file.
  - Currently, if the file has more numbers, they will not be read.
  - If the file has fewer numbers, what will happen?

A crash! Example output from a file with just 3 numbers:

```
16.2 to 23.5, change = 7.3
```

```
23.5 to 19.1, change = -4.4
```

```
Exception in thread "main"
```

```
java.util.NoSuchElementException
```

```
at java.util.Scanner.throwFor(Scanner.java:838)
```

```
at java.util.Scanner.next(Scanner.java:1347)
```

```
at Temperatures.main(Temperatures.java:12)
```

# Scanner exceptions

- ▶ `NoSuchElementException`
  - You read past the end of the input.
- ▶ `InputMismatchException`
  - You read the wrong type of token (e.g. read "hi" as an `int`).
- ▶ Finding and fixing these exceptions:
  - Read the exception text for line numbers in your code (the first line that mentions your file; often near the bottom):

```
Exception in thread "main"  
java.util.NoSuchElementException  
    at java.util.Scanner.throwFor(Scanner.java:838)  
    at java.util.Scanner.next(Scanner.java:1347)  
    at MyProgram.myMethodName(MyProgram.java:19)  
    at MyProgram.main(MyProgram.java:6)
```

# Scanner tests for valid input

Method	Description
<code>hasNext()</code>	returns <code>true</code> if there is a next token
<code>hasNextInt()</code>	returns <code>true</code> if there is a next token and it can be read as an <code>int</code>
<code>hasNextDouble()</code>	returns <code>true</code> if there is a next token and it can be read as a <code>double</code>

- These methods of the `Scanner` do not consume input; they just give information about what the next token will be.
  - Useful to see what input is coming, and to avoid crashes.
  - These methods can be used with a console `Scanner`, as well.
    - When called on the console, they sometimes pause waiting for input.

# Using hasNext methods

## ► Avoiding type mismatches:

```
Scanner console = new Scanner(System.in);
System.out.print("How old are you? ");
if (console.hasNextInt()) {
    int age = console.nextInt();    // will not crash!
    System.out.println("Wow, " + age + " is old!");
} else {
    System.out.println("You didn't type an integer.");
}
```

## ► Avoiding reading past the end of a file:

```
Scanner input = new Scanner(new File("example.txt"));
if (input.hasNext()) {
    String token = input.next();    // will not crash!
    System.out.println("next token is " + token);
}
```

# File input question 2

- ▶ Modify the temperature program to process the entire file, regardless of how many numbers it contains.
  - Example: If a ninth day's data is added, output might be:

```
16.2 to 23.5, change = 7.3
23.5 to 19.1, change = -4.4
19.1 to 7.4, change = -11.7
7.4 to 22.8, change = 15.4
22.8 to 18.5, change = -4.3
18.5 to -1.8, change = -20.3
-1.8 to 14.9, change = 16.7
14.9 to 16.1, change = 1.2
```

# File input answer 2

`// Displays changes in temperature from data in an input file.`

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class Temperatures {
    public static void main(String[] args)
        throws FileNotFoundException {
        Scanner input = new Scanner(new File("weather.txt"));
        double prev = input.nextDouble();    // fencepost
        while (input.hasNextDouble()) {
            double next = input.nextDouble();
            System.out.println(prev + " to " + next +
                               ", change = " + (next - prev));
            prev = next;
        }
    }
}
```



# File input question 3

- ▶ Modify the temperature program to handle files that contain non-numeric tokens (by skipping them).
- ▶ For example, it should produce the same output as before when given this input file, `weather2.txt`:

```
16.2    23.5  
Tuesday    19.1    Wed 7.4    THURS. TEMP: 22.8  
  
18.5    -1.8    <-- MIKE here is my data!    --Chris  
    14.9    :-)
```

- You may assume that the file begins with a double.
- **What if we didn't know that?**

# File input answer 3

```
// Displays changes in temperature from data in an input file.

import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class Temperatures2 {
    public static void main(String[] args)
        throws FileNotFoundException {
        Scanner input = new Scanner(new File("weather.txt"));
        double prev = input.nextDouble();    // fencepost
        while (input.hasNext()) {
            if (input.hasNextDouble()) {
                double next = input.nextDouble();
                System.out.println(prev + " to " + next +
                                   ", change = " + (next - prev));
                prev = next;
            } else {
                input.next();    // throw away unwanted token
            }
        }
    }
}
```

# "File" Input

- ▶ Reading from sources other than files is not very different
- ▶ For example we can read data from a web page about as easily as reading from a file
- ▶ Example, read stock information from a web page
  - often the hardest thing is finding the web page and the format of the url
  - once you have that the code is easy

# Read HTML from a Webpage

- ▶ <https://www.cnbc.com/>
- ▶ Read and print out the HTML from that web page
- ▶ Could easily alter to read data from any web page or via a web API
  - Service that allows a developer to read data via the web

# Display HTML

```
try {  
    System.out.println("Raw HTML from CNBC");  
    String urlAsString = "https://www.cnbc.com/";  
    URL url = new URL(urlAsString);  
    Scanner sc  
        = new Scanner(new InputStreamReader(url.openStream()));  
    while(sc.hasNextLine()) {  
        System.out.println(sc.nextLine());  
    }  
    sc.close();  
}  
catch(IOException e) {  
    System.out.println("UH OH: " + e);  
}
```

# Topic 19

## Line Based File Input

"Composing computer programs to solve scientific problems is like writing poetry. You must choose every word with care and link it with the other words in perfect syntax. There is no place for verbosity or carelessness. **To become fluent in a computer language demands almost the antithesis of modern loose thinking.** It requires many interactive sessions, the hands-on use of the device. You do not learn a foreign language from a book, rather you have to live in the country for year to let the language become an automatic part of you, and the same is true for computer languages. "

- James Lovelock



# Hours question

- ▶ Given a file `hours.txt` with the following contents:

```
123 Kim 12.5 8.1 7.6 3.2
456 Eric 4.0 11.6 6.5 2.7 12
789 Stef 8.0 8.0 8.0 8.0 7.5
```

- Consider the task of computing hours worked by each person:

```
Kim (ID#123) worked 31.4 hours (7.85 hours/day)
Eric (ID#456) worked 36.8 hours (7.36 hours/day)
Stef (ID#789) worked 39.5 hours (7.9 hours/day)
```

# Hours answer (flawed)

```
// This solution does not work!
import java.io.*;                // for File
import java.util.Scanner;
public class HoursWorked {
    public static void main(String[] args)
        throws FileNotFoundException {
        Scanner input = new Scanner(new File("hours.txt"));
        while (input.hasNext()) {
            // process one person
            int id = input.nextInt();
            String name = input.next();
            double totalHours = 0.0;
            int days = 0;
            while (input.hasNextDouble()) {
                totalHours += input.nextDouble();
                days++;
            }
            System.out.println(name + " (ID#" + id +
                               ") worked " + totalHours + " hours (" +
                               (totalHours / days) + " hours/day)");
        }
    }
}
```

123 Kim 12.5 8.1 7.6 3.2
456 Eric 4.0 11.6 6.5 2.7 12.3



# Clicker 1

- ▶ What happens when the solution on the previous slide is run given a file with this data?

```
123 Kim 12.5 8.1 7.6 3.2
```

```
456 Eric 4.0 11.6 6.5 2.7 12
```

```
789 Stef 8.0 8.0 8.0 8.0 7.5
```

- A. prints out correct answer
- B. no output due to syntax error
- C. some output then an `InputMismatchException`
- D. some output then a `NoSuchElementException`
- E. More than one of A - D is correct

# Flawed output

```
Kim (ID#123) worked 487.4 hours (97.48 hours/day)
Exception in thread "main"
java.util.InputMismatchException
    at java.util.Scanner.throwFor(Scanner.java:840)
    at java.util.Scanner.next(Scanner.java:1461)
    at java.util.Scanner.nextInt(Scanner.java:2091)
    at HoursWorked.main(HoursBad.java:9)
```

- The inner `while` loop is grabbing the next person's ID.
  - We want to process the tokens, but we also care about the line breaks (they mark the end of a person's data).
- A better solution is a hybrid approach:
- First, break the overall input into lines.
  - Then break each line into tokens.

# Line-based Scanner methods

Method	Description
<code>nextLine()</code>	returns next entire line of input (from cursor to <code>\n</code> )
<code>hasNextLine()</code>	returns <code>true</code> if there are any more lines of input to read (always <code>true</code> for console input)

```
Scanner input
    = new Scanner(new File("<filename>"));
while (input.hasNextLine()) {
    String line = input.nextLine();
    <process this line>;
}
```

# Consuming lines of input

```
23      3.14 John Smith      "Hello" world
      45.2 19
```

- ▶ The Scanner reads the lines as follows:

```
23\t3.14 John Smith\t"Hello" world\n\t\t45.2 19\n^
```

- String line = input.nextLine();

```
23\t3.14 John Smith\t"Hello" world\n\t\t45.2 19\n^
```

- String line2 = input.nextLine();

```
23\t3.14 John Smith\t"Hello" world\n\t\t45.2 19\n^
```

- Each \n character is consumed but not returned.

# Scanners on Strings

- ▶ A Scanner can tokenize the contents of a String:

```
Scanner <name> = new Scanner(<String>) ;
```

## – Example:

```
String text = "15 3.2 hello 9 27.5";  
Scanner scan = new Scanner(text) ;  
  
int num = scan.nextInt() ;  
System.out.println(num) ;           // 15  
  
double num2 = scan.nextDouble() ;  
System.out.println(num2) ;          // 3.2  
  
String word = scan.next() ;  
System.out.println(word) ;          // "hello"
```

# Mixing lines and tokens

Input file input.txt:	Output to console:
The quick brown fox jumps over the lazy dog.	Line has 6 words Line has 3 words

**// Counts the words on each line of a file**

```
Scanner input = new Scanner(new File("input.txt"));
while (input.hasNextLine()) {
    String line = input.nextLine();
    Scanner lineScan = new Scanner(line);

    // process the contents of this line
    int count = 0;
    while (lineScan.hasNext()) {
        String word = lineScan.next();
        count++;
    }
    System.out.println("Line has " + count + " words");
}
```

# Hours question

- Fix the `Hours` program to read the input file properly:

```
123 Kim 12.5 8.1 7.6 3.2
456 Eric 4.0 11.6 6.5 2.7 12
789 Stef 8.0 8.0 8.0 8.0 7.5
```

- Recall, it should produce the following output:

```
Kim (ID#123) worked 31.4 hours (7.85 hours/day)
Eric (ID#456) worked 36.8 hours (7.36 hours/day)
Stef (ID#789) worked 39.5 hours (7.9 hours/day)
```

# Hours answer, corrected

```
// Processes an employee input file and outputs each employee's hours.
import java.io.*;    // for File
import java.util.*;  // for Scanner

public class Hours {
    public static void main(String[] args) throws FileNotFoundException {
        Scanner input = new Scanner(new File("hours.txt"));
        while (input.hasNextLine()) {
            String line = input.nextLine();
            processEmployee(line);
        }
    }

    public static void processEmployee(String line) {
        Scanner lineScan = new Scanner(line);
        int id = lineScan.nextInt();           // e.g. 456
        String name = lineScan.next();         // e.g. "Eric"
        double sum = 0.0;
        int count = 0;
        while (lineScan.hasNextDouble()) {
            sum = sum + lineScan.nextDouble();
            count++;
        }

        double average = sum / count;
        System.out.println(name + " (ID#" + id + ") worked " +
            sum + " hours (" + average + " hours/day)");
    }
}
```



# File output

**reading: 6.4 - 6.5**

# Output to files

- ▶ **PrintStream**: An object in the `java.io` package that lets you print output to a destination such as a file.
  - Any methods you have used on `System.out` (such as `print`, `println`) will work on a `PrintStream`.

- ▶ **Syntax:**

```
PrintStream <name>  
    = new PrintStream(new File("<filename>"));
```

**Example:**

```
PrintStream output  
    = new PrintStream(new File("out.txt"));  
output.println("Hello, file!");  
output.println("This is a second line of output.");
```

# Details about `PrintStream`

```
PrintStream <name>
```

```
= new PrintStream(new File("<filename>")) ;
```

- If the given file does not exist, it is created.
- If the given file already exists, it is **overwritten**.
- The output you print appears in a file, not on the console. You have to open the file with an editor to see it.
- Do not open the same file for both reading (`Scanner`) and writing (`PrintStream`) at the same time.
  - You will overwrite your input file with an empty file (0 bytes).

# System.out and PrintStream

- ▶ The console output object, `System.out`, is a `PrintStream`.

```
PrintStream out1 = System.out;  
PrintStream out2 = new PrintStream(new File("data.txt"));  
out1.println("Hello, console!");    // goes to console  
out2.println("Hello, file!");       // goes to file
```

- A reference to it can be stored in a `PrintStream` variable.
  - Printing to that variable causes console output to appear.
- You can pass `System.out` to a method as a `PrintStream`.
  - Allows a method to send output to the console or a file.

# PrintStream question

- ▶ Modify our previous Hours program to use a `PrintStream` to send its output to the file `hours_out.txt`.
- ▶ The program will produce no console output.
- ▶ the file `hours_out.txt` will be created with the text:

```
Kim (ID#123) worked 31.4 hours (7.85 hours/day)
Eric (ID#456) worked 36.8 hours (7.36 hours/day)
Stef (ID#789) worked 39.5 hours (7.9 hours/day)
```

# PrintStream answer

```
// Processes an employee input file and outputs each employee's hours.
import java.io.*;    // for File
import java.util.*;  // for Scanner

public class Hours2 {
    public static void main(String[] args) throws FileNotFoundException {
        Scanner input = new Scanner(new File("hours.txt"));
        PrintStream out = new PrintStream(new File("hours_out.txt"));
        while (input.hasNextLine()) {
            String line = input.nextLine();
            processEmployee(out, line);
        }
    }

    public static void processEmployee(PrintStream out, String line) {
        Scanner lineScan = new Scanner(line);
        int id = lineScan.nextInt();           // e.g. 456
        String name = lineScan.next();         // e.g. "Eric"
        double sum = 0.0;
        int count = 0;
        while (lineScan.hasNextDouble()) {
            sum = sum + lineScan.nextDouble();
            count++;
        }

        double average = sum / count;
        out.println(name + " (ID#" + id + ") worked " +
            sum + " hours (" + average + " hours/day)");
    }
}
```

# Prompting for a file name

- ▶ We can ask the user to tell us the file to read.

- The filename might have spaces; use `nextLine()`, not `next()`

```
// prompt for input file name
```

```
Scanner console = new Scanner(System.in);  
System.out.print("Type a file name to use: ");  
String filename = console.nextLine();  
Scanner input = new Scanner(new File(filename));
```

- ▶ Files have an `exists` method to test for file-not-found:

```
File file = new File("hours.txt");  
if (!file.exists()) {  
    // try a second input file as a backup  
    System.out.print("hours file not found!");  
    file = new File("hours2.txt");  
}
```

# Topic 20

## More File Processing

**"We can only see a short distance ahead, but we can see plenty there that needs to be done."**

**- Alan Turing**





# Recall: Line-based methods

Method	Description
<code>nextLine()</code>	returns the next entire line of input
<code>hasNextLine()</code>	returns <code>true</code> if there are any more lines of input to read (always true for console input)

- ▶ `nextLine` consumes from the input cursor to the next `\n`.

```
Scanner input
    = new Scanner(new File("<filename>"));
while (input.hasNextLine()) {
    String line = input.nextLine();
    <process this line>;
}
```

# Recall: Tokenizing lines

- ▶ A `String Scanner` can tokenize each line of a file.

```
Scanner input
```

```
    = new Scanner(new File("<filename>")) ;
```

```
while (input.hasNextLine()) {
```

```
    String line = input.nextLine();
```

```
    Scanner lineScan = new Scanner(line);
```

```
    <process the contents of this line>;
```

```
}
```

# Clicker 1

- What is output by the following code if the input file contains

```
12  24
Christmas Eve
```

```
public static void print(Scanner sc) {
    int total = sc.nextInt();
    total += sc.nextInt();
    System.out.println(total + " " +
        sc.nextLine());
}
```

```
12 24\nChristmas Eve
```

- A. 36                      B. 36 Christmas Eve
- C. 24 Christmas Eve
- D. no output due to syntax error
- E. no output due to runtime error

# Hours v2 question

- ▶ Modify the `Hours` program to search for a person by ID:

- Example:

```
Enter an ID: 456
```

```
Eric worked 36.8 hours (7.36 hours/day)
```

- Example:

```
Enter an ID: 293
```

```
ID #293 not found
```

# Hours v2 answer 1

```
// This program searches an input file of employees' hours worked
// for a particular employee and outputs that employee's hours data.

import java.io.*;    // for File
import java.util.*;  // for Scanner

public class HoursWorked {
    public static void main(String[] args) throws FileNotFoundException {
        Scanner console = new Scanner(System.in);
        System.out.print("Enter an ID: ");
        int searchId = console.nextInt();           // e.g. 456

        Scanner input = new Scanner(new File("hours.txt"));
        String line = findPerson(input, searchId);
        if (line.length() > 0) {
            processLine(line);
        } else {
            System.out.println("ID #" + searchId + " was not found");
        }
    }
}

...
```

# Hours v2 answer 2

```
// Locates and returns the line of data about a particular person.
public static String findPerson(Scanner input, int searchId) {
    while (input.hasNextLine()) {
        String line = input.nextLine();
        Scanner lineScan = new Scanner(line);
        int id = lineScan.nextInt();           // e.g. 456
        if (id == searchId) {
            return line;                       // we found them!
        }
    }
    return "";                               // not found, so return an empty String
}

// Totals the hours worked by the person and outputs their info.
public static void processLine(String line) {
    Scanner lineScan = new Scanner(line);
    int id = lineScan.nextInt();               // e.g. 456
    String name = lineScan.next();             // e.g. "Brad"
    double hours = 0.0;
    int days = 0;
    while (lineScan.hasNextDouble()) {
        hours += lineScan.nextDouble();
        days++;
    }

    System.out.println(name + " worked " + hours + " hours ("
        + (hours / days) + " hours/day)");
}
```

# IMDb movies problem

- ▶ Consider the following Internet Movie Database (IMDb) data:

1	9.1	490,400	The Shawshank Redemption	(1994)
2	9.1	392,937	The Godfather	(1972)
3	9.0	232,741	The Godfather: Part II	(1974)

- ▶ Write a program that displays any movies containing a phrase:

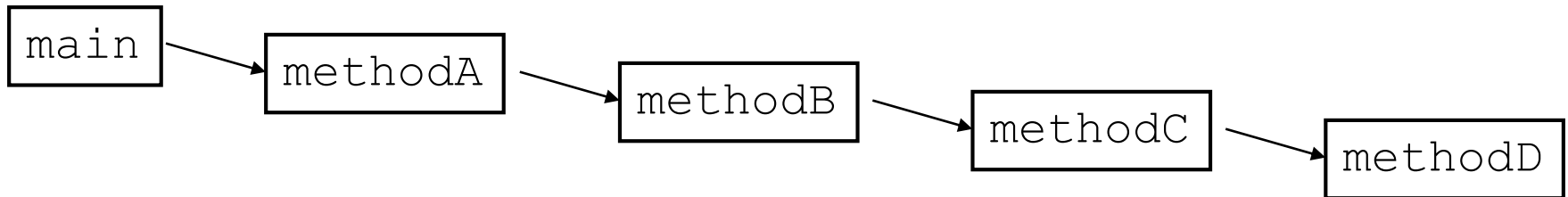
Search word? part

Rank	Votes	Rating	Title
3	232741	9.0	The Godfather: Part II (1974)
50	249709	8.4	The Departed (2006)
98	34736	8.3	The Apartment (1960)
241	48525	7.9	Spartacus (1960)

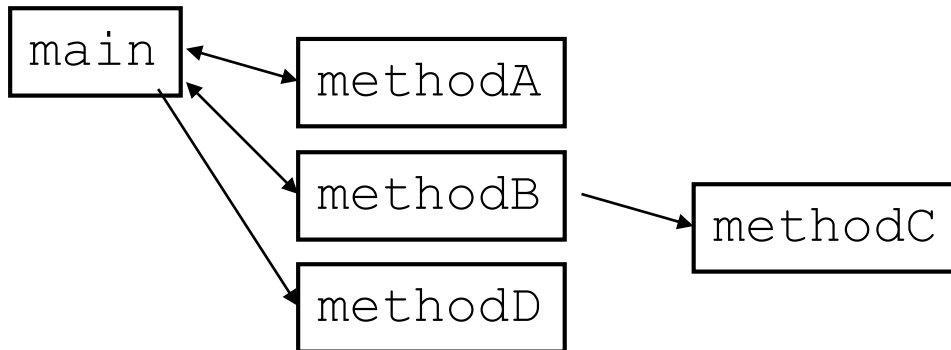
4 matches.

# Recall "Chaining"

- ▶ `main` should be a concise summary of your program.
  - It is generally poor style if each method calls the next without ever returning (*chaining*):



- ▶ A better structure has `main` make most of the calls.
  - Methods must return values to `main` to be passed on later.





# Bad IMDb "chained" code 1

```
// Displays IMDB's Top 250 movies that match a search string.
import java.io.*;          // for File
import java.util.*;        // for Scanner

public class Movies {
    public static void main(String[] args)
                                throws FileNotFoundException {
        getWord();
    }

    // Asks the user for their search word and returns it.
    public static void getWord() throws FileNotFoundException {
        System.out.print("Search word: ");
        Scanner console = new Scanner(System.in);
        String searchWord = console.next();
        searchWord = searchWord.toLowerCase();
        System.out.println();

        Scanner input = new Scanner(new File("imdb.txt"));
        search(input, searchWord);
    }
    ...
}
```

# Bad IMDb "chained" code 2

...

```
// Breaks apart each line, looking for lines that match the search word.
public static String search(Scanner input, String searchWord) {
    int matches = 0;
    while (input.hasNextLine()) {
        String line = input.nextLine();
        String lineLC = line.toLowerCase();           // case-insensitive match
        if (lineLC.indexOf(searchWord) >= 0) {
            matches++;
            System.out.println("Rank\tVotes\tRating\tTitle");
            display(line);
        }
    }

    System.out.println(matches + " matches.");
}

// Displays the line in the proper format on the screen.
public static void display(String line) {
    Scanner lineScan = new Scanner(line);
    int rank = lineScan.nextInt();
    double rating = lineScan.nextDouble();
    int votes = lineScan.nextInt();
    String title = "";
    while (lineScan.hasNext()) {
        title += lineScan.next() + " ";           // the rest of the line
    }
    System.out.println(rank + "\t" + votes + "\t" + rating + "\t" + title)
}
```

# Better IMDb answer 1

```
// Displays IMDB's Top 250 movies that match a search string.
import java.io.*;      // for File
import java.util.*;    // for Scanner

public class Movies {
    public static void main(String[] args)
        throws FileNotFoundException {
        String searchWord = getWord();
        Scanner input = new Scanner(new File("imdb.txt"));
        String line = search(input, searchWord);
        int matches = 0;

        if (line.length() > 0) {
            System.out.println("Rank\tVotes\tRating\tTitle");
            while (line.length() > 0) {
                matches++;
                display(line);
                line = search(input, searchWord);
            }
        }

        System.out.println(matches + " matches.");
    }
}
```

# Better IMDb answer 2

**// Asks the user for their search word and returns it.**

```
public static String getWord() {  
    System.out.print("Search word: ");  
    Scanner console = new Scanner(System.in);  
    String searchWord = console.next();  
    searchWord = searchWord.toLowerCase();  
    System.out.println();  
    return searchWord;  
}
```

**// Breaks apart each line, looking**

**// for lines that match the search word.**

```
public static String search(Scanner input, String searchWord)  
    while (input.hasNextLine()) {  
        String line = input.nextLine();  
        // case-insensitive match  
        String lineLC = line.toLowerCase();  
        if (lineLC.indexOf(searchWord) >= 0) {  
            return line;  
        }  
    }  
    return ""; // not found
```

# Better IMDb answer 3

```
// Displays the line in the proper format on the screen.
public static void display(String line) {
    Scanner lineScan = new Scanner(line);
    int rank = lineScan.nextInt();
    double rating = lineScan.nextDouble();
    int votes = lineScan.nextInt();
    String title = "";
    while (lineScan.hasNext()) {
        // the rest of the line
        title += lineScan.next() + " ";
    }
    System.out.println(rank + "\t" + votes
        + "\t" + rating + "\t" + title);
}
```

# Mixing tokens and lines

- ▶ Using `nextLine` in conjunction with the token-based methods on the same `Scanner` can cause unexpected results.

```
23      3.14
Joe      "Hello world"
          45.2      19
```

- ▶ You'd think you could read `23` and `3.14` with `nextInt` and `nextDouble`, then read `Joe "Hello world"` with `nextLine`.

```
System.out.println(input.nextInt());      // 23
System.out.println(input.nextDouble());   // 3.14
System.out.println(input.nextLine());     //
```

– But the `nextLine` call produces no output! Why?

# Mixing lines and tokens

- ▶ Avoid reading both tokens and lines from the **same** Scanner:

```
23    3.14
Joe    "Hello world"
           45.2    19
```

```
input.nextInt() // 23
23\t3.14\nJoe\t"Hello world"\n\t\t45.2    19\n  ^
```

```
input.nextDouble() // 3.14
23\t3.14\nJoe\t"Hello world"\n\t\t45.2    19\n  ^
```

```
input.nextLine() // "" (empty!)
23\t3.14\nJoe\t"Hello world"\n\t\t45.2    19\n  ^
```

```
input.nextLine() // "Joe\t\"Hello world\""
23\t3.14\nJoe\t"Hello world"\n\t\t45.2    19\n  ^
```

# Line-and-token example

```
Scanner console = new Scanner(System.in);
System.out.print("Enter your age: ");
int age = console.nextInt();

System.out.print("Now enter your name: ");
String name = console.nextLine();
System.out.println(name + " is " + age + " years old.");
```

## Log of execution (user input underlined):

```
Enter your age: 12
Now enter your name: Sideshow Bob
is 12 years old.
```

### ► Why?

- Overall input: 12\nSideshow Bob
- After `nextInt()` : **12**\nSideshow Bob  
                                  ^
- After `nextLine()` : 12\nSideshow Bob  
                                  ^



# Topic 21

## arrays - part 1

"Should array indices start at 0 or 1? My compromise of 0.5 was rejected without, I thought, proper consideration. "

- *Stan Kelly-Bootle*



# Can we solve this problem?

- Consider the following program (input underlined):

How many days' temperatures? 7

Day 1's high temp: 45

Day 2's high temp: 44

Day 3's high temp: 39

Day 4's high temp: 48

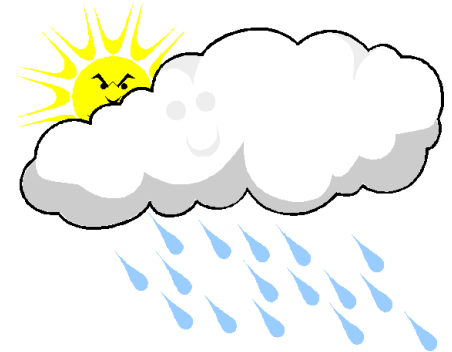
Day 5's high temp: 37

Day 6's high temp: 46

Day 7's high temp: 53

Average temp = 44.6

4 days were above average.



# Why the problem is hard

- ▶ We need each input value twice:
  - to compute the average (a cumulative sum)
  - to count how many were above average
- ▶ We could read each value into a variable...  
but we:
  - don't know how many days are needed until the program runs
  - don't know how many variables to declare
- ▶ **We need a way to declare many variables in one step.**

# Arrays

- ▶ **array**: object that stores many values of the same type.
  - **element**: One value in an array.
  - **index**: A **0-based integer** to access an element from an array.

<i>index</i>	0	1	2	3	4	5	6	7	8	9
<i>value</i>	12	49	-2	26	5	17	-6	84	72	3

element 0				element 4					element 9
-----------	--	--	--	-----------	--	--	--	--	-----------

# Array declaration

**<type>[] <name> = new <type>[<length>];**

– Example:

```
int[] numbers = new int[10];
```

<i>index</i>	0	1	2	3	4	5	6	7	8	9
<i>value</i>	0	0	0	0	0	0	0	0	0	0

# Array declaration, cont.

- ▶ The length can be any non-negative integer expression.

```
int x = 2 * 3 + 1;
```

```
int[] data = new int[x % 5 + 2];
```

- ▶ Each element initially gets a "zero-equivalent" value.

Type	Default value
int	0
double	0.0
boolean	false
String or other object	null (means, "no object")

# Accessing elements

**<name>** [ **<index>** ]    **// access**

**<name>** [ **<index>** ] = **<value>**;    **// modify**

– Example:

```
numbers[0] = 27;
```

```
numbers[3] = -6;
```

```
System.out.println(numbers[0]);
```

```
if (numbers[3] < 0) {
```

```
    System.out.println("Element 3 is negative.");
```

```
}
```

*index*    0    1    2    3    4    5    6    7    8    9

<i>value</i>	<b>27</b>	0	0	<b>-6</b>	0	0	0	0	0	0
--------------	-----------	---	---	-----------	---	---	---	---	---	---

# Arrays of other types

```
double[] results = new double[5];  
results[2] = 3.4;  
results[4] = -0.5;
```

<i>index</i>	0	1	2	3	4
<i>value</i>	0.0	0.0	<b>3.4</b>	0.0	<b>-0.5</b>

```
boolean[] tests = new boolean[6];  
tests[3] = true;
```

<i>index</i>	0	1	2	3	4	5
<i>value</i>	false	false	false	<b>true</b>	false	false



# Out-of-bounds

- ▶ Legal indexes: between **0** and the **array's length - 1**.
  - Reading or writing any index outside this range will throw an `ArrayIndexOutOfBoundsException`.

- ▶ Example:

```
int[] data = new int[10];  
System.out.println(data[0]);           // okay  
System.out.println(data[9]);           // okay  
System.out.println(data[-1]);          // exception  
System.out.println(data[10]);         // exception
```

<i>index</i>	0	1	2	3	4	5	6	7	8	9
<i>value</i>	0	0	0	0	0	0	0	0	0	0

# Accessing array elements

```
int[] numbers = new int[8];  
numbers[1] = 3;  
numbers[4] = 99;  
numbers[6] = 2;  
int x = numbers[1];  
numbers[x] = 42;  
numbers[numbers[6]] = 11; // use numbers[6] as index
```

x 

3
---

	<i>index</i>	0	1	2	3	4	5	6	7
<i>numbers</i>	<i>value</i>	0	3	11	42	99	0	2	0

# Clicker 1

► What is output by the following code?

```
String[] names = new String[5];  
names[1] = "Olivia";  
names[3] = "Isabelle";  
System.out.print(names[0].length());
```

- A. no output due to null pointer exception
- B. no output due to array index out of bounds exception
- C. no output due to a compile error (code can't run)
- D. 0
- E. 6

# Arrays and `for` loops

- ▶ It is common to use `for` loops to access array elements.

```
for (int i = 0; i < 8; i++) {  
    System.out.print(numbers[i] + " ");  
}  
System.out.println(); // output: 0 3 11 42 99 0 2 0
```

- ▶ Sometimes we assign each element a value in a loop.

```
for (int i = 0; i < 8; i++) {  
    numbers[i] = 2 * i;  
}
```

<i>index</i>	0	1	2	3	4	5	6	7
<i>value</i>	0	2	4	6	8	10	12	14

# The length field

- ▶ An array's `length` field stores its number of elements.

**<name>**.length

```
for (int i = 0; i < numbers.length; i++) {  
    System.out.print(numbers[i] + " ");  
}
```

// output: 0 2 4 6 8 10 12 14

- It does not use parentheses like a String's `.length()`.
- ▶ What expressions refer to:
  - The last element of any array?
  - The middle element?

# Weather question

- Use an array to solve the weather problem:

How many days' temperatures? 7

Day 1's high temp: 45

Day 2's high temp: 44

Day 3's high temp: 39

Day 4's high temp: 48

Day 5's high temp: 37

Day 6's high temp: 46

Day 7's high temp: 53

Average temp = 44.6

4 days were above average.

# Weather answer

```
// Reads temperatures from the user, computes average and # days above average.
import java.util.*;

public class Weather {
    public static void main(String[] args) {
        Scanner console = new Scanner(System.in);
        System.out.print("How many days' temperatures? ");
        int days = console.nextInt();

        int[] temps = new int[days];           // array to store days' temperatures
        int sum = 0;

        for (int i = 0; i < days; i++) {      // read/store each day's temperature
            System.out.print("Day " + (i + 1) + "'s high temp: ");
            temps[i] = console.nextInt();
            sum += temps[i];
        }
        double average = (double) sum / days;

        int count = 0;                       // see if each day is above average
        for (int i = 0; i < days; i++) {
            if (temps[i] > average) {
                count++;
            }
        }

        // report results
        System.out.printf("Average temp = %.1f\n", average);
        System.out.println(count + " days above average");
    }
}
```

# Quick array initialization

**<type>[] <name> = { <value>, <value>, ... <value> } ;**

– Example:

```
int[] numbers = {12, 49, -2, 26, 5, 17, -6};
```

<i>index</i>	0	1	2	3	4	5	6
<i>value</i>	12	49	-2	26	5	17	-6

- Useful when you know what the array's elements will be
- The compiler determines the length by counting the values



# "Array mystery" problem

- ▶ **traversal:** An examination of each element of an array.
- ▶ What element values are stored in the following array?

```
int[] a = {1, 7, 5, 6, 4, 14, 11};  
for (int i = 0; i < a.length - 1; i++) {  
    if (a[i] > a[i + 1]) {  
        a[i + 1] = a[i + 1] * 2;  
    }  
}
```

<i>index</i>	0	1	2	3	4	5	6
<i>value</i>	1	7	10	12	8	14	22

# Limitations of arrays

- ▶ You cannot resize an existing array:

```
int[] a = new int[4];  
a.length = 10;           // error
```

- ▶ You cannot compare arrays with `==` or `equals`:

```
int[] a1 = {42, -7, 1, 15};  
int[] a2 = {42, -7, 1, 15};  
if (a1 == a2) { ... }           // false!  
if (a1.equals(a2)) { ... }     // false!
```

- ▶ An array does not know how to print itself:

```
int[] a1 = {42, -7, 1, 15};  
System.out.println(a1);           // [I@98f8c4]
```

# The Arrays class

- ▶ Class `Arrays` in package `java.util` has useful static methods for manipulating arrays:

Method name	Description
<code>binarySearch(&lt;array&gt;, &lt;value&gt;)</code>	returns the index of the given value in a <i>sorted</i> array (or <code>&lt; 0</code> if not found)
<code>copyOf(&lt;array&gt;, &lt;length&gt;)</code>	returns a new copy of an array
<code>equals(&lt;array1&gt;, &lt;array2&gt;)</code>	returns <code>true</code> if the two arrays contain same elements in the same order
<code>fill(&lt;array&gt;, &lt;value&gt;)</code>	sets every element to the given value
<code>sort(&lt;array&gt;)</code>	arranges the elements into sorted order
<code>toString(&lt;array&gt;)</code>	returns a string representing the array, such as <code>"[10, 30, -25, 17]"</code>

- ▶ Syntax:

`Arrays.<methodName>(<parameters>)`

# Arrays.toString

- `Arrays.toString` accepts an array as a parameter and returns a `String` representation of its elements.

```
int[] e = {0, 2, 4, 6, 8};  
e[1] = e[3] + e[4];  
System.out.println("e is " + Arrays.toString(e));
```

Output:

```
e is [0, 14, 4, 6, 8]
```

– Must import `java.util.Arrays`;

# Weather question 2

- Modify the weather program to print the following output:

```
How many days' temperatures? 7
Day 1's high temp: 45
Day 2's high temp: 44
Day 3's high temp: 39
Day 4's high temp: 48
Day 5's high temp: 37
Day 6's high temp: 46
Day 7's high temp: 53
Average temp = 44.6
4 days were above average.
```

```
Temperatures: [45, 44, 39, 48, 37, 46, 53]
Two coldest days: 37, 39
Two hottest days: 53, 48
```

# Weather answer 2

```
// Reads temperatures from the user, computes average and # days above average.
import java.util.*;

public class Weather2 {
    public static void main(String[] args) {
        ...
        int[] temps = new int[days];           // array to store days' temperatures
        ...   (same as Weather program)

        // report results
        System.out.printf("Average temp = %.1f\n", average);
        System.out.println(count + " days above average");

        System.out.println("Temperatures: " + Arrays.toString(temps));
        Arrays.sort(temps);
        System.out.println("Two coldest days: " + temps[0] + ", " + temps[1]);
        System.out.println("Two hottest days: " + temps[temps.length - 1] +
                           ", " + temps[temps.length - 2]);
    }
}
```

# Topic 22

## arrays - part 2

# Swapping values

```
public static void main(String[] args) {  
    int a = 7;  
    int b = 35;  
    // swap a with b?  
    a = b;  
    b = a;  
    System.out.println(a + " " + b);  
}
```

– What is wrong with this code? What is its output?

► The red code should be replaced with:

```
int temp = a;  
a = b;  
b = temp;
```



# Array reversal question

- ▶ Write code that reverses the elements of an array.
  - For example, if the array initially stores:  
`[11, 42, -5, 27, 0, 89]`
  - Then after your reversal code, it should store:  
`[89, 0, 27, -5, 42, 11]`
  - The code should work for an array of any size.
  - Hint: think about swapping various elements...

# Algorithm idea

- ▶ Swap pairs of elements from the edges; work inwards:

<i>index</i>	0	1	2	3	4	5
<i>value</i>	89	0	27	-5	42	11
	↑	↑	↑	↑	↑	↑

# Flawed algorithm

- What's wrong with this code?

```
int[] numbers = [11, 42, -5, 27, 0, 89];  
  
// reverse the array  
for (int i = 0; i < numbers.length; i++) {  
    int temp = numbers[i];  
    numbers[i] = numbers[numbers.length - 1 - i];  
    numbers[numbers.length - 1 - i] = temp;  
}
```

- The loop goes too far and un-reverses the array! Fixed version:

```
for (int i = 0; i < numbers.length / 2; i++) {  
    int temp = numbers[i];  
    numbers[i] = numbers[numbers.length - 1 - i];  
    numbers[numbers.length - 1 - i] = temp;  
}
```

# Array reverse question 2

- ▶ Turn your array reversal code into a `reverse` method.
  - Accept the array of integers to reverse as a parameter.

```
int[] numbers = {11, 42, -5, 27, 0, 89};  
reverse(numbers);
```

- How do we write methods that accept arrays as parameters?
- Will we need to return the new array contents after reversal?

...

# Array parameter (declare)

```
public static <type> <method>(<type>[] <name>) {
```

## ► Example:

```
// Returns the average of the given array of numbers.
```

```
public static double average(int[] numbers) {  
    int sum = 0;  
    for (int i = 0; i < numbers.length; i++) {  
        sum += numbers[i];  
    }  
    return (double) sum / numbers.length;  
}
```

- You don't specify the array's length (but you can examine it).

# Array parameter (call)

**<methodName>** (**<arrayName>**) ;

## ► Example:

```
public class MyProgram {  
    public static void main(String[] args) {  
        // figure out the average TA IQ  
        int[] iq = {126, 84, 149, 167, 95};  
        double avg = average(iq);  
        System.out.println("Average IQ = " + avg);  
    }  
    ...  
}
```

- Notice that you don't write the `[]` when passing the array.

# Array return (declare)

```
public static <type>[] <method>(<parameters>) {
```

## ► Example:

```
// Returns a new array with two copies of each value.
```

```
// Example: [1, 4, 0, 7] -> [1, 1, 4, 4, 0, 0, 7, 7]
```

```
public static int[] stutter(int[] numbers) {  
    int[] result = new int[2 * numbers.length];  
    for (int i = 0; i < numbers.length; i++) {  
        result[2 * i] = numbers[i];  
        result[2 * i + 1] = numbers[i];  
    }  
    return result;  
}
```

# Array return (call)

**<type>[] <name> = <method>(<parameters>) ;**

## ► Example:

```
public class MyProgram {  
    public static void main(String[] args) {  
        int[] iq = {126, 84, 149, 167, 95};  
        int[] stuttered = stutter(iq);  
  
        System.out.println(Arrays.toString(stuttered));  
    }  
    ...  
}
```

## ► Output:

```
[126, 126, 84, 84, 149, 149, 167, 167, 95, 95]
```



# Reference semantics

**reading: 7.3**

# Clicker 1

► What is output by the following code?

```
int[] data = {1, 5, 3};  
foo(data);  
System.out.print(Arrays.toString(data));  
  
public static void foo(int[] d) {  
    int temp = d[0];  
    d[0] = d[d.length - 1];  
    d[d.length - 1] = temp;  
    System.out.print(Arrays.toString(d) + " ");  
}
```

A. [3, 5, 1] [1, 5, 3]

B. [3, 5, 1] [3, 5, 1]

C. [1, 5, 3] [1, 5, 3]

D. [5, 3, 1] [1, 5, 3]

E. Something else

## clicker 2

► What is output by the following code?

```
int[] data = {1, 5, 3};  
bar(data);  
System.out.print(Arrays.toString(data));  
  
public static void bar(int[] d) {  
    d[0]++;  
    d = new int[] {4, 6};  
    System.out.print(Arrays.toString(d) + " ");  
}
```

A. [4, 6] [2, 5, 3]

B. [4, 6] [ 4, 6]

C. [1, 5, 3] [1, 5, 3]

D. [2, 5, 3] [2, 5, 3]

E. Something else

# A swap method?

- Does the following swap method work? Why or why not?

```
public static void main(String[] args) {  
    int a = 7;  
    int b = 35;  
  
    // swap a with b?  
    swap(a, b);  
  
    System.out.println(a + " " + b);  
}  
  
public static void swap(int a, int b) {  
    int temp = a;  
    a = b;  
    b = temp;  
}
```

# Value semantics

- ▶ **value semantics:** Behavior where values are copied when assigned, passed as parameters, or returned.
  - All primitive types in Java use value semantics.
  - When one variable is assigned to another, its value is copied.
  - Modifying the value of one variable does not affect others.

```
int x = 5;
```

```
int y = x;
```

```
y = 17;
```

```
x = 8;
```

```
// x = 5, y = 5
```

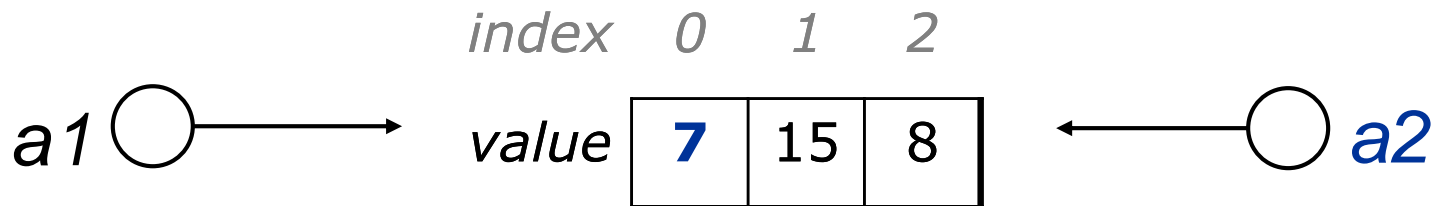
```
// x = 5, y = 17
```

```
// x = 8, y = 17
```

# Reference semantics (objects)

- ▶ **reference semantics:** Behavior where variables actually store the address of an object in memory.
  - When one variable is assigned to another, the object is *not* copied; both variables refer to the *same object*.
  - Modifying the value of one variable *will* affect others.

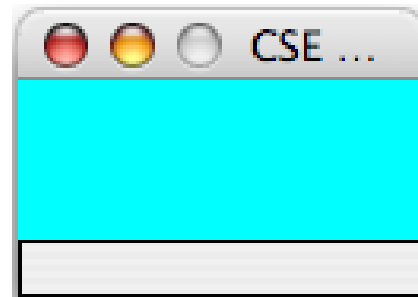
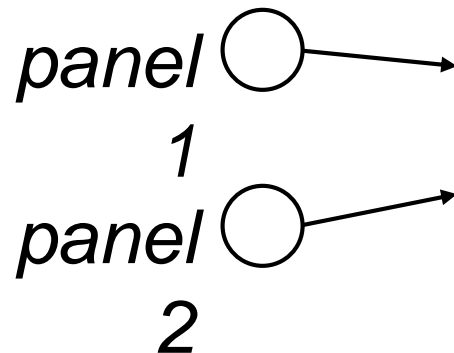
```
int[] a1 = {4, 15, 8};  
int[] a2 = a1;           // refer to same array as a1  
a2[0] = 7;  
System.out.println(Arrays.toString(a1)); // [7, 15, 8]
```



# References and objects

- ▶ Arrays and objects use reference semantics. Why?
  - *efficiency*. Copying large objects slows down a program.
  - *sharing*. It's useful to share an object's data among methods.

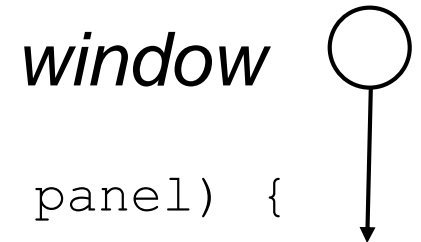
```
DrawingPanel panel1 = new DrawingPanel(80, 50);  
DrawingPanel panel2 = panel1;    // same window  
panel2.setBackground(Color.CYAN);
```



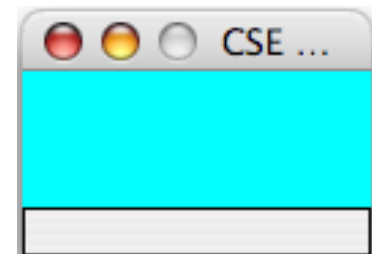
# Objects as parameters

- ▶ When an object is passed as a parameter, the object is *not* copied. The parameter refers to the same object.
  - If the parameter is modified, it *will* affect the original object.

```
public static void main(String[] args) {  
    DrawingPanel window = new DrawingPanel(80, 50);  
    window.setBackground(Color.YELLOW);  
    example(window);  
}
```



```
public static void example(DrawingPanel panel) {  
    panel.setBackground(Color.CYAN);  
    ...  
}
```

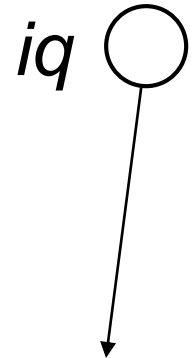




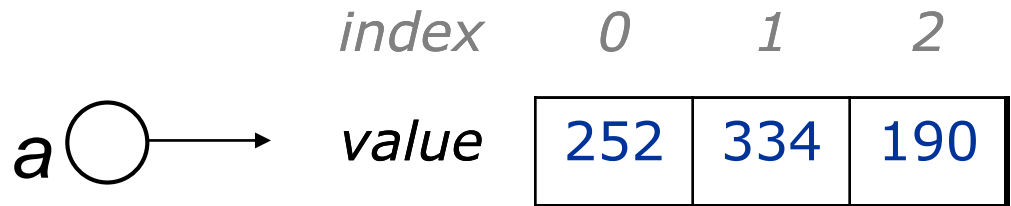
# Copy of a reference

- ▶ Array variables are references
- ▶ A parameter is a copy of the same reference the argument stores.
- ▶ Changes made in the method **to the elements** are also seen by the caller.

```
public static void main(String[] args) {  
    int[] iq = {126, 167, 95};  
    increase(iq);  
    System.out.println(Arrays.toString(iq));  
}  
  
public static void increase(int[] a) {  
    for (int i = 0; i < a.length; i++) {  
        a[i] = a[i] * 2;  
    }  
}
```



– Output:  
[252, 334, 190]



# Array reverse question 2

- ▶ Turn your array reversal code into a `reverse` method.
  - Accept the array of integers to reverse as a parameter.

```
int[] numbers = {11, 42, -5, 27, 0, 89};  
reverse(numbers);
```

- ▶ Solution:

```
public static void reverse(int[] numbers) {  
    for (int i = 0; i < numbers.length / 2; i++) {  
        int temp = numbers[i];  
        numbers[i] = numbers[numbers.length - 1 - i];  
        numbers[numbers.length - 1 - i] = temp;  
    }  
}
```

# Array parameter questions

- ▶ Write a method `swap` that accepts an array of integers and two indexes and swaps the elements at those indexes.

```
int[] a1 = {12, 34, 56};  
swap(a1, 1, 2);  
System.out.println(Arrays.toString(a1)); // [12, 56, 34]
```

- ▶ Write a method `swapAll` that accepts two arrays of integers as parameters and swaps their entire contents.

– Assume that the two arrays are the same length.

```
int[] a1 = {12, 34, 56};  
int[] a2 = {20, 50, 80};  
swapAll(a1, a2);  
System.out.println(Arrays.toString(a1)); // [20, 50, 80]  
System.out.println(Arrays.toString(a2)); // [12, 34, 56]
```

# Array parameter answers

**// Swaps the values at the given two indexes.**

```
public static void swap(int[] a, int i, int j) {  
    int temp = a[i];  
    a[i] = a[j];  
    a[j] = temp;  
}
```

**// Swaps the entire contents of a1 with those of a2.**

```
public static void swapAll(int[] a1, int[] a2) {  
    for (int i = 0; i < a1.length; i++) {  
        int temp = a1[i];  
        a1[i] = a2[i];  
        a2[i] = temp;  
    }  
}
```

# Array return question

- Write a method `merge` that accepts two arrays of integers and returns a new array containing all elements of the first array followed by all elements of the second.

```
int[] a1 = {12, 34, 56};  
int[] a2 = {7, 8, 9, 10};  
  
int[] a3 = merge(a1, a2);  
System.out.println(Arrays.toString(a3));  
// [12, 34, 56, 7, 8, 9, 10]
```

- Write a method `merge3` that merges 3 arrays similarly.

```
int[] a1 = {12, 34, 56};  
int[] a2 = {7, 8, 9, 10};  
int[] a3 = {444, 222, -1};  
  
int[] a4 = merge3(a1, a2, a3);  
System.out.println(Arrays.toString(a4));  
// [12, 34, 56, 7, 8, 9, 10, 444, 222, -1]
```

# Array return answer 1

```
// Returns a new array containing all elements of a1
// followed by all elements of a2.
public static int[] merge(int[] a1, int[] a2) {
    int[] result = new int[a1.length + a2.length];
    for (int i = 0; i < a1.length; i++) {
        result[i] = a1[i];
    }
    for (int i = 0; i < a2.length; i++) {
        result[a1.length + i] = a2[i];
    }
    return result;
}
```

# Array return answer 2

// Returns a new array containing all elements of  
a1,a2,a3.

```
public static int[] merge3(int[] a1, int[] a2, int[] a3) {  
    int[] a4 = new int[a1.length + a2.length + a3.length];  
    for (int i = 0; i < a1.length; i++) {  
        a4[i] = a1[i];  
    }  
    for (int i = 0; i < a2.length; i++) {  
        a4[a1.length + i] = a2[i];  
    }  
    for (int i = 0; i < a3.length; i++) {  
        a4[a1.length + a2.length + i] = a3[i];  
    }  
    return a4;  
}
```

// Shorter version that calls merge.

```
public static int[] merge3(int[] a1, int[] a2, int[] a3) {  
    return merge(merge(a1, a2), a3);  
}
```

# Topic 23

## arrays - part 3 (tallying, text processing)

"42 million of *anything* is a lot."

-Doug Burger, circa 2003

(commenting on the number of transistors in the Pentium IV processor)

As of 2020 processors for personal computers have, on the order of, **billions** of transistors.





► What is output when method clicker2 is called?

```
public static void clicker2() {  
    int[] values = {1, 2};  
    arrayManip(values);  
    System.out.print(Arrays.toString(values));  
}
```

```
public static void arrayManip(int[] values) {  
    values[1] += 2;  
    values[0] -= 2;  
    System.out.print(Arrays.toString(values));  
    values = new int[3];  
    System.out.print(Arrays.toString(values));  
}
```

- A. [1, 2] [0, 0, 0] [1, 2]
- B. [1, 2] [1, 2] [1, 2]
- C. [-1, 4] [0, 0, 0] [0, 0, 0]
- D. [-1, 4] [0, 0, 0] [1, 2]
- E. [-1, 4] [0, 0, 0] [-1, 4]

# A multi-counter problem

- ▶ Problem: Write a method `mostFrequentDigit` that returns the digit that occurs most frequently in a number.
  - Example: The number 669260267 contains:  
one 0, two 2s, four 6es, one 7, and one 9.  
`mostFrequentDigit(669260267)` returns 6.
  - If there is a tie, return the digit with the lower value.  
`mostFrequentDigit(57135203)` returns 3.

# A multi-counter problem

- ▶ We could declare 10 counter variables ...

```
int counter0, counter1, counter2, counter3, counter4,  
    counter5, counter6, counter7, counter8, counter9;
```

- ▶ But a better solution is to use an array of size 10.

- The element at index  $i$  will store the counter for digit value  $i$ .
- Example for 669260267:

<i>index</i>	0	1	2	3	4	5	6	7	8	9
<i>value</i>	1	0	2	0	0	0	4	1	0	0

- How do we build such an array? And how does it help?

# Creating an array of tallies

```
// assume n = 669260267
int[] counts = new int[10];
while (n > 0) {
    // pluck off a digit and add to proper counter
    int digit = n % 10;
    counts[digit]++;
    n = n / 10;
}
```

*index*    0    1    2    3    4    5    6    7    8    9

<i>value</i>	1	0	2	0	0	0	4	1	0	0
--------------	---	---	---	---	---	---	---	---	---	---

# Tally solution

```
// Returns the digit value that occurs most frequently in n.
// Breaks ties by choosing the smaller value.
public static int mostFrequentDigit(int n) {
    int[] counts = new int[10];
    while (n > 0) {
        int digit = n % 10; // pluck off a digit and tally it
        counts[digit]++;
        n = n / 10;
    }

    // find the most frequently occurring digit
    int bestIndex = 0;
    for (int i = 1; i < counts.length; i++) {
        if (counts[i] > counts[bestIndex]) {
            bestIndex = i;
        }
    }

    return bestIndex;
}
```

# Tally Problem

- ▶ Write a method to pick random numbers from 0 to 99.
- ▶ A parameters specifies the number of random numbers to pick
- ▶ The method returns the difference between the number of times the most and least picked number
- ▶ **Clicker 2:** With 1,000,000 numbers what do you expect the difference to be?

A. 0                      B. 1 - 10                      C. 11 - 100  
D. 101 - 1000    E. > 1000

# Array histogram question

- ▶ Given a file of integer exam scores, such as:

82

66

79

63

83

Write a program that will print a histogram of stars indicating the number of students who earned each unique exam score.

85 : \* \* \* \* \*

86 : \* \* \* \* \* \* \* \* \* \* \* \*

87 : \* \* \*

88 : \*

91 : \* \* \* \*

# Array histogram answer

```
// Reads a file of test scores and shows a histogram of the score distribution.
import java.io.*;
import java.util.*;

public class Histogram {
    public static void main(String[] args) throws FileNotFoundException {
        Scanner input = new Scanner(new File("midterm.txt"));
        int[] counts = new int[101];           // counters of test scores 0 - 100

        while (input.hasNextInt()) {          // read file into counts array
            int score = input.nextInt();
            counts[score]++;                 // if score is 87, then counts[87]++
        }

        for (int i = 0; i < counts.length; i++) {    // print star histogram
            if (counts[i] > 0) {
                System.out.print(i + ": ");
                for (int j = 0; j < counts[i]; j++) {
                    System.out.print("*");
                }
                System.out.println();
            }
        }
    }
}
```



# Text processing

**reading: 4.3**

# Type char

- **char** : A primitive type representing single characters.
  - A `String` is stored internally as an array of `char`

```
String s = "Ali G.";
```

<i>index</i>	0	1	2	3	4	5
<i>value</i>	'A'	'l'	'i'	' '	'G'	'.'

- It is legal to have variables, parameters, returns of type `char`
  - surrounded with apostrophes: `'a'` or `'4'` or `'\n'` or `'\''`

```
char letter = 'T';
```

```
System.out.println(letter);
```

```
// T
```

```
System.out.println(letter + "exas!");
```

```
// Texas!
```

# The charAt method

- The chars in a String can be accessed using the charAt method.
  - accepts an int index parameter and returns the char at that index

```
String food = "cookie";  
char firstLetter = food.charAt(0);    // 'c'  
System.out.println(firstLetter + " is for " + food);
```

- You can use a for loop to print or examine each character.

```
String major = "CS!";  
for (int i = 0; i < major.length(); i++) {  
    char c = major.charAt(i);  
    System.out.println(c);  
}  
// output:  
// C  
// S  
// !
```

# Comparing char values

- ▶ You can compare chars with ==, !=, and other operators:

```
String word = console.next();  
char last = word.charAt(word.length() - 1);  
if (last == 's') {  
    System.out.println(word  
        + " is plural.");  
}
```

```
// prints the alphabet  
for (char c = 'a'; c <= 'z'; c++) {  
    System.out.print(c);  
}
```

# char VS. int

- ▶ Each `char` is mapped to an integer value internally
  - Called an **ASCII value**

'A' is 65

'B' is 66

' ' is 32

'a' is 97

'b' is 98

'\*' is 42

- Mixing `char` and `int` causes automatic conversion to `int`.

'a' + 10 is 107,  
is 130

'A' + 'A'

- To convert an `int` into the equivalent `char`, type-cast it.

(char) ('a' + 2) is 'c'

# char VS. String

- ▶ "h" is a String, but 'h' is a char (they are different)

- ▶ A String is an object; it contains methods.

```
String s = "h";  
s = s.toUpperCase();           // "H"  
int len = s.length();         // 1  
char first = s.charAt(0);     // 'H'
```

- ▶ A char is primitive; you can't call methods on it.

```
char c = 'h';  
c = c.toUpperCase();           // ERROR  
s = s.charAt(0).toUpperCase(); // ERROR
```

- What is `s + 1` ? What is `c + 1` ?
- What is `s + s` ? What is `c + c` ?

# String traversals

- ▶ We can write algorithms to traverse strings to compute information.
- ▶ What useful information might the following string have?

"GDRGRRGDRRGDLGDGRRRRGRGRGGDGDDRDRRDGDGGD"

# Data takes many forms

```
// string stores voters' votes
// (R)EPUBLICAN, (D)EMOCRAT, (G)REEN, (L)IBERTARIAN
String votes =
"GDRGRRRGDRRGDLGDGRRRRGRGRGGDGDDRDRDRRDGDGGD";
int[] counts = new int[4]; // R -> 0, D -> 1, G -> 2, L -> 3
for (int i = 0; i < votes.length(); i++) {
    char c = votes.charAt(i);
    if (c == 'R') {
        counts[0]++;
    } else if (c == 'D') {
        counts[1]++;
    } else if (c == 'B') {
        counts[2]++;
    } else { // c == 'M'
        counts[3]++;
    }
}

System.out.println(Arrays.toString(counts));
```

## Output:

[13, 12, 14, 1]



# Section attendance question

- Read a file of section attendance (see *next slide*):

```
yyynyynayayynyayanyaynayyayyanayyyanyayna  
ayyanayyyayanaayyanayyyananayayaynyayayyny  
yyayaynyyayyanynnyyyayyanayaynannnyyayyayny
```

- And produce the following output:

Section 1

Student points: [20, 17, 19, 16, 13]

Student grades: [100.0, 85.0, 95.0, 80.0, 65.0]

Section 2

Student points: [17, 20, 16, 16, 10]

Student grades: [85.0, 100.0, 80.0, 80.0, 50.0]

Section 3

Student points: [17, 18, 17, 20, 16]

Student grades: [85.0, 90.0, 85.0, 100.0, 80.0]

- Students earn 3 points for each section attended up to 20.

# Section input file

<b>student</b>		1	2	3	4	5	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5						
<b>week</b>		1	2	3	4	5	6	7	8	9																																
<b>section</b>	1	y	y	n	y	y	n	a	y	a	y	n	y	y	y	a	y	a	n	y	y	y	a	y	n	a	y	y	a	y	n	a	y	y	a	n						
<b>section</b>	2	a	y	y	a	n	y	y	y	y	a	y	a	n	a	a	y	y	a	n	a	y	y	y	a	n	a	n	a	y	a	y	n	y	a	y	n					
<b>section</b>	3	y	y	a	y	a	n	y	y	a	y	y	a	n	y	n	n	y	y	y	a	y	y	a	n	a	y	a	y	n	a	n	n	n	y	y	a	y	a	y	n	y

- Each line represents a section.
- A line consists of 9 weeks' worth of data.
  - Each week has 5 characters because there are 5 students.
- Within each week, each character represents one student.
  - a means the student was absent (+0 points)
  - n means they attended but didn't do the problems (+2 points)
  - y means they attended and did the problems (+3 points)

# Section attendance answer

```
import java.io.*;
import java.util.*;

public class Sections {
    public static void main(String[] args) throws FileNotFoundException {
        Scanner input = new Scanner(new File("sections.txt"));
        int section = 1;
        while (input.hasNextLine()) {
            String line = input.nextLine();           // process one section
            int[] points = new int[5];
            for (int i = 0; i < line.length(); i++) {
                int student = i % 5;
                int earned = 0;
                if (line.charAt(i) == 'y') {           // c == 'y' or 'n'
                    earned = 3;
                } else if (line.charAt(i) == 'n') {
                    earned = 2;
                }
                points[student] = Math.min(20, points[student] + earned);
            }

            double[] grades = new double[5];
            for (int i = 0; i < points.length; i++) {
                grades[i] = 100.0 * points[i] / 20.0;
            }

            System.out.println("Section " + section);
            System.out.println("Student points: " + Arrays.toString(points));
            System.out.println("Student grades: " + Arrays.toString(grades));
            System.out.println();
            section++;
        }
    }
}
```

# Data transformations

- ▶ In many problems we transform data between forms.
  - Example: digits  $\rightarrow$  count of each digit  $\rightarrow$  most frequent digit
  - Often each transformation is computed/stored as an array.
  - For structure, a transformation is often put in its own method.
- ▶ Sometimes we map between data and array indexes.
  - by position (store the  $i^{\text{th}}$  value we read at index  $i$ )
  - tally (if input value is  $i$ , store it at array index  $i$ )
  - explicit mapping (count 'J' at index 0, count 'X' at index 1)
- ▶ *Exercise:* Modify the Sections program to use static methods that use arrays as parameters and returns.

# Array param/return answer

```
// This program reads a file representing which students attended
// which discussion sections and produces output of the students'
// section attendance and scores.

import java.io.*;
import java.util.*;

public class Sections2 {
    public static void main(String[] args) throws FileNotFoundException {
        Scanner input = new Scanner(new File("sections.txt"));
        int section = 1;
        while (input.hasNextLine()) {
            // process one section
            String line = input.nextLine();
            int[] points = countPoints(line);
            double[] grades = computeGrades(points);
            results(section, points, grades);
            section++;
        }
    }

    // Produces all output about a particular section.
    public static void results(int section, int[] points, double[] grades) {
        System.out.println("Section " + section);
        System.out.println("Student scores: " + Arrays.toString(points));
        System.out.println("Student grades: " + Arrays.toString(grades));
        System.out.println();
    }

    ...
}
```

# Array param/return answer

...

**// Computes the points earned for each student for a particular section.**

```
public static int[] countPoints(String line) {  
    int[] points = new int[5];  
    for (int i = 0; i < line.length(); i++) {  
        int student = i % 5;  
        int earned = 0;  
        if (line.charAt(i) == 'y') {           // c == 'y'   or   c == 'n'  
            earned = 3;  
        } else if (line.charAt(i) == 'n') {  
            earned = 2;  
        }  
        points[student] = Math.min(20, points[student] + earned);  
    }  
    return points;  
}
```

**// Computes the percentage for each student for a particular section.**

```
public static double[] computeGrades(int[] points) {  
    double[] grades = new double[5];  
    for (int i = 0; i < points.length; i++) {  
        grades[i] = 100.0 * points[i] / 20.0;  
    }  
    return grades;  
}
```

```
}
```

# Topic 24

## Sorting and Searching arrays

"There's nothing in your head the sorting hat can't see. So try me on and I will tell you where you ought to be."

-The Sorting Hat,  
*Harry Potter and  
the Sorcerer's Stone*



# Searching

- ▶ Given an array of ints find the index of the first occurrence of a target int

<i>index</i>	0	1	2	3	4	5
<i>value</i>	89	0	27	-5	42	11

- ▶ Given the above array and a target of 27 the method returns 2
- ▶ What if not present?
- ▶ What if more than one occurrence?



# Clicker 1

- ▶ Given an array with 1,000,000 distinct elements in random order, how many elements do you expect to look at (on average) when searching if:

item present

item not present

- |    |           |           |
|----|-----------|-----------|
| A. | 1         | 1,000,000 |
| B. | 500,000   | 1,000,000 |
| C. | 1,000,000 | 1,000,000 |
| D. | 1,000     | 500,000   |
| E. | 20        | 1,000,000 |

# linear or sequential search

# Sorting

## XKCD

<http://xkcd.com/1185/>

### INEFFECTIVE SORTS

```
DEFINE HALFHEARTEDMERGESORT(LIST):  
  IF LENGTH(LIST) < 2:  
    RETURN LIST  
  PIVOT = INT(LENGTH(LIST) / 2)  
  A = HALFHEARTEDMERGESORT(LIST[:PIVOT])  
  B = HALFHEARTEDMERGESORT(LIST[PIVOT:])  
  // UMMMMMM  
  RETURN [A, B] // HERE. SORRY.
```

```
DEFINE FASTBOGOSORT(LIST):  
  // AN OPTIMIZED BOGOSORT  
  // RUNS IN O(N LOG N)  
  FOR N FROM 1 TO LOG(LENGTH(LIST)):  
    SHUFFLE(LIST):  
    IF ISSORTED(LIST):  
      RETURN LIST  
  RETURN "KERNEL PAGE FAULT (ERROR CODE: 2)"
```

```
DEFINE JOBSITEINTERVIEWQUICKSORT(LIST):  
  OK SO YOU CHOOSE A PIVOT  
  THEN DIVIDE THE LIST IN HALF  
  FOR EACH HALF:  
    CHECK TO SEE IF IT'S SORTED  
    NO, WAIT, IT DOESN'T MATTER  
    COMPARE EACH ELEMENT TO THE PIVOT  
    THE BIGGER ONES GO IN A NEW LIST  
    THE EQUAL ONES GO INTO, UH  
    THE SECOND LIST FROM BEFORE  
    HANG ON, LET ME NAME THE LISTS  
    THIS IS LIST A  
    THE NEW ONE IS LIST B  
    PUT THE BIG ONES INTO LIST B  
    NOW TAKE THE SECOND LIST  
    CALL IT LIST, UH, A2  
    WHICH ONE WAS THE PIVOT IN?  
    SCRATCH ALL THAT  
    IT JUST RECURSIVELY CALLS ITSELF  
    UNTIL BOTH LISTS ARE EMPTY  
    RIGHT?  
    NOT EMPTY, BUT YOU KNOW WHAT I MEAN  
    AM I ALLOWED TO USE THE STANDARD LIBRARIES?
```

```
DEFINE PANICSORT(LIST):  
  IF ISSORTED(LIST):  
    RETURN LIST  
  FOR N FROM 1 TO 10000:  
    PIVOT = RANDOM(0, LENGTH(LIST))  
    LIST = LIST[PIVOT:] + LIST[:PIVOT]  
    IF ISSORTED(LIST):  
      RETURN LIST  
  IF ISSORTED(LIST):  
    RETURN LIST  
  IF ISSORTED(LIST): // THIS CAN'T BE HAPPENING  
    RETURN LIST  
  IF ISSORTED(LIST): // COME ON COME ON  
    RETURN LIST  
  // OH JEEZ  
  // I'M GONNA BE IN SO MUCH TROUBLE  
  LIST = [ ]  
  SYSTEM("SHUTDOWN -H +5")  
  SYSTEM("RM -RF ./")  
  SYSTEM("RM -RF ~/*")  
  SYSTEM("RM -RF /")  
  SYSTEM("RD /S /Q C:\*") // PORTABILITY  
  RETURN [1, 2, 3, 4, 5]
```

# Sorting

- ▶ A fundamental application for computers
- ▶ Done to make finding data (searching) faster
- ▶ Many different algorithms for sorting
- ▶ One of the difficulties with sorting is working with a fixed size storage container (array)
  - if resize, that is expensive (slow)
  - Trying to apply a human technique of sorting can be difficult
  - try sorting a pile of papers and clearly write out the algorithm you follow

# Selection Sort

- ▶ To sort a list into ascending order:
  - Find the smallest item in an array, the minimum
  - Put that value in the first element of the array
    - Where to put the value that was in the first location?
  - And now...?

# Selection Sort in Practice

44 68 191 119 119 37 83 82 191 45 158 130 76 153 39 25

<http://tinyurl.com/d7kxxxf>

animation of selection sort algorithm

# Implementation of Selection Sort

- ▶ Include println commands to trace the sort

## Clicker 2

► Determine how long it takes to sort an array with 100,000 elements in random order using selection sort. When the number of elements is increased to 200,000 how long will it take to sort the array?

- A. About the same
- B. 1.5 times as long
- C. 2 times as long
- D. 4 times as long
- E. 8 times as long



# Insertion Sort

- ▶ Another of the Simple sort
- ▶ The first item is sorted
- ▶ Compare the second item to the first
  - if smaller swap
- ▶ Third item, compare to item next to it
  - need to swap
  - after swap compare again
- ▶ And so forth...

# Insertion Sort in Practice

44 68 191 119 119 37 83 82 191 45 158 130 76 153 39 25

<http://tinyurl.com/d8spm2l>

animation of insertion sort algorithm

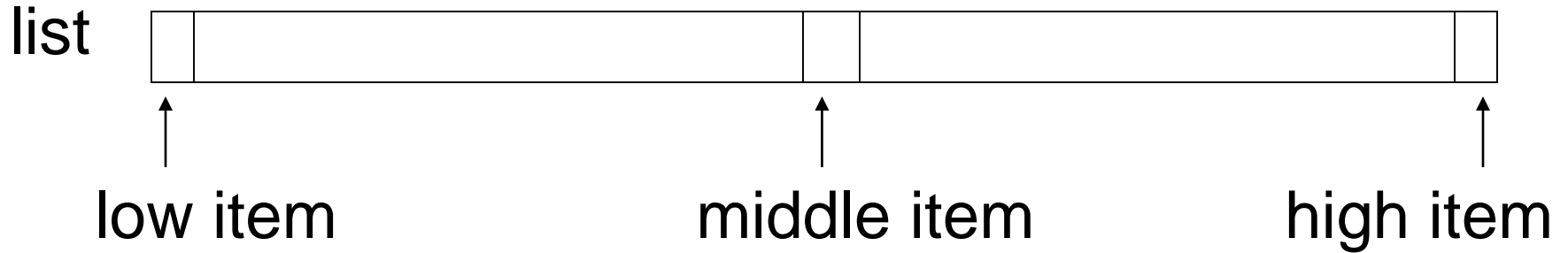
# Binary Search



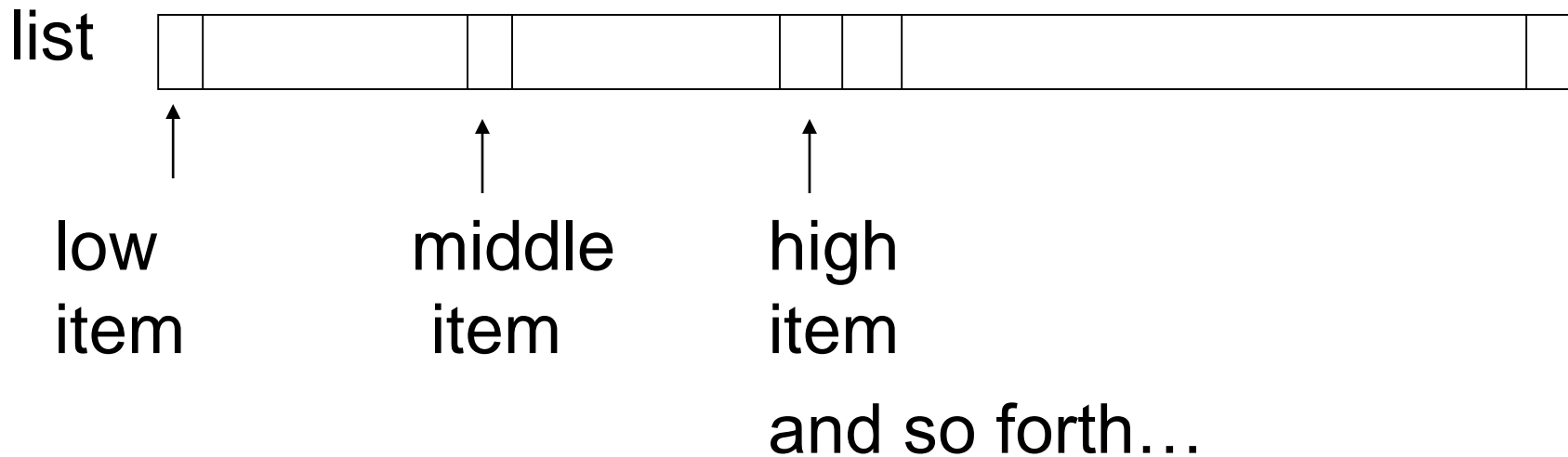
# Searching in a Sorted List

- ▶ If items are sorted then we can *divide and conquer*
- ▶ dividing your work in half with each step
  - generally a good thing
- ▶ The Binary Search on List in Ascending order
  - Start at middle of list
  - is that the item?
  - If not is it less than or greater than the item?
  - less than, move to second half of list
  - greater than, move to first half of list
  - repeat until found or sub list size = 0

# Binary Search



Is middle item what we are looking for? If not is it more or less than the target item? (Assume lower)



# Implement Binary Search

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	3	5	7	11	13	17	19	23	29	31	37	41	43	47	53

Trace When Key == 3  
Trace When Key == 30

Variables of Interest?

# Clicker 3

- ▶ Given an array with 1,000,000 elements in sorted order, how many elements do you expect to look at when searching (with binary search) for a value if:

	item present once	item not present
--	-------------------	------------------

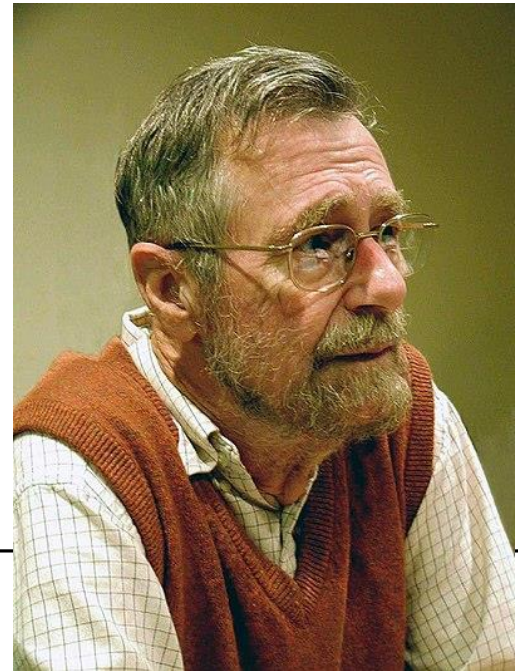
- |    |       |           |
|----|-------|-----------|
| A. | 1     | 500,000   |
| B. | 20    | 20        |
| C. | 1     | 1,000,000 |
| D. | 1,000 | 500,000   |
| E. | 1,000 | 1,000     |



# Topic 25 - more array algorithms

"The art of programming is the art of organizing complexity, of mastering multitude, and avoiding its [awful] chaos as effectively as possible.

- Edsger W. Dijkstra



# More array problems

- ▶ write a method to change an array to a sub-array, similar to substring method
- ▶ "rotate" elements in an array a given amount
- ▶ **determine how many elements in an array of Strings variables are set to null**
- ▶ determine if the elements in an array of ints or doubles are in sorted ascending order
- ▶ **Determine which character occurs most frequently in a file. Clicker 1:**
  - A. e      B. i      C. s
  - D. t      E. something else

# More array problems

- ▶ shuffle an array
- ▶ **determine the longest run length in an array of booleans (longest run of all booleans the same)**
- ▶ ensure all elements in an array are within a given range
- ▶ given an array with ints 1 to N determine if there are any duplicates in the array

# More array problems

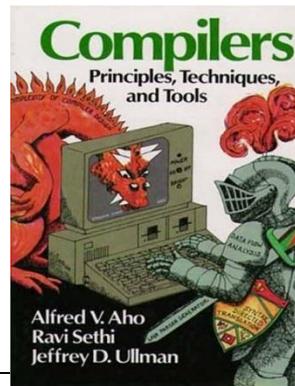
- ▶ **given an array, create and return an array the same as the original expect all duplicates are removed**
- ▶ implement the sieve of Eratosthenes to find prime numbers
- ▶ We'll say that an element in an array is "alone" if there are values before and after it, and those values are different from it. Return a version of the given array where every instance of the given value which is alone is replaced by whichever value to its left or right is larger. (from coding bat)

# Topic 26

## Two Dimensional Arrays

"Computer Science is a science of abstraction  
-creating the right model for a problem and  
devising the appropriate mechanizable  
techniques to solve it."

-Alfred Aho and Jeffery Ullman



# 2D Arrays in Java

- Arrays with multiple dimensions may be declared and used

```
int[][] mat = new int[3][4];
```

- the number of pairs of square brackets indicates the dimension of the array.
- by convention, in a 2D array the first number indicates the row and the second the column

# Two Dimensional Arrays

	0	1	2	3	column
0	0	0	0	0	
1	0	0	0	0	
2	0	0	0	0	
row					

This is our abstract picture of the 2D array and treating it this way is acceptable.

(actual implementation is different)

```
mat[2][1] = 12;
```

# What is What?

```
int[][] mat = new int[10][12];
```

```
// mat is a reference to the whole 2d array
```

```
// mat[0] or mat[r] are references to a single row
```

```
// mat[0][1] or mat[r][c] are references to  
// single elements
```

```
// no way to refer to a single column
```



# 2D Array Problems

- ▶ Write a method to find the max value in a 2d array of ints
- ▶ Write a method that finds the sum of values in each column of a 2d array of doubles
- ▶ Write a method to print out the elements of a 2d array of ints in row order.
  - row 0, then row 1, then row 2 ...
- ▶ Write a method to print out the elements of a 2d array of ints in column order
  - column 0, then column 1, then column 2 ...

# Clicker 1

► What is output by the following code?

```
String[][] strTable = new String[5][8];  
System.out.print(strTable.length + " ");  
System.out.print(strTable[0].length + " ");  
System.out.print(strTable[2][3].length());
```

A. 40 0 0

B. 8 5 0

C. 5 8 0

D. 5 8 then a runtime error occurs

E. No output due to a syntax error.

# Use of Two Dimensional Arrays

- ▶ 2D arrays are often used when I need a table of data or want to represent things that have 2 dimensions.
- ▶ For instance an area of a simulation

# Example of using a 2D array

- ▶ Conway's Game of Life
  - a cellular automaton designed by John Conway, a mathematician
  - not really a game
  - a simulation
  - takes place on a 2d grid
  - each element of the grid is occupied or empty

# Simulation

- ▶ <http://www.cuug.ab.ca/dewara/life/life.html>
- ▶ Select pattern from menu
- ▶ Select region in large area with mouse by pressing the control key and left click at the same time
- ▶ Select the paste button

# Generation 0

	0	1	2	3	4	5
0	.	*	.	*	.	*
1	*	.	*	*	*	*
2	.	.	*	*	.	*
3	.	*	*	*	.	*

\* indicates occupied, . indicates empty

Or

	0	1	2	3	4	5
0						
1						
2						
3						

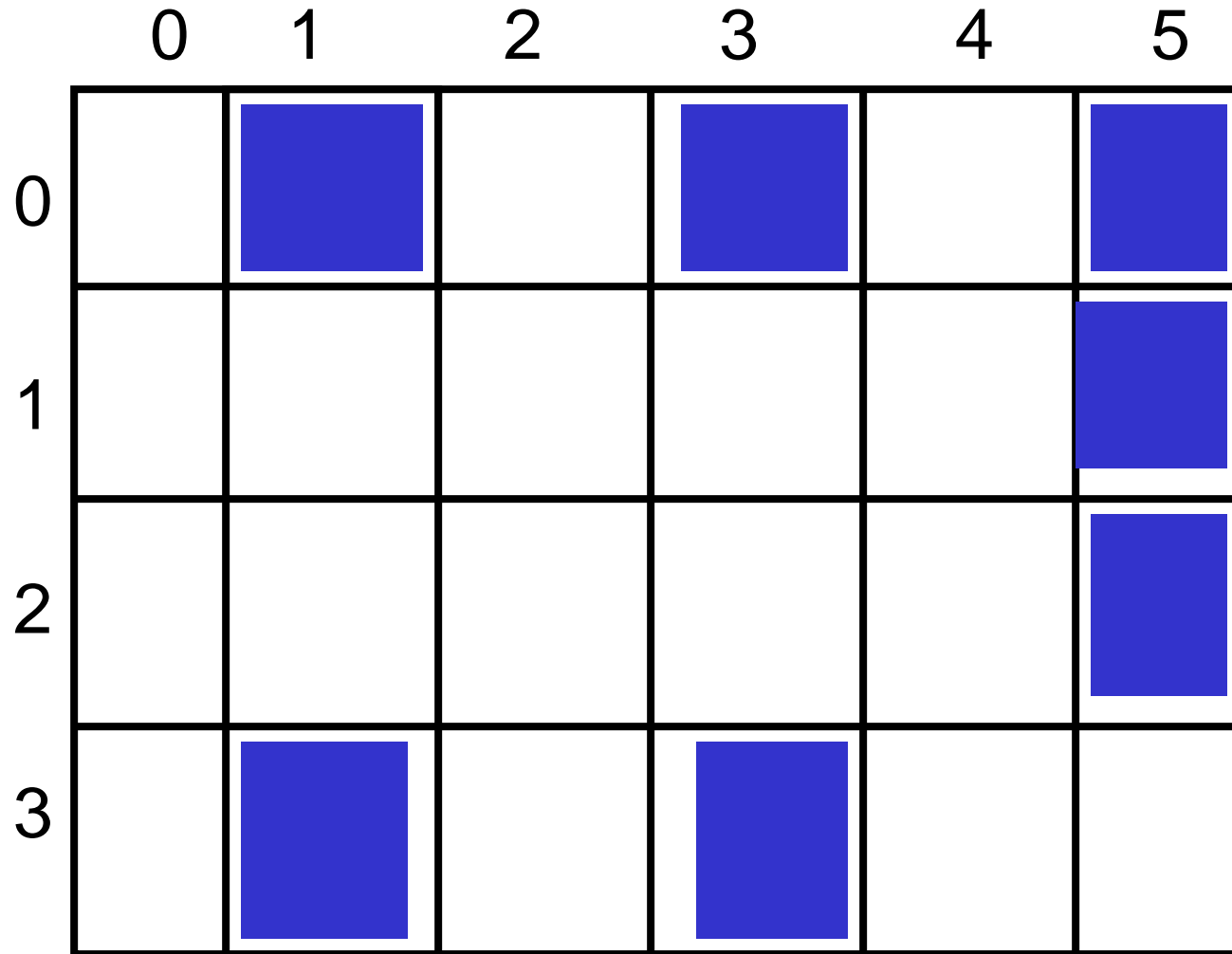
# Generation 1

	0	1	2	3	4	5
0	.	*	.	*	.	*
1	.	.	.	.	.	*
2	.	.	.	.	.	*
3	.	*	.	*	.	.

\* indicates occupied, . indicates empty



# Or , Generation 1



# Rules of the Game

- ▶ If a cell is occupied in this generation.
  - it survives if it has 2 or 3 neighbors in this generation
  - it dies if it has 0 or 1 neighbors in this generation
  - it dies if it has 4 or more neighbors in this generation
- ▶ If a cell is unoccupied in this generation.
  - there is a birth if it has exactly 3 neighboring cells that are occupied in this generation
- ▶ Neighboring cells are up, down, left, right, and diagonal. In general a cell has 8 neighboring cells

## Clicker 2

- ▶ Implement a program to run the simulation
- ▶ What data type do you want to use for the elements of the 2d array?

A. String

B. char

C. int

D. boolean

E. double

## Clicker 3

► Do you want to use a buffer zone on the edges?

A.No

B.Yes

# Topic 27

## classes and objects, state and behavior

"A 'class' is where we teach an 'object' to behave."

-Rich Pattis



# Object Oriented Programming

- ▶ "Object-oriented programming is a method of programming based on a hierarchy of classes, and well-defined and cooperating objects. "
- ▶ What is a class?
- ▶ "A class is a structure that defines the data and the methods to work on that data. When you write programs in the Java language, all program data is wrapped in a class, whether it is a class you write or a class you use from the Java platform API libraries."
  - a new data type

# Object Oriented Programming

- ▶ In other words break the problem up based on the things / data types that are part of the problem
- ▶ Not the only way
- ▶ One of many different kinds of strategies or *paradigms* for software development
  - functional, procedural, event driven, data flow, formal methods, agile or extreme, ...

# Clicker 1

▶ What kind of assignment handout do you prefer?

A. A long assignment handout

B. A short assignment handout

▶ Why?



# Example - Monopoly



If we had to start from scratch what classes would we need to create?

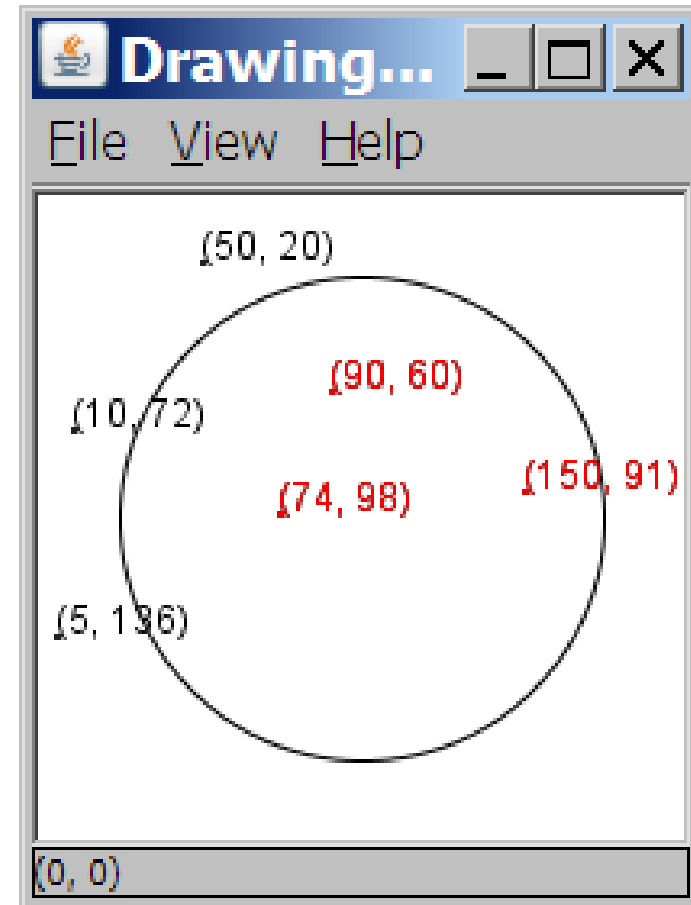
# A programming problem

- Given a file of cities' (x, y) coordinates, which begins with the number of cities:

```
6
50 20
90 60
10 72
74 98
5 136
150 91
```

- Write a program to draw the cities on a `DrawingPanel`, then a terrible event (zombie apocalypse, nuclear meltdown) that turns all cities red that are within a given radius:

```
Ground zero x: 100
Ground zero y: 100
Area of effect: 75
```



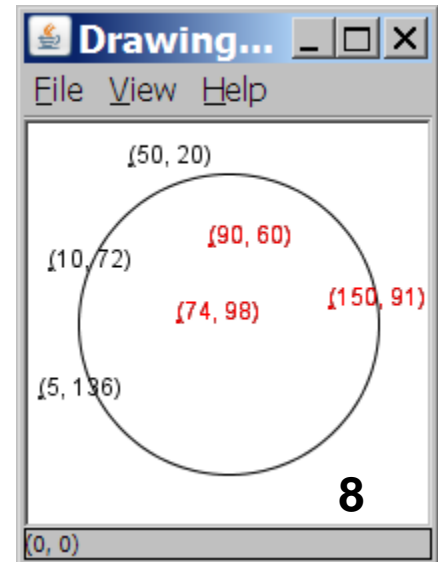
# A solution

```
Scanner input
    = new Scanner(new File("cities.txt"));
int cityCount = input.nextInt();
int[] xCoords = new int[cityCount];
int[] yCoords = new int[cityCount];
for (int i = 0; i < cityCount; i++) {
    xCoords[i] = input.nextInt();
    yCoords[i] = input.nextInt();
}
...
```

- **parallel arrays**: 2+ arrays with related data at same indexes.
  - Considered poor style. (Relationship exists in the programmer's mind, but not explicit in the program.)

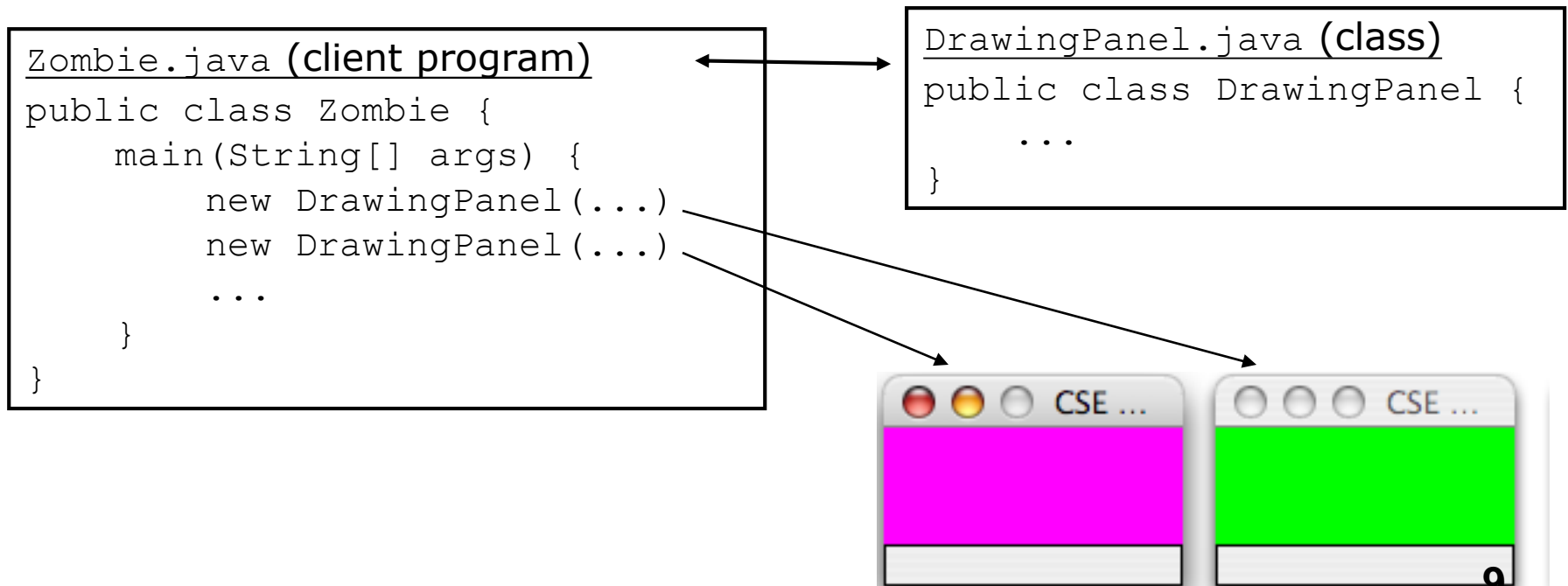
# Observations

- ▶ The data in this problem is a set of points.
- ▶ An alternative is to store them as `Point` objects.
  - A `Point` would store a city's x/y data.
  - We could compare distances between `Points` to see whether the terrible event affects a given city.
  - Each `Point` would know how to draw itself.
  - The driver program would be shorter and cleaner.



# Clients of objects

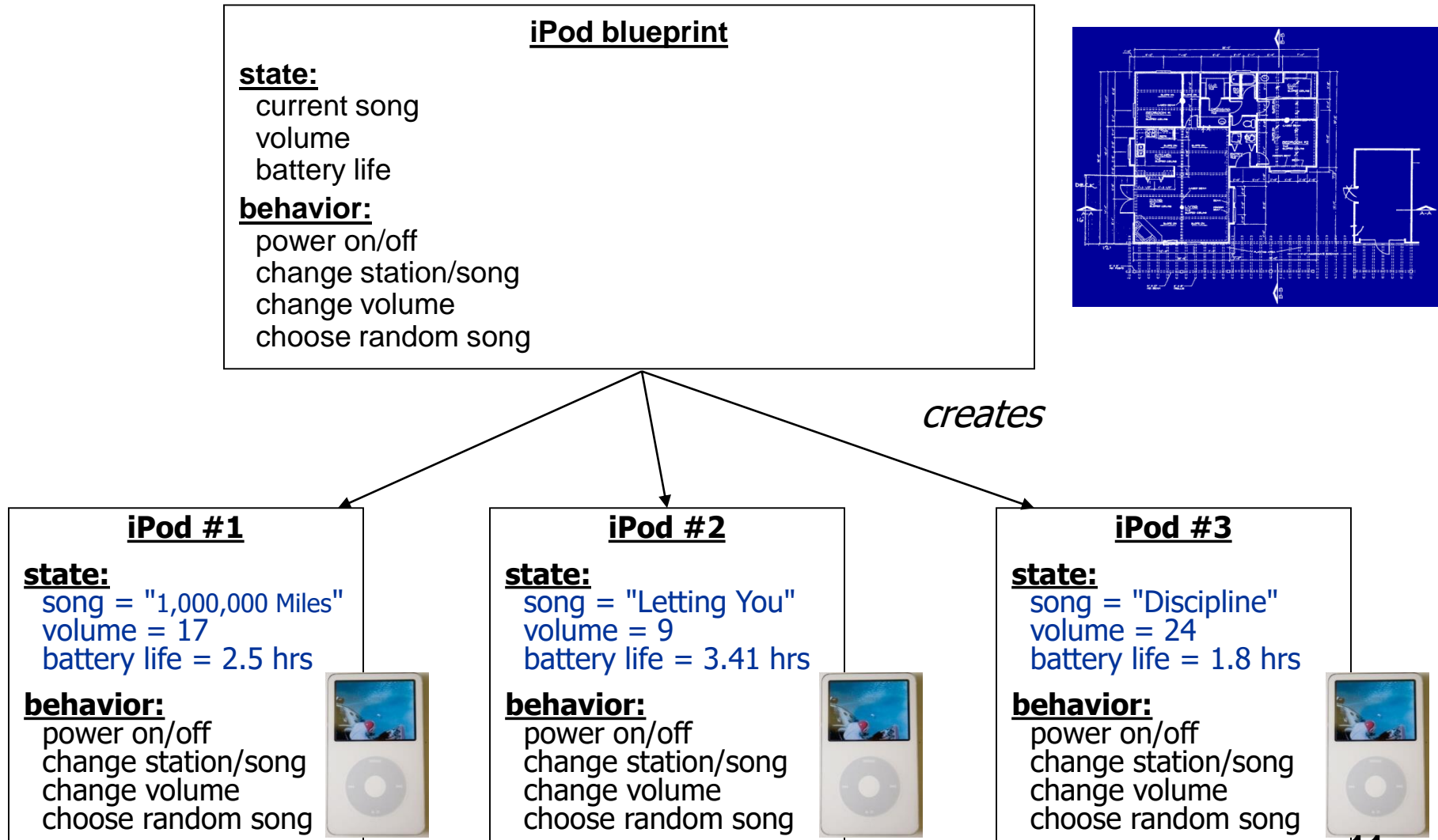
- **client program:** A program that uses objects.
  - **Example:** `Zombies` is a client of `DrawingPanel` and `Graphics`.



# Classes and objects

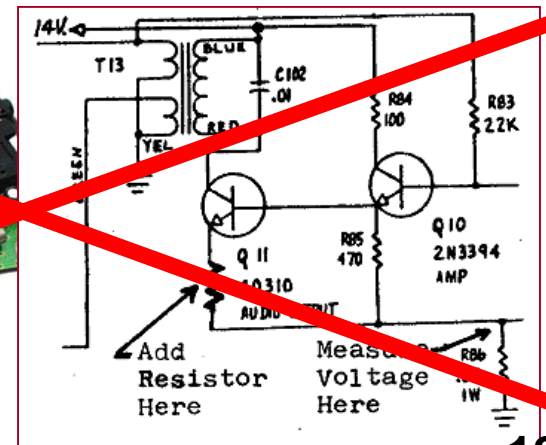
- ▶ **class:** A program entity that represents either:
  1. A program / module, or
  2. **A template for a new type of objects.**
  - The `DrawingPanel` class is a template for creating `DrawingPanel` objects.
  - **Other classes: `String`, `Random`, `Scanner`, `File`, ...**
- ▶ **object:** An entity that combines state and behavior.
  - **object-oriented programming (OOP):** Programs that perform their behavior as interactions between objects.

# Blueprint analogy



# Abstraction

- ▶ **abstraction:** A distancing between ideas and details.
  - We can use objects without knowing how they work.
- ▶ abstraction in an iPhone:
  - You understand its external behavior (buttons, screen).
  - You may not understand its inner details,  
***and you don't need to if you just want to use it.***





# Our task

- ▶ In the following slides, we will implement a `Point` class as a way of learning about defining classes.
  - We will define a type of objects named `Point`.
  - Each `Point` object will contain x/y data called **fields**.
  - Each `Point` object will contain behavior called **methods**.
  - **Client programs** will use the `Point` objects.

# Point objects (desired)

```
Point p1 = new Point(5, -2);
```

```
Point p2 = new Point(); // origin, (0, 0)
```

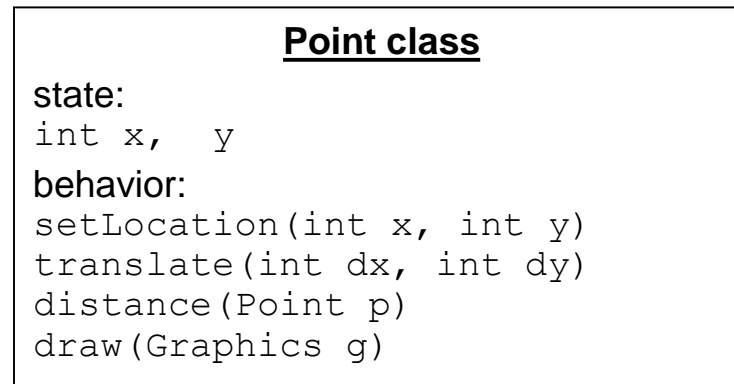
## ► Data in each `Point` object:

Field name	Description
<code>x</code>	the point's x-coordinate
<code>y</code>	the point's y-coordinate

## ► Methods in each `Point` object:

Method name	Description
<code>setLocation(<b>x</b>, <b>y</b>)</code>	sets the point's x and y to the given values
<code>translate(<b>dx</b>, <b>dy</b>)</code>	adjusts the point's x and y by the given amounts
<code>distance(<b>p</b>)</code>	how far away the point is from point <i>p</i>
<code>draw(<b>g</b>)</code>	displays the point on a drawing panel

# Point class as blueprint



**Point object #1**

state:  
x = 5, y = -2

behavior:  
setLocation(int x, int y)  
translate(int dx, int dy)  
distance(Point p)  
draw(Graphics g)

**Point object #2**

state:  
x = -245, y = 1897

behavior:  
setLocation(int x, int y)  
translate(int dx, int dy)  
distance(Point p)  
draw(Graphics g)

**Point object #3**

state:  
x = 18, y = 42

behavior:  
setLocation(int x, int y)  
translate(int dx, int dy)  
distance(Point p)  
draw(Graphics g)

- The class (blueprint) will describe how to create objects.
- Each object will contain its own data and methods.

**Clicker 2** What is output by the following code?

```
Point p1 = new Point();  
Point p2 = new Point();  
boolean b1 = (p1 == p2);  
System.out.print(b1);
```

- A. Syntax error
- B. Runtime error
- C. false
- D. true
- E. no output

Object state:  
Fields

# Point class, version 1

```
public class Point {  
    private int x;  
    private int y;  
}
```

- Save this code into a file named `Point.java`.
- ▶ The above code creates a new type named `Point`.
  - Each `Point` object contains two pieces of data:
    - an `int` named `x`, and
    - an `int` named `y`.
  - `Point` objects do not contain any behavior (yet).

# Fields

- ▶ **field**: A variable inside an object that is part of its state.
  - Each object has *its own copy* of each field.
- ▶ Declaration syntax:

**access\_modifier type name;**

- Example:

```
public class Student {  
    // each Student object has a name and  
    // gpa field (instance variable)  
    private String name;  
    private double gpa;  
}
```

# Accessing fields

- ▶ Other classes can access/modify an object's fields.
  - *depending on the access modifier*
  - access: **variable.field**
  - modify: **variable.field = value;**

## ▶ Example:

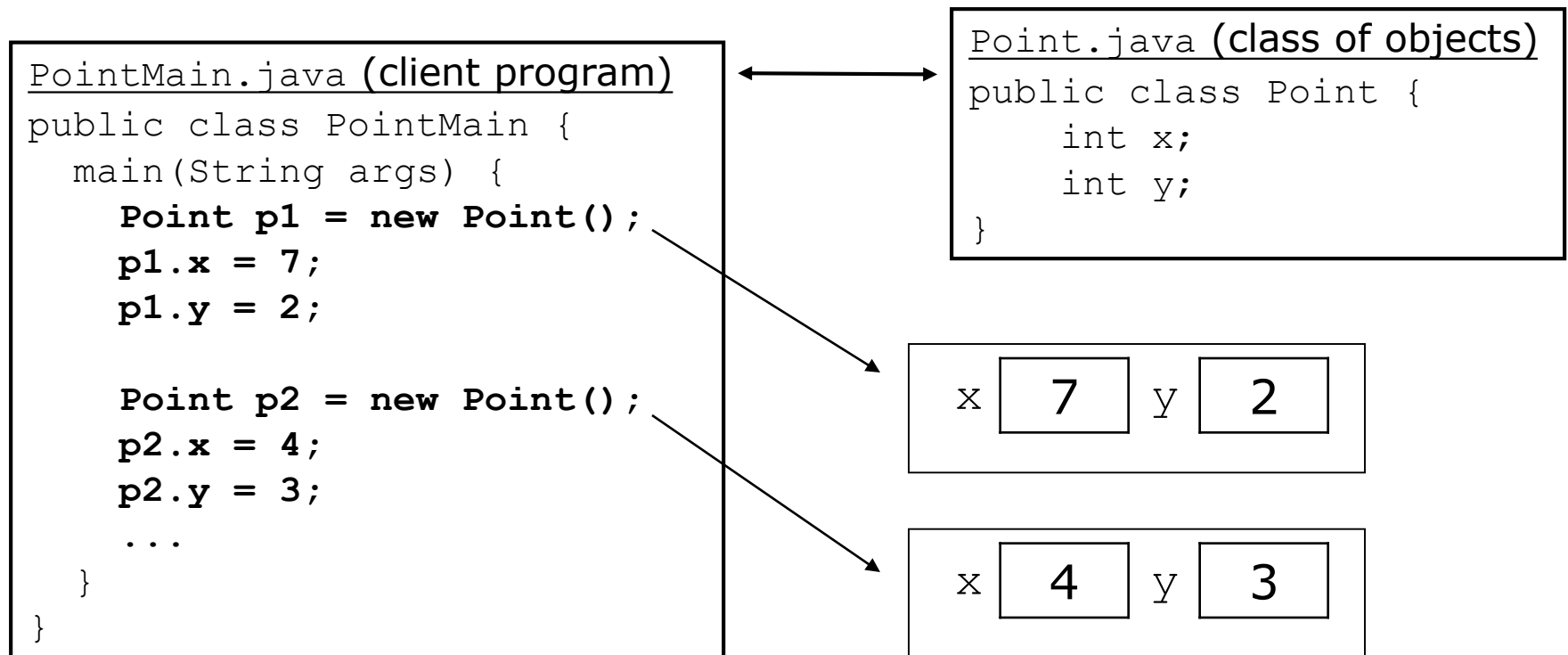
```
Point p1 = new Point();  
Point p2 = new Point();  
System.out.println("the x-coord is " + p1.x);  
p2.y = 13;
```

// access  
// modify



# A class and its client

- `Point.java` is not, by itself, a runnable program.
  - A class can be used by **client** programs.



# Object behavior: Methods

# Client code redundancy

- ▶ Suppose our client program wants to draw Point objects:

```
// draw each city
Point p1 = new Point();
p1.x = 15;
p1.y = 37;
g.fillOval(p1.x, p1.y, 3, 3);
g.drawString("(" + p1.x + ", " + p1.y + ")", p1.x, p1.y);
```

- ▶ To draw other points, the same code must be repeated.
  - We can remove this redundancy using a method.

# Eliminating redundancy, v1

- ▶ We can eliminate the redundancy with a static method:

```
// Draws the given point on the DrawingPanel.  
public static void draw(Point p, Graphics g) {  
    g.fillOval(p.x, p.y, 3, 3);  
    g.drawString("(" + p.x + ", " + p.y + ")", p.x, p.y);  
}
```

- ▶ `main` would call the method as follows:

```
draw(p1, g);
```

# Problems with static solution

- ▶ We are missing a major benefit of objects: code reuse.
  - Every program that draws `Points` would need a `draw` method.
- ▶ The syntax doesn't match how we're used to using objects.

```
draw(p1, g);    // static (bad)
```

- ▶ The point of classes is to combine state and behavior.
  - The `draw` behavior is closely related to a `Point`'s data.
  - The method belongs *inside* each `Point` object.

```
p1.draw(g);    // inside the object (better)
```

# Instance methods

- ▶ **instance method** (or **object method**): Exists inside each object of a class and gives behavior to each object.

```
public type name(parameters) {  
    statements;  
}
```

- same syntax as static methods, but without `static` keyword

Example:

```
public void shout() {  
    System.out.println("HELLO THERE!");  
}
```

# Instance method example

```
public class Point {  
    private int x;  
    private int y;  
  
    // Draws this Point object with the given pen.  
    public void draw(Graphics g) {  
        ...  
    }  
}
```

- ▶ The `draw` method no longer has a `Point p` parameter.
- ▶ How will the method know which point to draw?
  - How will the method access that point's x/y data?

# Point objects w/ method

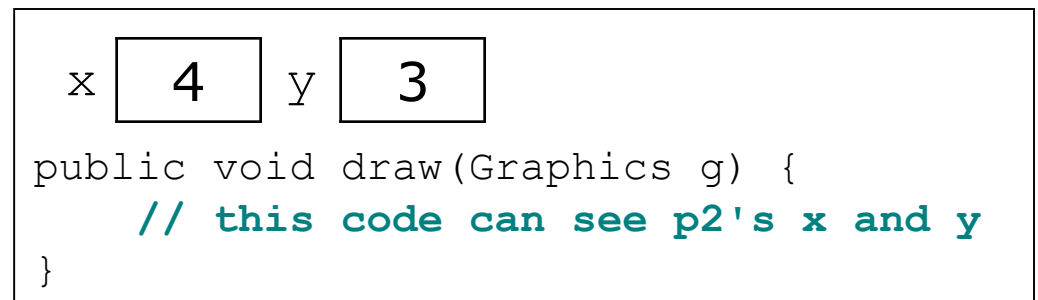
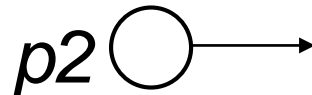
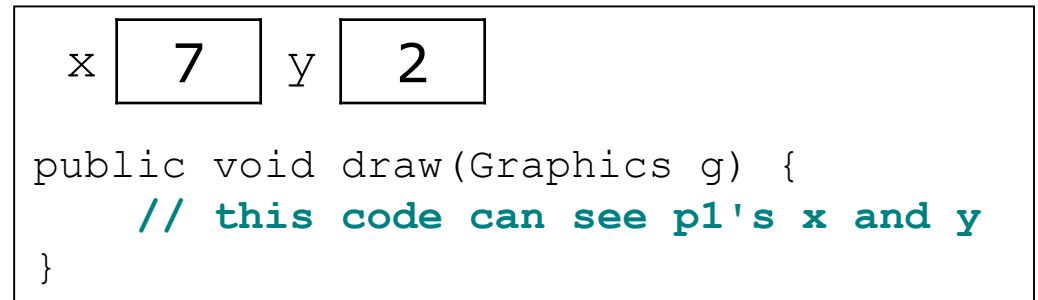
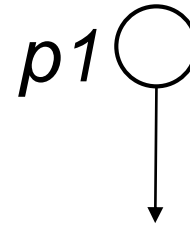
- Each `Point` object has its own copy of the `draw` method, which operates on that object's state:

```
Point p1 = new Point(7, 2);
```

```
Point p2 = new Point(4, 3);
```

```
p1.draw(g);
```

```
p2.draw(g);
```





# The implicit parameter

## ► implicit parameter:

The object on which an instance method is called.

- During the call `p1.draw(g)` ;  
the object referred to by `p1` is the implicit parameter.
- During the call `p2.draw(g)` ;  
the object referred to by `p2` is the implicit parameter.
- The instance method can refer to that object's fields.
  - We say that it executes in the *context* of a particular object.
  - `draw` can refer to the `x` and `y` of the object it was called on.

# Point class, version 2

```
public class Point {  
    int x;  
    int y;  
  
    // Changes the location of this Point object.  
    public void draw(Graphics g) {  
        g.fillOval(x, y, 3, 3);  
        g.drawString("(" + x + ", " + y + ")", x, y);  
    }  
}
```

- Each `Point` object contains a `draw` method that draws that point at its current `x/y` position.

# method questions

- ▶ Write a method `translate` that changes a `Point`'s location by a given  $dx$ ,  $dy$  amount.
- ▶ Write a method `distanceFromOrigin` that returns the distance between a `Point` and the origin,  $(0, 0)$ .

Use the formula:  $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$

- Modify the `Point` and client code to use these methods.

# Class method answers

```
public class Point {  
    int x;  
    int y;  
  
    public void translate(int dx, int dy) {  
        x = x + dx;  
        y = y + dy;  
    }  
  
    public double distanceFromOrigin() {  
        return Math.sqrt(x * x + y * y);  
    }  
}
```

# Topic 28

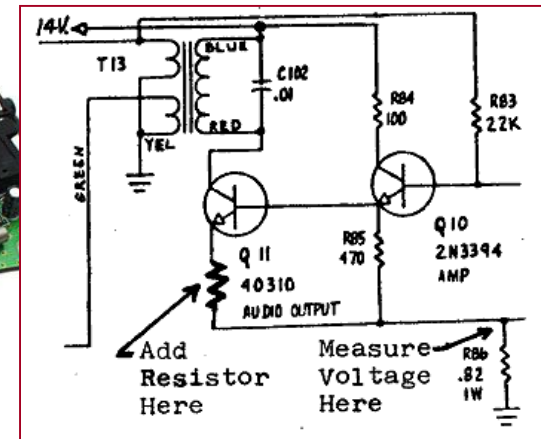
## classes and objects, part 2

# Encapsulation

► **encapsulation:** Hiding implementation details from clients.

– Encapsulation forces *abstraction*.

- separates external view (behavior) from internal view (state)
- protects the integrity of an object's data



# Private fields

*A field that cannot be accessed from outside the class*

**private** type name;

– Examples:

```
private int id;
```

```
private String name;
```

► Client code won't compile if it accesses private fields:

```
PointMain.java:11: x has private access in Point  
System.out.println(p1.x);
```

^

# Accessing private state

```
// A "read-only" access to the x field ("accessor")
public int getX() {
    return x;
}
```

```
// Allows clients to change the x field ("mutator")
public void setX(int newX) {
    x = newX;
}
```

- Client code will look more like this:

```
System.out.println(p1.getX()) ;
p1.setX(14) ;
```



# Point class, version 4

```
// A Point object represents an (x, y) location.
public class Point {
    private int x;
    private int y;

    public Point(int initialX, int initialY) {
        x = initialX;
        y = initialY;
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

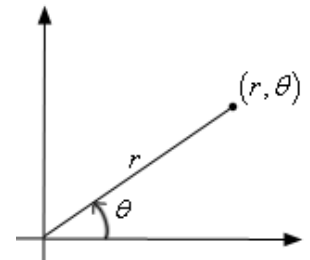
    public double distanceFromOrigin() {
        return Math.sqrt(x * x + y * y);
    }

    public void setLocation(int newX, int newY) {
        x = newX;
        y = newY;
    }

    public void translate(int dx, int dy) {
        setLocation(x + dx, y + dy);
    }
}
```

# Benefits of encapsulation

- ▶ Abstraction between object and clients
- ▶ Protects object from unwanted access
  - Example: Can't fraudulently increase an `Account`'s balance.
- ▶ Can change the class implementation later
  - Example: `Point` could be rewritten in polar coordinates  $(r, \theta)$  with the same methods.
- ▶ Can constrain objects' state (**invariants**)
  - Example: Only allow `Accounts` with non-negative balance.
  - Example: Only allow `Dates` with a month from 1-12.



# Clicker 1

- ▶ What is output by the following client code?
  - The code is not part of the Point class.

```
Point p1 = new Point(5, 10); // x, y  
p1.x = 12;  
System.out.println(p1.x);
```

- A. 0
- B. 5
- C. 12
- D. no output due to syntax error
- E. no output due to runtime error

The keyword `this`

**reading: 8.3**

# The `this` keyword

- ▶ **`this`** : Refers to the implicit parameter inside your class.

*(a variable that stores the object on which a method is called)*

- Refer to a field:        `this.field`
- Call a method:        `this.method (parameters) ;`
- One constructor can call another:        `this (parameters) ;`

# Variable shadowing

- **shadowing**: 2 variables with same name in same scope.
  - Normally illegal, except when one variable is a field.

```
public class Point {  
    private int x;  
    private int y;  
  
    ...  
  
    // this is legal  
    public void setLocation(int x, int y) {  
        ...  
    }  
}
```

- In most of the class, `x` and `y` refer to the fields.
- In `setLocation`, `x` and `y` refer to the method's parameters.

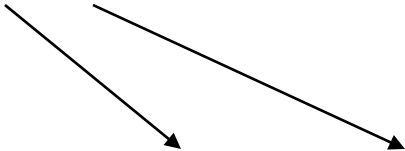
# Fixing shadowing

```
public class Point {  
    private int x;  
    private int y;  
  
    ...  
  
    public void setLocation(int x, int y)  
    {  
        this.x = x;  
        this.y = y;  
    }  
}
```

- Inside `setLocation`,
  - To refer to the data field `x`, say `this.x`
  - To refer to the parameter `x`, say `x`

# Calling another constructor

```
public class Point {  
    private int x;  
    private int y;  
  
    public Point() {  
        this(0, 0); // calls (x, y) constructor  
    }  
  
    public Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    ...  
}
```



- Avoids redundancy between constructors
- Only a constructor (not a method) can call another constructor



# Topic 29

## classes and objects, part 3

“And so, from Europe, we get things such as ... object-oriented analysis and design (a clever way of breaking up software programming instructions and data into small, reusable objects, based on certain abstraction principles and design hierarchies.)”

*-Michael A. Cusumano,  
The Business Of Software*



```
public static void cp(Point p) {  
    p.translate(2, 3); // add to x, y  
    p = new City(4, 7);  
}  
// client code of cp  
Point c1 = new Point(1, 2); // x, y  
cp(c1);  
System.out.println(c1);
```

A. (3, 5)

B. (1, 5)

C. (4, 7)

D. (6, 10)

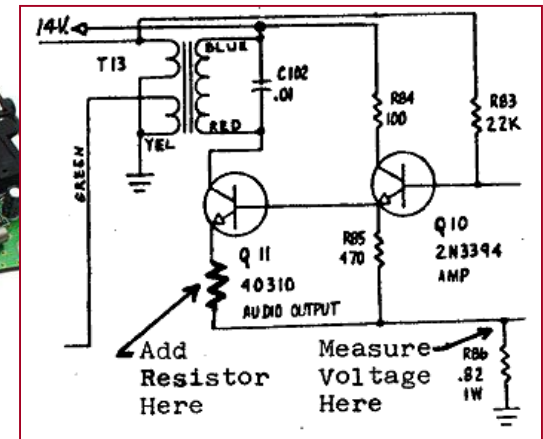
E. **error (syntax error or runtime error)** <sup>2</sup>

# Encapsulation

► **encapsulation:** Hiding implementation details from clients.

– Encapsulation forces *abstraction*.

- separates external view (behavior) from internal view (state)
- protects the integrity of an object's data



# Private fields

*A field that cannot be accessed from outside the class*

**private** type name;

– Examples:

```
private int id;
```

```
private String name;
```

► Client code won't compile if it accesses private fields:

```
PointMain.java:11: x has private access in Point
System.out.println(p1.x);
```

^

# Accessing private state

```
// A "read-only" access to the x field ("accessor")
public int getX() {
    return x;
}
```

```
// Allows clients to change the x field ("mutator")
public void setX(int newX) {
    x = newX;
}
```

- Client code will look more like this:

```
System.out.println(p1.getX()) ;
p1.setX(14) ;
```

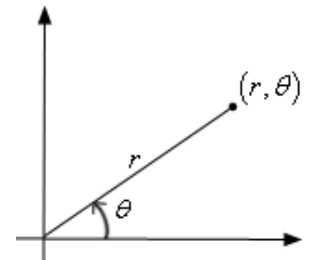
# Point class, version 4

// A Point object represents an (x, y) location.

```
public class Point {  
    private int x;  
    private int y;  
  
    public Point(int initialX, int initialY) {  
        x = initialX;  
        y = initialY;  
    }  
  
    public int getX() {  
        return x;  
    }  
  
    public int getY() {  
        return y;  
    }  
  
    public double distanceFromOrigin() {  
        return Math.sqrt(x * x + y * y);  
    }  
  
    public void setLocation(int newX, int newY) {  
        x = newX;  
        y = newY;  
    }  
  
    public void translate(int dx, int dy) {  
        setLocation(x + dx, y + dy);  
    }  
}
```

# Benefits of encapsulation

- ▶ Abstraction between object and clients
- ▶ Protects object from unwanted access
  - Example: Can't fraudulently increase an `Account`'s balance.
- ▶ Can change the class implementation later
  - Example: `Point` could be rewritten in polar coordinates  $(r, \theta)$  with the same methods.
- ▶ Can constrain objects' state (**invariants**)
  - Example: Only allow `Accounts` with non-negative balance.
  - Example: Only allow `Dates` with a month from 1-12.



The keyword `this`

**reading: 8.3**



# The `this` keyword

- ▶ **`this`** : Refers to the implicit parameter inside your class.

*(a variable that stores the object on which a method is called)*

- Refer to a field:        `this.field`
- Call a method:        `this.method (parameters) ;`
- One constructor can call another:        `this (parameters) ;`

# Variable shadowing

- **shadowing**: 2 variables with same name in same scope.
  - Normally illegal, except when one variable is a field.

```
public class Point {  
    private int x;  
    private int y;  
  
    ...  
  
    // this is legal  
    public void setLocation(int x, int y) {  
        ...  
    }  
}
```

- In most of the class, `x` and `y` refer to the fields.
- In `setLocation`, `x` and `y` refer to the method's parameters.

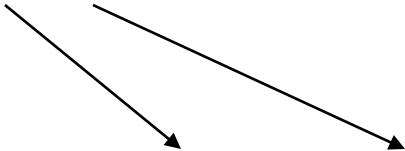
# Fixing shadowing

```
public class Point {  
    private int x;  
    private int y;  
  
    ...  
  
    public void setLocation(int x, int y)  
    {  
        this.x = x;  
        this.y = y;  
    }  
}
```

- Inside `setLocation`,
  - To refer to the data field `x`, say `this.x`
  - To refer to the parameter `x`, say `x`

# Calling another constructor

```
public class Point {  
    private int x;  
    private int y;  
  
    public Point() {  
        this(0, 0); // calls (x, y) constructor  
    }  
  
    public Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    ...  
}
```



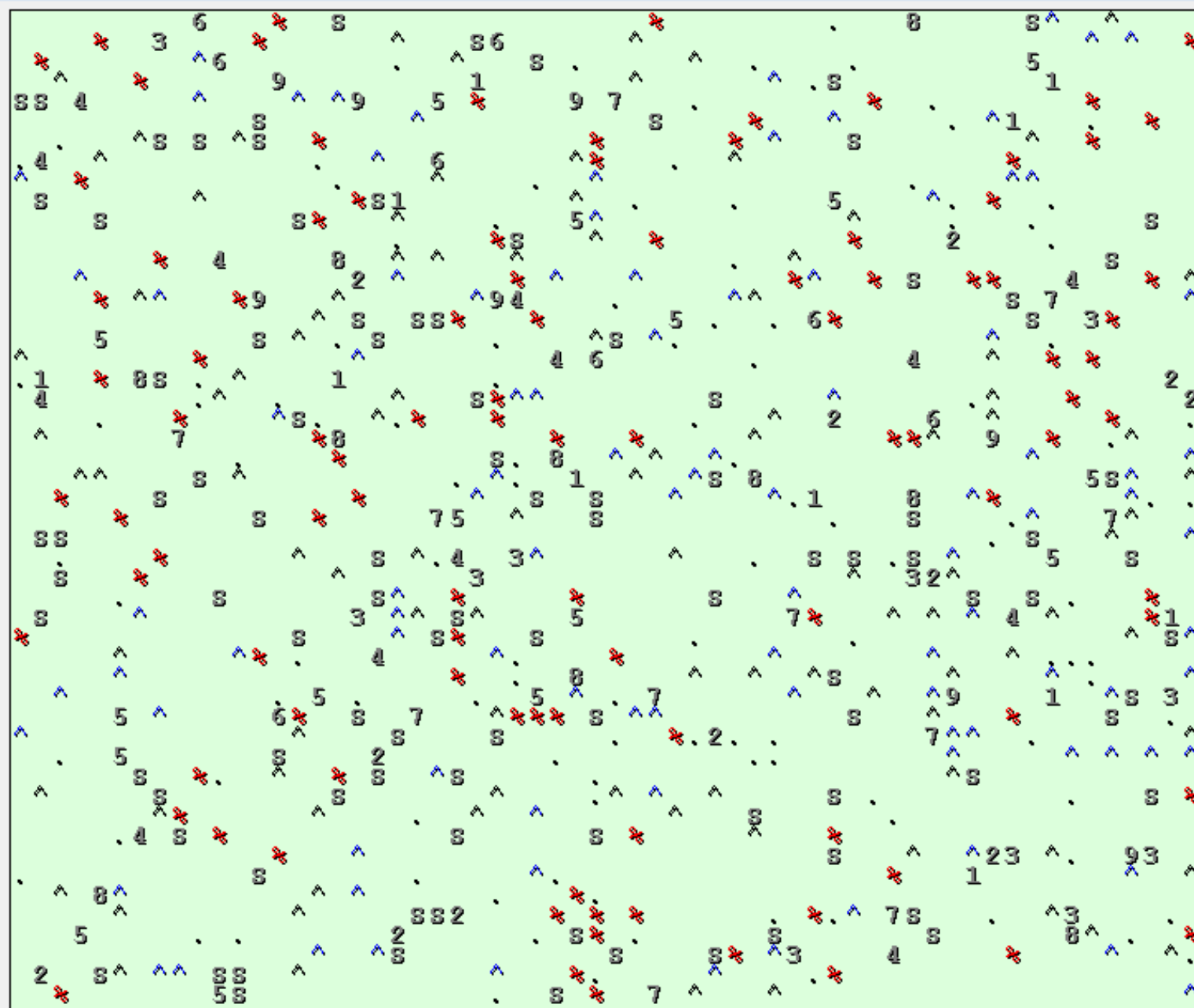
- Avoids redundancy between constructors
- Only a constructor (not a method) can call another constructor

# Assignment 11: Critters

## **HW11 Assignment Overview**

# Critters

- A simulation world with animal objects with behavior:
  - `fight` animal fighting
  - `getColor` color to display
  - `getMove` movement
  - `toString` letter to display
  - `eat` eat food?
- You must implement:
  - Ant
  - Bird
  - Vulture
  - Hippo
  - Longhorn (Wins Overall and Creative)

**Ant**

100 alive (- 0)  
+ 0 kills  
+ 0 food  
=100 TOTAL

**Bird**

100 alive (- 0)  
+ 0 kills  
+ 0 food  
=100 TOTAL

**Hippo**

100 alive (- 0)  
+ 0 kills  
+ 0 food  
=100 TOTAL

**Stone**

100 alive (- 0)  
+ 0 kills  
+ 0 food  
=100 TOTAL

**Vulture**

100 alive (- 0)  
+ 0 kills  
+ 0 food  
=100 TOTAL

Speed:



0 moves

Go

Stop

Tick

Reset



# How the simulator works

- When you press "Go", the simulator enters a loop:
  - move each animal once (`getMove`), in random order
  - if the animal has moved onto an occupied square, `fight`!
- Key concept: The simulator is in control, NOT your animal.
  - Example: `getMove` can return only one move at a time.  
`getMove` can't use loops to return a sequence of moves.
    - It wouldn't be fair to let one animal make many moves in one turn!
  - Your animal must keep state (as fields, instance variables) so that it can make a single move, and know what moves to make later.



# Scoring

- Score for each species:
- For all animals of that species
- Number of animals alive
- Number of fights won
- Pieces of food eaten

# Food

- Simulator places food randomly around world
- Eating food increases score for species, but ...
- Critters sleep after eating
  - simulator (CriticMain) handles this
- A Critter that gets in a fight while sleeping always loses
  - simulator handles this



# Mating

- Two Critters of same species next to each other mate and produce a baby Critter
- Simulator handles this
- Critters not asked if they want to mate
- Critters vulnerable while mating (heart graphic indicates mating)
  - automatically lose fight
- The Simulator handles all of this
  - You don't write any code to deal with mating

# Critter Class

```
public abstract class Critter {  
    public boolean eat() {  
        return false;  
    }  
    public Attack fight(String opponent) {  
        return Attack.FORFEIT;  
    }  
    public Color getColor() {  
        return Color.BLACK;  
    }  
    public Direction getMove() {  
        return Direction.CENTER;  
    }  
    public String toString() {  
        return "?";  
    }  
}
```

# Enums

- Critter class has two ***nested*** Enums for Direction of movement and how to fight

```
// constants for directions
public static enum Direction {
    NORTH, SOUTH, EAST, WEST, CENTER
};
```

```
// constants for fighting
public static enum Attack {
    ROAR, POUNCE, SCRATCH, FORFEIT
};
```



# Nested Enums

- To access a Direction or Attack a class external to Critter would use the following syntax:
- Critter.Direction.NORTH
- Critter.Attack.POUNCE
- Classes that are descendants of Critter (like the ones you implement) do not have to use the Critter.
  - it is implicit
- Direction.SOUTH, Attack.ROAR

# A Critter class

```
public class name extends Critter {  
    ...  
}
```

- `extends Critter` tells the simulator your class is a critter
- override methods from Critter based on Critter spec
- Critter has a number of methods not required by the 4 simple Critter classes (Ant, Bird, Vulture, Hippo)
- ... but you should use them to create an interesting and successful Longhorn



# Critter exercise: Stone

- Write a critter class `Stone`(the dumbest of all critters):

Method	Behavior
constructor	<code>public Stone()</code>
fight	<b>Always</b> <code>Attack.ROAR</code>
getColor	<b>Always</b> <code>Color.GRAY</code>
getMove	<b>Always</b> <code>Direction.CENTER</code>
toString	<code>"S"</code>
eat	<b>Always</b> <code>false</code>





# Ideas for state

- You must not only have the right state, but update that state properly when relevant actions occur.
- Counting is helpful:
  - How many total moves has this animal made?
  - How many times has it fought?
- Remembering recent actions in fields is helpful:
  - Which direction did the animal move last?
    - How many times has it moved that way?

# Clicker 1

- We want to implement a Critter that moves West until it is in a fight. After the fight the Critter moves East until it is in another fight. Each time the Critter is in a fight it shifts its direction for the next move from West to East or East to West.
- Will the move method have a loop?
  - A. No
  - B. Yes
  - C. Maybe



# Keeping state

- How can a critter move west until it fights?

```
public Direction getMove() {  
    while (animal has not fought) {  
        return Direction.EAST;  
    }  
    while (animal has not fought a second time) {  
        return Direction.EAST;  
    }  
}
```

```
private int fights;    // total times Critter has fought  
public Direction getMove() {  
    if (fights % 2 == 0) {  
        return Direction.WEST;  
    } else {  
        return Direction.EAST;  
    }  
}
```

# Testing critters

- Use the MiniMain to create String based on actions and print those out
- Focus on one specific critter of one specific type
  - Only spawn 1 of each animal, for debugging
- Make sure your fields update properly
  - Use `println` statements to see field values
- Look at the behavior one step at a time
  - Use "Tick" rather than "Go"



# Critter exercise: Snake

Method	Behavior
constructor	<code>public Snake(boolean northSnake)</code>
fight	alternates between SCRATCH and POUNCE
getColor	Yellow
getMove	north bound snakes: 5 steps north, pause 5 ticks, 5 steps north, pause 5 ticks, ...  otherwise: 5 steps west, pause 5 ticks, 5 steps west, pause 5 ticks, ...
eat	always eats
toString	"K"



# Determining necessary fields

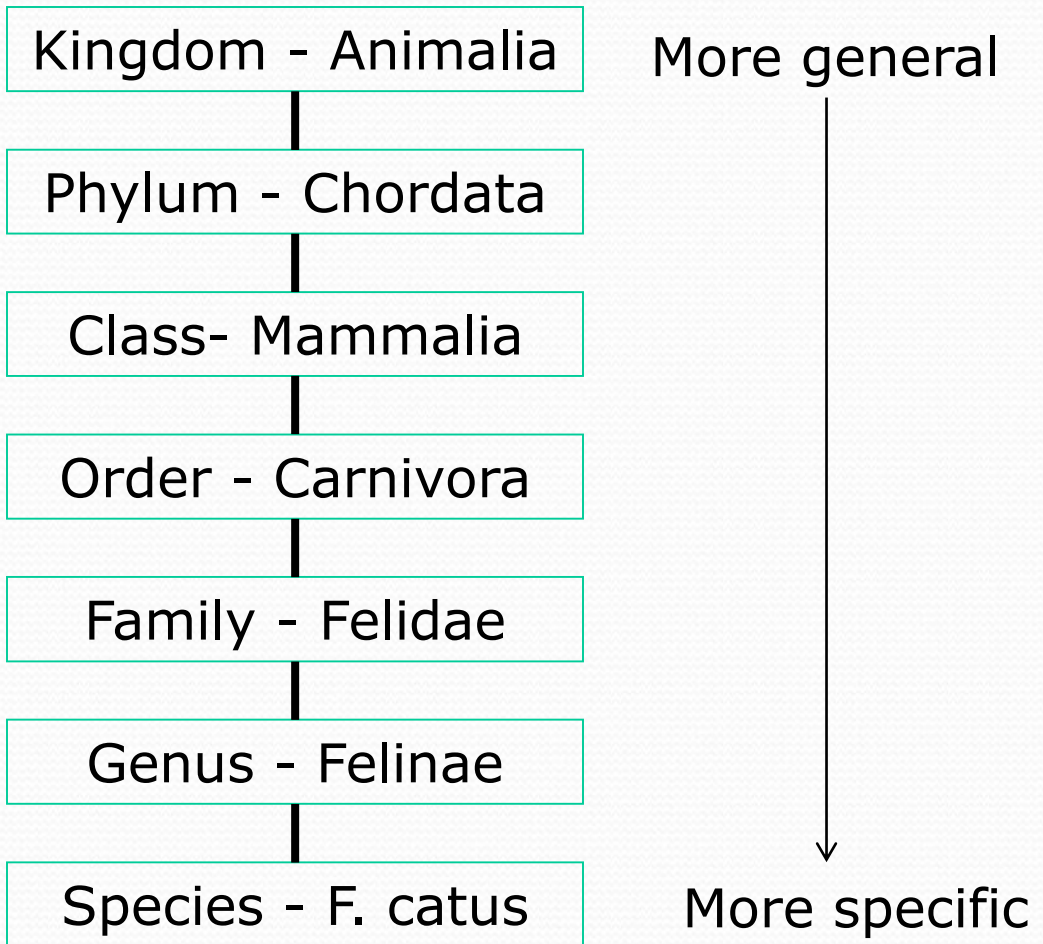
- Information required to decide what move to make?
  - Direction to go in
  - Length of current cycle
  - Number of moves made in current cycle
- Remembering things you've done in the past:
  - an `int` counter?
  - a `boolean` flag?



# Topic 31 - inheritance

# Hierarchies

- Hierarchies used to organize information





# Hierarchies

- Object oriented languages provide a mechanism to create hierarchies among data types in a program and in code libraries
- Used for organization, modeling the problem, and to avoid redundant code
- When a new data type is a specialization or variation on an existing data type use inheritance to capture the relationship and avoid redundancy of code

# Inheritance in Practice

1. extends keyword
2. inheritance of instance methods
3. inheritance of instance variables
4. object initialization and constructors
5. calling a parent constructor with **super()**
6. overriding methods
7. partial overriding, **super.parentMethod()**
8. inheritance requirement in Java
9. the **Object** class
10. inheritance hierarchies



# Pretty Stone

- Implement a Pretty Stone class
- Same as a Stone except alternates Color every  $(N + 1)$  times based on an int parameter to the constructor
- If parameter is  $[0..2]$  alternates between BLUE and RED
- If parameter  $[3..5]$  alternates between GREEN and YELLOW
- If parameter  $> 5$  alternates between MAGENTA and ORANGE
- Pretty stones always return true when asked to eat.

# Rolling Stone

- Implement a Rolling Stone class
- Same as a Pretty Stone ...
- except when a Rolling Stone is created it picks a random number of turns based on the int sent to the constructor.  
0 -> 0-99, 1 -> 100-199, 2 -> 200-299
- Stays still until asked to move that number of times, then moves North.
- fights: if not moving same as pretty stone, otherwise a random attack

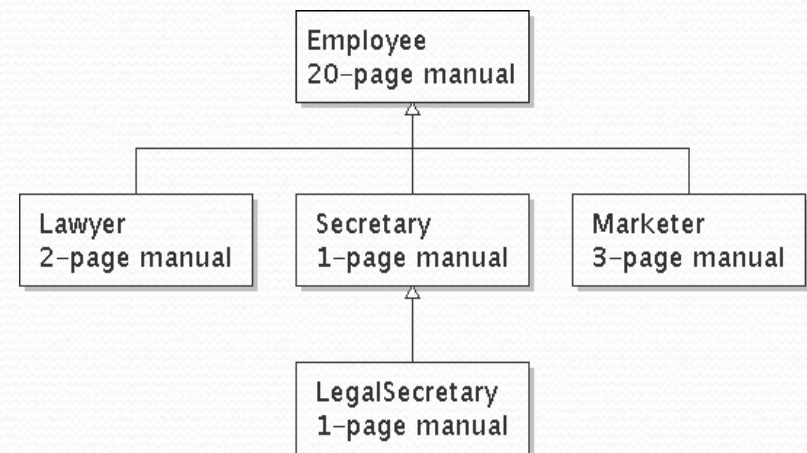


# Another Example

Following slides contain another example of an inheritance hierarchy and Java syntax for implementing it.

# Law firm employee analogy

- common rules: hours, vacation, benefits, regulations ...
  - all employees attend a common orientation to learn general company rules
  - each employee receives a 20-page manual of common rules
- each subdivision also has specific rules:
  - employee receives a smaller (1-3 page) manual of these rules
  - smaller manual adds some new rules and also changes some rules from the large manual





# Employee regulations

- Consider the following employee regulations:
  - Employees work 40 hours / week.
  - Employees make \$40,000 per year, except legal secretaries who make \$5,000 extra per year (\$45,000 total), and marketers who make \$10,000 extra per year (\$50,000 total).
  - Employees have 2 weeks of paid vacation leave per year, except lawyers who get an extra week (a total of 3).
  - Employees should use a yellow form to apply for leave, except for lawyers who use a pink form.
- Each type of employee has some unique behavior:
  - Lawyers know how to sue.
  - Marketers know how to advertise.
  - Secretaries know how to take dictation.
  - Legal secretaries know how to prepare legal documents.

# An Employee class

**// A class to represent employees in general (20-page manual).**

```
public class Employee {  
    public int getHours() {  
        return 40;           // works 40 hours / week  
    }  
  
    public double getSalary() {  
        return 40000.0;      // $40,000.00 / year  
    }  
  
    public int getVacationDays() {  
        return 10;           // 2 weeks' paid vacation  
    }  
  
    public String getVacationForm() {  
        return "yellow";     // use the yellow form  
    }  
}
```

- **Exercise:** Implement class `Secretary`, based on the previous employee regulations. (Secretaries can take dictation.)



# Redundant Secretary class

**// A redundant class to represent secretaries.**

```
public class Secretary {  
    public int getHours() {  
        return 40;           // works 40 hours / week  
    }  
  
    public double getSalary() {  
        return 40000.0;      // $40,000.00 / year  
    }  
  
    public int getVacationDays() {  
        return 10;          // 2 weeks' paid vacation  
    }  
  
    public String getVacationForm() {  
        return "yellow";     // use the yellow form  
    }  
  
    public void takeDictation(String text) {  
        System.out.println("Taking dictation of text: " + text);  
    }  
}
```

# Desire for code-sharing

- `takeDictation` is the only unique behavior in `Secretary`.
- We'd like to be able to say:

*// A class to represent secretaries.*

```
public class Secretary {
```

***<copy all the contents from the Employee class>***

```
    public void takeDictation(String text) {
```

```
        System.out.println("Taking dictation of text: " + text);
```

```
    }
```

```
}
```



# Inheritance

- **inheritance:** A way to form new classes based on existing classes, taking on their attributes/behavior.
  - a way to group related classes
  - a way to share code between two or more classes
- One class can *extend* another, absorbing its data/behavior.
  - **superclass:** The parent class that is being extended.
  - **subclass:** The child class that extends the superclass and inherits its behavior.
    - Subclass gets a copy of every field and method from superclass

# Inheritance syntax

```
public class <name> extends <superclass> {
```

- Example:

```
public class Secretary extends Employee {  
    ...  
}
```

- By extending `Employee`, each `Secretary` object now:
  - receives a `getHours`, `getSalary`, `getVacationDays`, and `getVacationForm` method automatically
  - can be treated as an `Employee` by client code (seen later)



# Improved Secretary code

// A class to represent secretaries.

```
public class Secretary extends Employee {  
    public void takeDictation(String text) {  
        System.out.println("Taking dictation of text: " + text);  
    }  
}
```

- Now we only write the parts unique to each type.
  - Secretary **inherits** getHours, getSalary, getVacationDays, and getVacationForm **methods** from Employee.
  - Secretary **adds** the takeDictation **method**.

# Implementing Lawyer

- Consider the following lawyer regulations:
  - Lawyers who get an extra week of paid vacation (a total of 3).
  - Lawyers use a pink form when applying for vacation leave.
  - Lawyers have some unique behavior: they know how to sue.
- Problem: We want lawyers to inherit *most* behavior from employee, but we want to replace parts with new behavior.



# Lawyer class

```
// A class to represent lawyers.
public class Lawyer extends Employee {
    // overrides getVacationForm from Employee class
    public String getVacationForm() {
        return "pink";
    }

    // overrides getVacationDays from Employee class
    public int getVacationDays() {
        return 15;           // 3 weeks vacation
    }

    public void sue() {
        System.out.println("I'll see you in court!");
    }
}
```

- Exercise: Complete the `Marketer` class. Marketers make \$10,000 extra (\$50,000 total) and know how to advertise.

# Marketer class

**// A class to represent marketers.**

```
public class Marketer extends Employee {  
    public void advertise() {  
        System.out.println("Act now while supplies last!");  
    }  
  
    public double getSalary() {  
        return 50000.0;        // $50,000.00 / year  
    }  
}
```



# Levels of inheritance

- Multiple levels of inheritance in a hierarchy are allowed.
  - Example: A legal secretary is the same as a regular secretary but makes more money (\$5,000 more) and can file legal briefs.

```
public class LegalSecretary extends Secretary {  
    ...  
}
```

- Exercise: Complete the `LegalSecretary` class.

# LegalSecretary class

**// A class to represent legal secretaries.**

```
public class LegalSecretary extends Secretary {  
    public void fileLegalBriefs() {  
        System.out.println("I could file all day!");  
    }  
  
    public double getSalary() {  
        return 45000.0;        // $45,000.00 / year  
    }  
}
```

# Changes to common behavior

- Imagine a company-wide change affecting all employees.

Example: Everyone is given a \$10,000 raise due to inflation.

- The base employee salary is now \$50,000.
  - Legal secretaries now make \$55,000.
  - Marketers now make \$60,000.
- We must modify our code to reflect this policy change.



# Modifying the superclass

```
// A class to represent employees in general (20-page manual).
public class Employee {
    public int getHours() {
        return 40;                // works 40 hours / week
    }

    public double getSalary() {
        return 50000.0;           // $50,000.00 / year
    }

    ...
}
```

- Are we finished?
- The `Employee` subclasses are still incorrect.
  - They have overridden `getSalary` to return other values.

# An unsatisfactory solution

```
public class LegalSecretary extends Secretary {  
    public double getSalary() {  
        return 55000.0;  
    }  
    ...  
}  
  
public class Marketer extends Employee {  
    public double getSalary() {  
        return 60000.0;  
    }  
    ...  
}
```

- Problem: The subclasses' salaries are based on the Employee salary, but the `getSalary` code does not reflect this.



# Calling overridden methods

- Subclasses can call overridden methods with `super`

`super.<method> (<parameters>)`

- Example:

```
public class LegalSecretary extends Secretary {  
    public double getSalary() {  
        double baseSalary = super.getSalary();  
        return baseSalary + 5000.0;  
    }  
    ...  
}
```

- Exercise: Modify `Lawyer` and `Marketer` to use `super`.

# Improved subclasses

```
public class Lawyer extends Employee {
    public String getVacationForm() {
        return "pink";
    }

    public int getVacationDays() {
        return super.getVacationDays() + 5;
    }

    public void sue() {
        System.out.println("I'll see you in court!");
    }
}

public class Marketer extends Employee {
    public void advertise() {
        System.out.println("Act now while supplies last!");
    }

    public double getSalary() {
        return super.getSalary() + 10000.0;
    }
}
```



Given the Employee class  
to the right what is output  
by the following code?

```
Employee e1;  
e1 = new Employee("#1");  
String str;  
str = e1.toString();  
System.out.println(str);
```

- A. #1
- B. "#1"
- C. Output varies each time.
- D. Syntax error
- E. Runtime error

```
// A class to represent employees  
public class Employee {  
  
    private String id;  
  
    public Employee(String id) {  
        this.id = id;  
    }  
  
    public int getHours() {  
        return 40;  
    }  
  
    public double getSalary() {  
        return 40000.0;  
    }  
  
    public int getVacationDays() {  
        return 10;  
    }  
  
    public String getVacationForm() {  
        return "yellow";  
    }  
}
```



# Topic 32 - Polymorphism

# Clicker 1

- What is output by the following code?

```
Critter c1 = new Hippo(7);  
System.out.print(c1.toString());
```

- A. 7
- B. ?
- C. null
- D. No output due to a syntax error
- E. No output due to a runtime error



# Polymorphism

- **polymorphism**: Ability for the same code to be used with different types of objects and behave differently with each.
  - `System.out.println` can print any type of object.
    - Each one displays in its own way on the console.
  - `CritterMain` can interact with any type of critter.
    - Each one moves, fights, etc. in its own way.

# Coding with polymorphism

- **A variable of type *T* can refer to an object of any subclass of *T*.**

```
Critter c1 = new Hippo(7);
```

- You can call any methods from the `Critter` class on `c1`.
- When a method is called on `c1`, it behaves as a `Hippo`.

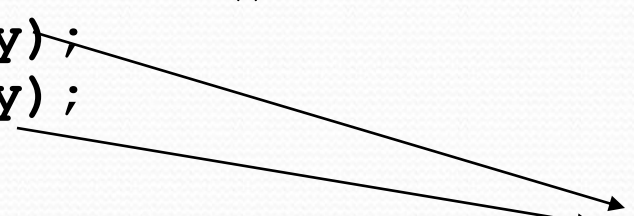
```
System.out.println(c1.getColor);           // GRAY  
System.out.println(c1.toString());         // 7
```



# Polymorphism and parameters

- You can pass any subtype of a parameter's type.

```
public class CriiterMain {  
    public static void main(String[] args) {  
        Hippo henry = new Hippo(7);  
        Bird angry = new Bird();  
        printInfo(henry);  
        printInfo(angry);  
    }  
  
    public static void printInfo(Critter crit) {  
        System.out.println(" eat?: " + crit.eat());  
        System.out.println(" fight: " + crit.fight("?"));  
        System.out.println(" move: " + crit.getMove());  
        System.out.println();  
    }  
}
```

A diagram consisting of two arrows. The first arrow originates from the **printInfo(henry);** line in the main method and points to the parameter **Critter** in the printInfo method signature. The second arrow originates from the **printInfo(angry);** line and points to the parameter **crit** in the same signature. This illustrates that both **Hippo** and **Bird** subtypes are being passed to a method that expects a **Critter** parameter.

OUTPUT???

# Polymorphism and arrays

- Arrays of superclass types can store any subtype as elements.

```
public class CritterMain2 {  
    public static void main(String[] args) {  
        Critter[] crits = { new Bird(),    new Vulture(),  
                           new Hippo(7), new Ant(true) };  
  
        for (int i = 0; i < crits.length; i++) {  
            System.out.println(" color: " + crits[i].getColor());  
            System.out.println("  move: " + crits[i].getMove());  
            System.out.println();  
        }  
    }  
}
```

Output:



# A polymorphism problem

```
public class Foo {  
    public void method1() {  
        System.out.println("foo 1");  
    }  
    public void method2() {  
        System.out.println("foo 2");  
    }  
    public String toString() {  
        return "foo";  
    }  
}  
  
public class Bar extends Foo {  
    public void method2() {  
        System.out.println("bar 2");  
    }  
}
```

```
public class Baz extends Foo {  
    public void method1() {  
        System.out.println("baz 1");  
    }  
    public String toString() {  
        return "baz";  
    }  
}  
  
public class Mumble extends Baz {  
    public void method2() {  
        System.out.println("mumble 2");  
    }  
}
```

# A polymorphism problem

```
}
```

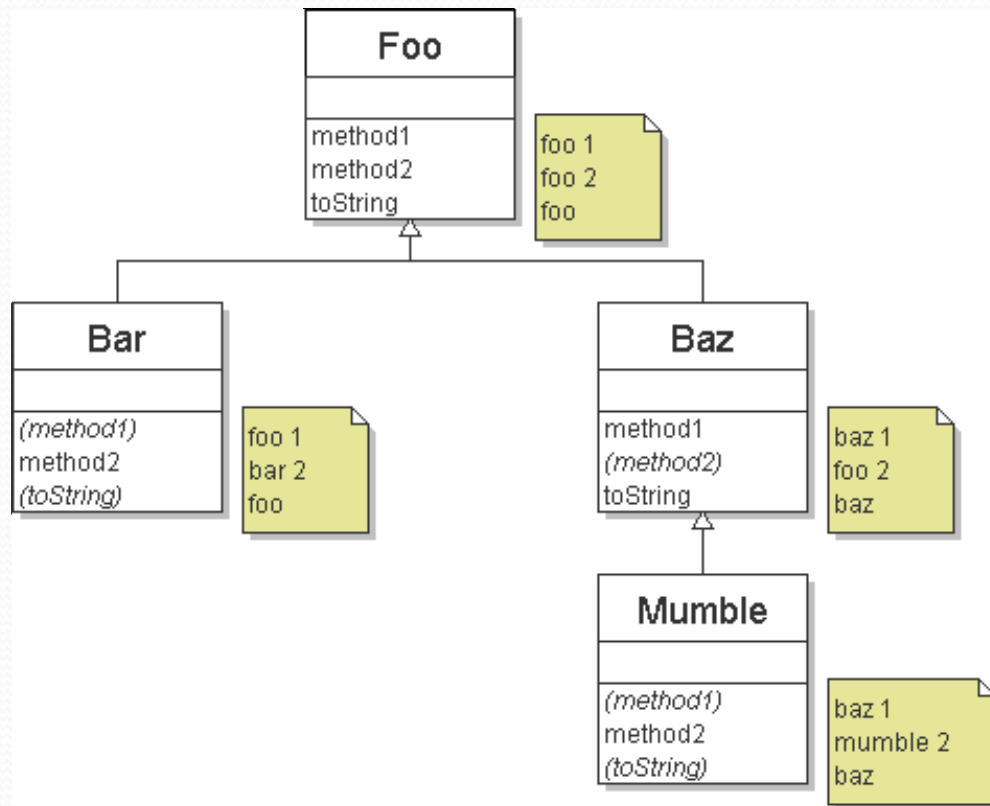
- What would be the output of the following client code?

```
Foo[] foos = {new Baz(), new Bar(), new Mumble(), new Foo()};  
for (int i = 0; i < foos.length; i++) {  
    System.out.println(foos[i]);  
    foos[i].method1();  
    foos[i].method2();  
    System.out.println();  
}
```



# Diagramming the classes

- Add classes from top (superclass) to bottom (subclass).
- Include all inherited methods.



# Finding output with tables

<b>method</b>	<b>Foo</b>	<b>Bar</b>	<b>Baz</b>	<b>Mumble</b>
method1	foo 1	<i>foo 1</i>	baz 1	<i>baz 1</i>
method2	foo 2	bar 2	<i>foo 2</i>	mumble 2
toString	foo	<i>foo</i>	baz	<i>baz</i>

# Polymorphism answer

```
Foo[] foos = {new Baz(), new Bar(), new Mumble(), new Foo()};  
for (int i = 0; i < foos.length; i++) {  
    System.out.println(foos[i]);  
    foos[i].method1();  
    foos[i].method2();  
    System.out.println();  
}
```

- **Output:**

```
baz  
baz 1  
foo 2  
  
foo  
foo 1  
bar 2  
  
baz  
baz 1  
mumble 2  
  
foo  
foo 1  
foo 2
```



# Another problem

- The order of the classes is jumbled up.
- The methods sometimes call other methods (tricky!).

```
public class Lamb extends Ham {  
    public void b() {  
        System.out.print("Lamb b    ");  
    }  
}  
  
public class Ham {  
    public void a() {  
        System.out.print("Ham a    ");  
        b();  
    }  
    public void b() {  
        System.out.print("Ham b    ");  
    }  
    public String toString() {  
        return "Ham";  
    }  
}
```

# Another problem 2

```
public class Spam extends Yam {
    public void b() {
        System.out.print("Spam b    ");
    }
}

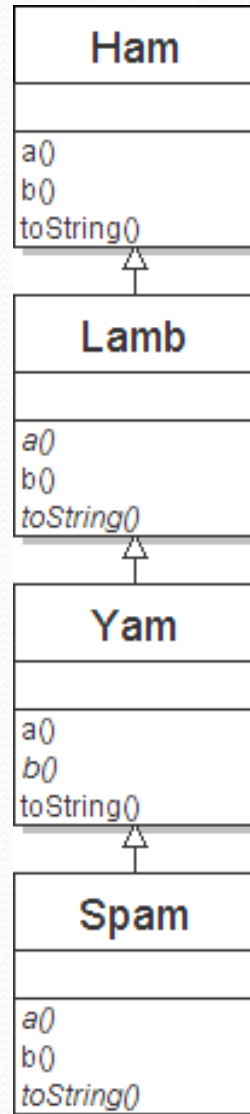
public class Yam extends Lamb {
    public void a() {
        System.out.print("Yam a    ");
        super.a();
    }
    public String toString() {
        return "Yam";
    }
}
```

- What would be the output of the following client code?

```
Ham[] food = {new Lamb(), new Ham(), new Spam(), new Yam()};
for (int i = 0; i < food.length; i++) {
    System.out.println(food[i]);
    food[i].a();
    System.out.println();           // to end the line of output
    food[i].b();
    System.out.println();           // to end the line of output
    System.out.println();
}
```



# Class diagram



# Polymorphism at work

- Lamb inherits Ham's a. a calls b. But Lamb overrides b...

```
public class Ham {  
    public void a() {  
        System.out.print("Ham a    ");  
        b();  
    }  
    public void b() {  
        System.out.print("Ham b    ");  
    }  
    public String toString() {  
        return "Ham";  
    }  
}  
  
public class Lamb extends Ham {  
    public void b() {  
        System.out.print("Lamb b    ");  
    }  
}
```

- Lamb's output from a:

Ham a      **Lamb b**



# The table

method	Ham	Lamb	Yam	Spam
a	Ham a <b>b()</b>	<i>Ham a</i> <b><i>b()</i></b>	Yam a Ham a <b>b()</b>	<i>Yam a</i> <i>Ham a</i> <b><i>b()</i></b>
b	Ham b	Lamb b	Lamb b	Spam b
toString	Ham	<i>Ham</i>	Yam	<i>Yam</i>



# The answer

```
Ham[] food = {new Lamb(), new Ham(), new Spam(), new Yam()};  
for (int i = 0; i < food.length; i++) {  
    System.out.println(food[i]);  
    food[i].a();  
    food[i].b();  
    System.out.println();  
}
```

- **Output:**

```
Ham  
Ham a      Lamb b  
Lamb b  
  
Ham  
Ham a      Ham b  
Ham b  
  
Yam  
Yam a      Ham a      Spam b  
Spam b  
  
Yam  
Yam a      Ham a      Lamb b  
Lamb b
```

# Overriding Object's equals Method

- The Object class contains this method:

```
public boolean equals(Object obj)
```

- many classes override this method
- many students mistakenly *overload* the method
- many headaches when placing objects in data structures



# Overriding Object's equals Method

- overriding equals correctly follows a pattern
- So, it isn't that hard, if you follow the pattern
- Override equals for a Standard Playing Card
- Override equals for a Snake Critter
  - Demo array of Critter objects

# Topic 33

~~ArrayLists~~

# Exercise

- Write a program that reads a file and displays the words of that file.
  - First display all words.
  - Then display them with all plurals (ending in "s") capitalized.
  - Then display them in reverse order.
  - Then create and return an array with all the words except the plural words.
- Can we solve this problem using an array?
  - Why or why not?
  - What would be hard?

# Naive solution

```
String[] allWords = new String[1000];  
int wordCount = 0;
```

```
Scanner input = new Scanner(new File("data.txt"));  
while (input.hasNext()) {  
    String word = input.next();  
    allWords[wordCount] = word;  
    wordCount++;  
}
```

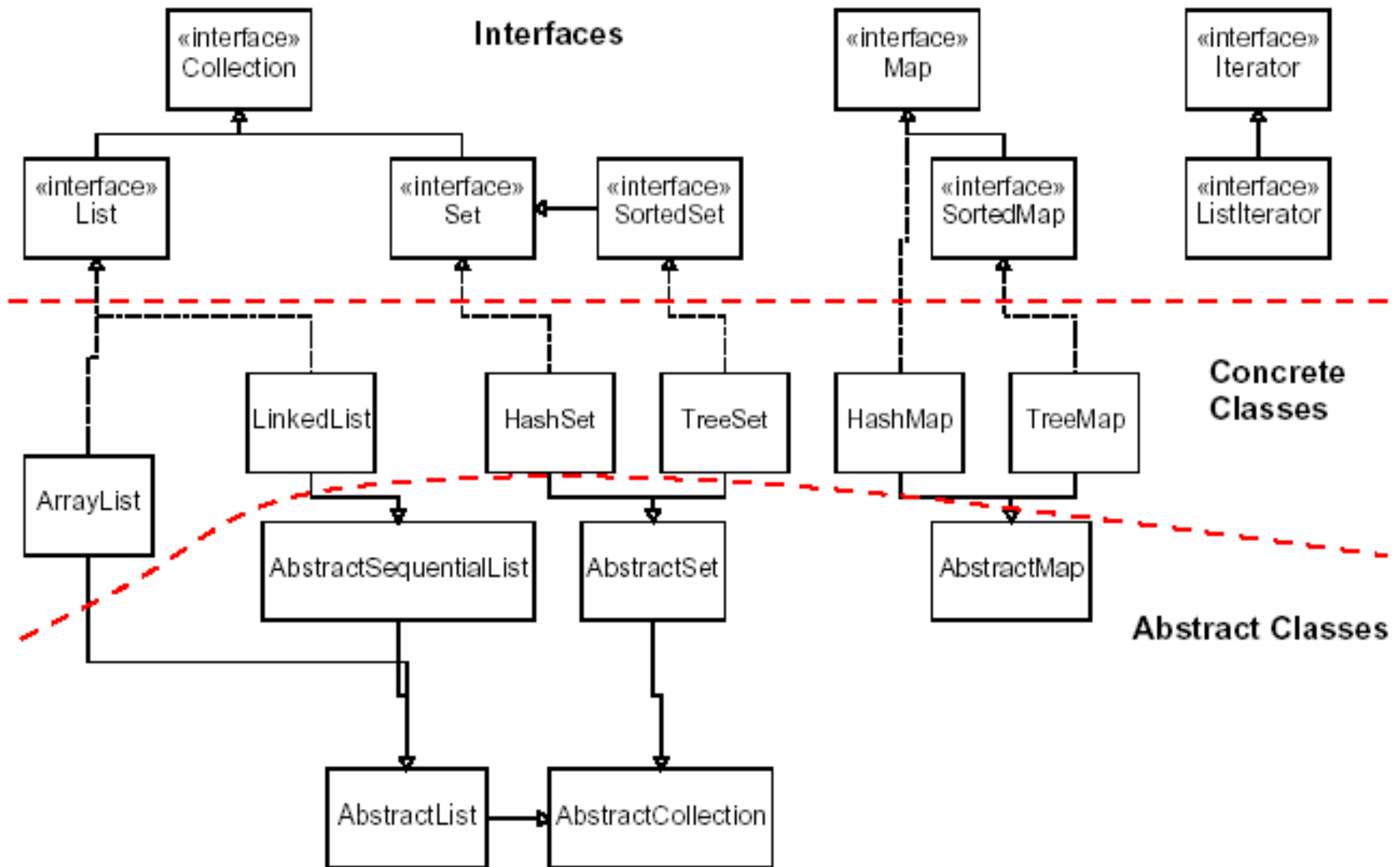
- Problem: You don't know how many words the file will have.
  - Hard to create an array of the appropriate size.
  - Later parts of the problem are more difficult to solve.
- Luckily, there are other ways to store data besides in an array.

# Collections

- **collection**: an object that stores data; a.k.a. "data structure"
  - the objects stored are called **elements**
  - some collections maintain an ordering; some allow duplicates
  - typical operations: *add*, *remove*, *clear*, *contains* (search), *size*
  - examples found in the Java class libraries:
    - `ArrayList`, `LinkedList`, `HashMap`, `TreeSet`, `PriorityQueue`
  - all collections are in the `java.util` package

```
import java.util.*;
```

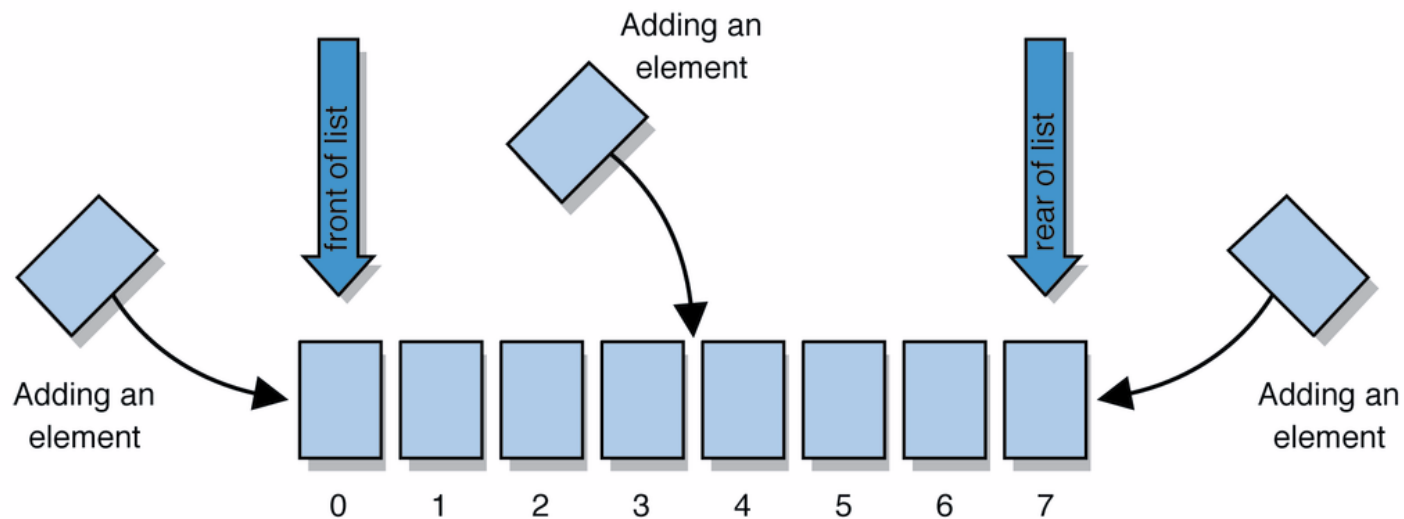
# Java collections framework





# Lists

- **list**: a collection storing an ordered sequence of elements
  - each element is accessible by a 0-based **index**
  - a list has a **size** (number of elements currently present)
  - elements can be added to the front, back, or in the middle
  - Java has several classes that are Lists such as **ArrayList**



# Concept of a list

- Rather than creating an array of elements, create an object that represents a "list" of items. (initially an empty list.)

`[]`

- You can add items to the list.
  - The default behavior is to add to the end of the list.

`[hello, ABC, goodbye, okay]`

- The list object keeps track of the element values that have been added to it, their order, indexes, and its total size.
  - Think of an "array list" as an automatically resizing array object.
  - Internally, the list is implemented using an array and a size field.

# ArrayList methods (10.1)

<code>add ( <b>value</b> )</code>	appends value at end of list
<code>add ( <b>index</b>, <b>value</b> )</code>	inserts given value just before the given index, shifting subsequent values to the right
<code>clear ()</code>	removes all elements of the list
<code>indexOf ( <b>value</b> )</code>	returns first index where given value is found in list (-1 if not found)
<code>get ( <b>index</b> )</code>	returns the value at given index
<code>remove ( <b>index</b> )</code>	removes/returns value at given index, shifting subsequent values to the left
<code>set ( <b>index</b>, <b>value</b> )</code>	replaces value at given index with given value
<code>size ()</code>	returns the number of elements in list
<code>toString ()</code>	returns a string representation of the list such as "[ 3, 42, -7, 15 ]"

# ArrayList methods 2

addAll ( <b>list</b> ) addAll ( <b>index</b> , <b>list</b> )	adds all elements from the given list to this list (at the end of the list, or inserts them at the given index)
contains ( <b>value</b> )	returns true if given value is found somewhere in this list
containsAll ( <b>list</b> )	returns true if this list contains every element from given list
equals ( <b>list</b> )	returns true if given other list contains the same elements
iterator() listIterator()	returns an object used to examine the contents of the list (seen later)
lastIndexOf ( <b>value</b> )	returns last index value is found in list (-1 if not found)
remove ( <b>value</b> )	finds and removes the given value from this list
removeAll ( <b>list</b> )	removes any elements found in the given list from this list
retainAll ( <b>list</b> )	removes any elements <i>not</i> found in given list from this list
subList ( <b>from</b> , <b>to</b> )	returns the sub-portion of the list between indexes <b>from</b> (inclusive) and <b>to</b> (exclusive)
toArray ()	returns the elements in this list as an array

# Type Parameters (Generics)

```
ArrayList<Type> name = new ArrayList<Type>();
```

- When constructing an `ArrayList`, you must specify the type of elements it will contain between `<` and `>`.
  - This is called a *type parameter* or a *generic* class.
  - Allows the same `ArrayList` class to store lists of different types.

```
ArrayList<String> names = new ArrayList<String>();  
names.add("Marty Stepp");  
names.add("Stuart Reges");
```

# ArrayList of primitives?

- The type you specify when creating an `ArrayList` must be an object type; it cannot be a primitive type.

```
// illegal -- int cannot be a type parameter  
ArrayList<int> list = new ArrayList<int>();
```

- But we can still use `ArrayList` with primitive types by using special classes called *wrapper* classes in their place.

```
// creates a list of ints  
ArrayList<Integer> list = new ArrayList<Integer>();
```

# Wrapper classes

Primitive Type	Wrapper Type
int	Integer
double	Double
char	Character
boolean	Boolean

- A wrapper is an object whose sole purpose is to hold a primitive value.
- Once you construct the list, use it with primitives as normal:

```
ArrayList<Double> grades = new ArrayList<Double>();  
grades.add(3.2);  
grades.add(2.7);  
...  
double myGrade = grades.get(0);
```

# Clicker 1

What is the output of the following code?

```
ArrayList<String> list = new ArrayList<>();  
list.add("D");  
list.add("X");  
list.add("C");  
list.add(1, "M");  
list.add(3, "P");  
list.remove(2);  
System.out.println(list);
```

A. [D, M, P, C]

B. []

C. [D, X, P, C]

D. [D, M, null, P, C]

E. [M, X]



# Clicker 2

What is the output of the following code?

```
ArrayList<Double> list = new ArrayList<>();  
for (int i = 1; i <= 8; i++) {  
    list.add((double) (i * 5));  
}  
for (int i = 0; i < list.size(); i++) {  
    list.remove(i);  
}  
System.out.println(list);
```

- A. [10.0, 20.0, 30.0, 40.0]
- B. []
- C. [5.0, 10.0, 15.0, 20.0]
- D. [40.0]
- E. No output due to syntax or runtime error.

# Learning about classes

- The [Java API Specification](http://java.sun.com/javase/6/docs/api/) contains the documentation for every Java class in the standard library and their methods.
  - The link to the API Specs is on the course web site.



# ArrayList vs. array

- construction

```
String[] names = new String[5];
```

```
ArrayList<String> list = new ArrayList<String>();
```

- storing a value

```
names[0] = "Jessica";
```

```
list.add("Jessica");
```

- retrieving a value

```
String s = names[0];
```

```
String s = list.get(0);
```

# ArrayList vs. array 2

- doing something to each value that starts with "B"

```
for (int i = 0; i < names.length; i++) {  
    if (names[i].startsWith("B")) { ... }  
}
```

```
for (int i = 0; i < list.size(); i++) {  
    if (list.get(i).startsWith("B")) { ... }  
}
```

- seeing whether the value "Benson" is found

```
for (int i = 0; i < names.length; i++) {  
    if (names[i].equals("Benson")) { ... }  
}
```

```
if (list.contains("Benson")) { ... }
```

# Exercise, revisited

- Write a program that reads a file and displays the words of that file as a list.
  - First display all words.
  - Then display them in reverse order.
  - Then display them with all plurals (ending in "s") capitalized.
  - Then display them with all plural words removed.

# Exercise solution (partial)

```
ArrayList<String> allWords = new ArrayList<String>();
Scanner input = new Scanner(new File("words.txt"));
while (input.hasNext()) {
    String word = input.next();
    allWords.add(word);
}
System.out.println(allWords);

// remove all plural words
for (int i = 0; i < allWords.size(); i++) {
    String word = allWords.get(i);
    if (word.endsWith("s")) {
        allWords.remove(i);
        i--; // Angel Tears
    }
}
```

# ArrayList as parameter

```
public static void name(ArrayList<Type> name) {
```

- Example:

```
// Removes all plural words from the given list.
```

```
public static void removePlural(ArrayList<String> list) {  
    for (int i = 0; i < list.size(); i++) {  
        String str = list.get(i);  
        if (str.endsWith("s")) {  
            list.remove(i);  
            i--;  
        }  
    }  
}
```

- You can also return a list:

```
public static ArrayList<Type> methodName(params)
```

# Exercise

- Write a program that reads a file full of numbers and displays all the numbers as a list, then:
  - Prints the average of the numbers.
  - Prints the highest and lowest number.
  - Filters out all of the even numbers (ones divisible by 2).



# Exercise solution (partial)

```
ArrayList<Integer> numbers = new ArrayList<Integer>();
Scanner input = new Scanner(new File("numbers.txt"));
while (input.hasNextInt()) {
    int n = input.nextInt();
    numbers.add(n);
}
System.out.println(numbers);
filterEvens(numbers);
System.out.println(numbers);
...

// Removes all elements with even values from the given list.
public static void filterEvens(ArrayList<Integer> list) {
    for (int i = list.size() - 1; i >= 0; i--) {
        int n = list.get(i);
        if (n % 2 == 0) {
            list.remove(i);
        }
    }
}
```

# Other Exercises

- Write a method `reverse` that reverses the order of the elements in an `ArrayList` of strings.
- Write a method `capitalizePlurals` that accepts an `ArrayList` of strings and replaces every word ending with an "s" with its uppercased version.
- Write a method `removePlurals` that accepts an `ArrayList` of strings and removes every word in the list ending with an "s", case-insensitively.

# Out-of-bounds

- Legal indexes are between **0** and the **list's size() - 1**.
  - Reading or writing any index outside this range will cause an `IndexOutOfBoundsException`.

```
ArrayList<String> names = new ArrayList<String>();  
names.add("Marty");    names.add("Kevin");  
names.add("Vicki");    names.add("Larry");  
System.out.println(names.get(0));           // okay  
System.out.println(names.get(3));           // okay  
System.out.println(names.get(-1));         // exception  
names.add(9, "Aimee");                     // exception
```

<i>index</i>	<i>0</i>	<i>1</i>	<i>2</i>	<i>3</i>
<i>value</i>	Marty	Kevin	Vicki	Larry

# ArrayList "mystery" 2

```
ArrayList<Integer> list = new ArrayList<Integer>();  
for (int i = 1; i <= 5; i++) {  
    list.add(2 * i);    // [2, 4, 6, 8, 10]  
}
```

- What is the output of the following code?

```
int size = list.size();  
for (int i = 0; i < size; i++) {  
    list.add(i, 42);    // add 42 at index i  
}  
System.out.println(list);
```

- Answer:

```
[42, 42, 42, 42, 42, 2, 4, 6, 8, 10]
```

# ArrayList as parameter

```
public static void name(ArrayList<Type> name) {
```

- Example:

```
// Removes all plural words from the given list.
```

```
public static void removePlural(ArrayList<String> list) {  
    for (int i = 0; i < list.size(); i++) {  
        String str = list.get(i);  
        if (str.endsWith("s")) {  
            list.remove(i);  
            i--;  
        }  
    }  
}
```

- You can also return a list:

```
public static ArrayList<Type> methodName(params)
```

# Exercise

- Write a method `addStars` that accepts an array list of strings as a parameter and places a `*` after each element.
  - Example: if an array list named `list` initially stores:  
`[the, quick, brown, fox]`
  - Then the call of `addStars(list);` makes it store:  
`[the, *, quick, *, brown, *, fox, *]`
- Write a method `removeStars` that accepts an array list of strings, assuming that every other element is a `*`, and removes the stars (undoing what was done by `addStars` above).

# Exercise solution

```
public static void addStars(ArrayList<String> list) {  
    for (int i = 0; i < list.size(); i += 2) {  
        list.add(i, "*");  
    }  
}
```

```
public static void removeStars(ArrayList<String> list) {  
    for (int i = 0; i < list.size(); i++) {  
        list.remove(i);  
    }  
}
```

# Exercise

- Write a method `intersect` that accepts two sorted array lists of integers as parameters and returns a new list that contains only the elements that are found in both lists.
  - Example: if lists named `list1` and `list2` initially store:  
[1, **4**, 8, 9, **11**, 15, 17, **28**, 41, **59**]  
[**4**, 7, **11**, **17**, 19, 20, 23, **28**, 37, **59**, 81]
  - Then the call of `intersect(list1, list2)` returns the list:  
[4, 11, 17, 28, 59]



# Other Exercises

- Write a method `reverse` that reverses the order of the elements in an `ArrayList` of strings.
- Write a method `capitalizePlurals` that accepts an `ArrayList` of strings and replaces every word ending with an "s" with its uppercased version.
- Write a method `removePlurals` that accepts an `ArrayList` of strings and removes every word in the list ending with an "s", case-insensitively.

# Objects storing collections

- An object can have an array, list, or other collection as a field.

```
public class Course {  
    private double[] grades;  
    private ArrayList<String> studentNames;  
  
    public Course() {  
        grades = new double[4];  
        studentNames = new ArrayList<String>();  
        ...  
    }  
}
```

- Now each object stores a collection of data inside it.

# The compareTo method (10.2)

- The standard way for a Java class to define a comparison function for its objects is to define a `compareTo` method.
  - Example: in the `String` class, there is a method:  

```
public int compareTo(String other)
```
- A call of **A**.`compareTo`(**B**) will return:
  - a value  $< 0$  if **A** comes "before" **B** in the ordering,
  - a value  $> 0$  if **A** comes "after" **B** in the ordering,
  - or 0 if **A** and **B** are considered "equal" in the ordering.

# Using compareTo

- `compareTo` can be used as a test in an `if` statement.

```
String a = "alice";  
String b = "bob";  
if (a.compareTo(b) < 0) { // true  
    ...  
}
```

Primitives	Objects
<code>if (a &lt; b) { ...</code>	<code>if (a.compareTo(b) &lt; 0) { ...</code>
<code>if (a &lt;= b) { ...</code>	<code>if (a.compareTo(b) &lt;= 0) { ...</code>
<code>if (a == b) { ...</code>	<code>if (a.compareTo(b) == 0) { ...</code>
<code>if (a != b) { ...</code>	<code>if (a.compareTo(b) != 0) { ...</code>
<code>if (a &gt;= b) { ...</code>	<code>if (a.compareTo(b) &gt;= 0) { ...</code>
<code>if (a &gt; b) { ...</code>	<code>if (a.compareTo(b) &gt; 0) { ...</code>

# compareTo and collections

- You can use an array or list of strings with Java's included binary search method because it calls `compareTo` internally.

```
String[] a = {"al", "bob", "cari", "dan", "mike"};  
int index = Arrays.binarySearch(a, "dan"); // 3
```

- Java's `TreeSet/Map` use `compareTo` internally for ordering.

```
Set<String> set = new TreeSet<String>();  
for (String s : a) {  
    set.add(s);  
}  
System.out.println(s);  
// [al, bob, cari, dan, mike]
```

# Ordering our own types

- We cannot binary search or make a `TreeSet/Map` of arbitrary types, because Java doesn't know how to order the elements.
  - The program compiles but crashes when we run it.

```
Set<HtmlTag> tags = new TreeSet<HtmlTag>();  
tags.add(new HtmlTag("body", true));  
tags.add(new HtmlTag("b", false));  
...
```

```
Exception in thread "main" java.lang.ClassCastException  
at java.util.TreeSet.add(TreeSet.java:238)
```

# Comparable (10.2)

```
public interface Comparable<E> {  
    public int compareTo(E other);  
}
```

- A class can implement the `Comparable` interface to define a natural ordering function for its objects.
- A call to your `compareTo` method should return:
  - a value  $< 0$  if the `other` object comes "before" this one,
  - a value  $> 0$  if the `other` object comes "after" this one,
  - or 0 if the `other` object is considered "equal" to this.
- If you want multiple orderings, use a `Comparator` instead (see Ch. 13.1)

# Comparable template

```
public class name implements Comparable<name> {  
  
    ...  
  
    public int compareTo(name other) {  
        ...  
    }  
}
```



# Comparable example

```
public class Point implements Comparable<Point> {  
    private int x;  
    private int y;  
    ...  
  
    // sort by x and break ties by y  
    public int compareTo(Point other) {  
        if (x < other.x) {  
            return -1;  
        } else if (x > other.x) {  
            return 1;  
        } else if (y < other.y) {  
            return -1;    // same x, smaller y  
        } else if (y > other.y) {  
            return 1;    // same x, larger y  
        } else {  
            return 0;    // same x and same y  
        }  
    }  
}
```

# compareTo tricks

- *subtraction trick* - Subtracting related numeric values produces the right result for what you want `compareTo` to return:

```
// sort by x and break ties by y
public int compareTo(Point other) {
    if (x != other.x) {
        return x - other.x;    // different x
    } else {
        return y - other.y;    // same x; compare y
    }
}
```

## – The idea:

- if  $x > \text{other.x}$ , then  $x - \text{other.x} > 0$
- if  $x < \text{other.x}$ , then  $x - \text{other.x} < 0$
- if  $x == \text{other.x}$ , then  $x - \text{other.x} == 0$

– NOTE: This trick doesn't work for `doubles` (but see `Math.signum`)

# compareTo tricks 2

- *delegation trick* - If your object's fields are comparable (such as strings), use their `compareTo` results to help you:

```
// sort by employee name, e.g. "Jim" < "Susan"
public int compareTo(Employee other) {
    return name.compareTo(other.getName());
}
```

- *toString trick* - If your object's `toString` representation is related to the ordering, use that to help you:

```
// sort by date, e.g. "09/19" > "04/01"
public int compareTo(Date other) {
    return toString().compareTo(other.toString());
}
```

# Exercises

- Make the `HtmlTag` class from HTML Validator comparable.
  - Compare tags by their elements, alphabetically by name.
  - For the same element, opening tags come before closing tags.

```
// <b></b><i><b></b><br/></i></body>
Set<HtmlTag> tags = new TreeSet<HtmlTag>();
tags.add(new HtmlTag("body", true));    // <b>
tags.add(new HtmlTag("b", true));       // </b>
tags.add(new HtmlTag("b", false));      // <i>
tags.add(new HtmlTag("i", true));       // <b>
tags.add(new HtmlTag("b", false));      // </b>
tags.add(new HtmlTag("br"));            // <br/>
tags.add(new HtmlTag("i", false));      // </i>
tags.add(new HtmlTag("body", false));   // </body>
System.out.println(tags);
// [<b>, </b>, <b>, </b>, <br/>, <i>, </i>]
```

# Exercise solution

```
public class HtmlTag implements Comparable<HtmlTag> {
    ...
    // Compares tags by their element ("body" before "head"),
    // breaking ties with opening tags before closing tags.
    // Returns < 0 for less, 0 for equal, > 0 for greater.
    public int compareTo(HtmlTag other) {
        int compare = element.compareTo(other.getElement());
        if (compare != 0) {
            // different tags; use String's compareTo result
            return compare;
        } else {
            // same tag
            if ((isOpenTag == other.isOpenTag()) {
                return 0;    // exactly the same kind of tag
            } else if (other.isOpenTag()) {
                return 1;    // he=open, I=close; I am after
            } else {
                return -1;   // I=open, he=close; I am before
            }
        }
    }
}
```