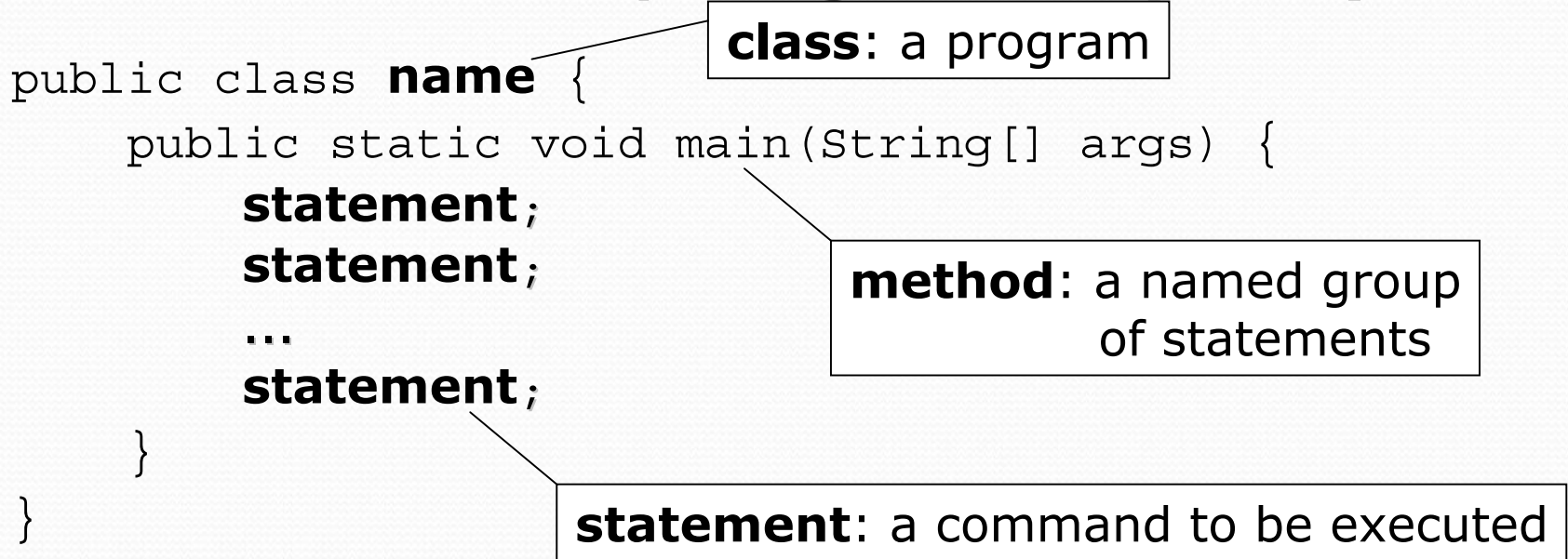


# Building Java Programs

Lecture 1: Java Review

**reading: Ch. 1-9**

# A Java program (1.2)



- Every executable Java program consists of a **class**,
  - that contains a **method** named `main`,
  - that contains the **statements** (commands) to be executed.

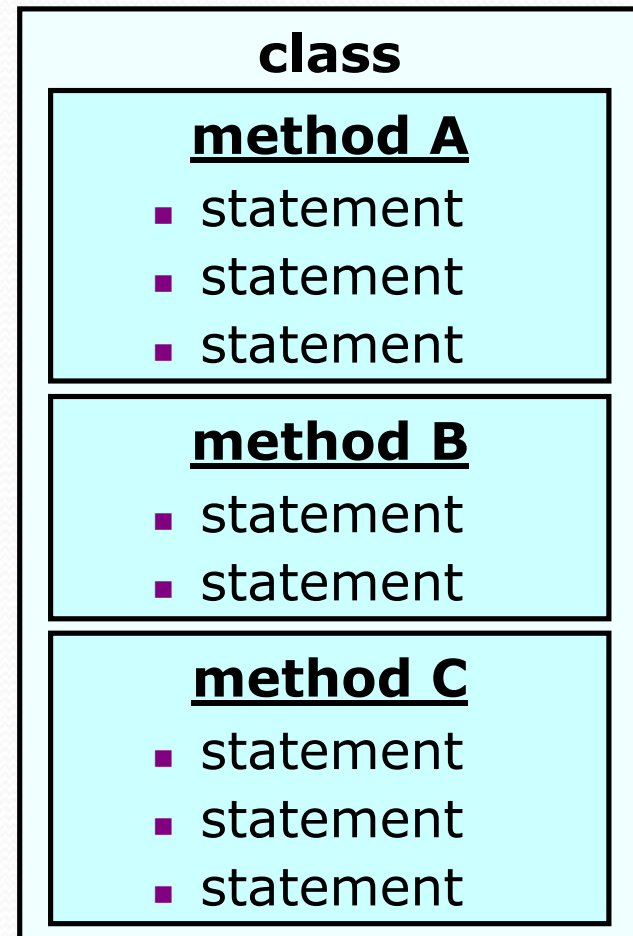
# System.out.println

- A statement that prints a line of output on the console.
  - pronounced "print-linn"
  - sometimes called a "println statement" for short
- 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.



# Static methods (1.4)

- **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
- Writing a static method is like adding a new command to Java.



# 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();
```

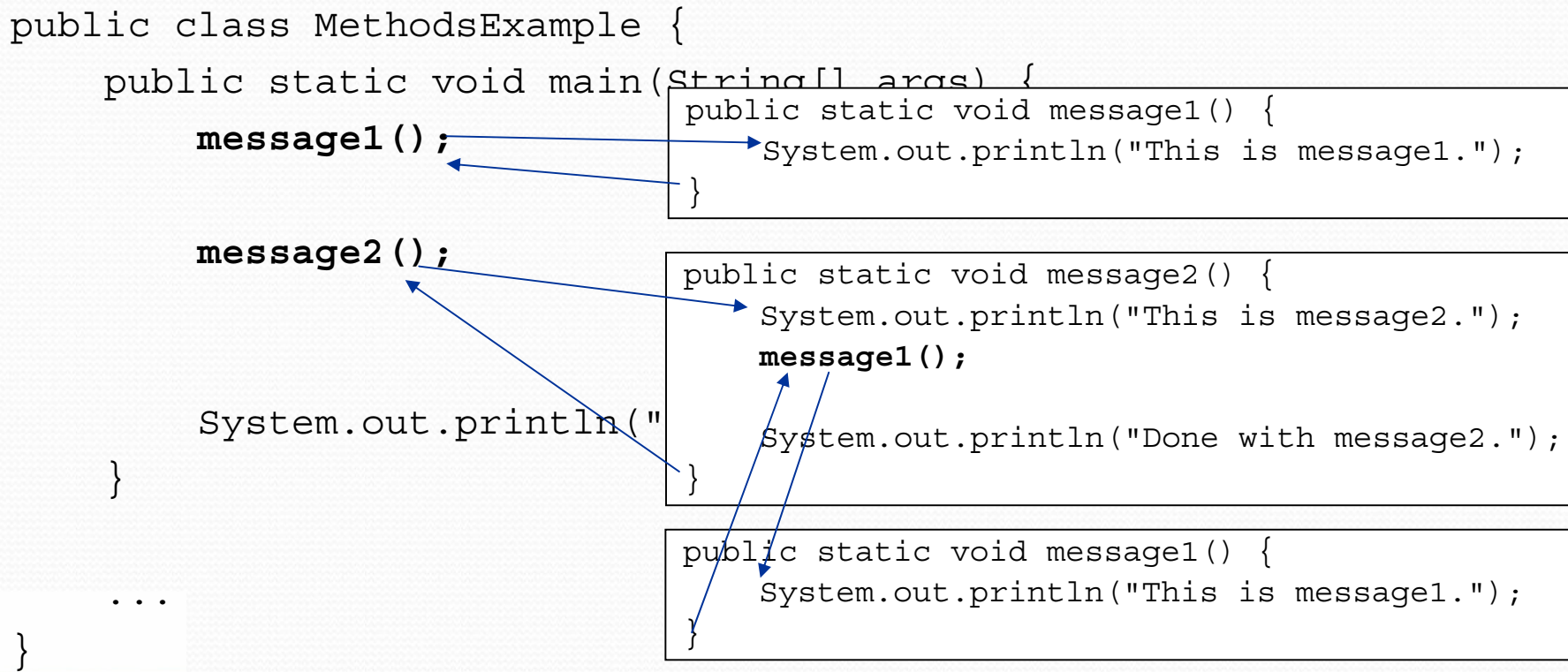
- Output:

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



# 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.



# Java's primitive types (2.1)

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

<b>Name</b>	<b>Description</b>	<b>Examples</b>
<code>int</code>	integers	<code>42, -3, 0, 926394</code>
<code>double</code>	real numbers	<code>3.1, -0.25, 9.4e3</code>
<code>char</code>	single text characters	<code>'a', 'X', '?', '\n'</code>
<code>boolean</code>	logical values	<code>true, false</code>

- Why does Java distinguish integers vs. real numbers?



# Expressions

- **expression:** A value or operation that computes a value.
  - Examples:  $1 + 4 * 5$   
 $(7 + 2) * 6 / 3$   
42
  - The simplest expression is a *literal value*.
  - A complex expression can use operators and parentheses.

# Integer division with /

- When we divide integers, the quotient is also an integer.
  - $14 / 4$  is 3, not 3.5

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

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

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

- More examples:

- $32 / 5$  is 6
- $84 / 10$  is 8
- $156 / 100$  is 1

- Dividing by 0 causes an error when your program runs.

# Integer remainder with %

- The % operator computes the remainder from integer division.

- $14 \% 4$  is 2

- $218 \% 5$  is 3

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

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

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



# Precedence

- **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  $11$

# 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

# Variables (2.2)

- **variable**: A piece of the computer's memory that is given a name and type, and can store a value.
- A variable can be declared/initialized in one statement.
- Syntax:

**type name = value;**

- `double myGPA = 3.95;`

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

x	14
---	----

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



# Type casting

- **type cast:** A conversion from one type to another.
  - To promote an `int` into a `double` to get exact division from `/`
  - 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
```

# Increment and decrement

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

## Shorthand

**variable**++;

**variable**--;

```
int x = 2;
```

```
x++;
```

```
double gpa = 2.5;
```

```
gpa--;
```

## Equivalent longer version

**variable** = **variable** + 1;

**variable** = **variable** - 1;

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

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

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

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

# Modify-and-assign operators

*shortcuts to modify a variable's value*

## Shorthand

**variable += value;**

**variable -= value;**

**variable \*= value;**

**variable /= value;**

**variable %= value;**

## Equivalent longer version

**variable = variable + value;**

**variable = variable - value;**

**variable = variable \* value;**

**variable = variable / value;**

**variable = variable % value;**

```
x += 3;
```

```
gpa -= 0.5;
```

```
number *= 2;
```

```
// x = x + 3;
```

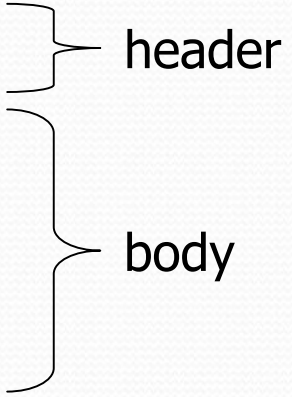
```
// gpa = gpa - 0.5;
```

```
// number = number * 2;
```



# for loops (2.3)

```
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 **statements**.
  - Perform the **update**.

# 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

# Nested loops

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

```
for (int i = 1; i <= 4; i++) {  
    for (int j = 1; j <= 5; j++) {  
        System.out.print((i * j) + "\t");  
    }  
    System.out.println(); // to end the line  
}
```

- **Output:**

1	2	3	4	5
2	4	6	8	10
3	6	9	12	15
4	8	12	16	20

- Statements in the outer loop's body are executed 4 times.
  - The inner loop prints 5 numbers each time it is run.



# Variable 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
```



x's scope

# Class constants (2.4)

- **class constant:** A value visible to the whole program.
  - value can only be set at declaration
  - value can't be changed while the program is running

- Syntax:

```
public static final type name = value;
```

- name is usually in ALL\_UPPER\_CASE

- Examples:

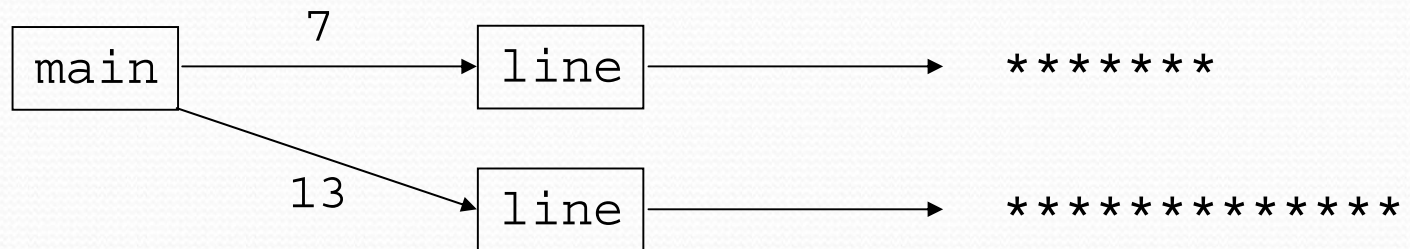
```
public static final int DAYS_IN_WEEK = 7;
```

```
public static final double INTEREST_RATE = 3.5;
```

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

# Parameters (3.1)

- **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.





# Passing parameters

- Declaration:

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

- Call:

```
methodName (value, value, ..., value);
```

- Example:

```
public static void main(String[] args) {  
    sayPassword(42);           // The password is: 42  
    sayPassword(12345);       // The password is: 12345  
}  
  
public static void sayPassword(int code) {  
    System.out.println("The password is: " + code);  
}
```

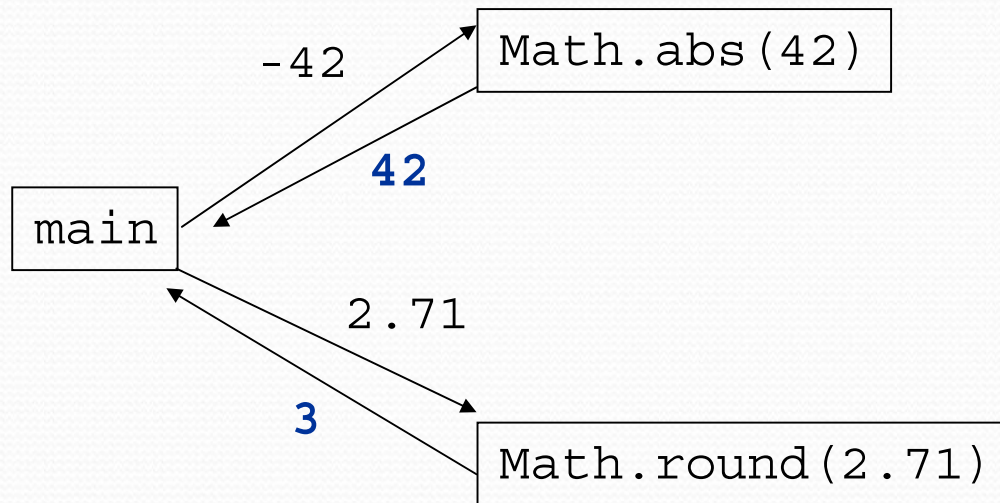
# Java's Math class (3.2)

Method name	Description
<code>Math.abs(<i>value</i>)</code>	absolute value
<code>Math.round(<i>value</i>)</code>	nearest whole number
<code>Math.ceil(<i>value</i>)</code>	rounds up
<code>Math.floor(<i>value</i>)</code>	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.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
<code>Math.random()</code>	random double between 0 and 1

Constant	Description
<code>Math.E</code>	2.7182818...
<code>Math.PI</code>	3.1415926...

# Return (3.2)

- **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.





# Returning a value

```
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;  
}
```

# Strings (3.3)

- **string**: An object storing a sequence of text characters.

```
String name = "text";
```

```
String name = expression;
```

- Characters of a string are numbered with 0-based *indexes*:

```
String name = "P. Diddy";
```

index	0	1	2	3	4	5	6	7
char	P	.		D	i	d	d	y

- The first character's index is always 0
- The last character's index is 1 less than the string's length
- The individual characters are values of type `char`

# 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 it is not there)
<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> 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 gangsta = "Dr. Dre";  
System.out.println(gangsta.length());    // 7
```



# 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("Dr. ")) {
    System.out.println("Are you single?");
} else if (name.equalsIgnoreCase("LUMBERG")) {
    System.out.println("I need your TPS reports.");
}
```

# 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("I love you, you love me,");
    System.out.println("We're a happy family!");
}
```

- Technically this is a method that returns a value of type `boolean`, the type used in logical tests.



# Type char (4.4)

- `char` : A primitive type representing single characters.
  - Each character inside a `String` is stored as a `char` value.
  - Literal `char` values are surrounded with apostrophe (single-quote) marks, such as `'a'` or `'4'` or `'\n'` or `'\''`
  - It is legal to have variables, parameters, returns of type `char`

```
char letter = 'S';  
System.out.println(letter);           // S
```

- `char` values can be concatenated with strings.

```
char initial = 'P';  
System.out.println(initial + " Diddy"); // P Diddy
```



# char VS. String

- "h" is a String  
'h' is a char (the two behave differently)

- 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'
```

- char is primitive; you can't call methods on it

```
char c = 'h';  
c = c.toUpperCase();          // ERROR: "cannot be dereferenced"
```

- What is `s + 1`? What is `c + 1`?
- What is `s + s`? What is `c + c`?

# System.out.printf (4.4)

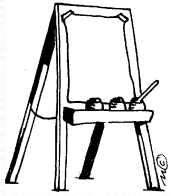
```
System.out.printf("format string", parameters);
```

- A format string contains *placeholders* to insert parameters into it:
  - %d an integer
  - %f a real number
  - %s a string
  - %8d an integer, 8 characters wide, right-aligned
  - %-8d an integer, 8 characters wide, left-aligned
  - %.4f a real number, 4 characters after decimal
  - %6.2f a real number, 6 characters wide, 2 after decimal

- Example:

```
int x = 3, y = 2;  
System.out.printf("(%d, %d)\n", x, y); // (3, 2)  
System.out.printf("%4d %4.2f\n", x, y); // 3 2.00
```

# DrawingPanel (3G)



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

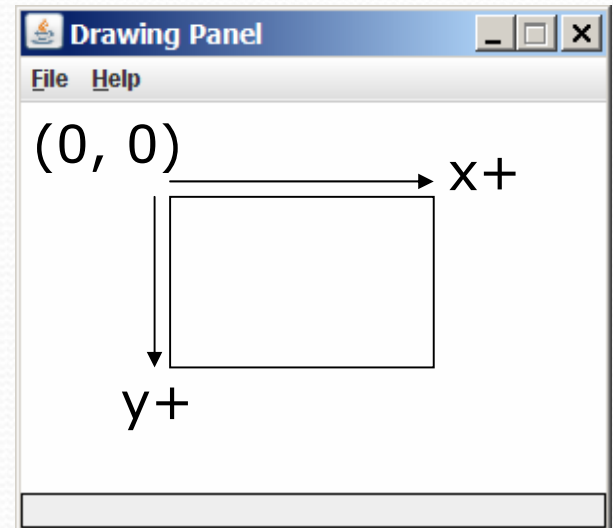
- To create a window:

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

Example:

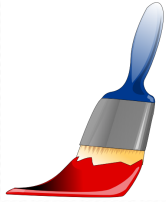
```
DrawingPanel panel = new DrawingPanel(300, 200);
```

- The window has nothing on it.
  - We can draw shapes and lines on it using another object of type Graphics.





# Graphics



*"Pen" objects that can draw lines and shapes*

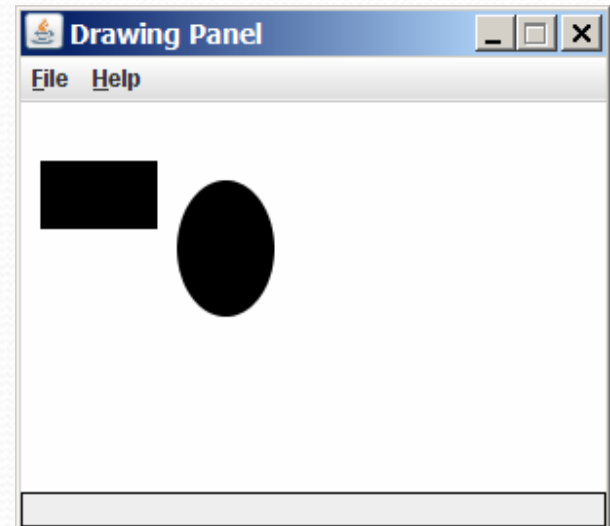
- Access it by calling `getGraphics` on your `DrawingPanel`.

```
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);
```



# Graphics methods

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

# Color



- Create one using Red-Green-Blue (RGB) values from 0-255

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

- Example:

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

- Or use a predefined `Color` class constant (more common)

```
Color.CONSTANT_NAME
```

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

- BLACK, BLUE, CYAN, DARK\_GRAY, GRAY,  
GREEN, LIGHT\_GRAY, MAGENTA, ORANGE,  
PINK, RED, WHITE, or YELLOW



# Scanner (3.3)

- `System.out`
  - An object with methods named `println` and `print`
- `System.in`
  - not intended to be used directly
  - We use a second object, from a class `Scanner`, to help us.
- 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 a token of user input as an <code>int</code>
<code>nextDouble()</code>	reads a token of user input as a <code>double</code>
<code>next()</code>	reads a token of user input as a <code>String</code>
<code>nextLine()</code>	reads a <i>line</i> of user input as a <code>String</code>

- Each method waits until the user presses Enter.
  - The value typed is returned.

```
System.out.print("How old are you? "); // prompt
int age = console.nextInt();
System.out.println("You'll be 40 in " +
    (40 - age) + " years.");
```

- **prompt:** A message telling the user what input to type.



# Testing for valid input (5.3)

- Scanner methods to see what the next token will be:

Method	Description
<code>hasNext ()</code>	returns <code>true</code> if there are any more tokens of input to read <i>(always true for console input)</i>
<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>
<code>hasNextLine ()</code>	returns <code>true</code> if there are any more lines of input to read <i>(always true for console input)</i>

- These methods do not consume input; they just give information about the next token.
  - Useful to see what input is coming, and to avoid crashes.



# Cumulative sum (4.1)

- A loop that adds the numbers from 1-1000:

```
int sum = 0;
for (int i = 1; i <= 1000; i++) {
    sum = sum + i;
}
System.out.println("The sum is " + sum);
```

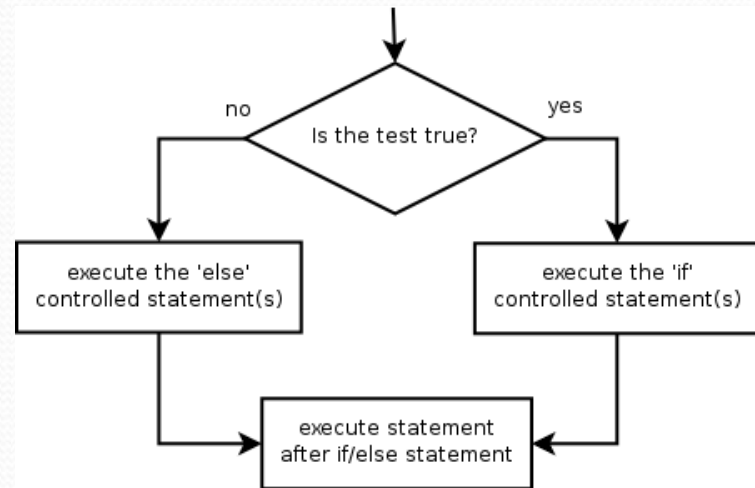
## Key idea:

- Cumulative sum variables must be declared *outside* the loops that update them, so that they will exist after the loop.

# if/else (4.2)

*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

- A **test** in an `if` is the same as in a `for` loop.

```
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
==	equals	1 + 1 == 2	true
!=	does not equal	3.2 != 2.5	true
<	less than	10 < 5	false
>	greater than	10 > 5	true
<=	less than or equal to	126 <= 100	false
>=	greater than or equal to	5.0 >= 5.0	true



# Logical operators: `&&`, `||`, `!`

- Conditions can be combined using *logical operators*:

Operator	Description	Example	Result
<code>&amp;&amp;</code>	and	<code>(2 == 3) &amp;&amp; (-1 &lt; 5)</code>	false
<code>  </code>	or	<code>(2 == 3)    (-1 &lt; 5)</code>	true
<code>!</code>	not	<code>!(2 == 3)</code>	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

# Type boolean (5.2)

- **boolean**: A logical type whose values are `true` and `false`.
  - A **test** in an `if`, `for`, or `while` is a boolean expression.
  - You can create boolean variables, pass boolean parameters, return boolean values from methods, ...

```
boolean minor = (age < 21);
boolean expensive = iPhonePrice > 200.00;
boolean iLoveCS = true;

if (minor) {
    System.out.println("Can't purchase alcohol!");
}
if (iLoveCS || !expensive) {
    System.out.println("Buying an iPhone");
}
```

# De Morgan's Law

- **De Morgan's Law:**

Rules used to *negate* or *reverse* boolean expressions.

- Useful when you want the opposite of a known boolean 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:

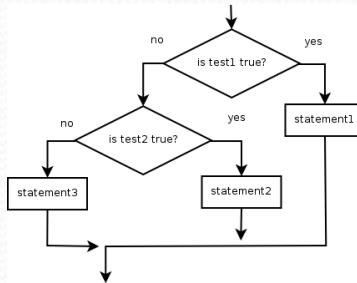
Original Code	Negated Code
<pre>if (x == 7 &amp;&amp; y &gt; 3) {     ... }</pre>	<pre>if (x != 7    y &lt;= 3) {     ... }</pre>



# if/else Structures

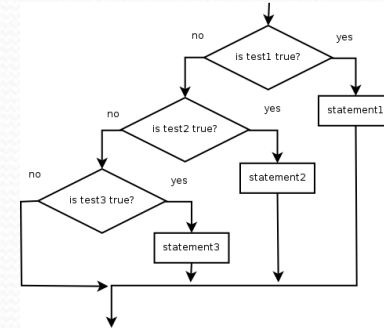
- Exactly 1 path: (mutually exclusive)

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



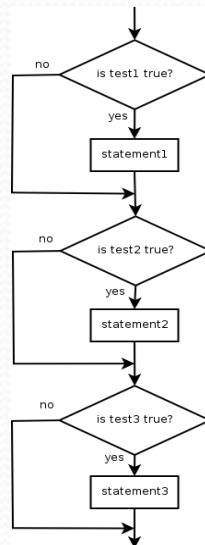
- 0 or 1 path:

```
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);  
}
```



# Fencepost loops (4.1)

- **fencepost problem:** When we want to repeat two tasks, one of them  $n$  times, another  $n-1$  or  $n+1$  times.
  - Add a statement outside the loop to place the initial "post."
  - Also called a *fencepost loop* or a "loop-and-a-half" solution.
- Algorithm template:

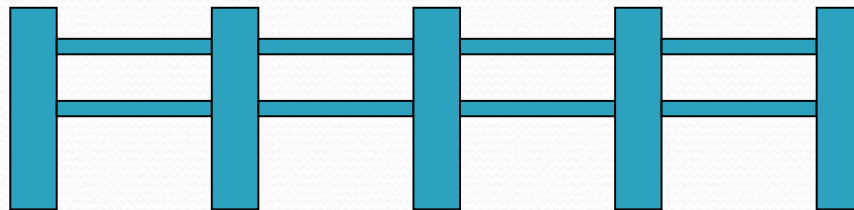
***place a post.***

***for (length of fence - 1) {***

***place some wire.***

***place a post.***

***}***



# Fencepost method solution

- 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
```

- 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  
}
```



# while loops (5.1)

- **while loop:** Repeatedly executes its body as long as a logical test is true.

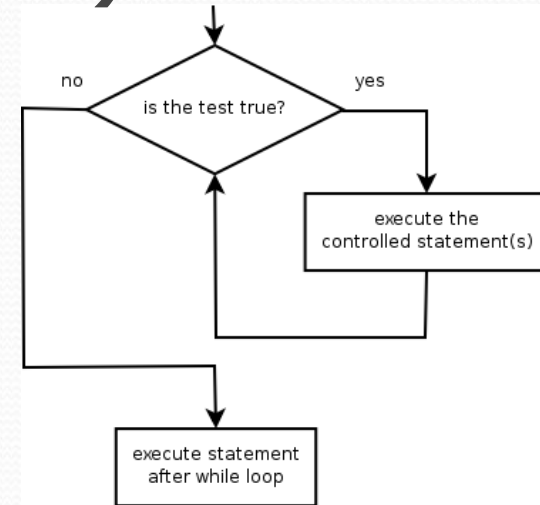
```
while (test) {  
    statement(s);  
}
```

- Example:

```
int num = 1;  
while (num <= 200) {  
    System.out.print(num + " ");  
    num = num * 2;  
}
```

- OUTPUT:

1 2 4 8 16 32 64 128



```
// initialization  
// test  
  
// update
```

# do/while loops (5.4)

- **do/while loop:** Executes statements repeatedly while a condition is `true`, testing it at the *end* of each repetition.

```
do {  
    statement(s);  
} while (test);
```

- Example:

```
// prompt until the user gets the right password  
String phrase;  
do {  
    System.out.print("Password: ");  
    phrase = console.next();  
} while (!phrase.equals("abracadabra"));
```

# The Random class (5.1)

- A Random object generates pseudo-random\* numbers.
  - Class Random is found in the `java.util` package.

```
import java.util.*;
```

Method name	Description
<code>nextInt()</code>	returns a random integer
<code>nextInt(max)</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
```



# "Boolean Zen"

- Students new to `boolean` often test if a result is `true`:

```
if (bothOdd(7, 13) == true) {    // bad
    ...
}
```

- But this is unnecessary and redundant. Preferred:

```
if (bothOdd(7, 13)) {          // good
    ...
}
```

- A similar pattern can be used for a `false` test:

```
if (bothOdd(7, 13) == false) { // bad
if (!bothOdd(7, 13)) {         // good
```

# "Boolean Zen", part 2

- Methods that return boolean often have an if/else that returns true or false:

```
public static boolean bothOdd(int n1, int n2) {  
    if (n1 % 2 != 0 && n2 % 2 != 0) {  
        return true;  
    } else {  
        return false;  
    }  
}
```

- Observation: The if/else is unnecessary.
  - Our logical test is itself a boolean value; so return that!

```
public static boolean bothOdd(int n1, int n2) {  
    return (n1 % 2 != 0 && n2 % 2 != 0);  
}
```

# break (5.4)

- **break** statement: Immediately exits a loop.
  - Can be used to write a loop whose test is in the middle.
  - Such loops are often called "*forever*" loops because their header's boolean test is often changed to a trivial `true`.

```
while (true) {  
    statement(s);  
    if (test) {  
        break;  
    }  
    statement(s);  
}
```

- Some programmers consider `break` to be bad style.



# Reading files (6.1)

- To read a file, pass a `File` when constructing a `Scanner`.

```
Scanner name = new Scanner(new File("file name"));
```

Example:

```
File file = new File("mydata.txt");
```

```
Scanner input = new Scanner(file);
```

or, better yet:

```
Scanner input = new Scanner(new File("mydata.txt"));
```

# The throws clause

- **throws clause:** Keywords on a method's header that state that it may generate an exception.

- Syntax:

```
public static type name (params) 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 it happens."*

# Input tokens (6.2)

- **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 `numbers.txt` that contains this text:

```
308.2
  14.9 7.4 2.8
3.9 4.7 -15.4
  2.8
```

- A Scanner views all input as a stream of characters:

```
308.2\n 14.9 7.4 2.8\n\n3.9 4.7 -15.4\n 2.8\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.

```
308.2\n 14.9 7.4 2.8\n\n3.9 4.7 -15.4\n 2.8\n^
```

```
double x = input.nextDouble(); // 308.2
```

```
308.2\n 14.9 7.4 2.8\n\n3.9 4.7 -15.4\n 2.8\n^
```

```
String s = input.next(); // "14.9"
```

```
308.2\n 14.9 7.4 2.8\n\n3.9 4.7 -15.4\n 2.8\n^
```

# Scanner exceptions

- `InputMismatchException`
  - You read the wrong type of token (e.g. read "hi" as `int`).
- `NoSuchElementException`
  - You read past the end of the input.
- 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 CountTokens.sillyMethod(CountTokens.java:19)
    at CountTokens.main(CountTokens.java:6)
```



# Output to files (6.4)

- **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("file name"));
```

**Example:**

```
PrintStream output = new PrintStream(new File("out.txt"));  
output.println("Hello, file!");  
output.println("This is a second line of output.");
```

# System.out and PrintStream

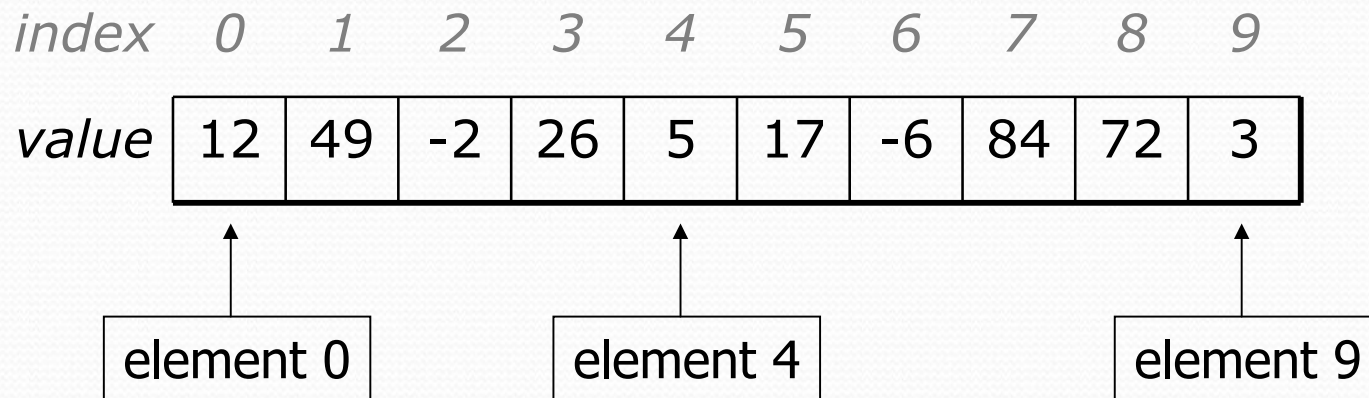
- The console output object, `System.out`, is a `PrintStream`.

```
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` as a parameter to a method expecting a `PrintStream`.
  - Allows methods that can send output to the console or a file.

# Arrays (7.1)

- **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.





# Array declaration

**type** [] **name** = new **type** [**length**] ;

- Example:

```
int [] numbers = new int [10] ;
```

*index*    0    1    2    3    4    5    6    7    8    9

<i>value</i>	0	0	0	0	0	0	0	0	0	0
--------------	---	---	---	---	---	---	---	---	---	---

# 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.");  
}
```

<i>index</i>	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

# 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



# 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().

# 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 figures out the size by counting the values.

# The Arrays class

- Class `Arrays` in package `java.util` has useful static methods for manipulating arrays:

Method name	Description
<code>binarySearch(array, value)</code>	returns the index of the given value in a sorted array (< 0 if not found)
<code>equals(array1, array2)</code>	returns <code>true</code> if the two arrays contain the same elements in the same order
<code>fill(array, value)</code>	sets every element in the array to have the given value
<code>sort(array)</code>	arranges the elements in the array into ascending order
<code>toString(array)</code>	returns a string representing the array, such as "[10, 30, 17]"



# Arrays as parameters

- Declaration:

```
public static type methodName(type[] name) {
```

- Example:

```
public static double average(int[] numbers) {  
    ...  
}
```

- Call:

```
methodName(arrayName);
```

- Example:

```
int[] scores = {13, 17, 12, 15, 11};  
double avg = average(scores);
```

# Arrays as return

- Declaring:

```
public static type[] methodName(parameters) {
```

- Example:

```
public static int[] countDigits(int n) {  
    int[] counts = new int[10];  
    ...  
    return counts;  
}
```

- Calling:

```
type[] name = methodName(parameters);
```

- Example:

```
public static void main(String[] args) {  
    int[] tally = countDigits(229231007);  
    System.out.println(Arrays.toString(tally));  
}
```

# Value semantics (primitives)

- **value semantics:** Behavior where values are copied when assigned to each other or passed as parameters.
  - When one primitive 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
```

x

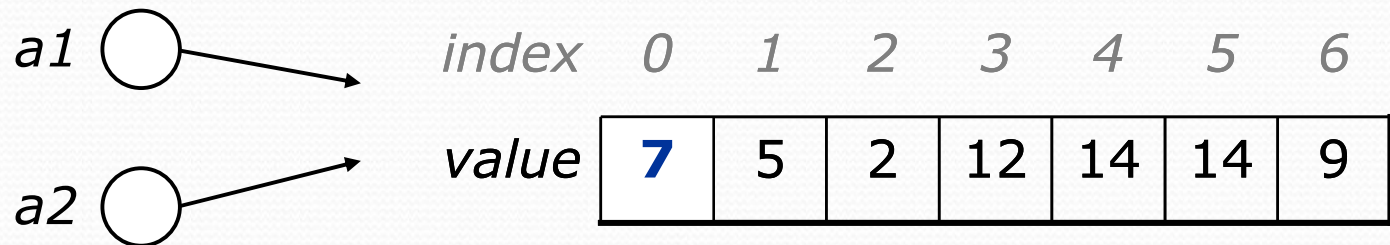
y



# Reference semantics (objects)

- **reference semantics:** Behavior where variables actually store the address of an object in memory.
  - When one reference 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.

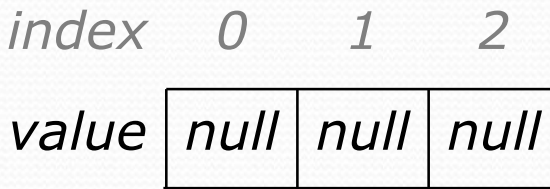
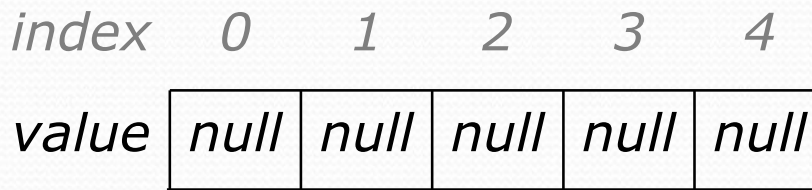
```
int[] a1 = {4, 5, 2, 12, 14, 14, 9};  
int[] a2 = a1;      // refer to same array as a1  
a2[0] = 7;  
System.out.println(a1[0]); // 7
```



# Null

- `null` : A reference that does not refer to any object.
  - Fields of an object that refer to objects are initialized to `null`.
  - The elements of an array of objects are initialized to `null`.

```
String[] words = new String[5];  
DrawingPanel[] windows = new DrawingPanel[3];
```





# Null pointer exception

- **dereference:** To access data or methods of an object with the dot notation, such as `s.length()`.
  - It is illegal to dereference `null` (causes an exception).
  - `null` is not any object, so it has no methods or data.

```
String[] words = new String[5];  
System.out.println("word is: " + words[0]);  
words[0] = words[0].toUpperCase();
```

## Output:

```
word is: null
```

```
Exception in thread "main"  
java.lang.NullPointerException  
    at Example.main(Example.java:8)
```



# Classes and objects (8.1)

- **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.
- **object:** An entity that combines state and behavior.
  - **object-oriented programming (OOP):** Programs that perform their behavior as interactions between objects.

# Fields (8.2)

- **field**: A variable inside an object that is part of its state.
  - Each object has *its own copy* of each field.
  - **encapsulation**: Declaring fields `private` to hide their data.
- Declaration syntax:

```
private type name;
```

- Example:

```
public class Student {  
    private String name;           // each object now has  
    private double gpa;           // a name and gpa field  
}
```

# Instance methods

- **instance method:** One that exists inside each object of a class and defines behavior of that object.

```
public type name (parameters) {  
    statements;  
}
```

- same syntax as static methods, but without `static` keyword

Example:

```
public void shout() {  
    System.out.println("HELLO THERE!");  
}
```



# A Point class

```
public class Point {  
    private int x;  
    private 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 data fields named `x` and `y`.
- Each `Point` object contains a method named `draw` that draws that point at its current `x/y` position.

# 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.

# Kinds of methods

- Instance methods take advantage of an object's state.
  - Some methods allow clients to access/modify its state.
- **accessor**: A method that lets clients examine object state.
  - Example: A `distanceFromOrigin` method that tells how far a `Point` is away from (0, 0).
  - Accessors often have a non-void return type.
- **mutator**: A method that modifies an object's state.
  - Example: A `translate` method that shifts the position of a `Point` by a given amount.



# Constructors (8.4)

- **constructor:** Initializes the state of new objects.

```
public type (parameters) {  
    statements;  
}
```

- Example:

```
public Point(int initialX, int initialY) {  
    x = initialX;  
    y = initialY;  
}
```

- runs when the client uses the `new` keyword
- does not specify a return type; implicitly returns a new object
- If a class has no constructor, Java gives it a *default constructor* with no parameters that sets all fields to 0.

# toString method (8.6)

- tells Java how to convert an object into a String

```
public String toString() {  
    code that returns a suitable String;  
}
```

- Example:

```
public String toString() {  
    return "(" + x + ", " + y + ")";  
}
```

- called when an object is printed/concatenated to a String:

```
Point p1 = new Point(7, 2);  
System.out.println("p1: " + p1);
```

- Every class has a toString, even if it isn't in your code.
  - Default is class's name and a hex number: Point@9e8c34

# this keyword (8.7)

- **this** : A reference to the implicit parameter.
  - *implicit parameter*: object on which a method is called
- Syntax for using `this`:
  - To refer to a field:  
`this.field`
  - To call a method:  
`this.method (parameters) ;`
  - To call a constructor from another constructor:  
`this (parameters) ;`



# Static methods

- **static method:** Part of a class, not part of an object.
  - shared by all objects of that class
  - good for code related to a class but not to each object's state
  - does not understand the *implicit parameter*, `this`; therefore, cannot access an object's fields directly
  - if `public`, can be called from inside or outside the class
- Declaration syntax:

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

# Inheritance (9.1)

- **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 (9.1)

```
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)



# Overriding methods (9.1)

- **override:** To write a new version of a method in a subclass that replaces the superclass's version.
  - No special syntax required to override a superclass method. Just write a new version of it in the subclass.

```
public class Secretary extends Employee {  
    // overrides getVacationForm in Employee class  
    public String getVacationForm() {  
        return "pink";  
    }  
    ...  
}
```

# super keyword (9.3)

- 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;
    }
    ...
}
```



# Polymorphism

- **polymorphism:** Ability for the same code to be used with different types of objects and behave differently with each.
  - Example: `System.out.println` can print any type of object.
    - Each one displays in its own way on the console.

- A variable of type *T* can hold an object of any subclass of *T*.

```
Employee ed = new LegalSecretary();
```

- You can call any methods from `Employee` on `ed`.
- You can *not* call any methods specific to `LegalSecretary`.
- When a method is called, it behaves as a `LegalSecretary`.

```
System.out.println(ed.getSalary()); // 55000.0  
System.out.println(ed.getVacationForm()); // pink
```