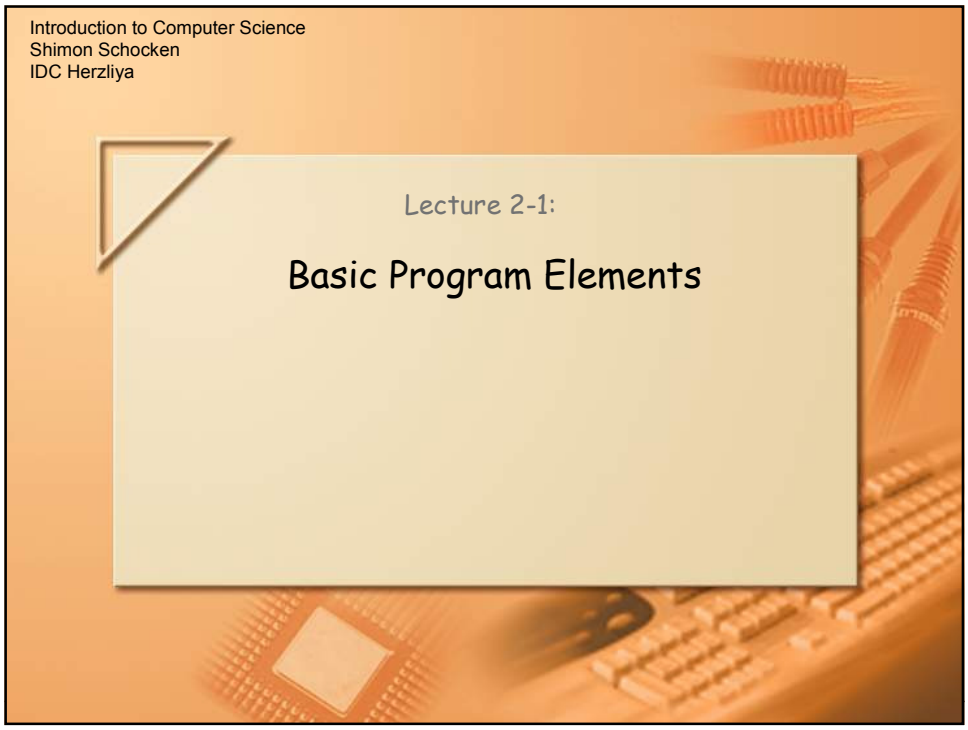


Basic Program Elements: lecture outline

- Binary representation
- Variables
- Data types
  - Numbers
  - Characters
- Constants
- Assignment
- Arithmetic operators and expressions
- Casting
- Input / Output
- Program readability
- Identifiers.

Binary Representation (first approximation)

Example: 8-bit memory

0	10011100
1	00101000
2	11010100
3	00000011
4	10010010
...	11011011

- Digital devices and networks are built to handle two values only; We call these values 0 and 1
- Fortunately, practically anything can be represented by 0's and 1's:
  - Integers: 1 can be represented by 00000001, 2 by 00000010, etc.
  - Characters: 'A' can be represented by 00100001, 'B' by 00100010, etc.
  - Images: can be represented by bitmap / vector graphics
  - More: music, video, smell, … almost anything can be digitized
- The number of bits used in the representation determines how many different values can be represented:
  - 1 bit can represent 2 values: 0,1
  - 2 bits can represent 4 values: 00,01,10,11
  - 8 bits (a *byte*) can represent 256 values: 00000000,...,11111111
  - 32 bits can represent more than 4,000,000,000 values
  - 64 bits can represent a lot more values

In the most part, we don't have to worry about binary numbers. Why?

- Because high level programming languages like Java are designed to abstract the low level complexity away from the programmer
- An important abstraction: the high-level concept of a variable.

Variables

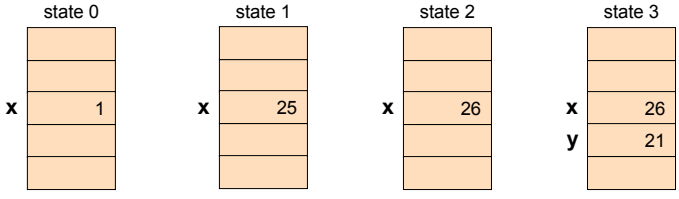
```
public class VariableDemo {
  public static void main(String[] args) {
    int x = 1;       // state 0
    x = 25;          // state 1
    x = x + 1;       // state 2
   int y = x - 5;    // State 3
  }
}
```

Variable abstraction:
A named and typed container

Variable implementation:
A memory location associated with the variable name and type

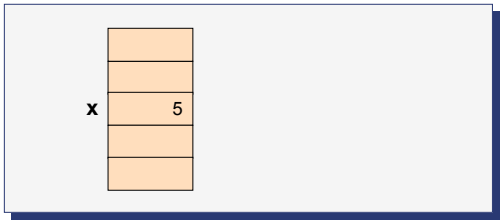
Variable declaration:
Initializes the memory allocation, name, type, and (optionally) a value.

Memory states:



Variables (behind the scene)

Abstraction (high level)



None of this should interest the high-level programmer

Implementation (low level, on a 32-bit memory)

Main memory

20578925	10110110110100110011011111010011
20578926	00110111110100110011011111010011
20578927	00000000000000000000000000000101
20578928	00110111110100110011011111010011
20578929	11110111110100111011011000010011

(Memory addresses are an arbitrary example)

Behind the scene housekeeping

- The compiler maintains a *symbol table*
- Symbol table:
  variable x is mapped on address 20578927
- Whenever the program assigns a new value to x:
  1. Look up x's address in the symbol table
  2. Update the contents of this address

In languages that employ a *virtual machine*, like Java, the VM adds another layer of abstraction and housekeeping.
More about this, later.

Variable declaration

Syntax:

```
type variableName;
```

```
type variableName1, variableName1 ... ;
```

```
type variableName1 = expression;
```

Examples:

```
int weight;
```

```
char c1, c2, s;
```

```
String city = "LA"
```

- Variables are declared as needed, by the programmer
- Each variable is declared to hold values of a certain *type*
- Variables can be declared anywhere in the program, according to the program's needs
- Local variables are not initialized unless you initialize them.

The assignment operation

Syntax:

```
variableName  =  expression;
```

Examples:

```
int a = 5, b = 9, c, d, f, celsius;
c = -3;
a = b;
d = b * b - 4 * a * c;
f = scan.nextInt();
celsius = (f - 32) * 5 / 9
// Side comment: f looks like a badly chosen variable name
```

Assignment anatomy:

- First, the expression on the right hand side is evaluated
- The resulting value is then assigned to the variable on the left hand side, overwriting its current value

Don't get confused:

The assignment operator `"="` has nothing to do with algebra's operator `"="`

Types (so far)

Type = a set of values and operations that can be performed on them

Types fall into two categories:

- Primitive types (fixed, part of the language) ← This lecture
- Class types (extensible, defined by programmers) ← Later in the course

Examples:

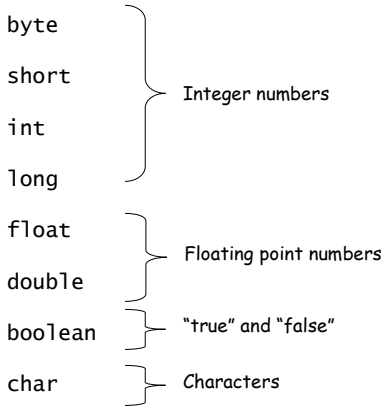
- The primitive type `short` is the set of values `-32768` ... `32767` and arithmetic operations like `+, -, *, /`
- The class type `Turtle` is the set of all possible `Turtle` objects; each object is characterized by a screen coordinates and a direction and operations like `hide()`, `show()`, `moveForward()`, ...

  (comment: that's *our* notion of turtles!)

- Primitive types are documented in the language specification
- Class types are documented by their respective class API's

Primitive types

Java features 8 primitive types:



- Primitive types are *abstractions*, provided by the Java language and compiler
- At the low-level implementation, all data values, irrespective of type, are represented in computer memory by bits
- High-level languages hide this low-level complexity and allow the programmer to perform type-oriented operations on data values, according to their type.

Integer types

Type	Bits	Value Range
`byte`	8	`-128 ... 127`
`short`	16	`-32768 ... 32767`
`int`	32	`-2,147,483,648 ... 2,147,483,647`
`long`	64	$< -9 \times 10^{18}$ ... $> 9 \times 10^{18}$

- The four integer types differ in the amount of memory allocated to store their data values, and in the resulting data range
- Why not define all integers as `long`?
  - To promote storage and processing efficiency (occasionally not important)
  - To make programs more readable (always critically important)

Floating Point types

Type	Bits	Range	Precision (significant digits)
`float`	32	-3.4×10^{38} ... 3.4×10^{38}	7
`double`	64	-1.7×10^{308} ... 1.7×10^{308}	15

The floating point method uses a binary representation.

It cannot represent 1/10 accurately.

Best practice advice:

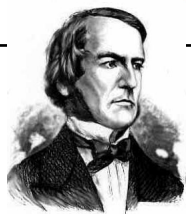
- Use `double` instead of `float`. The precision gain outweighs the performance loss, which is negligible
- For most scientific / engineering calculations, `double` is sufficiently accurate
- For code that deals with money, use `int`, `long`, or `BigDecimal`.

Example (thx to Josh Bloch):

```
System.out.println(1.03 - 0.42);

// will print 0.6100000000000001;

System.out.println(1.00 - 9 * .10);

// will print 0.0999999999999998;
```

Boolean

The `Boolean` type consists of the two values `true` , `false` and operations `!` (not), `&&` (and), `||` (or)



George Boole
(1815-1864)

Inventor of a 2-valued arithmetic now called "Boolean logic"

Examples:

```
boolean lightOn = false, roomIsEmpty = false;
...
lightOn = true;  // Turn the lights on
...
lightOn = !(lightOn);  // Switch the current light state to the other
...
if (roomIsEmpty && lightOn)
   System.out.println("There is an empty room with the lights on");
...
if (roomIsEmpty)
    lightOn = false;  // Turn the light off
...
```

Basic Program Elements: lecture outline

- Binary representation
- Variables
- Data types
  - Numbers
  - Characters
- Constants
- Assignment
- Arithmetic operators and expressions
- Casting
- Input / Output
- Program readability
- Identifiers.

Characters

Characters

- Characters = symbols that can be printed on output devices (printer, screen, ...)
- Control characters (unprintable) = symbols that are used to control output devices

Every one of these symbols is represented internally by an agreed-upon numeric code

Character set = an ordered mapping between characters and their numeric codes

ASCII (late 1960's)

- An agreed-upon computing industry standard character set, based on a 7-bit coding scheme, giving a total of 128 possible codes
- For example: `'a'` is represented by 97, `'b'` by 98, `'c'` by 99, ...
  `'A'` by 65, `'B'` by 66, `'C'` by 66, ...
  `'0'` by 48, `'1'` by 49, `'2'` by 50, ...
  `';'` by 59, `'<'` by 60, and so on (see next slide)
- Problem: 128 codes are not enough for representing the characters of the different languages of the world

UNICODE (late 1980's):

- An agreed-upon 16-bit character set standard, representing more than 100,000 different characters.

ASCII table

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

- The first 31 codes represent control characters

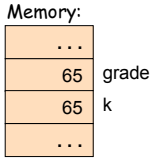
The char type

- A `char` variable represents a single Unicode value
- `char` values are implemented (stored in memory) as 16-bit numbers
- When a Java statement instructs to display a `char` value, the screen driver, which is part of the OS, uses the default font to create a pattern of pixels which is then rendered on the screen
- When a Java statement instructs to display a numeric value (e.g. an `int` variable), the OS parses the value and displays each digit character separately.

Example:

```
char grade = 'A';
int k = 65;
System.out.println(grade);
System.out.println(k);
```

Memory:



Control characters (aka "escape sequences")

Desired output:

Won't work:

```
System.out.println("I don't like the word "no"")
```

Will work:

```
System.out.println("I don\'t like\nthe word \"no\"")
```

In Java, control characters are called *escape sequences*, beginning with a backslash (\):

Escape Sequence	Meaning
\b	backspace
\t	tab
\n	newline
\r	carriage return
\"	double quote
\'	single quote
\\	Backslash

A few more …

Lecture outline

- Binary representation
- Variables
- Data types
  - Numbers
  - Characters
- Constants
- Assignment
- Arithmetic operators and expressions
- Casting
- Input / Output
- Program readability
- Identifiers (Noam started with it, should be at end)

Constants

A variable declaration can include the prefix `FINAL`

This means that the variable value is set once and can never change thereafter

```
final double PI = 3.14159;
final int LONDON_OLYMPICS_YEAR = 2012;
final boolean SUMMER_SAVINGS_TIME = true;
final double G = 6.67428e-11;
```

Advantages:

- Readability
- Consistency
- Error minimization
- Efficiency in code development and maintenance.

Arithmetic operators

An operator is a mapping that maps one or more operands onto a single value:

```
a + b      // adds a and b
a - b      // subtracts b from a
a * b      // multiplies a and b
a / b      // divides a by b
a % b      // the reminder of dividing a by b
-a         // The negative value of a
```

The types of the operands determine the type of the resulting expression
(the compiler takes care of this)

For example: if either operand is a `double`, the expression type will also be `double`.

Integer division

When two integers (`byte`, `short`, `int`, `long`) are divided, the result is truncated into an integer

Example:

```
double x = 5.0;
int n = 5;
System.out.println(x / 2);   // 2.5
System.out.println(n / 2);   // 2
```


Arithmetic expressions

Arithmetic expression is something that evaluates to a numeric value.

```
int a, b, q, a, y, z;
double PI, r, weight;
. . .
// Examples of arithmetic expressions:
17
b
weight * 2.73
2 * PI * r
a - (7 - b)
1 + 2 + 3 + 4
(x + y) * ((2 - z + (5 - q)) * -(1 - x)) / Math.sqrt(z)
```

Arithmetic expressions can be as complex as needed.

Operator precedence

```
10 - 7 - 1        // 2
10 - (7 - 1)      // 4
1 + 2 * 3         // 7
(1 + 2) * 3       // 9
x = 1 - 2 * 3 + 4 * 5  // 15
y = 10 % 3 + 5 * -2    // -9
fahrenheit = 9 / 5 * celsius + 32
```

- Order of evaluation:

  Precedence level 1: unary +, unary –

  Precedence level 2: * / %

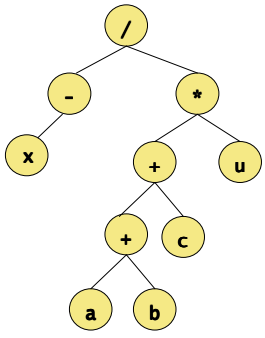
  Precedence level 3: + - , and + (string concatenation)

  Precedence level 4: = (assignment)

- Operators with the same precedence are evaluated left to right
- Parenthesis are used to force a desired order of evaluation.

Expression trees

```
-x / (a + b + c) * u
```



- The evaluation of any given expression can be viewed as an *expression tree*
- The operators lower in the tree have higher precedence and are evaluated first.

Basic Program Elements: lecture outline

- Binary representation
- Variables
- Data types
  - Numbers
  - Characters
- Constants
- Assignment
- Arithmetic operators and expressions
- Casting
- Input / Output
- Program readability
- Identifiers.

Data type conversion

- We often have to deal with an expression that is composed from values that have different data types:

```
final double PI = 3.14159;
int radius = 5;

double area = PI * radius * radius;
System.out.println("Area is: " + area);

int areaInt = (int) (PI * radius * radius);
System.out.println("Area (truncated) is: " + areaInt);
```

widening

narrowing

Output:

```
Area is: 78.53975
Area (truncated) is: 78
```

- Expressions are allowed to contain mixed types.
  This means that, occasionally, types must be converted from one to another
- In some cases, the conversion is done implicitly, by Java, based on the context of the underlying operation
- In other cases, the programmer has to convert the data type explicitly. This is called casting.

Another casting example

```
char c = 'G';
System.out.println(c);
System.out.println((int) c);
System.out.println(c + 1);
```

Output:

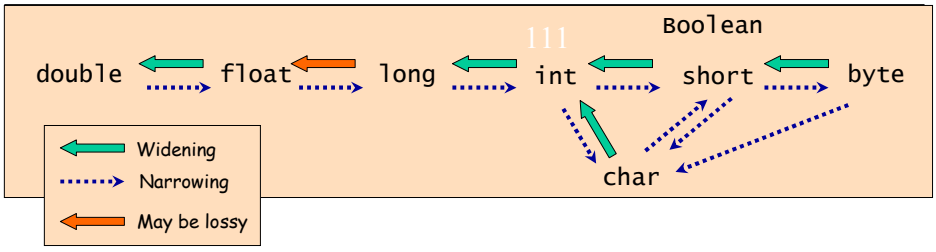
```
G
71
72
```

Important:

Casting operations don't change the type of the variable

Rather, they change the type of its computed *value*.

Data conversion



Widening: Values can be converted to a wider type in three ways :

(in the examples below assume that all variables are `int`, unless said otherwise)

- Arithmetic promotion: `9.0/5.0 * celsius + 32`
- Assignment conversion: `float rate = n + 1` (also occurs in parameter passing)
- Casting: `System.out.println( (double) (x / y) )`

Widening is safe: there is no loss of data

Narrowing: Values can be converted to a narrower type in one way only:

- Casting: `nStudentsNextYear = (int) nStudentsThisYear * 1.1`

Narrowing is lossy and should be used with care.

Basic Program Elements: lecture outline

- Binary representation
- Variables
- Data types
  - Numbers
  - Characters
- Constants
- Assignment
- Arithmetic operators and expressions
- Casting
- Input / Output
- Program readability
- Identifiers.

Two basic forms of user interaction

Console-based textual user interface

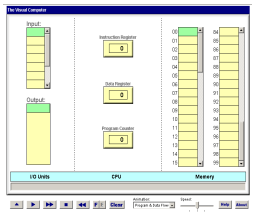
- Output: the programs displays characters on the screen
- Input: user types characters on the keyboard

```
Enter your name: John Brown
Enter your age: 23
Enter your GPA: 3.75
Your name is John Brown.
You are 23 years old
and your GPA is 3.75
```

Done using Java classes designed to read and write textual data

Graphical User Interface ("GUI")

Windows, buttons, mouse, etc.



Done using Java classes designed to manage graphical components

Getting input values using the `Scanner` class

The Java class library includes a class called `Scanner`

This class features various methods for parsing and scanning a stream of characters

The characters stream comes from a certain source: keyboard, file, modem, ...

In Java, the "standard input stream" is represented by `System.in`

To read data from the keyboard, we construct a `Scanner` object and initialize it to read data from `System.in`:

```
import java.util.Scanner;

public class IODemo {
  public static void main (String[] args) {

    Scanner scan = new Scanner(System.in);
    ...
  }
}
```

This code creates a new `Scanner` object, which we call `scan`, through which we can now invoke various methods that get input values from the keyboard.

Getting input values using the `Scanner` class

```
import java.util.Scanner;
public class VarDemo {
    public static void main (String args[]) {
        Scanner scan = new Scanner(System.in);
        System.out.print("Enter your name: ");
        String name = scan.nextLine();

        System.out.print("Enter your age: ");
        int age = scan.nextInt();
        System.out.print("Enter your GPA: ");
        double gpa = scan.nextDouble();
        System.out.println("Your name is " + name + ". You are " + age
                           + " years old and your GPA is "+ gpa);
    }
}
```

Scanner API (partial):

- `String nextLine()` returns all the characters till the end of the line
- `int nextInt()` returns the next token, as an integer
- Other `Scanner` methods are designed to return other types

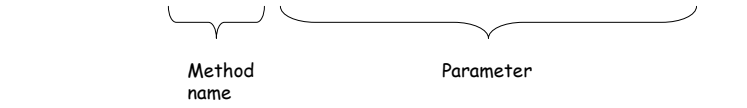
`Scanner` is an example of a "singleton class"

Output:

```
Enter your name: John Van Leer
Enter your age: 23
Enter your GPA: 3.75
Your name is John Van Leer. You are 23 years old and your GPA is 3.75
```

Displaying outputs using `print` and `println`

```
System.out.println("opportunty favors the prepared")
```



Three popular printing methods:

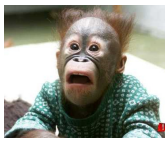
- `Println(x)`: prints the value of x and advances to the next line
- `Print(x)`: prints the value of x
- `Printf(`*displayFormat*`, x)`: prints the value of `x` according to some display format.

Lecture outline

- Variables
- Data types
  - Numeric
  - Textual
- Constants
- Arithmetic expressions
- Casting
- Input / Output
- Program readability
- Identifiers

The importance of program readability

```
class A {

  public static void main(String[] args){
    Scanner g = new Scanner(System.in);
    double u, v = 0; g5 = 0;

    while(true) {
      System.out.print("Enter a number:");
      u = g.nextInt();
      v = v + u ;
      g5 = g5 + 1;
    }
    System.out.println(v / g5);
  }
}
```



The importance of program readability

```
class Average {

  public static void main(String[] args){
    Scanner scan = new Scanner(System.in);
    double x, sum = 0; count = 0;

    while(true) {
      System.out.println("Enter a number:");
      x = scan.nextIn();
      sum = sum + x ;
      count = count + 1;
    }
    System.out.println(sum / count);
  }
}
```

Choosing good identifiers is critically important for program's readability.

Best practice advice: don't use "mispar", "cadur", etc. Use English words.

Identifiers

- Identifiers = labels that the programmer chooses for naming her classes, methods, and variables
- The identifier can be made up of any length of
  - letters
  - digits
  - underscore character (_)
  - dollar sign ($)

  But it cannot begin with a digit
- Java is *case sensitive:* `Count` and `count` are completely different identifiers
- The identifiers that occur in Java programs come from two sources:
  - Chosen by you:

    `x, y, sum, transfer, BankAccount, ...`
  - Chosen by other programmers whose classes we use:

    `in, out, String, nextInt, StringTokenizer, NullPointerException ...`

Best practice advice: different organizations have different variable naming conventions. Read the organization's *Programming Style Guidelines*.

ERROR: undefined
OFFENDING COMMAND:

STACK: