

<https://introc.cs.princeton.edu>

## 1.2 BUILT-IN DATA TYPES

---

- ▶ *strings*
- ▶ *integers*
- ▶ *floating-point numbers*
- ▶ *booleans*
- ▶ *type conversion*

# Questions during (or after) lecture

---



raise your hand and ask



ask on Ed



attend office hours (or stay after lecture)



# Built-in data types

---

A **data type (type)** is a set of values and a set of operations on those values.

| type                      | set of values                  | example values                     | examples of operations  |
|---------------------------|--------------------------------|------------------------------------|---|
| <i>String</i>             | <i>sequences of characters</i> | "Hello, World"<br>"COS 125 is fun" | <i>concatenate</i>  |
| <i>int</i><br><i>long</i> | <i>integers</i>                | 17<br>-12345L                      | <i>add, subtract, multiply, divide,</i><br><i>compare, equality</i> |
| <i>double</i>             | <i>floating-point numbers</i>  | 2.5<br>-0.125                      | <i>add, subtract, multiply, divide,</i><br><i>compare, equality</i> |
| <i>boolean</i>            | <i>truth values</i>            | true<br>false                      | <i>and, or, not,</i><br><i>equality</i>                             |

Java's built-in data types  
(that we use regularly in this course)

# Programming terminology

---

**Program.** Sequence of statements. ← *for now*

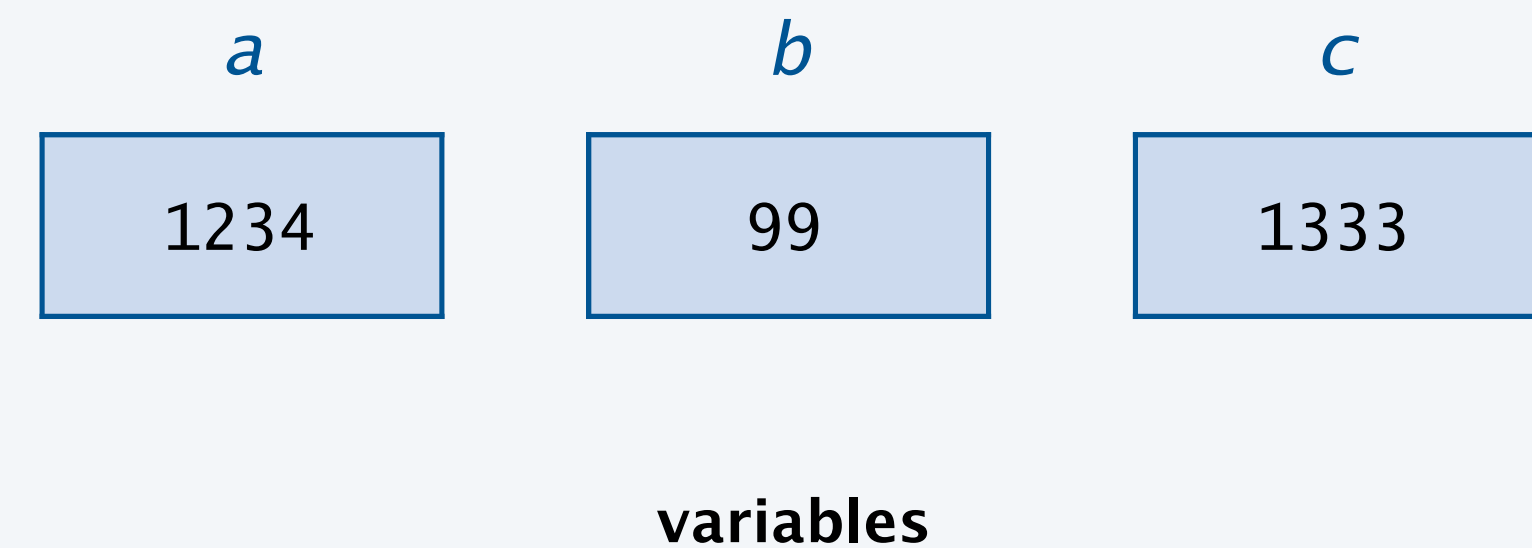
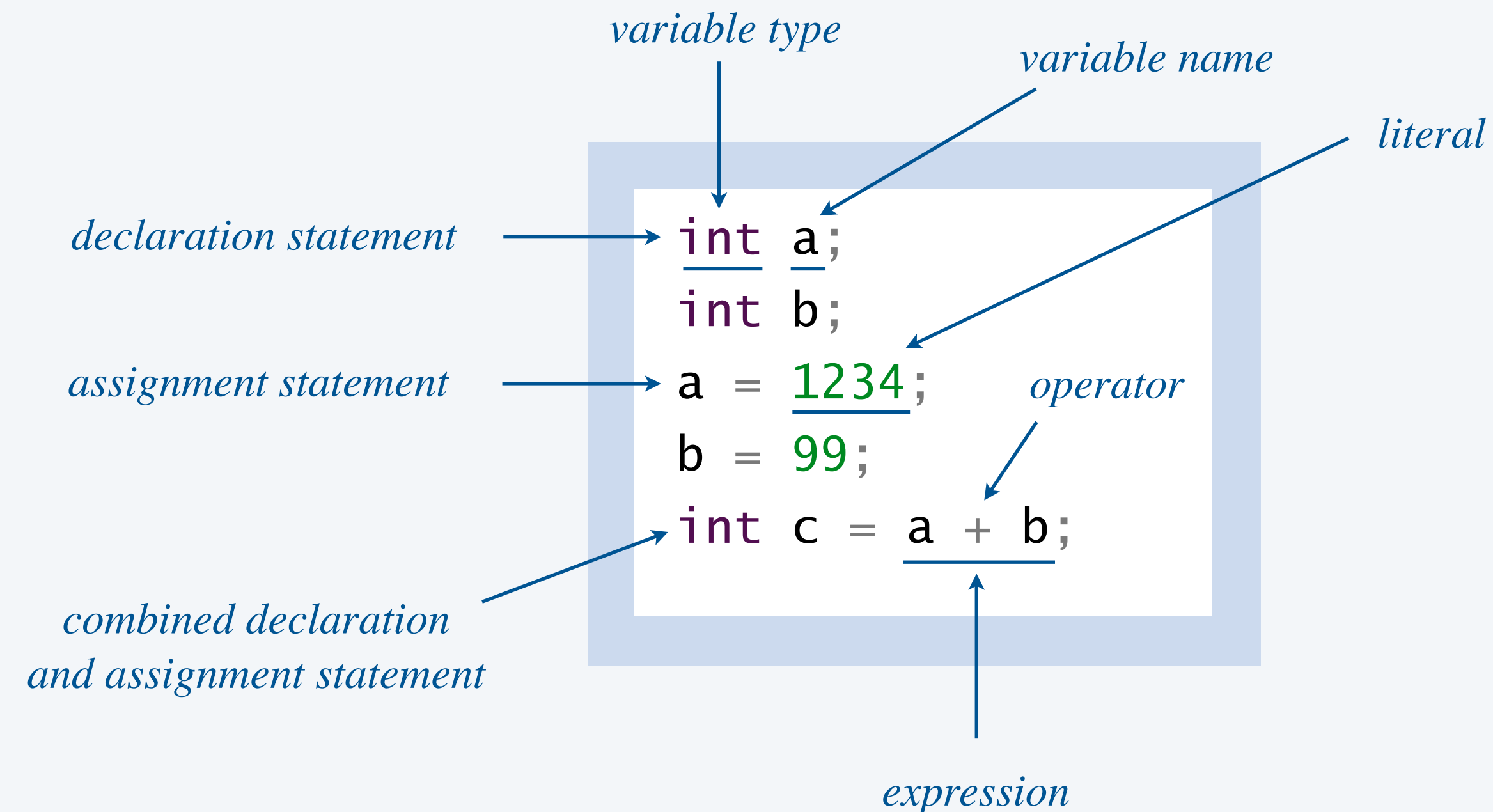
**Declaration statement.** Associates a variable with a name and type.

**Variable.** A storage location for a data-type value.

**Assignment statement.** Stores a value in a variable.

**Literal.** Programming-language representation of a data-type value.

**Expression.** A combination of variable names, literals, operators, etc. that evaluates to a value.

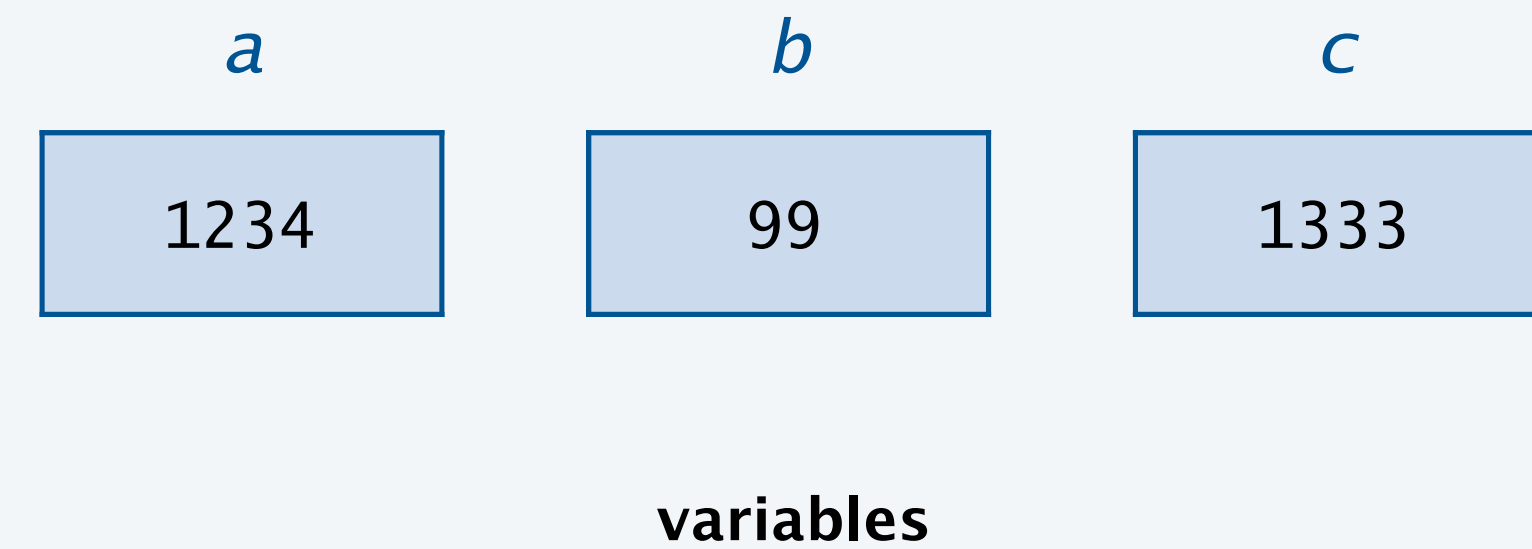
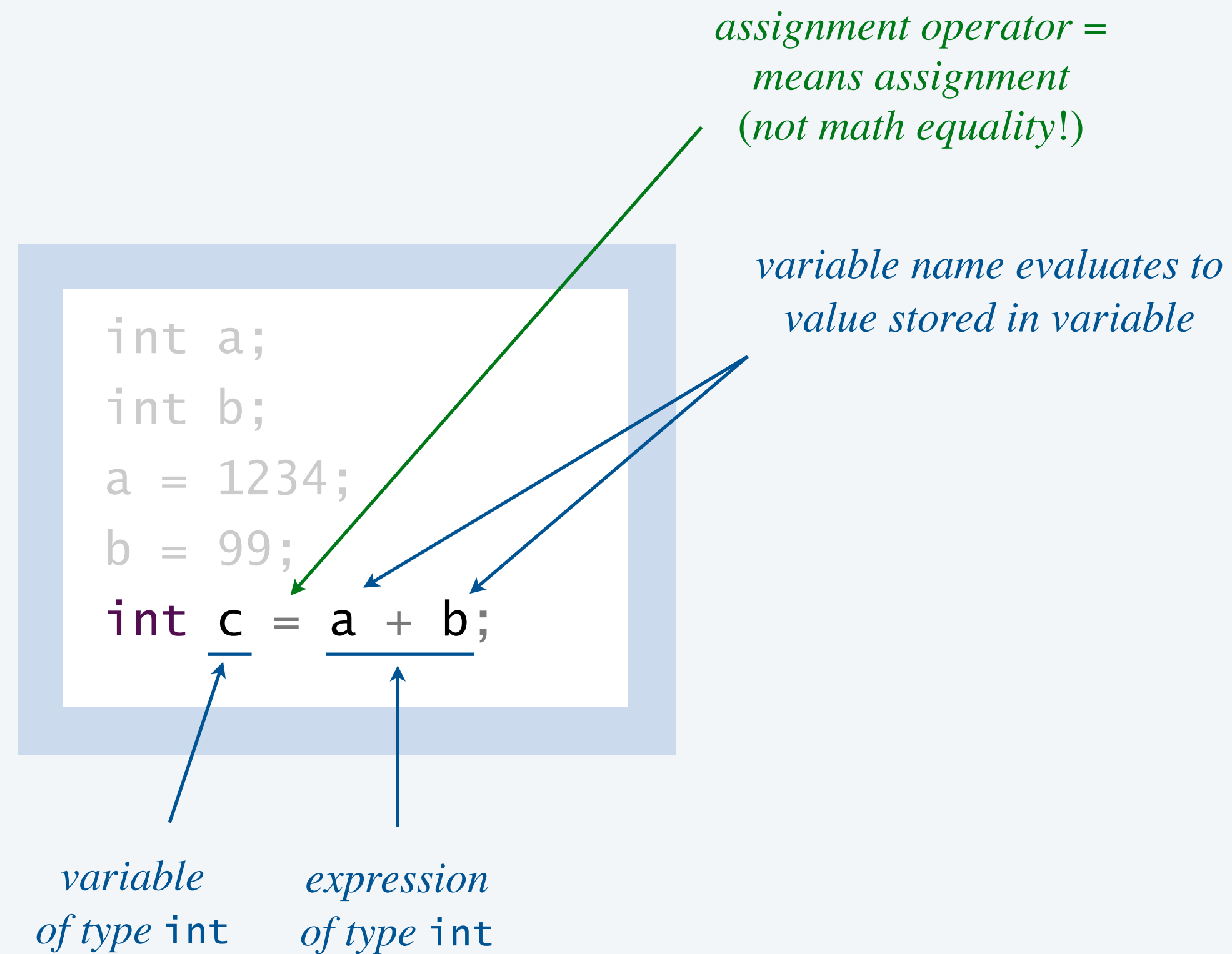


# Assignment statements

Q. How does an assignment statement work?

A. Java evaluates the **expression on the RHS** and assigns that value to the **variable on the LHS**.

↑  
*expression type must be compatible with variable type*



# Valid and invalid assignment statements

---

Q. Which of these independent code fragments are valid?

| statements                            | compiles? | remark  |
|---------------------------------------|-----------|---|
| <pre>int a = 1; 123 = a;</pre>        | 😞         | <i>LHS is not a variable<br/>(= does not mean math equality)</i>                      |
| <pre>double a = 2.5; int b = a;</pre> | 😞         | <i>RHS type is incompatible with LHS type</i>   |
| <pre>String s = 123;</pre>            | 😞         | <i>RHS type is incompatible with LHS type</i>   |
| <pre>int b = 2; int a = 3 * b;</pre>  | 😍         | <i>RHS can be an expression</i>   |
| <pre>int a = 3; a = 2 * a;</pre>      | 😍         | <i>a variable can be reassigned<br/>(that's why it's called a variable!)</i>          |
| <pre>int a = 2 * a;</pre>             | 😞         | <i>a variable must be assigned a value<br/>before it can be used in an expression</i> |

# Tracing the execution of a program



Q. What does this code fragment do?

A. Let's **trace** the variables during execution of the code. ← *table of variable values*

```
int a = 123;  
int b = 456;  
int temp = a;  
a = b;  
b = temp;
```

*this idiom exchanges  
the values stored in the  
variables a and b*

|                               | <i>a</i>          | <i>b</i>          | <i>temp</i>       |
|-------------------------------|-------------------|-------------------|-------------------|
| <i>start of code fragment</i> | <i>undeclared</i> | <i>undeclared</i> | <i>undeclared</i> |
| <code>int a = 123;</code>     | 123               | <i>undeclared</i> | <i>undeclared</i> |
| <code>int b = 456;</code>     | 123               | 456               | <i>undeclared</i> |
| <code>int temp = a;</code>    | 123               | 456               | 123               |
| <code>a = b;</code>           | 456               | 456               | 123               |
| <code>b = temp;</code>        | 456               | 123               | 123               |

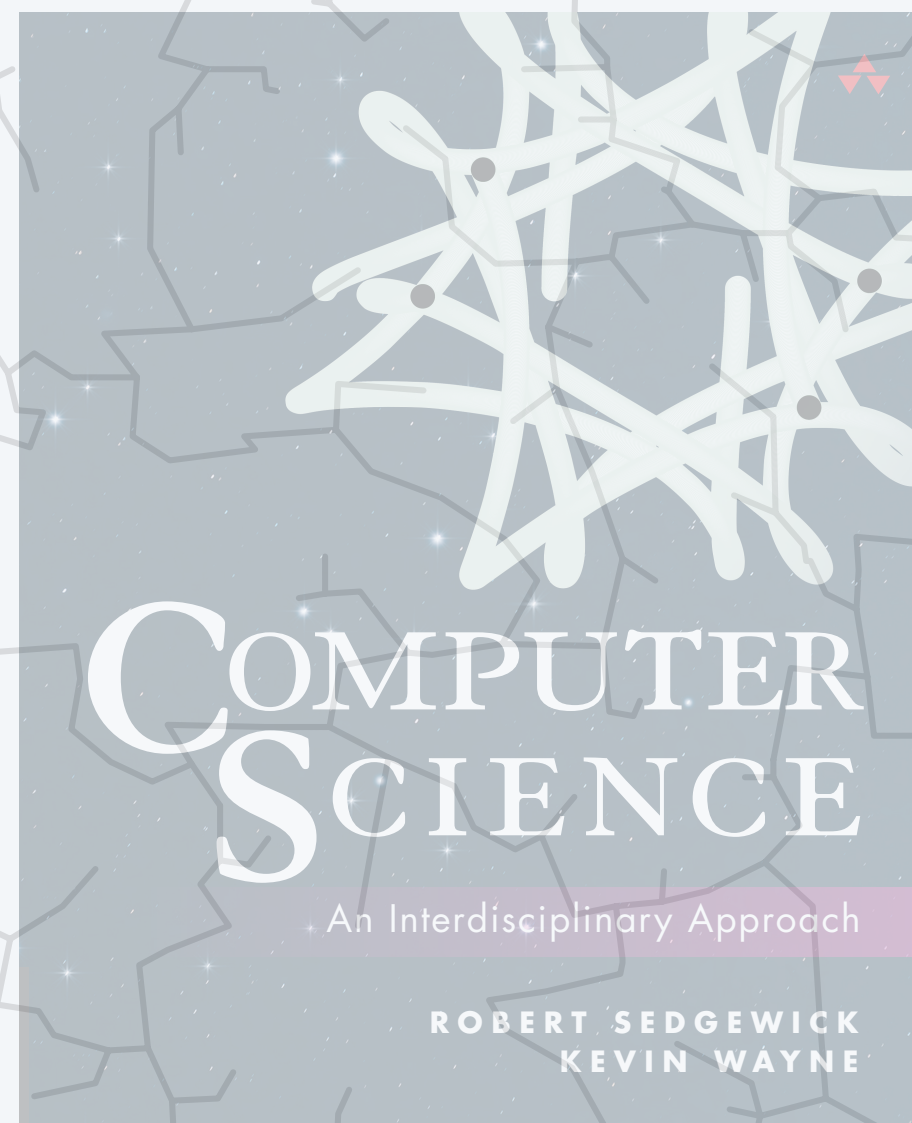
**trace of variables  
(after each statement)**



What are the values stored in the variables *a* and *b* after the code fragment is executed?

- A. 123 and 456.
- B. 456 and 123.
- C. 579 and 579.
- D. 579 and 123.
- E. Compile-time error.

```
int a = 123;  
int b = 456;  
a = a + b;  
b = a - b;  
a = a - b;
```



<https://introcs.cs.princeton.edu>

## BUILT-IN DATA TYPES

---

- ▶ *strings*
- ▶ *integers*
- ▶ *floating-point numbers*
- ▶ *booleans*
- ▶ *type conversion*

# The *String* data type

---

Typical usage. Program input and output; text processing.

|                  |                                |
|------------------|--------------------------------|
| values           | <i>sequences of characters</i> |
| typical literals | "Hi" "1234" "Nǐ hǎo" "💩💩💩"     |
| operations       | <i>concatenation</i>           |
| operator         | +                              |

| expression         | value          | remark   |
|--------------------|----------------|--|
| "My " + "Precious" | "My Precious"  | <i>spaces within a string literal matter</i>                       |
| "1234" + "99"      | "123499"       | <i>strings are not integers</i>                                    |
| "A" + "B" + "C"    | "ABC"          | <i>can concatenate several strings together, in one expression</i> |
| "ᠠᠯᠤᠯᠤ" + "ᠠᠨᠠᠨᠠ!" | "ᠠᠯᠤᠯᠤ ᠠᠨᠠᠨᠠ!" | <i>Unicode supported</i>   |



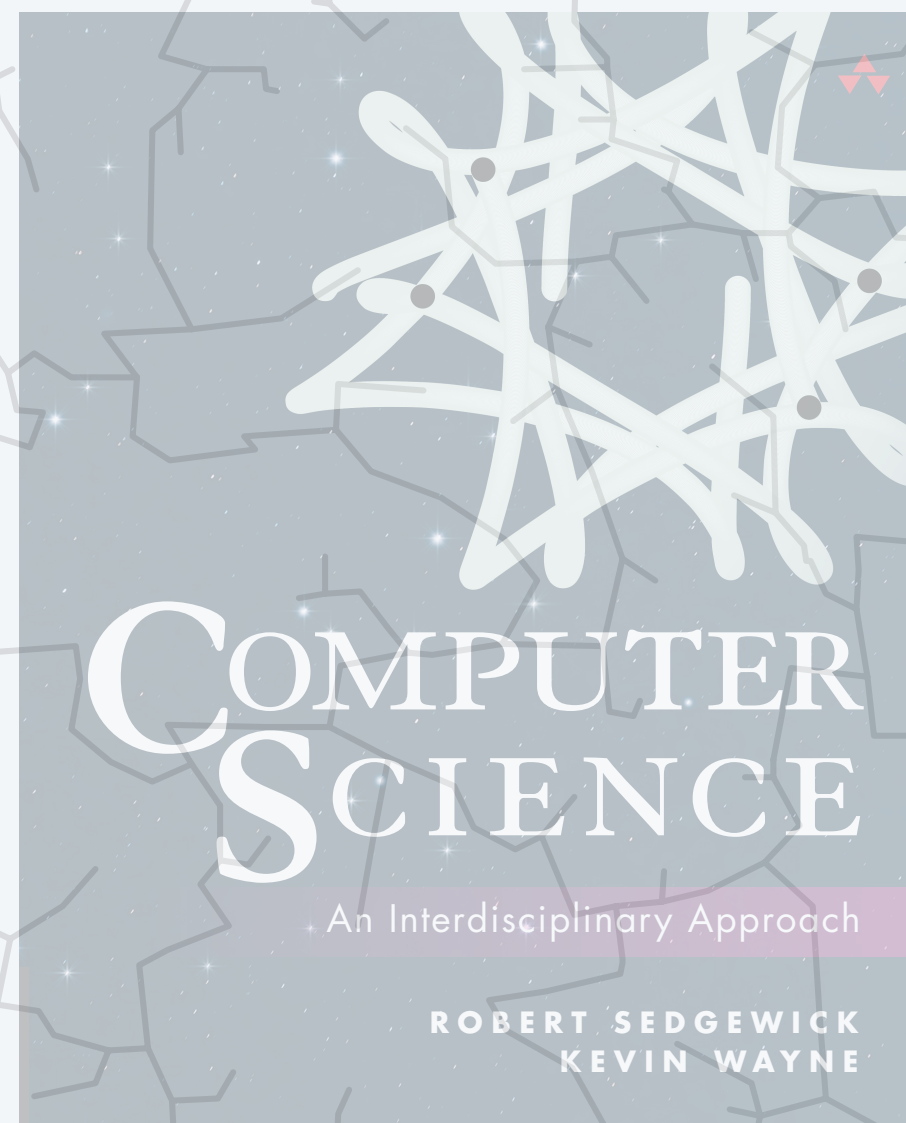
# Command-line arguments are strings

Command-line arguments. The variables `args[0]`, `args[1]`, `args[2]`, ... are of type *String*.

*we'll revisit in Section 1.4 (arrays)*

```
public class CommandLineArguments {  
    public static void main(String[] args) {  
        String a = args[0];  
        String b = args[1];  
        String c = args[2];  
        String result = a + "-" + b + "-" + c;  
        System.out.println(result);  
    }  
}
```

```
~/cos125/datatypes> java CommandLineArguments A B C  
A-B-C  
                                     ^  
                                     args[0]  
  
~/cos125/datatypes> java CommandLineArguments do re mi  
do-re-mi  
  
~/cos125/datatypes> java CommandLineArguments  
Exception in thread "main"  
java.lang.ArrayIndexOutOfBoundsException:  
Index 0 out of bounds for length 0 at  
CommandLineArguments.main(CommandLineArguments.java:3)  
                                     line number  
                                     of error  
                                     ↓
```



<https://introc.cs.princeton.edu>

## BUILT-IN DATA TYPES

---

- ▶ *strings*
- ▶ *integers*
- ▶ *floating-point numbers*
- ▶ *booleans*
- ▶ *type conversion*

# The *int* data type

Typical usage: math calculations involving integers; program control flow.

|                  |  |                 |                 |               |                  |
|------------------|--|-----------------|-----------------|---------------|------------------|
| values           | <i>integers between <math>-2^{31}</math> and <math>2^{31} - 1</math></i> |                 |                 |               |                  |
| typical literals | 1234 99 0 1000000 -3   |                 |                 |               |                  |
| operations       | <i>add</i>   | <i>subtract</i> | <i>multiply</i> | <i>divide</i> | <i>remainder</i> |
| operators        | +  | -               | *               | /             | %                |

*only  $2^{32}$  different int values  
(not quite the same as integers)*

| expression            | value       | remark                        |
|-----------------------|-------------|-------------------------------|
| 20 + 3                | 23          |                               |
| 20 - 3                | 17          |                               |
| 20 * 3                | 60          |                               |
| 20 / 3                | 6           | <i>drop fractional part</i>   |
| 20 % 3                | 2           | <i>remainder</i>              |
| 20 / 0                | -           | <i>division-by-zero error</i> |
| <u>2147483647</u> + 1 | -2147483648 | <i>integer overflow</i>       |
| $2^{31} - 1$          |             |                               |

*applying an int operator  
to two int operands  
always results in an int  
(or division-by-zero error)*

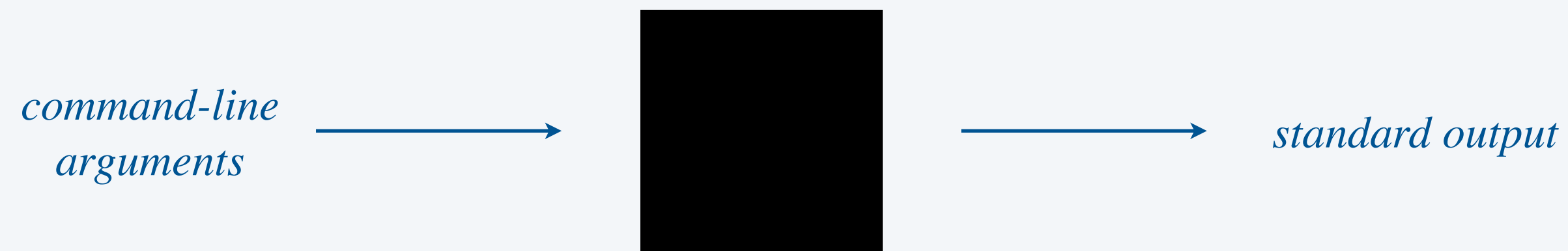
*don't use int with very large integers*

# Input and output

---

## Java I/O model. [for now]

- Read strings from the command line.
- Print strings to standard output. (Or display an image. Later: play music!)



Q. How to read integers from the command line?

A. The system method `Integer.parseInt()` converts from a `String` to an `int`.

Q. How to print integers to standard output?

A. When a `String` is concatenated with an `int`, Java converts the `int` to a `String`.

# Input and output with integers

```
public class IntOps {  
    public static void main(String[] args) {  
        int a = Integer.parseInt(args[0]);  
        int b = Integer.parseInt(args[1]);  
        int sum = a + b;  
        int prod = a * b;  
        int quot = a / b;  
        int rem = a % b;  
        System.out.println(a + " + " + b + " = " + sum);  
        System.out.println(a + " * " + b + " = " + prod);  
        System.out.println(a + " / " + b + " = " + quot);  
        System.out.println(a + " % " + b + " = " + rem);  
    }  
}
```

*converts from  
String to int*

*converts from  
int to String*

```
~/cos125/datatypes> java IntOps 20 3
```

```
20 + 3 = 23
```

```
20 * 3 = 60
```

```
20 / 3 = 6
```

```
20 % 3 = 2
```

*← 20 = 6 · 3 + 2*

```
~/cos125/datatypes> java IntOps 1234 99
```

```
1234 + 99 = 1333
```

```
1234 * 99 = 122166
```

```
1234 / 99 = 12
```

```
1234 % 99 = 46
```

*← 1234 = 12 · 99 + 46*

```
~/cos125/datatypes> java IntOps 1234 Hello
```

```
Exception in thread "main"
```

```
java.lang.NumberFormatException:
```

```
For input string: "Hello"
```

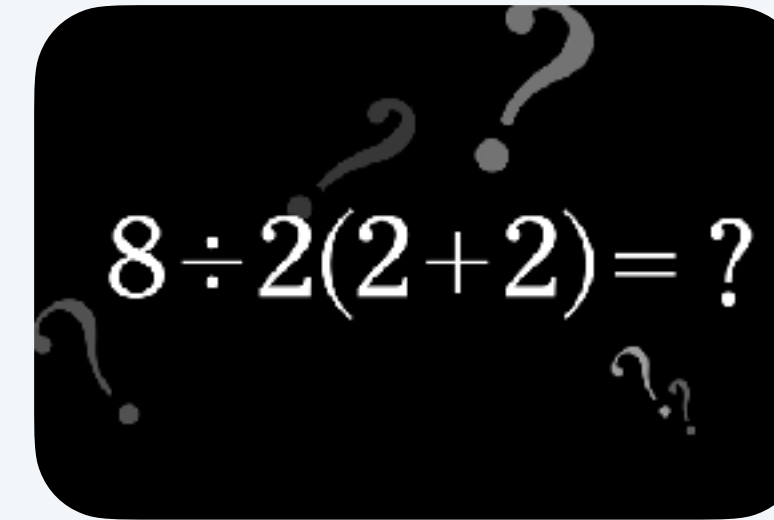
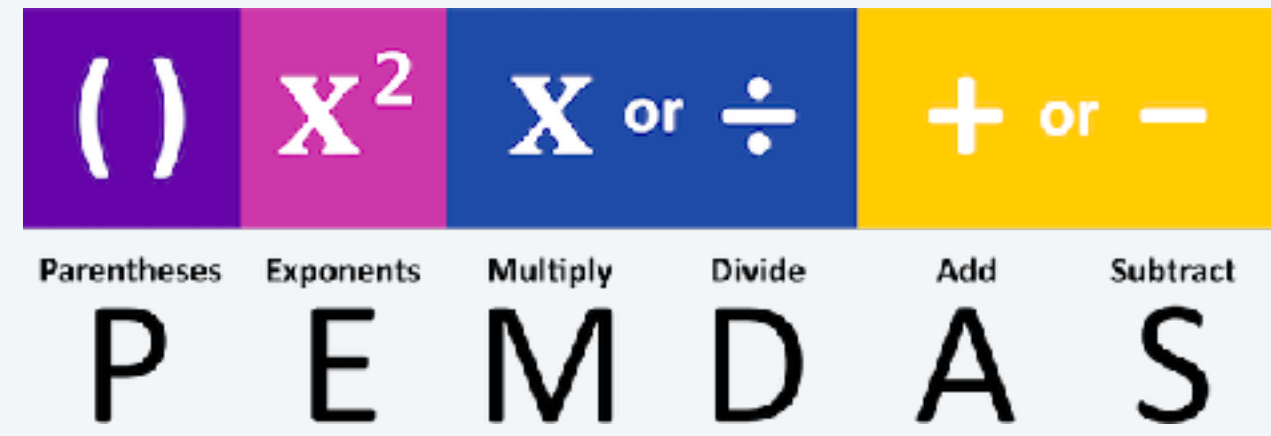
```
...
```

```
at IntOps.main(IntOps.java:4)
```

*← line number  
of error*

# Order of operations

**PEMDAS.** Rules for evaluating an arithmetic expression.



Engagement bait

**Operator precedence.** Priority for grouping operands with operators in an expression.

**Operator associativity.** Rule when two operators in an expression have same priority.

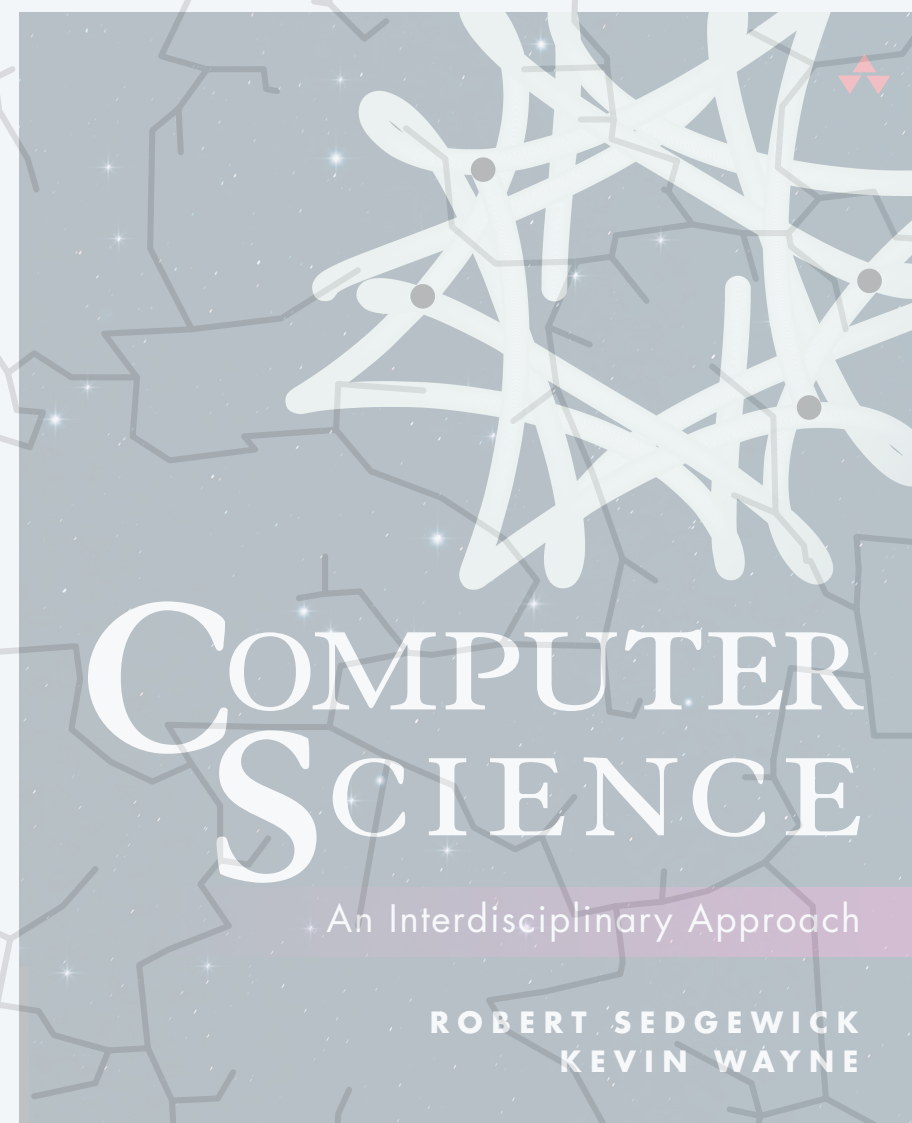
| expression        | equivalent to       | value | remark  |
|-------------------|---------------------|-------|---|
| $3 * 5 - 2$       | $(3 * 5) - 2$       | 13    | <i>* has higher precedence than -</i>                                     |
| $3 + 5 / 2$       | $3 + (5 / 2)$       | 5     | <i>/ has higher precedence than +</i>                                     |
| $3 - 5 - 2$       | $(3 - 5) - 2$       | -4    | <i>left-to-right associative</i>  |
| $(3 - 5) - 2$     | <i>itself</i>       | -4    | <i>better style</i>   |
| $8 / 2 * (2 + 2)$ | $(8 / 2) * (2 + 2)$ | 16    | <i>left-to-right associative</i><br><i>(* and / have same precedence)</i> |



What value does the following expression evaluate to?

```
1 + 2 + "ABC" + (3 + 4)
```

- A. "12ABC34"
- B. "3ABC7"
- C. "3ABC34"
- D. "12ABC7"
- E. Compile-time error.



<https://introcs.cs.princeton.edu>

## BUILT-IN DATA TYPES

---

- ▶ *strings*
- ▶ *integers*
- ▶ *floating-point numbers*
- ▶ *booleans*
- ▶ *type conversion*

# The *double* data type

Typical usage: scientific calculations involving real numbers.

|                  |                                    |                 |                    |               |                  |
|------------------|------------------------------------|-----------------|--------------------|---------------|------------------|
| values           | <i>IEEE floating-point numbers</i> |                 |                    |               |                  |
| typical literals | 18.25                              | -2.0            | 1.4142135623730951 | 6.022E23      |                  |
| operations       | <i>add</i>                         | <i>subtract</i> | <i>multiply</i>    | <i>divide</i> | <i>remainder</i> |
| operators        | +                                  | -               | *                  | /             | %                |

*only  $2^{64}$  different double values  
(not quite the same as real numbers)*

*$6.022 \times 10^{23}$   
(scientific notation)*

| expression | value              | remark                    |
|------------|--------------------|---------------------------|
| 1.5 + 0.25 | 1.75               |                           |
| 1.5 - 0.25 | 1.25               |                           |
| 1.5 * 2.0  | 3.0                |                           |
| 5.0 / 3.0  | 1.6666666666666667 | not exactly $\frac{5}{3}$ |
| -1.0 / 0.0 | -Infinity          | not an error              |
| 0.0 / 0.0  | NaN                | "not a number"            |

*applying a double operator  
to two double operands  
always results in a double  
(can't result in an error)*

*only binary fractional values  
can be represented exactly, such as  
 $\frac{1}{4} + \frac{1}{16} + \frac{1}{128} = 0.3203125$   
(but not  $\frac{5}{3}$ ,  $\frac{1}{10}$ , or  $\pi$ )*

# Excepts from Java's *Math* library

## Math library function

## description

static double abs(double a)  
 static double max(double a, double b)  
 static double min(double a, double b)

*absolute value of a*  
*maximum of a and b*  
*minimum of a and b*

← *also defined for int*

static double sin(double theta)  
 static double cos(double theta)  
 static double tan(double theta)

*sine (sin  $\theta$ )*  
*cosine (cos  $\theta$ )*  
*tangent (tan  $\theta$ )*

← *inverse functions also available:  
 asin(), acos(), and atan()*

← *degrees in radians;  
 to convert, use Math.toDegrees() and Math.toRadians()*

static double exp(double a)  
 static double log(double a)  
 static double sqrt(double a)  
 static double pow(double a, double b)

*exponential ( $e^a$ )*  
*natural logarithm ( $\log_e a$ )*  
*positive square root ( $\sqrt{a}$ )*  
*power ( $a^b$ )*

static long round(double a)  
 static double random()

*round to the nearest integer*  
*pseudorandom number in [0, 1)*

static double E  
 static double PI

*value of e (constant)*  
*value of  $\pi$  (constant)*

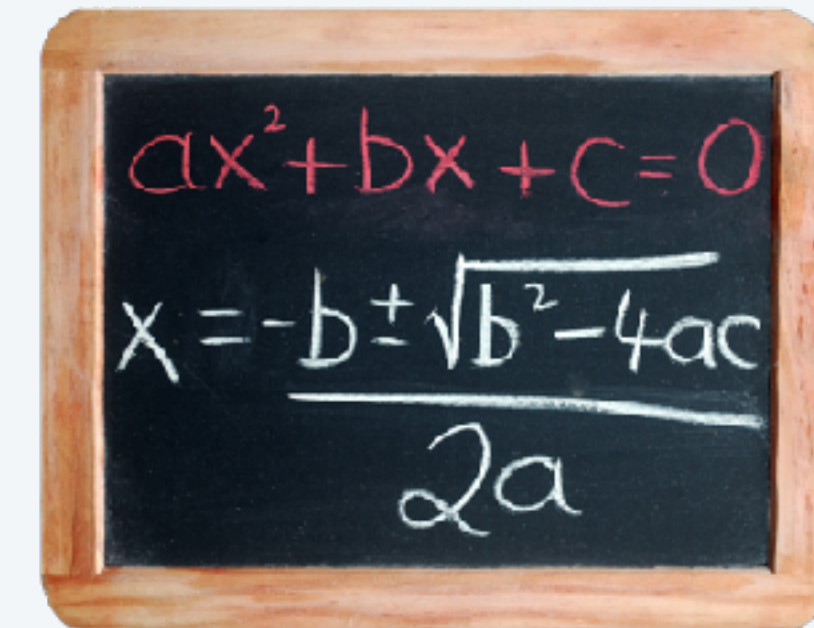


**You can discard your calculator now (please).**

| expression         | value              |
|--------------------|--------------------|
| Math.max(1.0, 2.5) | 2.5                |
| Math.cos(0.0)      | 1.0                |
| Math.sqrt(2.0)     | 1.4142135623730951 |
| Math.random()      | 0.7707780210347349 |
| Math.PI            | 3.141592653589793  |

# Quadratic equation

**Goal.** Print the solutions to the equation  $ax^2 + bx + c = 0$ , assuming  $a \neq 0$ .



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

        // Parse coefficients from command-line.
        double a = Double.parseDouble(args[0]);
        double b = Double.parseDouble(args[1]);
        double c = Double.parseDouble(args[2]);

        // Calculate roots of ax^2 + bx + c.
        double discriminant = b*b - 4.0*a*c;
        double d = Math.sqrt(discriminant);
        double root1 = (-b + d) / (2.0*a);
        double root2 = (-b - d) / (2.0*a);

        // Print them out.
        System.out.println(root1);
        System.out.println(root2);
    }
}
```

```
~/cos125/datatypes> java Quadratic 1.0 -3.0 2.0
2.0
1.0
```

$$x^2 - 3x + 2$$

```
~/cos125/datatypes> java Quadratic 1.0 -1.0 -1.0
1.618033988749895
-0.6180339887498949
```

$$\longleftarrow \frac{1 \pm \sqrt{5}}{2}$$

$$x^2 - x - 1$$

```
~/cos125/datatypes> java Quadratic 1.0 1.0 1.0
NaN
NaN
```

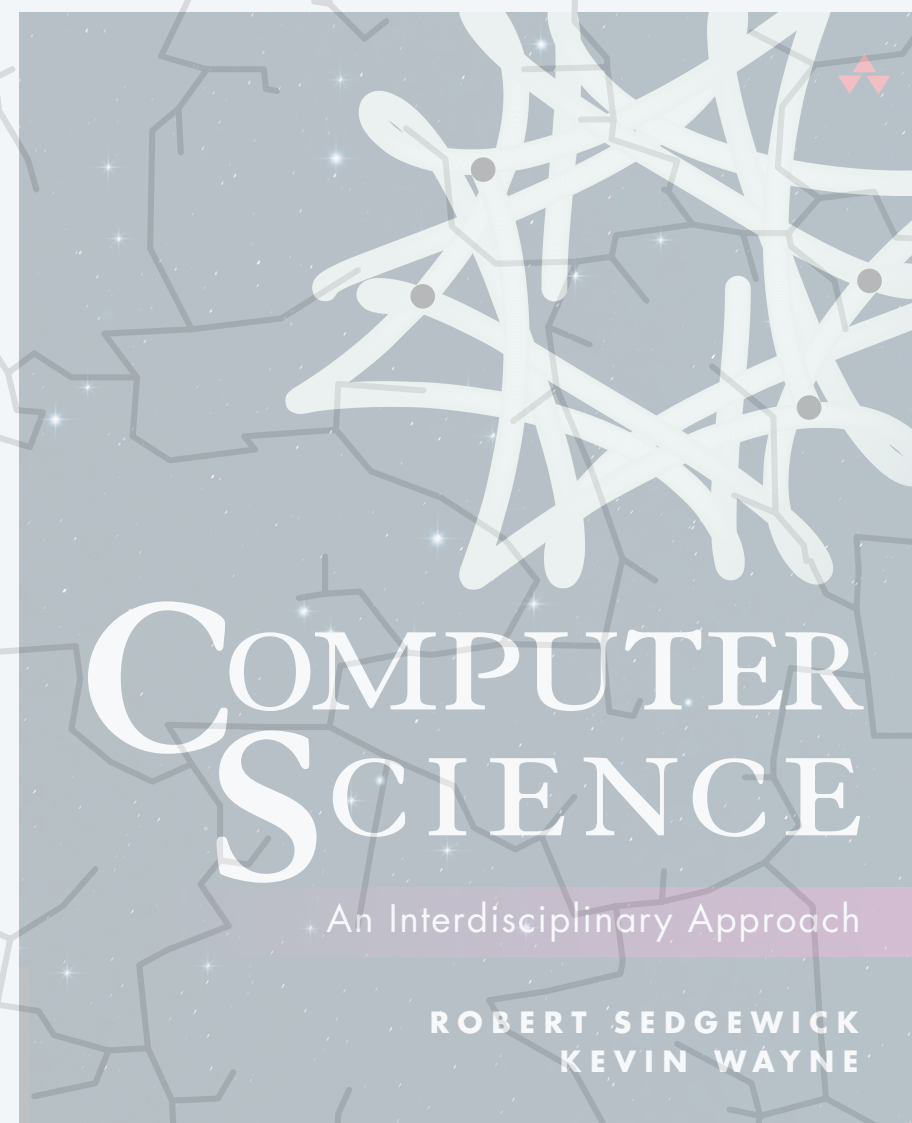
$$\longleftarrow \frac{-1 \pm 3i}{2}$$

$$x^2 + x + 1$$

```
~/cos125/datatypes> java Quadratic 1.0 2.8 1.96
NaN
NaN
```

$\longleftarrow$  floating-point roundoff error  
( $x = -\frac{7}{5}$  is a double root)

$$x^2 + \frac{14}{5}x + \frac{49}{25}$$



<https://introcs.cs.princeton.edu>

## BUILT-IN DATA TYPES

---

- ▶ *strings*
- ▶ *integers*
- ▶ *floating-point numbers*
- ▶ ***booleans***
- ▶ *type conversion*

# The *boolean* data type

Typical usage: decision making in a program. *← stay tuned for conditionals and loops*

|            |                       |            |                            |
|------------|-----------------------|------------|----------------------------|
| values     | <i>true and false</i> |            |                            |
| literals   | true false            |            |                            |
| operations | <i>not</i>            | <i>and</i> | <i>or</i>                  |
| operators  | !                     | &&         | <i>← logical operators</i> |

| expression          | value |
|---------------------|-------|
| <code>!false</code> | true  |
| <code>!true</code>  | false |

truth table for NOT

| expression                          | value |
|-------------------------------------|-------|
| <code>false &amp;&amp; false</code> | false |
| <code>false &amp;&amp; true</code>  | false |
| <code>true &amp;&amp; false</code>  | false |
| <code>true &amp;&amp; true</code>   | true  |

truth table for AND


| expression                  | value |
|-----------------------------|-------|
| <code>false    false</code> | false |
| <code>false    true</code>  | true  |
| <code>true    false</code>  | true  |
| <code>true    true</code>   | true  |

truth table for OR

# Equality and comparison operators

---

Equality and comparison operators. To **compare** numeric values.

- Operands: two numeric expressions.  *can be literals, variable, or arbitrary expressions*
- Evaluates to: a value of type *boolean*.

| operator           | meaning                      | true                   | false                  |
|--------------------|------------------------------|------------------------|------------------------|
| <code>==</code>    | <i>equal</i>                 | <code>2 == 2</code>    | <code>2 == 3</code>    |
| <code>!=</code>    | <i>not equal</i>             | <code>3 != 2</code>    | <code>2 != 2</code>    |
| <code>&lt;</code>  | <i>less than</i>             | <code>2 &lt; 13</code> | <code>13 &lt; 2</code> |
| <code>&lt;=</code> | <i>less than or equal</i>    | <code>2 &lt;= 2</code> | <code>3 &lt;= 2</code> |
| <code>&gt;</code>  | <i>greater than</i>          | <code>13 &gt; 2</code> | <code>2 &gt; 13</code> |
| <code>&gt;=</code> | <i>greater than or equal</i> | <code>2 &gt;= 2</code> | <code>2 &gt;= 3</code> |

equality and comparison operators in Java

# Equality and comparison operators: examples

|                               |  |  |
|-------------------------------|--|--|
| zero denominator?             | <code>denominator == 0</code>                                |  |
| non-negative discriminant?    | <code>(b*b - 4.0*a*c) &gt;= 0.0</code>                       | parentheses for clarity:<br>arithmetic operators have<br>higher precedence than<br>equality/comparison operators |
| divisible by 60?              | <code>(minutes % 60) == 0</code>                             |  |
| RGB color is not black?       | <code>(red &gt; 0)    (green &gt; 0)    (blue &gt; 0)</code> | compound boolean expressions   |
| valid month?                  | <code>(month &gt;= 1) &amp;&amp; (month &lt;= 12)</code>     |  |
| invalid month?                | <code>!((month &gt;= 1) &amp;&amp; (month &lt;= 12))</code>  |  |
| floating-point roundoff error | <code>(0.1 * 3.0) == 0.3</code>                              | don't do this!<br>(evaluates to false)   |
| string equality               | <code>args[0] == "Hello"</code>                              | or this!<br>(always evaluates to false)  |

# Example of computing with booleans: leap year test

---

Q. Is a given year a leap year? ← *Gregorian calendar*

A. Yes if either (1) divisible by 400 or (2) divisible by 4 but not 100.

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

        int year = Integer.parseInt(args[0]);
        boolean isLeapYear;

        // divisible by 4 but not 100
        isLeapYear = (year % 4 == 0) && (year % 100 != 0);

        // or divisible by 400
        isLeapYear = isLeapYear || (year % 400 == 0);

        System.out.println(isLeapYear);
    }
}
```

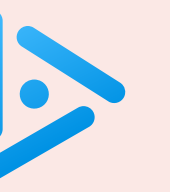
*if argument to System.out.println() is of type boolean,  
it prints either true or false*

```
~/cos125/datatypes> java LeapYear 2024
true

~/cos125/datatypes> java LeapYear 2023
false

~/cos125/datatypes> java LeapYear 1900
false

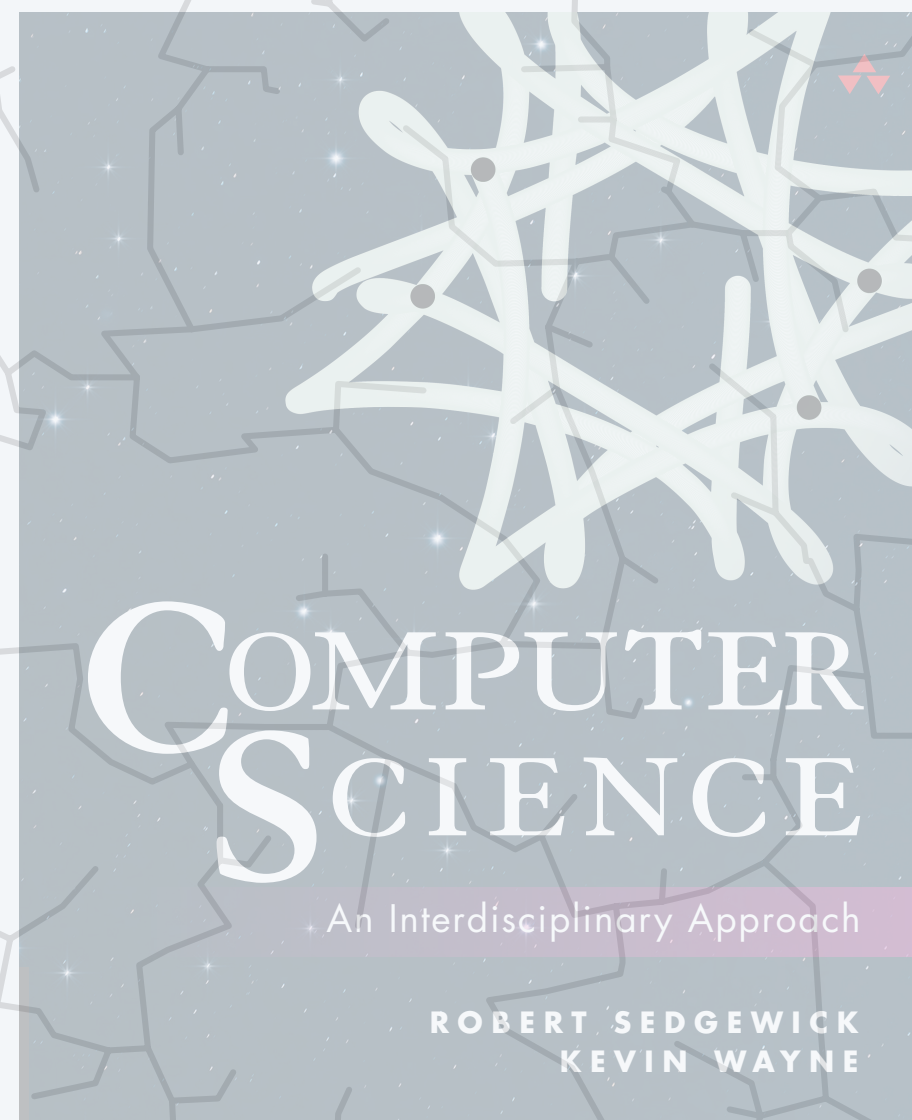
~/cos125/datatypes> java LeapYear 2000
true
```



What does the following expression evaluate to?

```
year % 4 == 0 && year % 100 != 0 || year % 400 == 0
```

- A. Works: computes whether `year` is a leap year correctly.
- B. “Works:” compiles, but the result may not be correct.
- C. Doesn’t work: equivalent to `(year % 4) == ((0 && year) % (100 != 0)) || (year % 400 == 0)` (compile-time error).



<https://introcs.cs.princeton.edu>

## BUILT-IN DATA TYPES

---

- ▶ *strings*
- ▶ *integers*
- ▶ *floating-point numbers*
- ▶ *booleans*
- ▶ *type conversion*

# Data types

---

**Types** limit the allowable operations on values and determine the meaning of those operations.

```
public class StringMultiply {  
    public static void main(String[] args) {  
        String s = "123" * "456";  
    }  
}
```

```
~/cos125/datatypes> javac StringMultiply.java  
StringMultiply.java:3: error: bad operand types  
for binary operator '*'  
    String s = "123" * "456";  
                    ^  
    first type: String  
    second type: String  
1 error
```

**Java compiler.** The compiler checks for type mismatch errors in your code.

# Data types

**Types** limit the allowable operations on values and determine the meaning of those operations.

| operator | int                     | double                | boolean            | String               |
|----------|-------------------------|-----------------------|--------------------|----------------------|
| +        | <i>addition</i>         | <i>addition</i>       | ✗                  | <i>concatenation</i> |
| -        | <i>subtraction</i>      | <i>subtraction</i>    | ✗                  | ✗                    |
| *        | <i>multiplication</i>   | <i>multiplication</i> | ✗                  | ✗                    |
| /        | <i>integer division</i> | <i>division</i>       | ✗                  | ✗                    |
| &&       | ✗                       | ✗                     | <i>logical AND</i> | ✗                    |
|          | ✗                       | ✗                     | <i>logical OR</i>  | ✗                    |
| !        | ✗                       | ✗                     | <i>logical NOT</i> | ✗                    |
| <        | <i>less than</i>        | <i>less than</i>      | ✗                  | ✗                    |
| ⋮        | ⋮                       | ⋮                     | ⋮                  | ⋮                    |

← can't subtract, multiply, or divide two String or boolean values (compile-time errors)

**Static typing.** Every Java variable and expression has a type that is known at compile time.

- Benefit: compiler catches entire class of programming errors automatically.
- Drawback: extra boilerplate code.

# Type conversions with built-in types

Type conversion is an essential aspect of programming.

## Automatic type conversions.

- String conversion: from any type to *String* (via string concatenation).
- Numeric promotion: from *int* to *double* (when a *double* is expected).
- from *int* to *long* ← every *int* can be exactly represented as a *double* or *long*

| expression  | type   | value    |
|-------------|--------|----------|
| "x = " + 99 | String | "x = 99" |
| 11 * 0.25   | double | 2.75     |
| 22 / 7L     | long   | 3L       |

## System methods.

- Integer.parseInt()* from *String* to *int*.
- Double.parseDouble()* from *String* to *double*.

| expression                        | type   | value |
|-----------------------------------|--------|-------|
| <i>Integer.parseInt</i> ("126")   | int    | 126   |
| <i>Double.parseDouble</i> ("2.5") | double | 2.5   |

## Explicit casts from one type to another.

- Cast from *double* to *int*. ← discards fractional part
- Cast from *int* to *double*.

| expression        | type   | value   |
|-------------------|--------|---------|
| (int) 2.71828;    | int    | 2       |
| (double) sum / n; | double | average |

cast operator

cast has higher precedence

two int variables

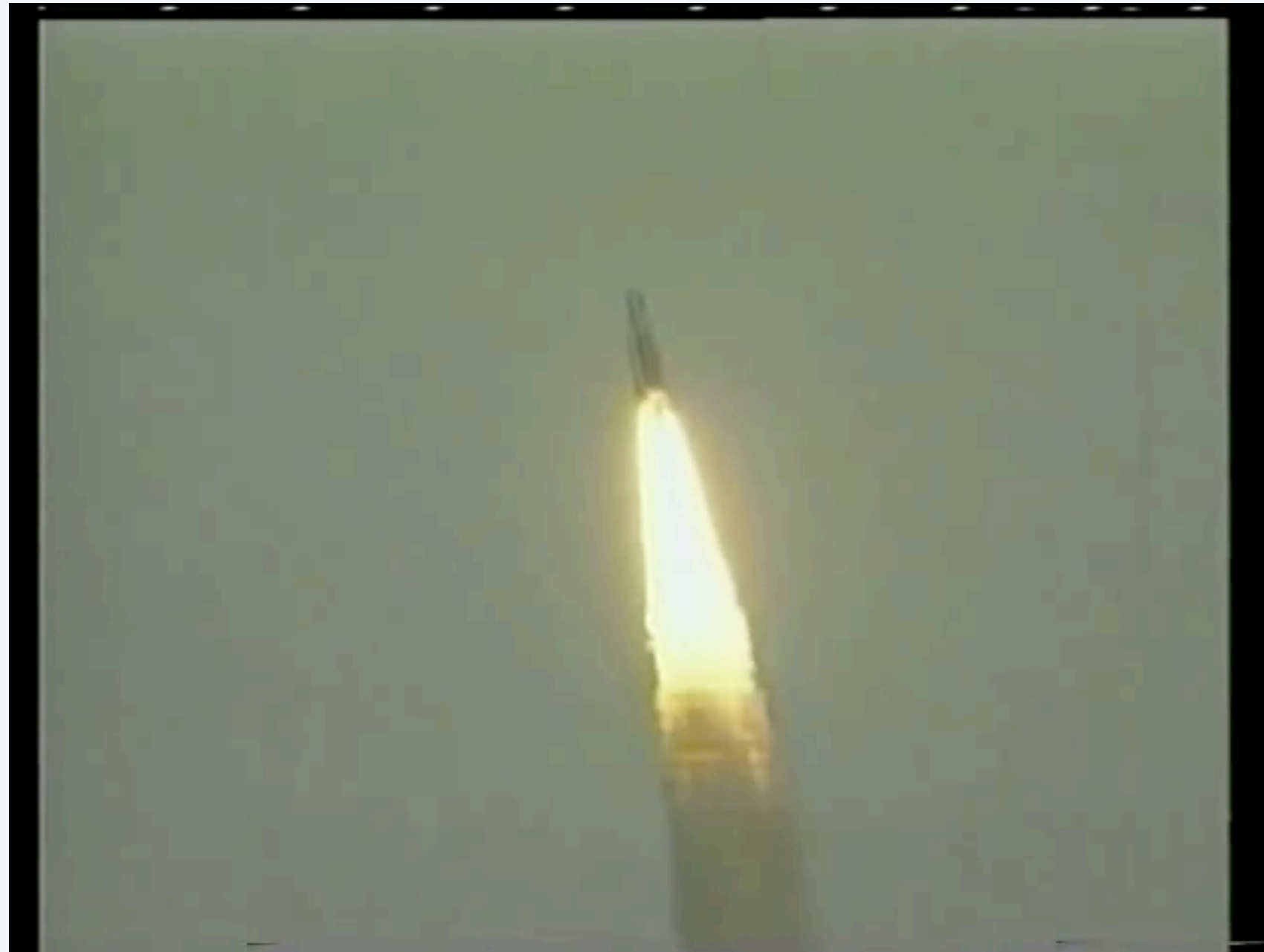
# Type-conversion catastrophe

---

## Ariane 5 rocket.

- European Space Agency spent a decade and \$7 billion in research and development.
- Rocket self-destructed 39 seconds after first launch.
- Source of bug: unsafe type conversion of 64-bit floating-point number to 16-bit integer.

*code worked fine in Ariane 4  
(but Ariane 5 velocity was much higher)*



[https://www.youtube.com/watch?v=PK\\_yguLapgA](https://www.youtube.com/watch?v=PK_yguLapgA)

## Example of type conversion

---

Q. What is type and value of each expression on the left?

| expression                 | type                      | value | remark   |
|----------------------------|---------------------------|-------|--|
| <code>(7 / 2) * 2.0</code> | double                    | 6.0   | <i>integer division;<br/>then promotion to double</i>        |
| <code>(7 / 2.0) * 2</code> | double                    | 7.0   | <i>promotion to double;<br/>then floating-point division</i> |
| <code>"12" + 6</code>      | String                    | "126" | <i>conversion to String</i>                                  |
| <code>0 == false</code>    | <i>compile-time error</i> |       | <i>can't compare<br/>int to boolean</i>                      |

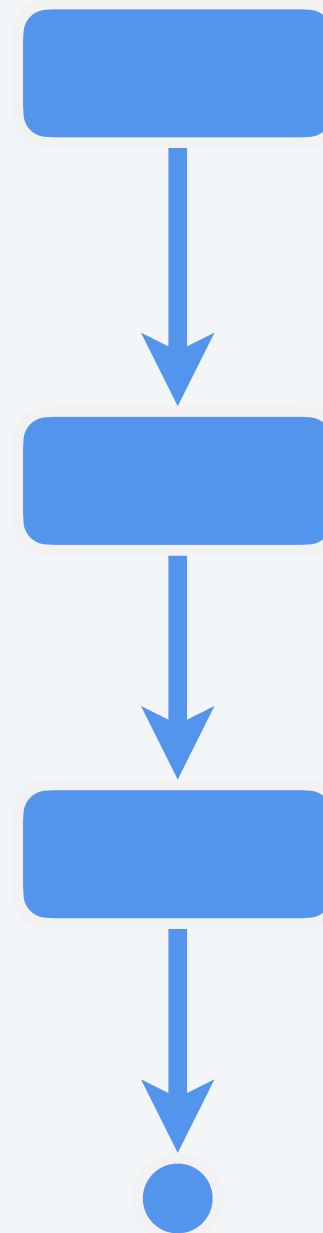
# Overview

---

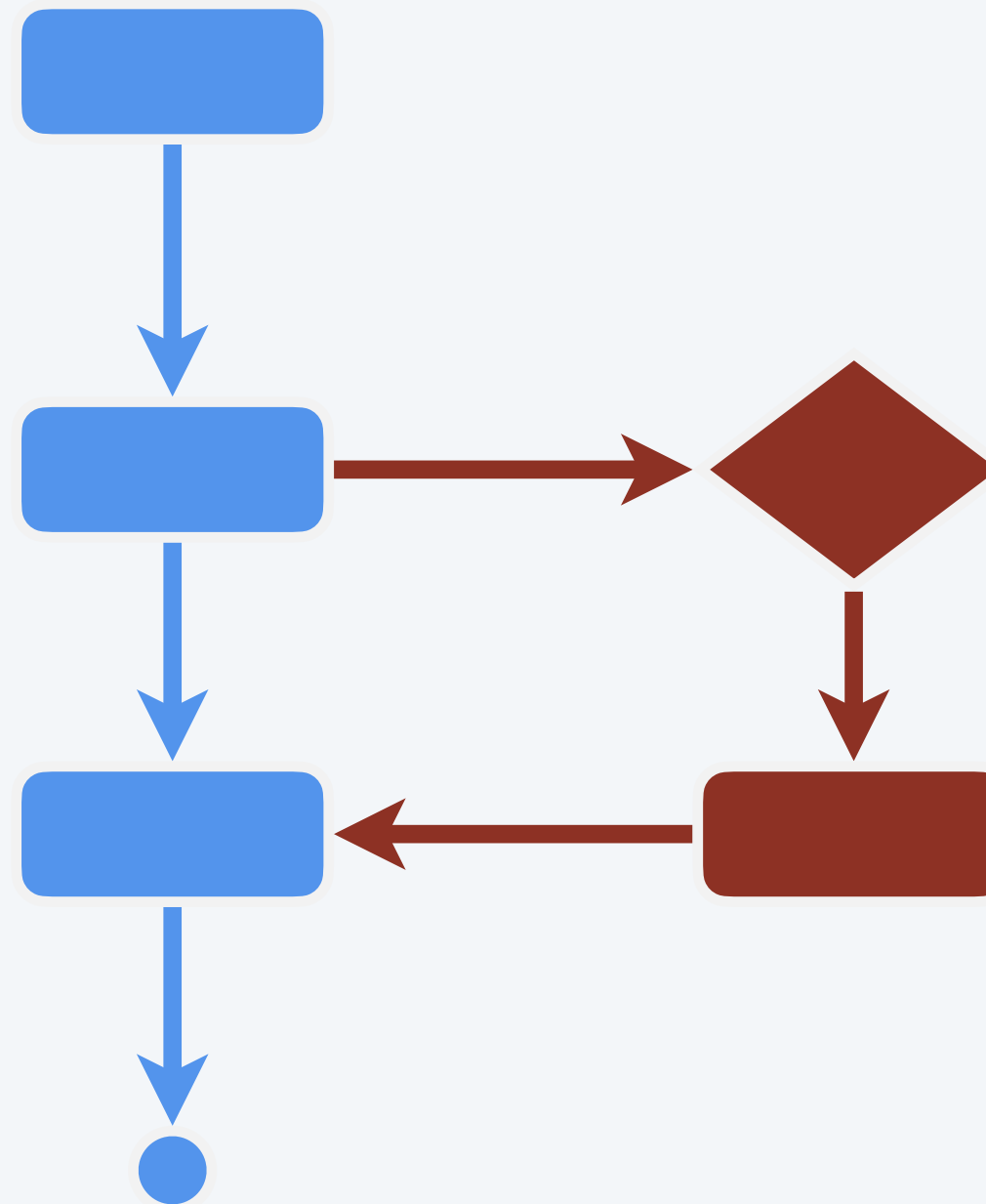
This lecture. Write programs with **declaration**, **assignment**, and **print** statements.

Next lecture. Write programs with **conditionals**.

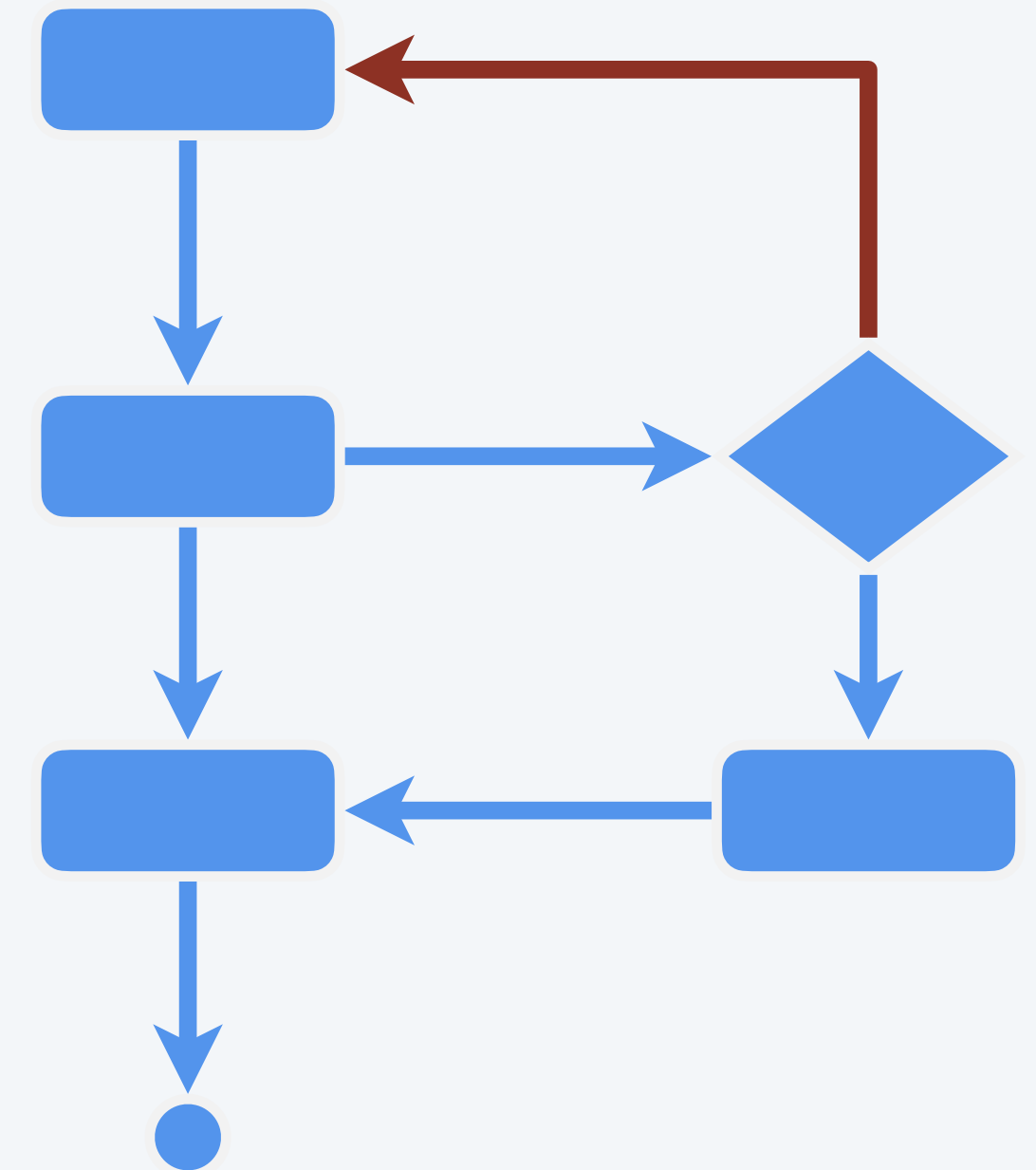
Next week. Write programs with **conditionals** and **loops**.



straight-line control flow



control flow with conditionals



control flow with conditionals and loops

## Recap + teaser: data types

---

A **data type (type)** is a set of values and a set of operations on those values.

| type   | set of values                  |
|--|--------------------------------|
| <i>byte</i><br><i>short</i><br><i>int</i><br><i>long</i> | <i>integers</i>                |
| <i>float</i><br><i>double</i>                            | <i>floating-point numbers</i>  |
| <i>boolean</i>   | <i>truth values</i>            |
| <i>char</i>  | <i>character</i>               |
| <i>String</i>  | <i>sequences of characters</i> |

# Credits

---

| <b>media</b>                       | <b>source</b>                | <b>license</b>           |
|------------------------------------|------------------------------|--------------------------|
| <i>PEMDAS</i>                      | <u>Mometrix</u>              |                          |
| <i>PEMDAS meme</i>                 | <u>New York Times</u>        |                          |
| <i>Scientific Calculator</i>       | <u>Wikimedia</u>             | <u>CC BY-SA 3.0</u>      |
| <i>Solving Quadratic Equations</i> | <u>Adobe Stock</u>           | <u>education license</u> |
| <i>Ariane 5 Rocket Launch</i>      | <u>European Space Agency</u> |                          |