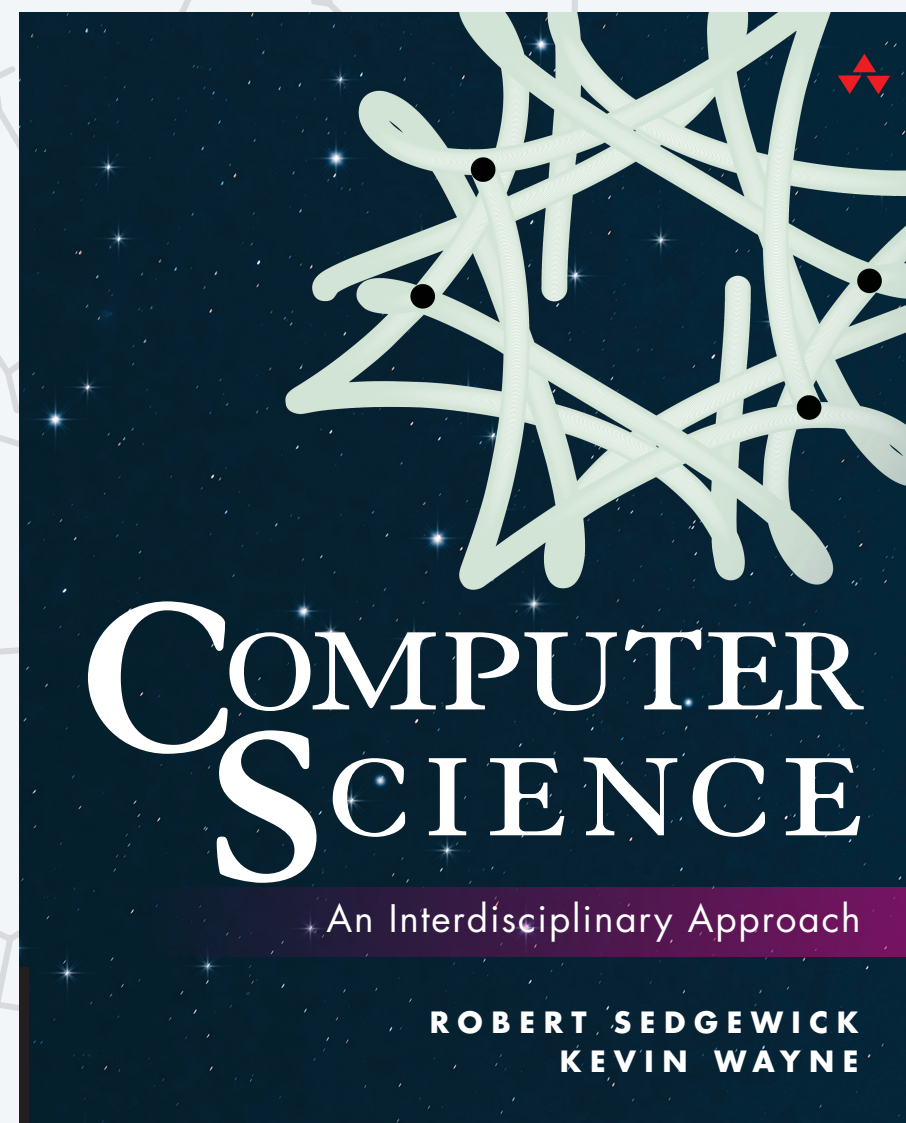




<https://introcs.cs.princeton.edu>

1.2 BUILT-IN DATA TYPES

- *types, variables, and assignments*
- *strings*
- *integers*
- *floating-point numbers*
- *type conversion*

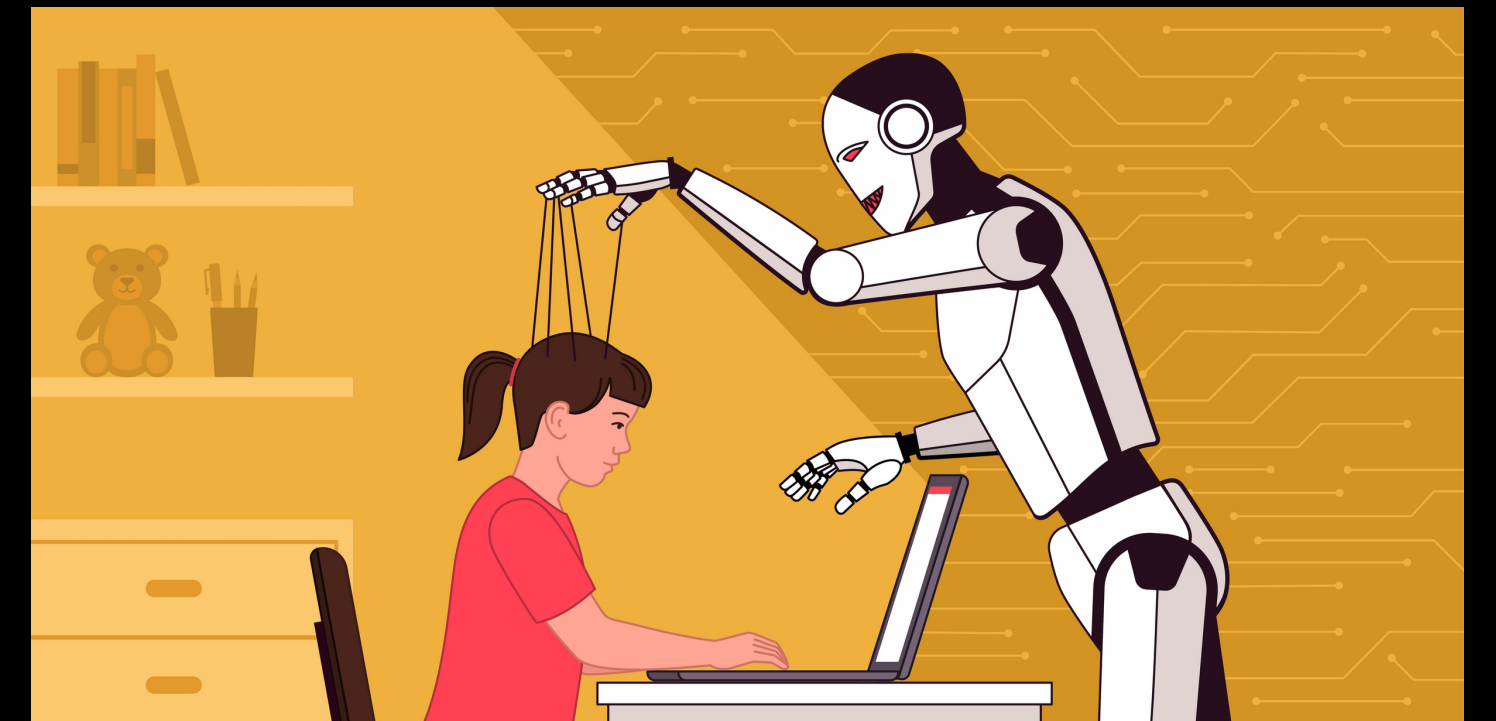
Reminders



Don't use devices
(except for iClicker, taking notes,
running code!)



Use Ed to communicate with course staff
(exception: emailing preceptors)



Using AI will hurt your learning!

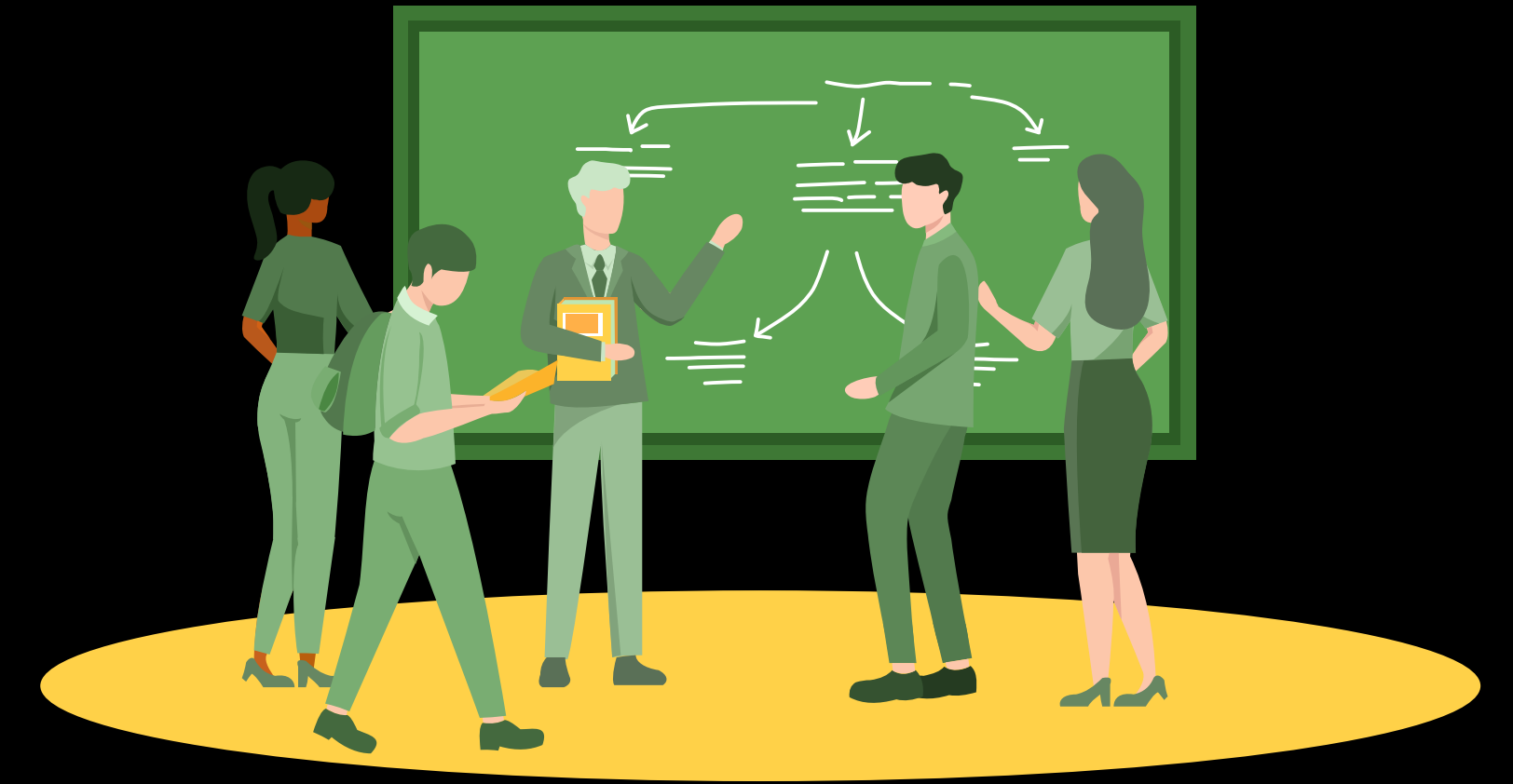
Questions during (or after) lecture



raise your hand and ask



ask (and answer) on Ed



attend office hours (or stay after lecture)



×

Programming Assignment Workshop

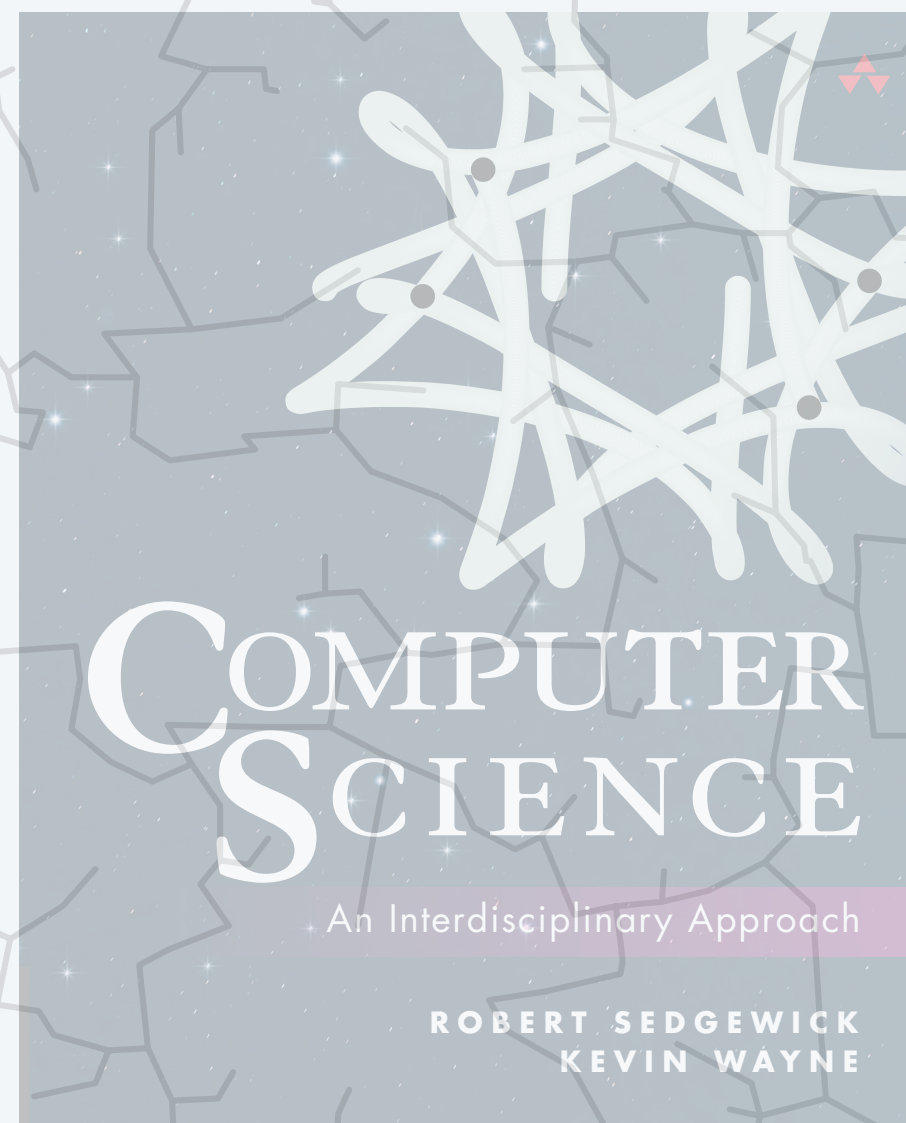
Thursday, September 10 · 7:30 – 9:30pm

Friend Center 008

Get a smart start on this week's assignment with the guidance of a preceptor. Our goal is to complete the assignment before the weekend. This workshop will be held for the individual assignments.

More details

Copy to my calendar



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

1.2 BUILT-IN DATA TYPES

- *types, variables, and assignments*
- *strings*
- *integers*
- *floating-point numbers*
- *type conversion*

Java's core language types

A **data type** (or **type**) is a set of values together with operations on those values.

type	set of values	example values	typical operations
String	<i>sequences of characters</i>	"Hello, World" "I ❤️ COS 126!"	concatenate
int	<i>integers</i>	17 -12345	add, subtract, multiply, divide, compare, equality
double	<i>floating-point numbers</i>	2.5 -0.125	add, subtract, multiply, divide, compare, equality
boolean	<i>truth values</i>	true false	and, or, not, equality
char	<i>characters</i>	'A' '中'	compare

←

next lecture

←

later in the course

Programming terminology

Program. A sequence of statements executed in order. ← *for now*

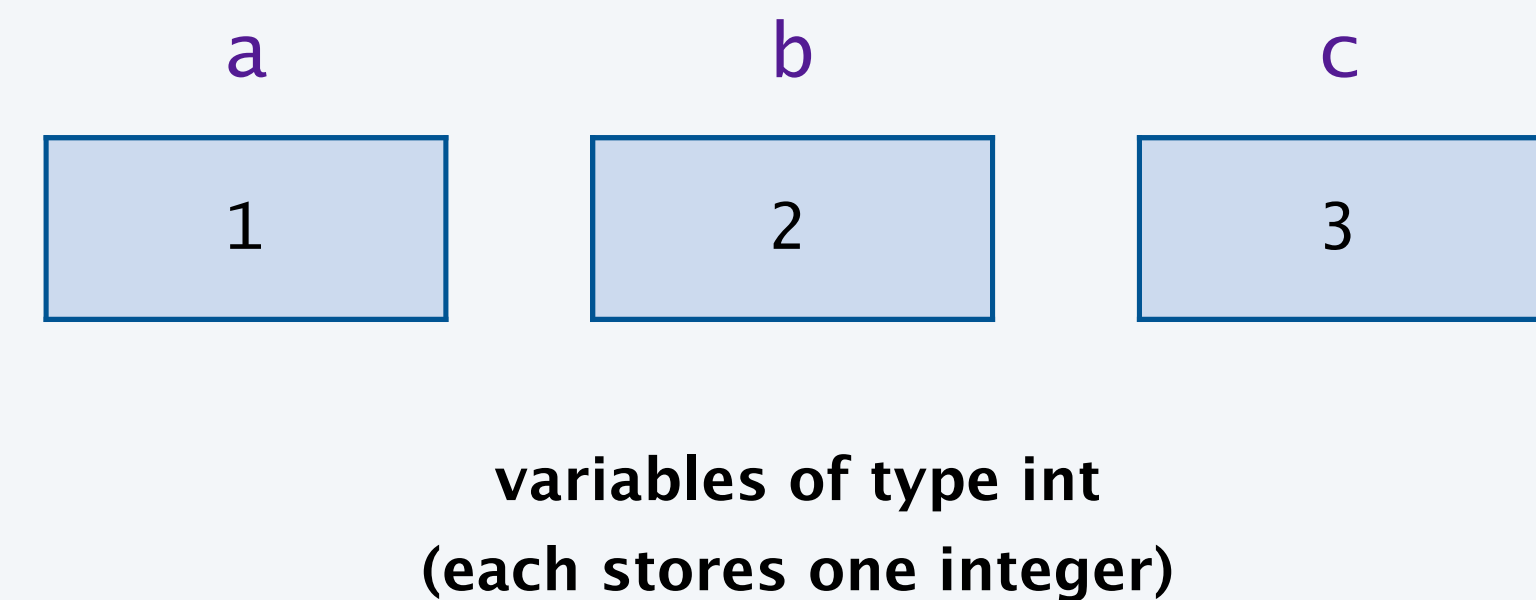
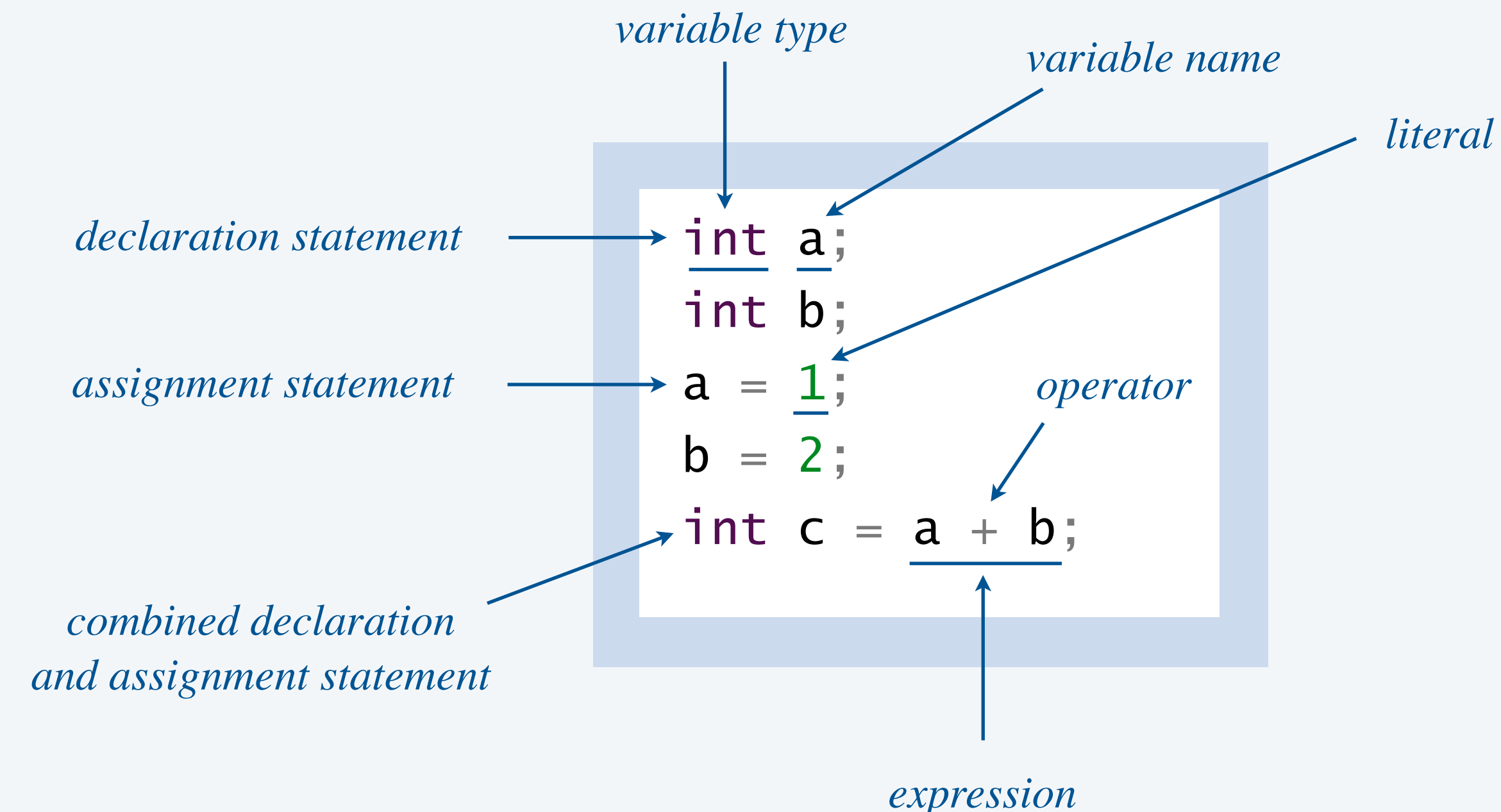
Declaration statement. Associates a variable with a name and type.

Variable. A place to store a data-type value.

Assignment statement. Stores a value in a variable.

Literal. A programming-language representation of a data-type value.

Expression. Code that can be evaluated to produces a value.

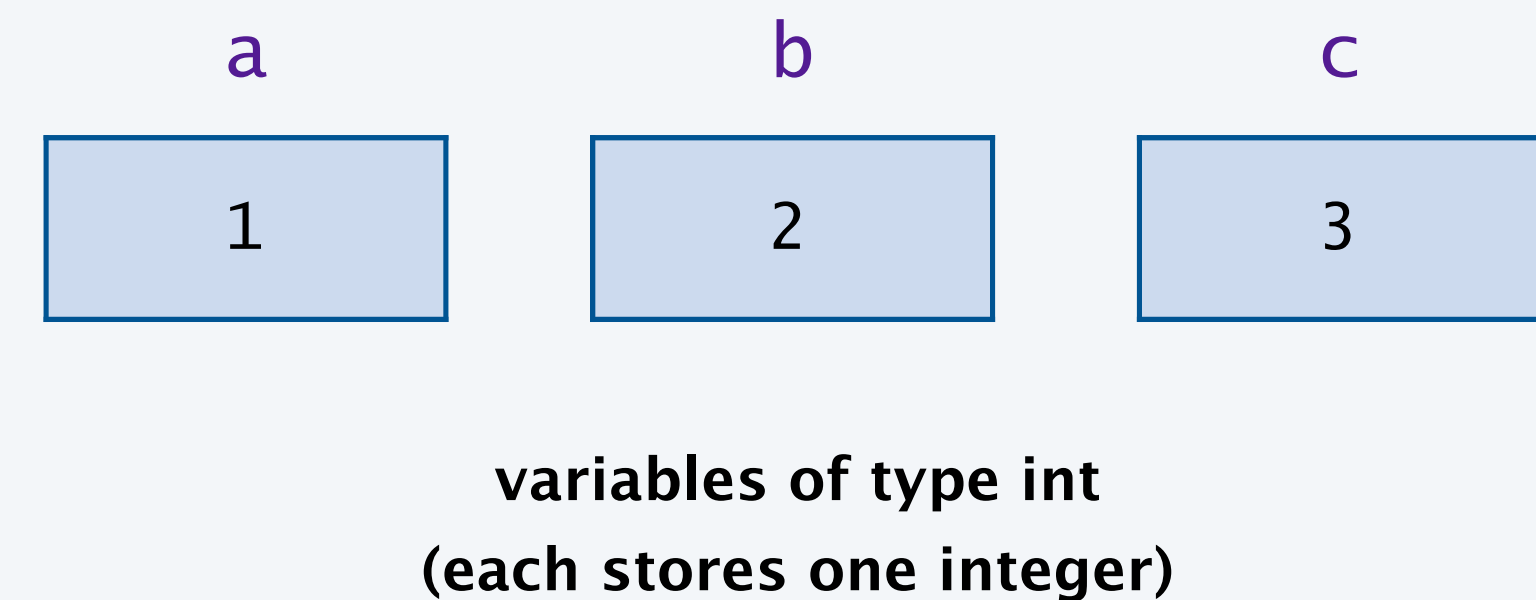
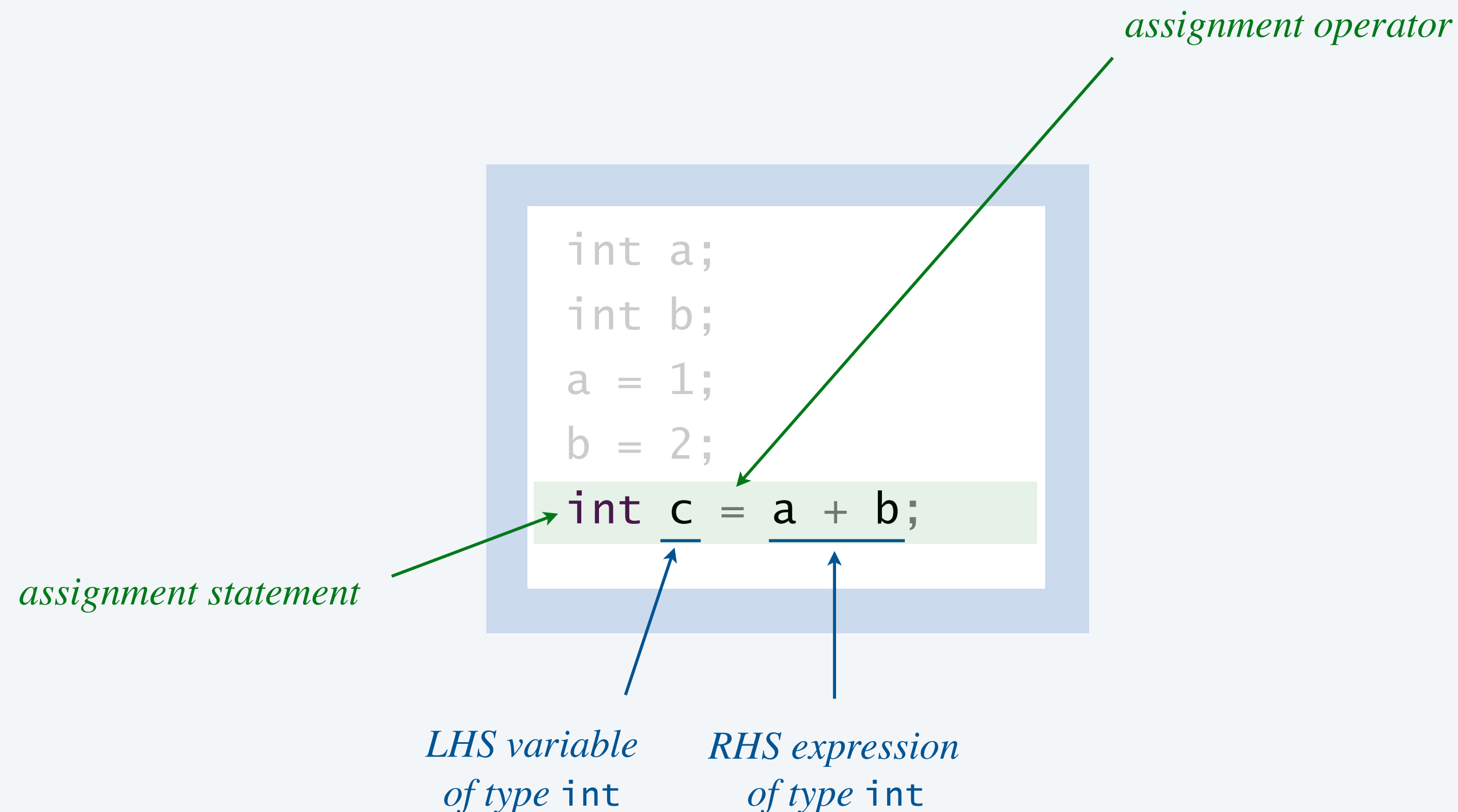


Assignment statements

Q. How does Java execute an **assignment statement**?

A. Java evaluates the RHS **expression**, then stores the resulting value in the LHS **variable**.

Requirement. The RHS type must be compatible with the variable on the LHS.



Valid and invalid assignment statements

Q. Which of these independent code fragments are legal in Java (compiles without errors)?

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

Tracing the execution of a program



Q. What does this code fragment do?

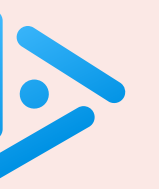
A. Let's **trace** the variables as the program executes. *← table of variable values*

```
int a = 100;
int b = 126;
int temp = a;
a = b;
b = temp;
```

*this classic idiom swaps
the values in a and b*

	a	b	temp
before execution begins	undeclared	undeclared	undeclared
int a = 100;	100	undeclared	undeclared
int b = 126;	100	126	undeclared
int temp = a;	100	126	100
a = b;	126	126	100
b = temp;	126	100	100
	126	100	100

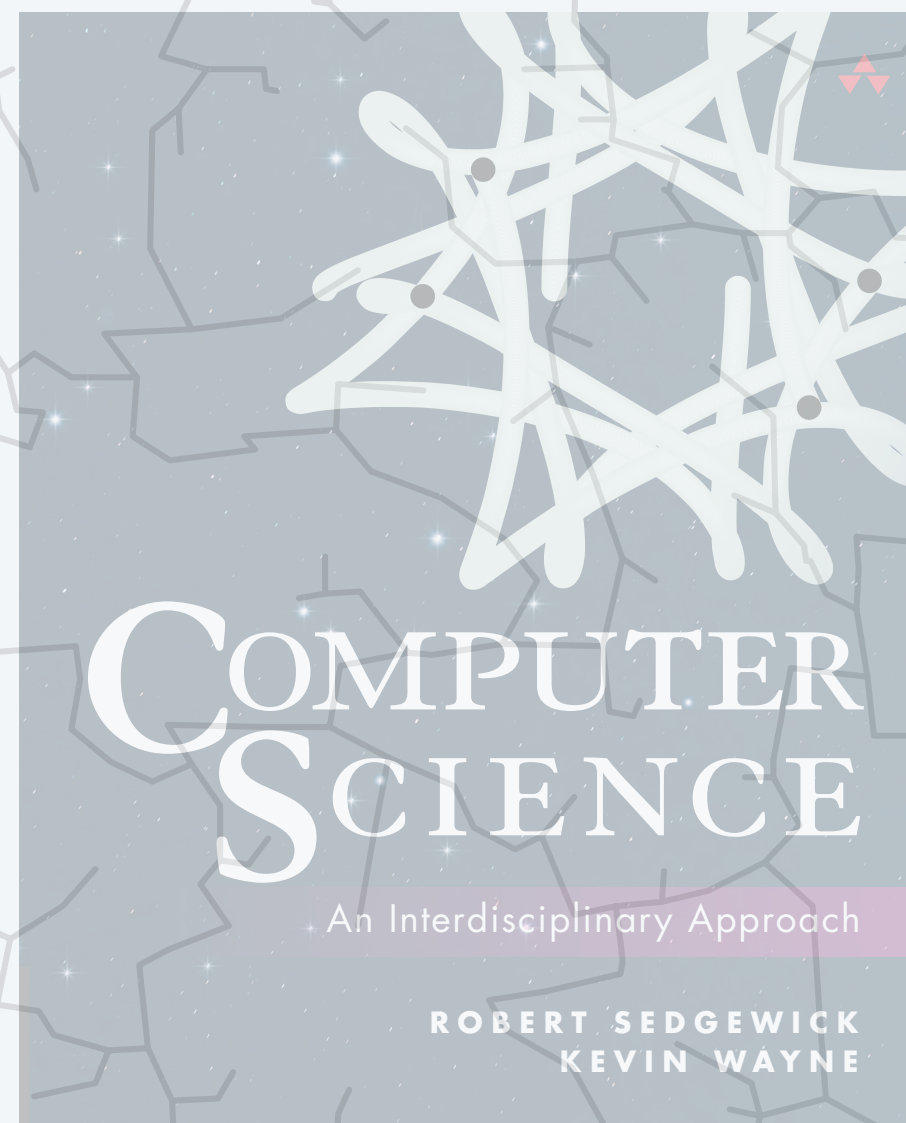
trace of variable values
(after each statement)



What values are stored in **a** and **b** after this code fragment executes?

- A. 100 and 126
- B. 126 and 100
- C. 226 and 126
- D. -26 and -26
- E. Compile-time error

```
int a = 100;  
int b = 126;  
a = a + b;  
b = a - b;  
a = a - b;
```

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

1.2 BUILT-IN DATA TYPES

- *types, variables, and assignments*
- *strings*
- *integers*
- *floating-point numbers*
- *type conversion*

The *String* data type

Typical usage. Program input, program output, and general text processing.

values	<i>sequences of characters</i>
example literals	"Hi" "1234" "Nĩ hăo" "💩💩💩"
operation	<i>concatenation</i>
operator	+

expression	value	remark
"My " + "Precious"	"My Precious"	<i>spaces within string literals matter</i>
"1234" + "99"	"123499"	<i>strings are not integers</i>
"A" + "B" + "C"	"ABC"	<i>can concatenate many strings together, in a single compound expression</i>
"ᠠᠯᠤᠰ " + "ᠠᠨᠠᠭᠤᠨᠠᠨᠢ"	"ᠠᠯᠤᠰ ᠠᠨᠠᠭᠤᠨᠠᠨᠢ"	<i>full Unicode support</i>

Command-line arguments are strings

Command-line arguments. When you run a program:

- Java automatically creates a few variables (named `args[0]`, `args[1]`, `args[2]`, ...), each of type `String`.
- Java initializes them to the text you type after the program name on the command line.

bracket notation explained in Section 1.4 (arrays)

```
public class CommandLineArguments {  
    public static void main(String[] args) {  
        String a = args[0]; // first argument  
        String b = args[1]; // second argument  
        String c = args[2]; // third argument  
  
        // concatenate strings, adding dashes  
        String result = a + "-" + b + "-" + c;  
        System.out.println(result);  
    }  
}
```

```
~/cos126/datatypes> java CommandLineArguments A B C  
A-B-C
```

args[0]

```
~/cos126/datatypes> java CommandLineArguments do re mi  
do-re-mi
```

*arguments separated
by whitespace*

```
~/cos126/datatypes> java CommandLineArguments  
Exception in thread "main"  
java.lang.ArrayIndexOutOfBoundsException:  
Index 0 out of bounds for length 0 at  
CommandLineArguments.main(CommandLineArguments.java:3)
```

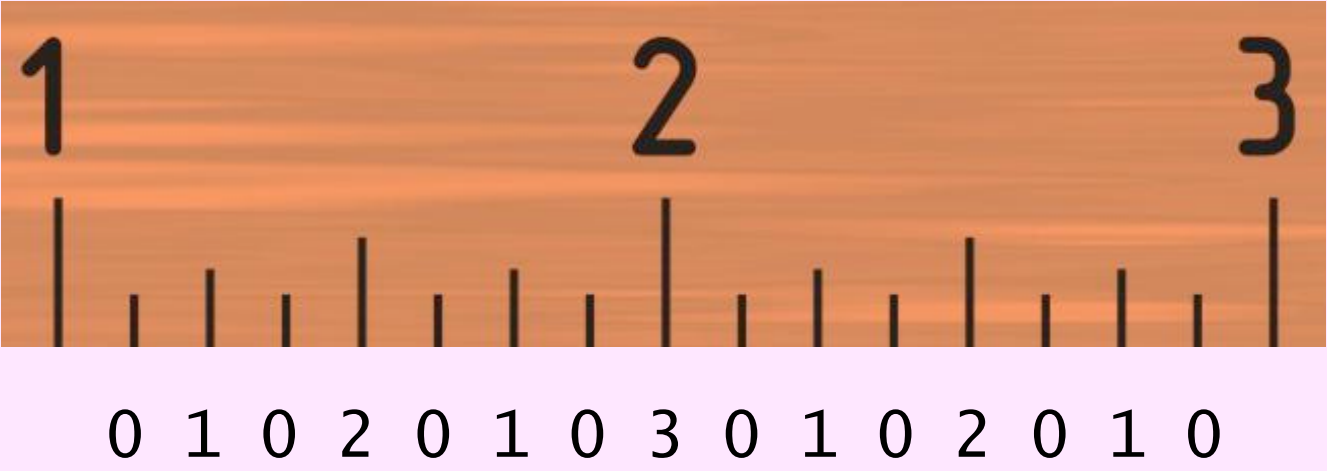
*line number
of error*

Building ruler markings by concatenating strings



```
public class Ruler {
    public static void main(String[] args) {
        String ruler0 = "0";
        String ruler1 = ruler0 + " 1 " + ruler0;
        String ruler2 = ruler1 + " 2 " + ruler1;
        String ruler3 = ruler2 + " 3 " + ruler2;
        System.out.println(ruler3);
    }
}
```

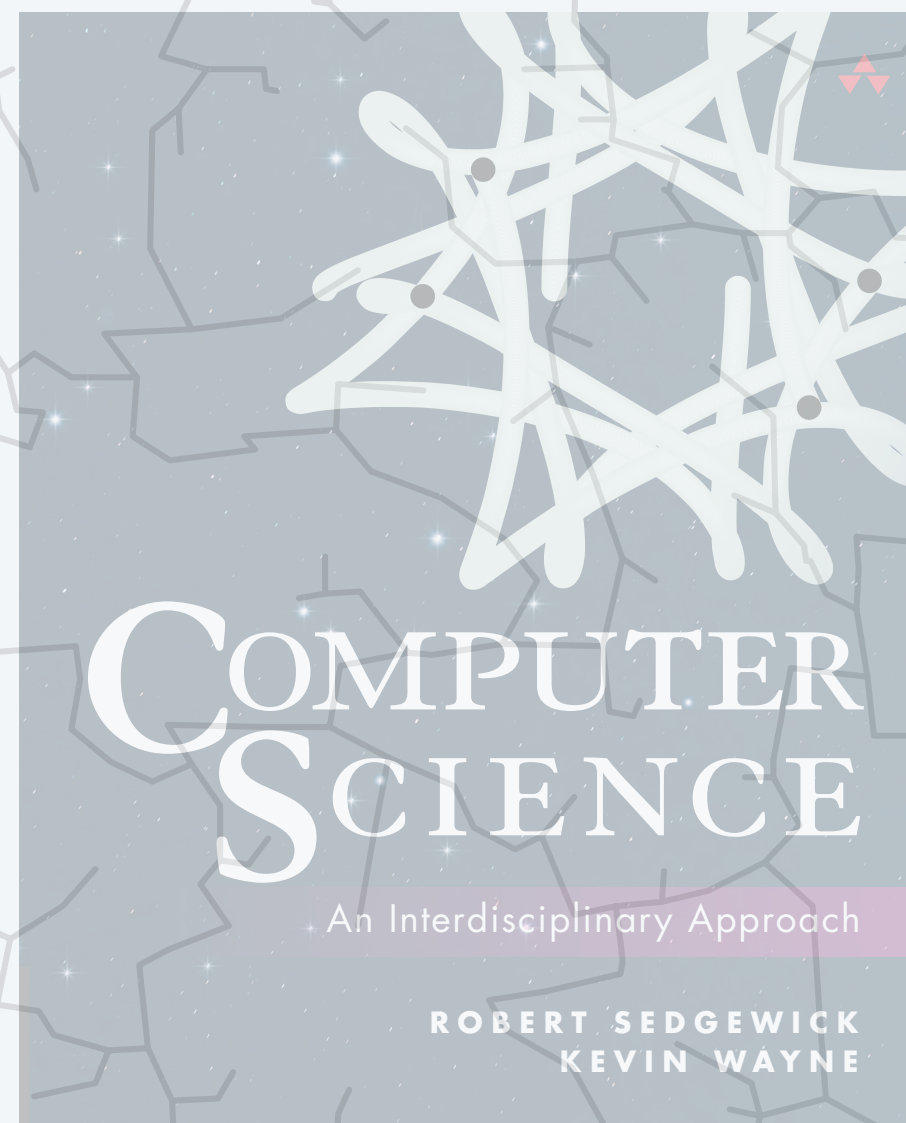
string concatenation



```
~/cos126/datatypes> java Ruler
0 1 0 2 0 1 0 3 0 1 0 2 0 1 0
```

ruler0	ruler1	ruler2	ruler3
<i>undeclared</i>	<i>undeclared</i>	<i>undeclared</i>	<i>undeclared</i>
"0"	<i>undeclared</i>	<i>undeclared</i>	<i>undeclared</i>
"0"	"0 1 0"	<i>undeclared</i>	<i>undeclared</i>
"0"	"0 1 0"	"0 1 0 2 0 1 0"	<i>undeclared</i>
"0"	"0 1 0"	"0 1 0 2 0 1 0"	"0 1 0 2 0 1 0 3 0 1 0 2 0 1 0"

trace of variable values (after each statement)



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

1.2 BUILT-IN DATA TYPES

- *types, variables, and assignments*
- *strings*
- *integers*
- *floating-point numbers*
- *type conversion*

The *int* data type

Typical usage: math with integers; counting steps in a loop; accessing elements in an array.

values	<i>integers between -2^{31} and $2^{31} - 1$</i>				
example literals	1234 99 0 1000000 -3				
arithmetic operations	<i>add</i>	<i>subtract</i>	<i>multiply</i>	<i>integer divide</i>	<i>remainder</i>
arithmetic 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	drop fractional part
20 % 3	2	remainder
20 / 0	-	ArithmeticException
<u>2147483647</u> + 1	-2147483648	integer overflow
$2^{31} - 1$		

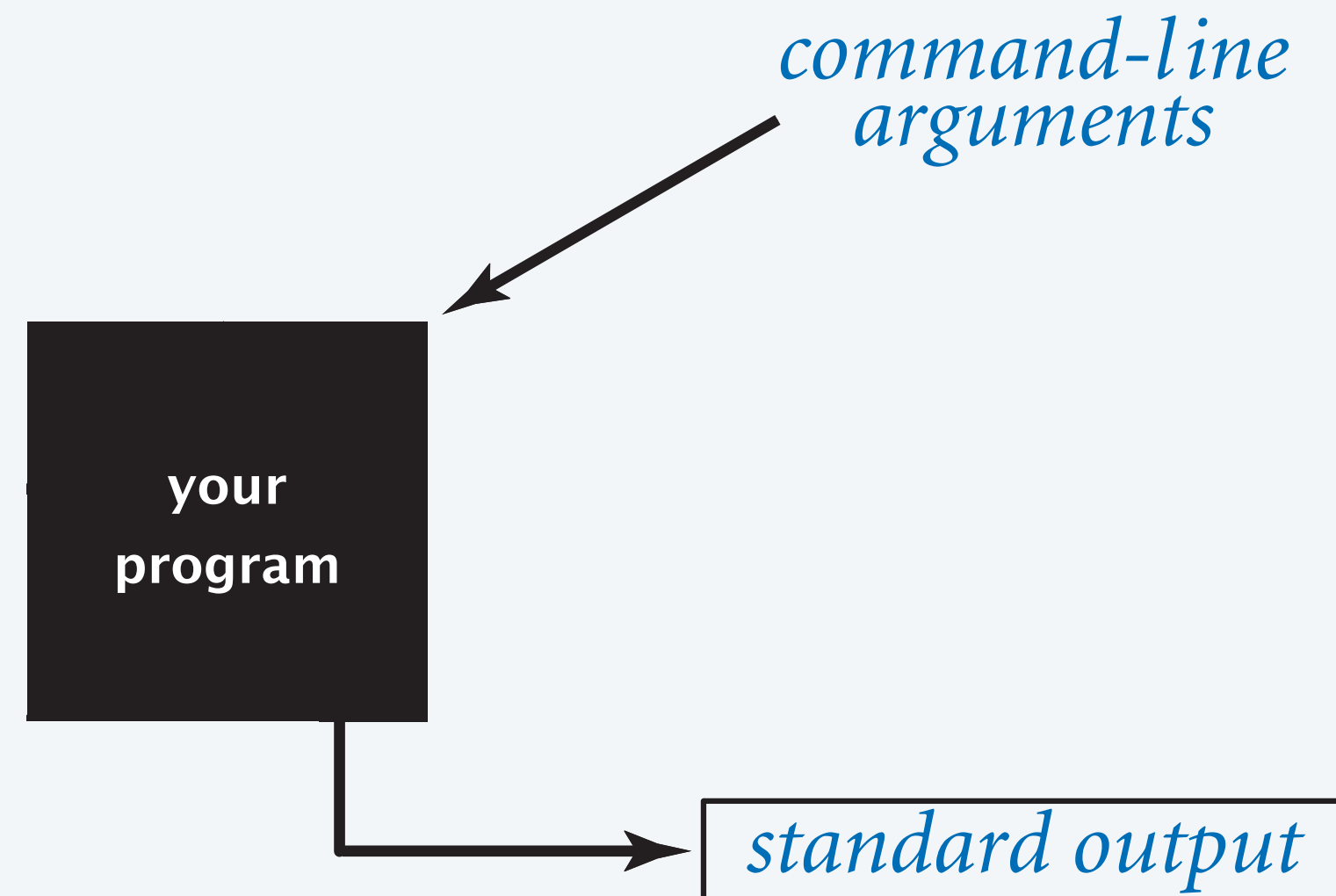
*arithmetic on `int` values produces an `int`
(or a runtime exception on divide by zero)*

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.



Q. How do we read integers from the command line?

A. Use the library method `Integer.parseInt()`, which converts a `String` to an `int`.

Q. How do we print integers?

A. Java converts an `int` to a `String` automatically when you concatenate it with 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*

*when either operand is a String,
Java interprets + as string concatenation
(converting the other operand to a String)*

```
~/cos126/datatypes> java IntOps 20 3
```

20 + 3 = 23

20 * 3 = 60

20 / 3 = 6

20 % 3 = 2

← 20 = 6×3 + 2

```
~/cos126/datatypes> java IntOps 1234 10
```

1234 + 10 = 1244

1234 * 10 = 12340

1234 / 10 = 123

1234 % 10 = 4

← 1234 = 123×10 + 4

```
~/cos126/datatypes> java IntOps 1234 Hello
```

Exception in thread "main"

java.lang.NumberFormatException:

For input string: "Hello"

...

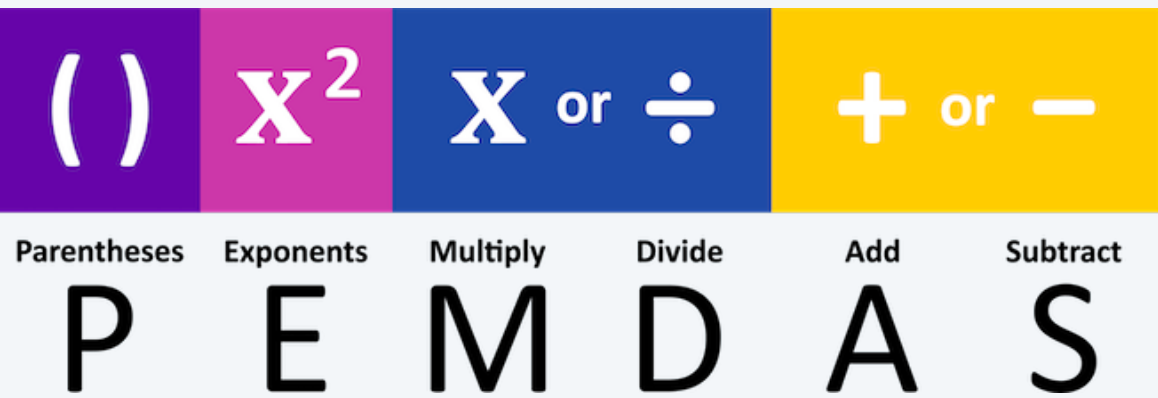
at IntOps.main(IntOps.java:4)

← *line number of
runtime error*

Evaluating expressions

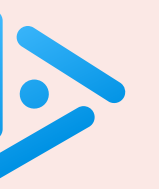
Q. How does Java evaluate an expression?

- **Literal:** evaluates to its value.
- **Variable name:** evaluates to the value stored in the variable.
- **Operator:** evaluates its operands, then computes a value.
- **Compound expression:** **precedence/associativity** determine which operator applies first.
- **Parenthesized expression:** evaluated first (overrides precedence/associativity).



expression	value	remark
1 + 2	3	apply operator to operands
1 + 2 * 6	13	* has higher precedence than +
1 - 2 - 6	-7	left-to-right associative
1 - (2 - 6)	5	parentheses

← Java has over 40 operators and 15 precedence levels (use parentheses for clarity)

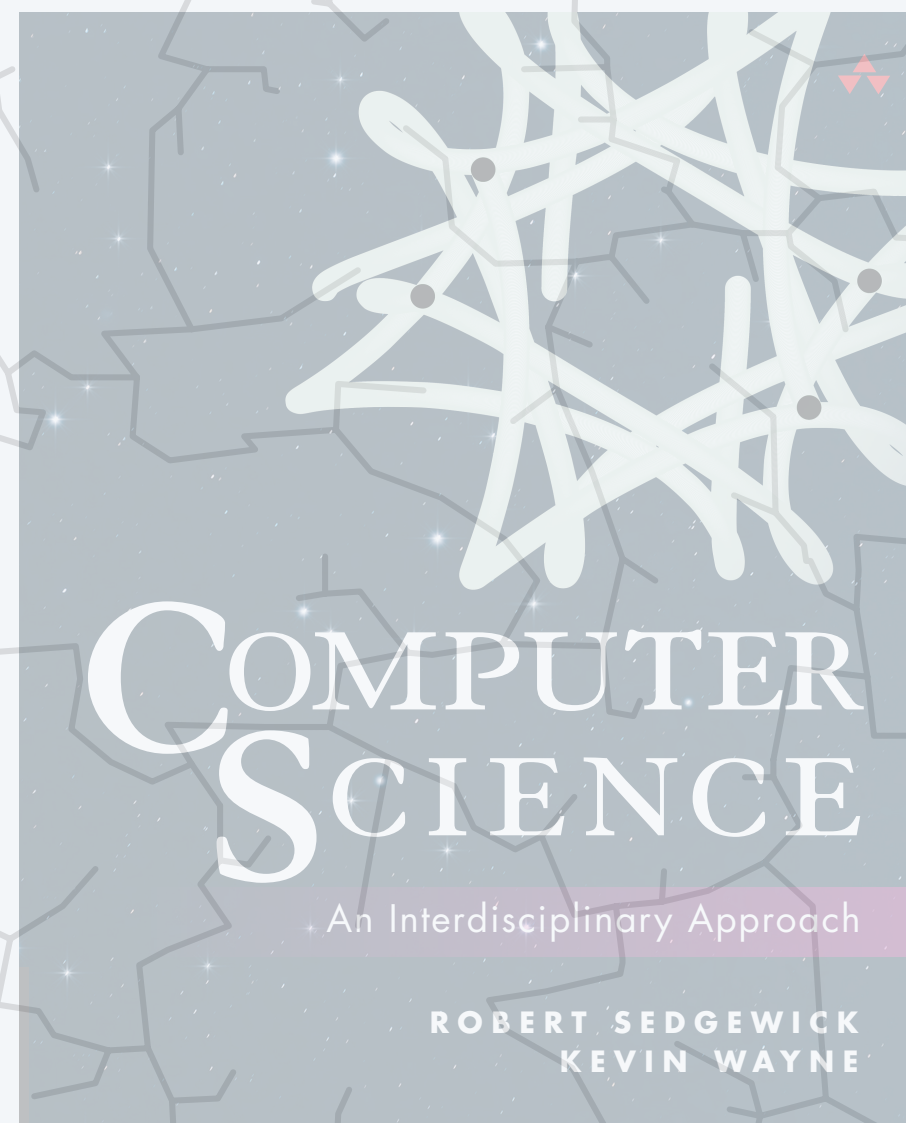


What value does the following Java expression evaluate to?

```
8 / 2*(2+2)
```

$8 \div 2(2+2) = ?$

- A. 1
- B. 1.0
- C. 16
- D. 16.0



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

1.2 BUILT-IN DATA TYPES

- *types, variables, and assignments*
- *strings*
- *integers*
- ***floating-point numbers***
- *type conversion*

The *double* data type

Typical usage: calculations with real numbers; measurements; scientific data.

values	IEEE floating-point numbers				
example literals	18.25	-2.0	1.4142135623730951	6.022E23	
arithmetic operations	add	subtract	multiply	divide	remainder
arithmetic operators	+	-	*	/	%

only 2^{64} possible `double` values
(most real numbers are not exactly representable)

6.022×10^{23}
(scientific notation)

expression	value	remark
<code>1.5 + 0.25</code>	1.75	
<code>1.5 - 0.25</code>	1.25	
<code>1.5 * 2.0</code>	3.0	
<code>5.0 / 3.0</code>	1.6666666666666667	not exactly $\frac{5}{3}$
<code>-1.0 / 0.0</code>	-Infinity	a well-defined value
<code>0.0 / 0.0</code>	NaN	“not a number”

arithmetic on `double` values always produces a `double`
(possibly a special value, but never a runtime exception)

exactly representable only if
denominator is a powers of 2
(e.g., $\frac{1}{2}$ or $\frac{9}{8}$; not $\frac{1}{10}$ or $\frac{5}{3}$)

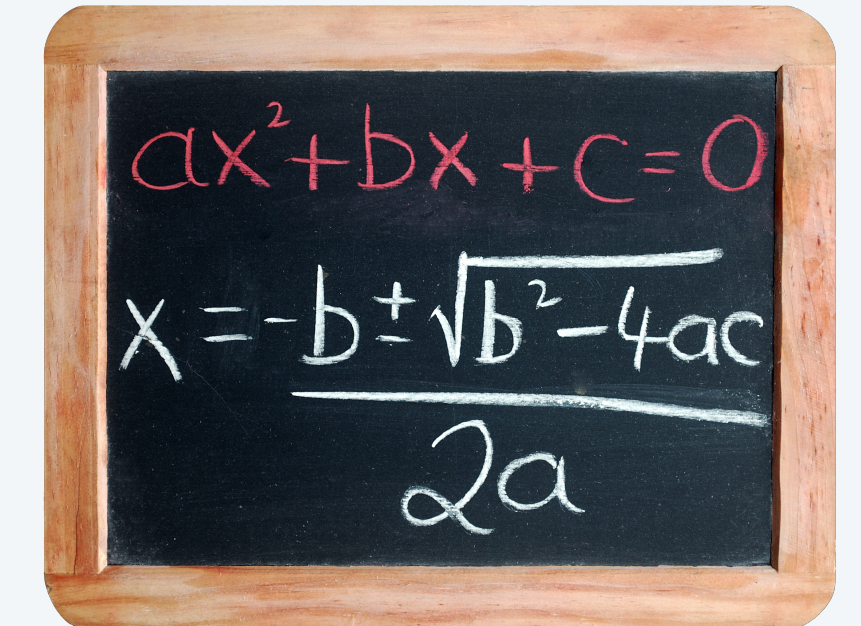
Excepts from Java's *Math* library

Math library function	description	
static double <code>abs</code> (double a)	absolute value of a	← also defined for int
static double <code>max</code> (double a, double b)	maximum of a and b	
static double <code>min</code> (double a, double b)	minimum of a and b	
static double <code>sin</code> (double theta)	sine (sin θ)	
static double <code>cos</code> (double theta)	cosine (cos θ)	
static double <code>tan</code> (double theta)	tangent (tan θ)	
static double <code>exp</code> (double a)	exponential (e^a)	
static double <code>log</code> (double a)	natural logarithm ($\log_e a$)	
static double <code>sqrt</code> (double a)	positive square root ($\sqrt{a}$)	
static double <code>pow</code> (double a, double b)	power (a^b)	
static long <code>round</code> (double a)	round to the nearest integer	
static double <code>random</code> ()	pseudorandom number in [0, 1)	
static double E	double value closest to e	
static double PI	double value closest to π	

expression	value
<code>Math.max(1.0, 2.5)</code>	2.5
<code>Math.cos(0.0)</code>	1.0
<code>Math.sqrt(2.0)</code>	1.4142135623730951
<code>Math.random()</code>	0.7707780210347349
<code>Math.PI</code>	3.141592653589793

Solving quadratic equations

Goal. Given a , b , and c , compute the real solutions to $ax^2 + bx + c = 0$. Assume $a \neq 0$.


$$ax^2 + bx + c = 0$$
$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

quadratic formula

```
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  $ax^2 + bx + c = 0$ .  
        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 the two roots.  
        System.out.println(root1);  
        System.out.println(root2);  
    }  
}
```

```
~/cos126/datatypes> java Quadratic 1.0 -3.0 2.0  
2.0  
1.0
```

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

```
~/cos126/datatypes> java Quadratic 1.0 -1.0 -1.0  
1.618033988749895  
-0.6180339887498949
```

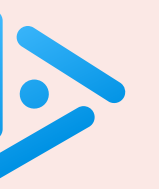
$$\longleftarrow \frac{1 \pm \sqrt{5}}{2}$$

$$x^2 - x - 1$$

```
~/cos126/datatypes> java Quadratic 1.0 1.0 1.0  
NaN  
NaN
```

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

$$x^2 + x + 1$$



What does `Quadratic` print for these command-line arguments?

```
~/cos126/datatypes> java Quadratic 1.0 -1.4 0.49  
???  
???
```

$$x^2 - 1.4x + 0.49 = (x - 0.7)^2$$

- A. 0.7
- B. 0.7 and 0.7
- C. Two numbers close to 0.7
- D. NaN and NaN

Floating-point catastrophe

How a floating-point error caused a Patriot missile to fail.

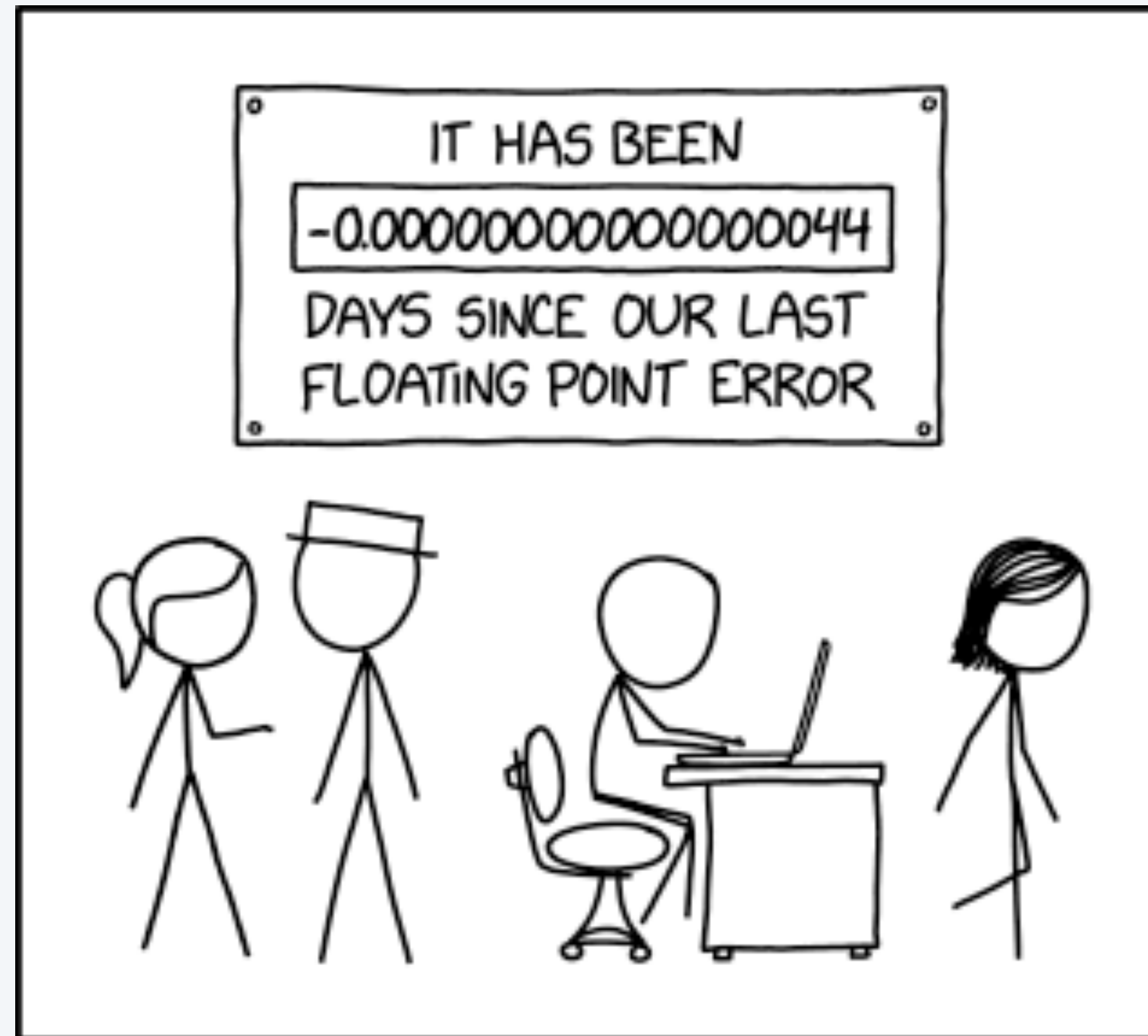
- Impact: 28 deaths; hundreds of injuries.
- Cause: $1/10$ second is not exactly representable (binary floating point).
- Outcome: tiny error $\times$ long uptime $\rightarrow$ clock drift $\rightarrow$ tracking error $\rightarrow$ failed intercept.



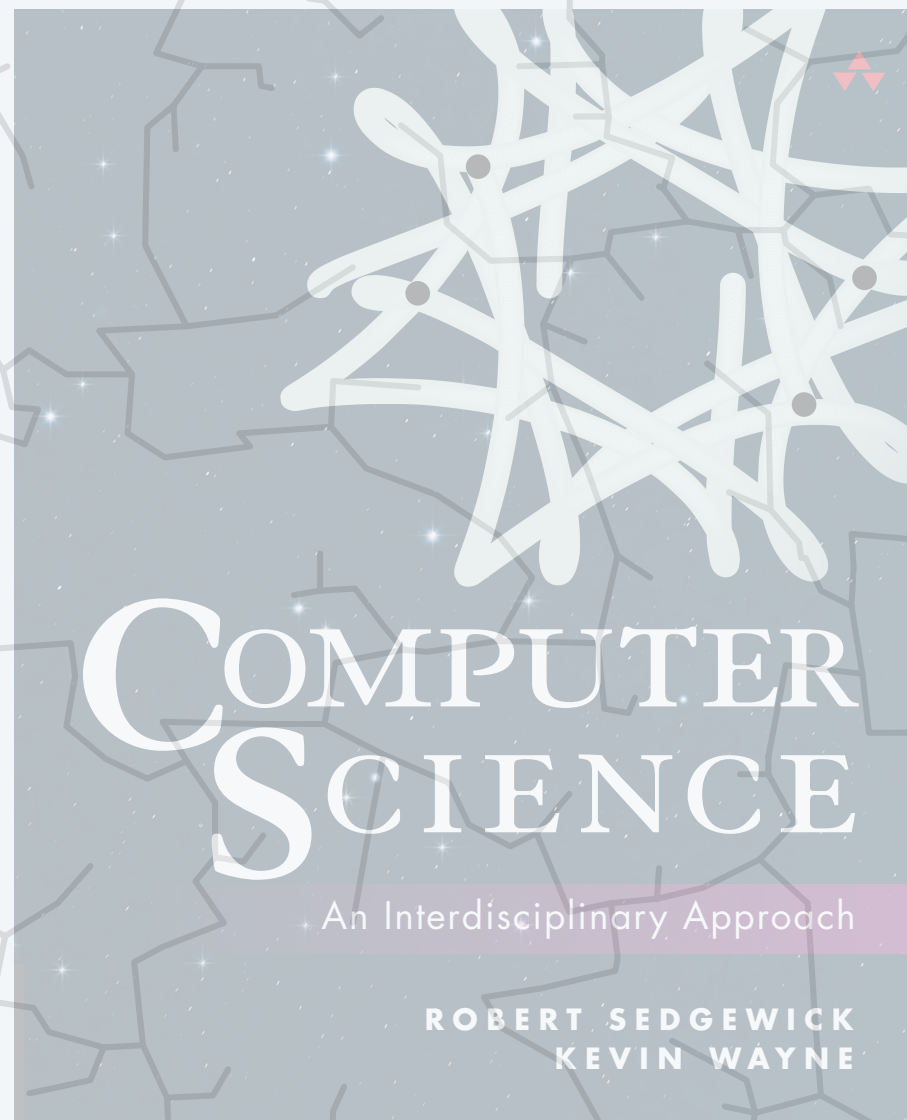
Scud Missile Hits a U.S. Barracks, Killing 27



Associated Press
An Iraqi Scud missile that demolished a barracks yesterday near Dhahran, Saudi Arabia, killed 27 American soldiers and wounded 98, according to officials. A soldier told photographers to leave the scene.



<https://xkcd.com/3228>



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

1.2 BUILT-IN DATA TYPES

- *types, variables, and assignments*
- *strings*
- *integers*
- *floating-point numbers*
- ***type conversion***

Types and type-checking

Types determine which operations are allowed on values (and what they mean).

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

```
~/cos126/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
```

compiler rejects the program because the types don't match

Java compiler. Checks your code for type mismatches.

Types and type-checking

Types determine which operations are allowed on values (and what they mean).

operator	int	double	String
+	<i>addition</i>	<i>addition</i>	<i>concatenation</i>
-	<i>subtraction</i>	<i>subtraction</i>	<i>no</i>
*	<i>multiplication</i>	<i>multiplication</i>	<i>no</i> ← <i>cannot multiply two String values (compile-time error)</i>
/	<i>integer division</i>	<i>division</i>	<i>no</i>
⋮	⋮	⋮	⋮

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

- Benefit: the compiler catches many errors automatically (at compile time).
- Drawback: you must declare types in your code.

Type conversion is a routine part of programming

Automatic type conversions.

- String conversion: from any type to `String` (via concatenation).
- Numeric promotion: from `int` to `double` (when a `double` is expected).

every int can be exactly represented as a double

expression	type	value
"x = " + 99	String	"x = 99"
11 * 0.25	double	2.75

Conversions via library methods.

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

expression	type	value
<code>Integer.parseInt("12345")</code>	<code>int</code>	12345
<code>Double.parseDouble("2.5")</code>	<code>double</code>	2.5

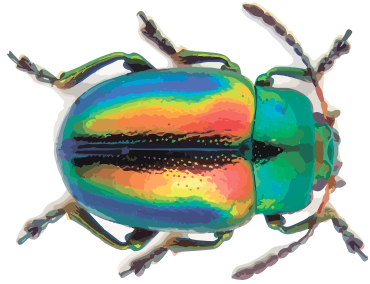
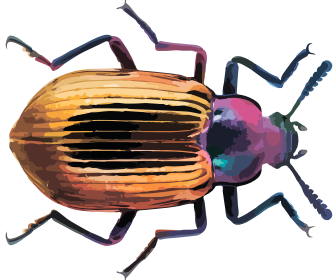
Explicit casts from one type to another.

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

	expression	type	value
<i>cast operator</i>	<code>(int) 2.71828;</code>	<code>int</code>	2
	<code>(double) 1 / 4;</code>	<code>double</code>	0.25

cast has higher precedence than /

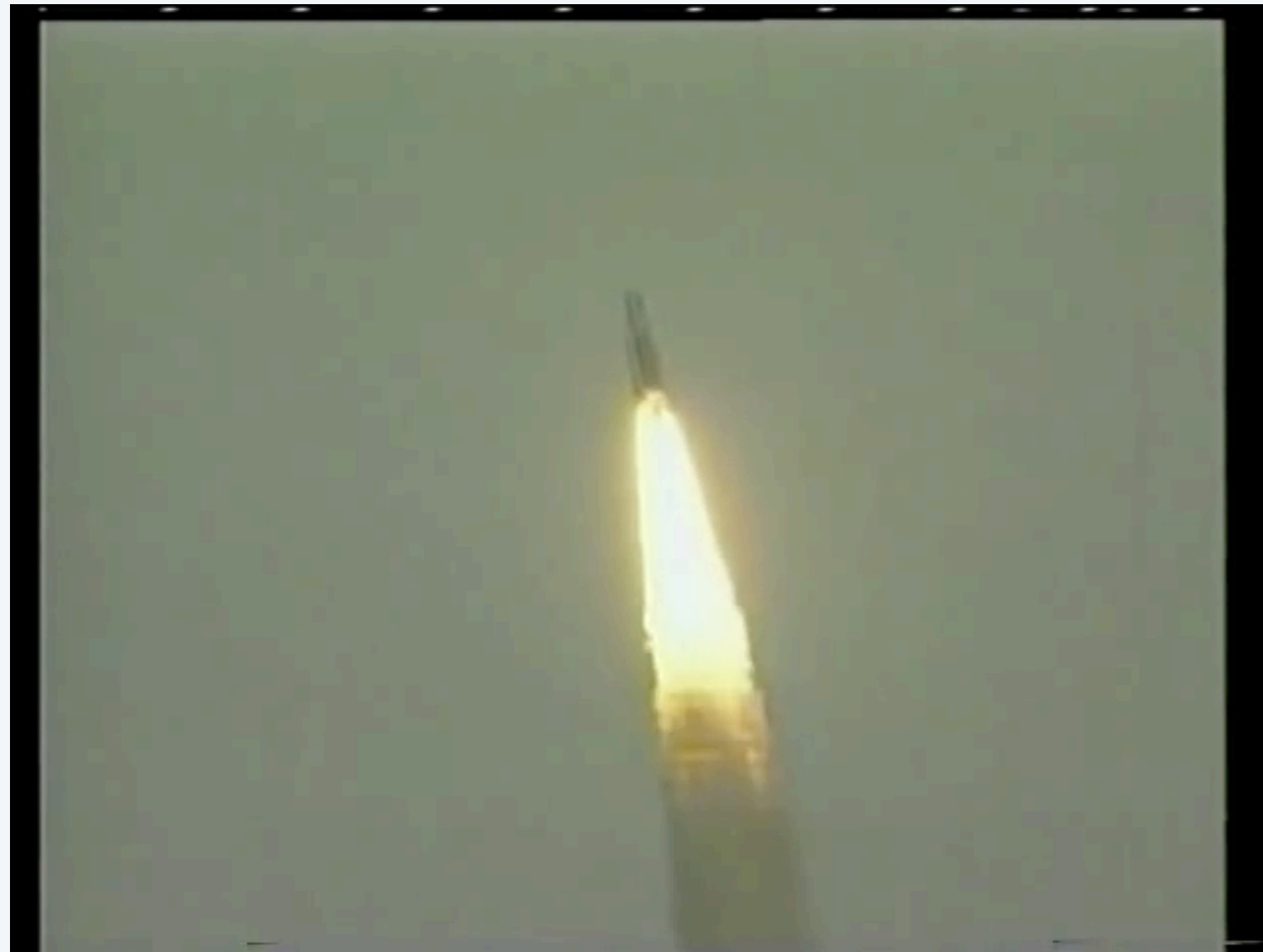
Common type-conversion bugs

bug	buggy Java code	result	explanation
	<pre>double x = Integer.parseInt("3.14");</pre>	<i>runtime exception</i>	<i>can't parse "3.14" as an integer</i>
	<pre>int a = 6; int b = 7; System.out.println("sum = " + a + b);</pre>	<i>prints "sum = 67"</i>	<i>string conversion, not integer addition</i>
	<pre>int accepted = 1782; int applied = 39644; double rate = (double) (accepted / applied);</pre>	<i>rate is 0.0</i>	<i>integer division; then cast to double</i>

Type-conversion catastrophe

Ariane 5 rocket failure.

- Self-destructed 39 seconds after liftoff.
- Cause: **unsafe conversion** from 64-bit floating-point to 16-bit integer.
- Impact: ~ \$7B loss after ~ 10 years of development.



*worked in Ariane 4;
failed in Ariane 5
(higher speeds → overflow)*

Lesson 1. Declared types help the compiler/IDE catch many bugs early.

Lesson 2. Conversions are a common source of bugs; treat them with care.

Simulating a fair roll

Goal. Given an integer $n > 0$, pick an integer $0, 1, 2, \dots, n - 1$ uniformly at random.

Die roll. Add 1 to shift the range to $1, 2, 3, \dots, n$.



$n = 6$



$n = 10$



$n = 100$

Generating pseudorandom integers

Goal. Given an integer $n > 0$, pick an integer $0, 1, 2, \dots, n - 1$ uniformly at random.

Useful library method. `Math.random()` returns a pseudorandom `double` value in `[0.0, 1.0)`. *← can return 0.0
can't return 1.0*

Approach. Scale to the desired range, then cast to `int` (truncates). *not truly random,
but close enough for
most applications*

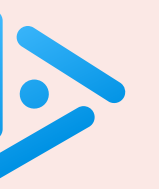
```
public class RandomInt {  
    public static void main(String[] args) {  
  
        int n = Integer.parseInt(args[0]);  
        double r = Math.random();  
        int result = (int) (r * n);  
  
        System.out.println(result);  
    }  
}
```

*String to int
(library method)*

*double to int
(cast)*

*int to double
(automatic)*

```
~/cos126/datatypes> java RandomInt 6  
3  
  
~/cos126/datatypes> java RandomInt 6  
0  
  
~/cos126/datatypes> java RandomInt 6  
5  
  
~/cos126/datatypes> java RandomInt 10000  
3184
```

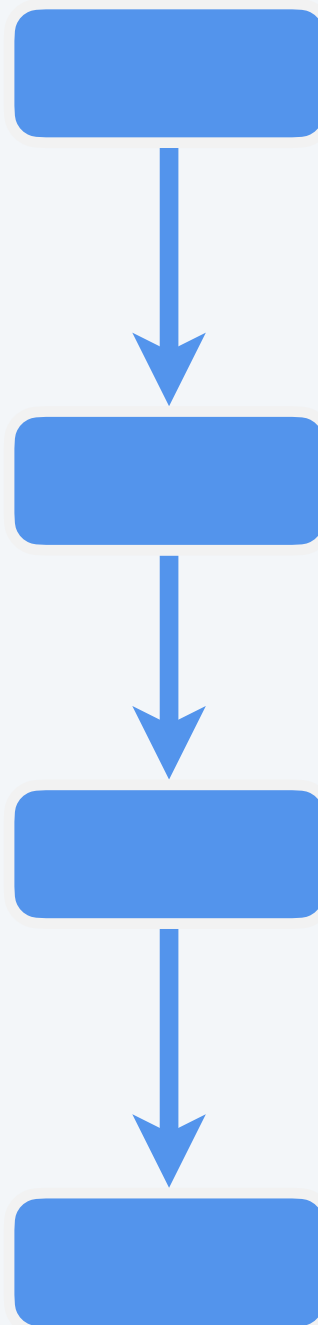
Which expression produces a pseudorandom **even** integer in $0, 2, 4, \dots, 2n - 2$?

- A. `2 * (int) n * Math.random()`
- B. `2 * (int) (n * Math.random())`
- C. `2 * n * (int) Math.random()`
- D. `(int) (2 * n * Math.random())`

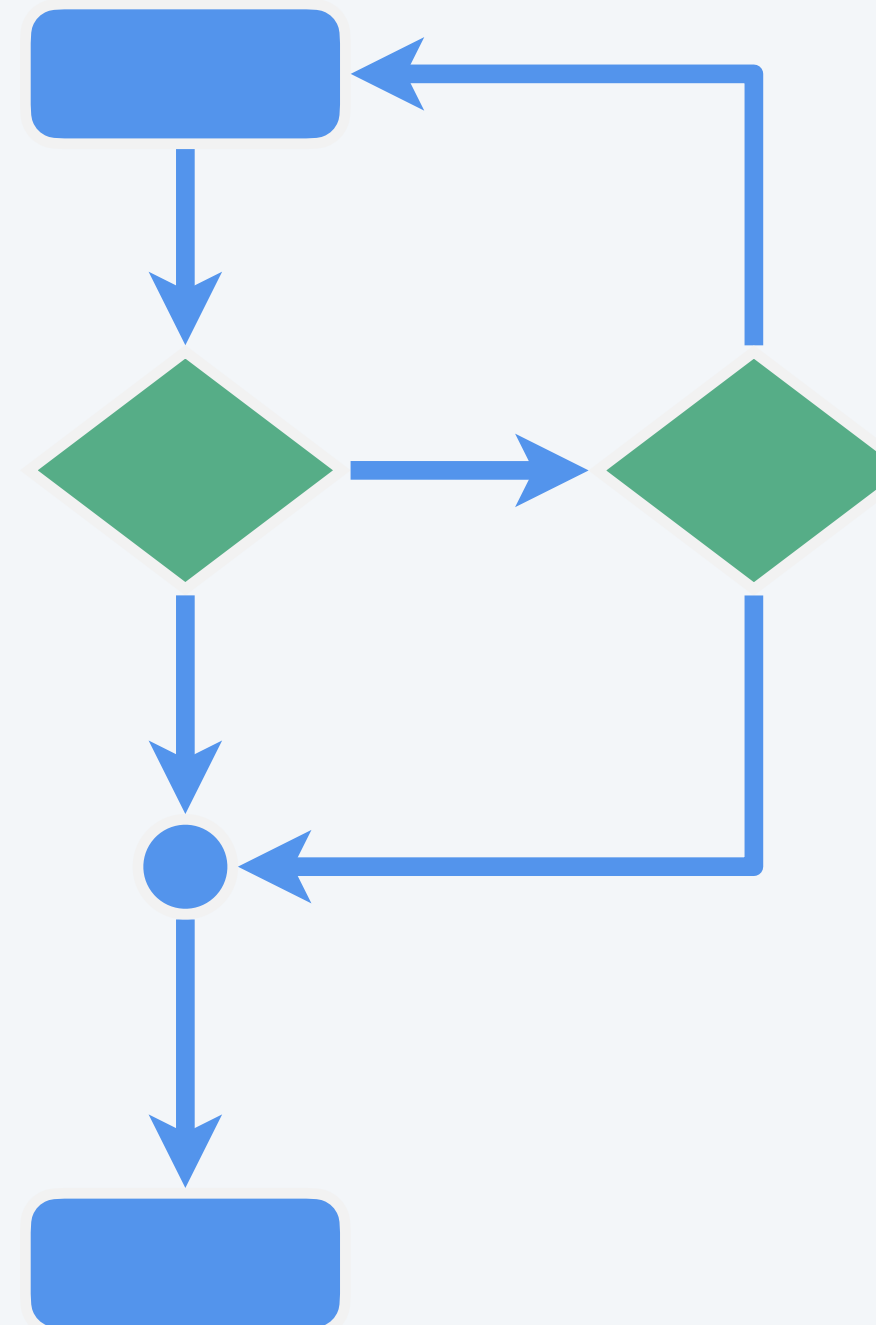
Overview

This lecture. Write programs using **variables**, **expressions**, and **print** statements.

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



straight-line control flow



control flow with conditionals and loops

Credits

media	source	license
<i>Phones Away</i>	<u>NorthernStar</u>	
<i>AI puppeteer</i>	<u>Getty Images</u>	
<i>PEMDAS</i>	<u>Mometrix</u>	
<i>PEMDAS meme</i>	<u>New York Times</u>	
<i>Solving Quadratic Equations</i>	<u>Adobe Stock</u>	<u>education license</u>
<i>Bugs</i>	<u>Adobe Stock</u>	<u>education license</u>
<i>Patriot Missile Launcher</i>	<u>Raytheon</u>	
<i>Scud Missile Hits a U.S. Barracks</i>	<u>New York Times</u>	
<i>Floating-Point Counter</i>	<u>xkcd</u>	<u>CC BY-NC 2.5</u>
<i>Ariane 5 Rocket Launch</i>	<u>European Space Agency</u>	
<i>Two Red Dice</i>	<u>Wikimedia</u>	<u>public domain</u>
<i>Ten-Sided Die</i>	<u>Wikimedia</u>	<u>CC BY-SA 3.0</u>
<i>Hundred-Sided Die</i>	<u>gameoutonline.com</u>	