INTERACTIVE LEARNING

- In this class, we will sometimes work in tandem between programming and discussion
- If you get lost, I will have the programs written in the slides, or you can ask
- To facilitate:
  - Make a new directory for the day
  - Open sublime and the terminal and be ready to go

CHAPTER 2
ELEMENTARY PROGRAMMING

ACKNOWLEDGEMENT: THESE SLIDES ARE ADAPTED FROM SLIDES PROVIDED WITH INTRODUCTION TO JAVA PROGRAMMING, LIANG (PEARSON 2014)

NEXT STEP IN PROGRAMMING

- Computations! Support for basic mathematics
- In pseudocode, we might write the following
  - $x \leftarrow 0$
  - $a \leftarrow \pi r^2$
- Here we make variables (store data) and give values ($\leftarrow$ means “gets”, note I do not use “equals” in pseudocode)

EXAMPLE: COMPUTING THE AREA

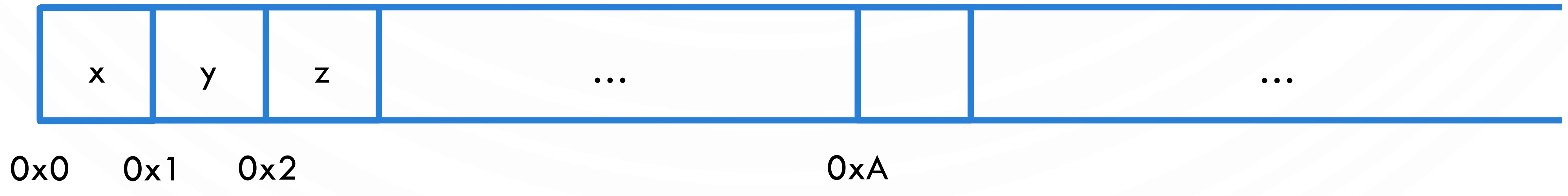
```java
public class ComputeArea {
  public static void main(String[] args) {
    double radius; // Declare radius
    double area; // Declare area

    // Assign a radius
    radius = 20; // New value is radius

    // Compute area
    area = radius * radius * 3.14159;

    // Display results
    System.out.println("The area for the circle of radius " +
      radius + " is " + area);
  }
}
```

TRACING

- Tracing is an activity used by computer scientists to follow a piece of code to check correctness, find errors, etc
- We will learn how to trace programs and follow execution of code exactly as a computer might
- But first, a note on memory

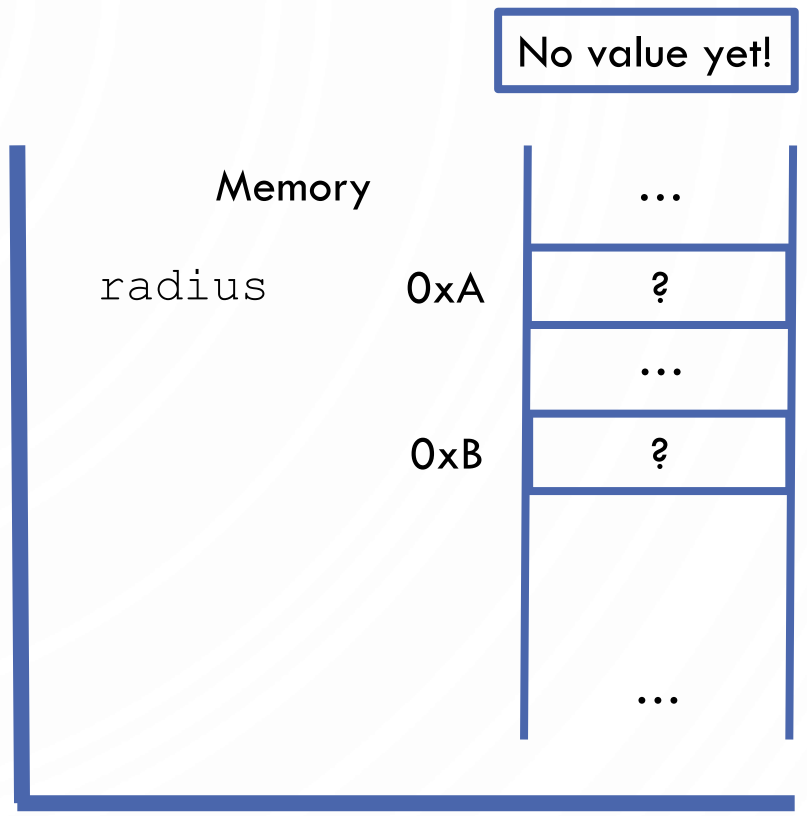
MEMORY

- **Memory** is storage for data and programs
- We will pretend that memory is an infinitely long piece of **tape** separated into different **cells**
- Each cell has an **address**, i.e., a location, and a **value**
- In the computer these values are represented in **binary** (0s and 1s) and addresses are located in **hexadecimal** (base 16, 0x)



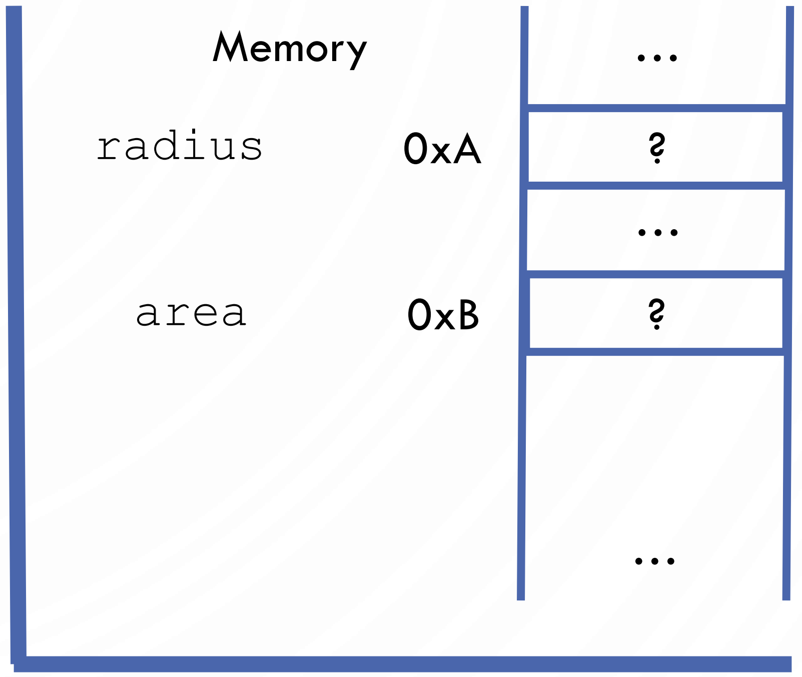
EXAMPLE: COMPUTING THE AREA

```java
public class ComputeArea {
  public static void main(String[] args) {
    double radius; // Declare radius
    double area; // Declare area

    // Assign a radius
    radius = 20; // New value is radius

    // Compute area
    area = radius * radius * 3.14159;

    // Display results
    System.out.println("The area for the circle of radius " +
      radius + " is " + area);
  }
}
```



EXAMPLE: COMPUTING THE AREA

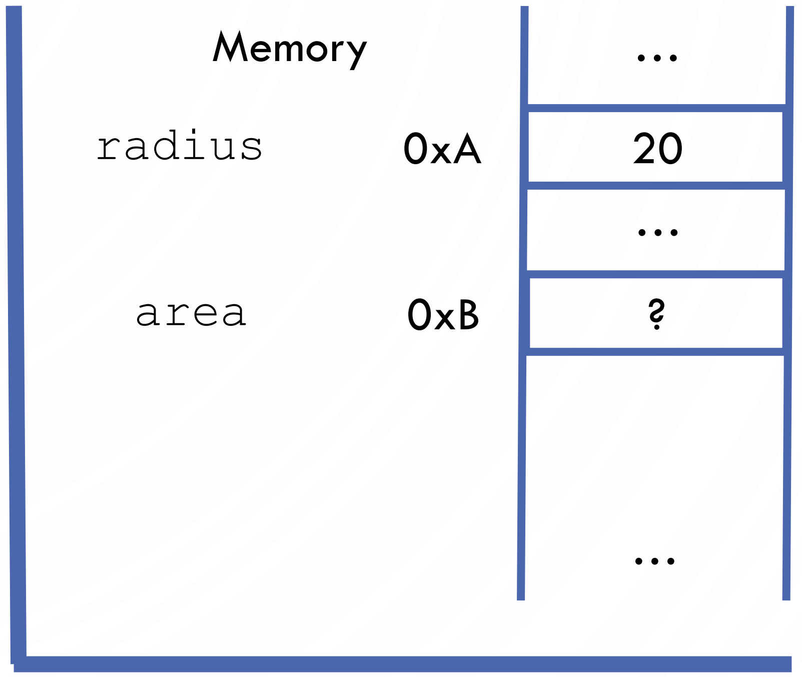
```java
public class ComputeArea {
  public static void main(String[] args) {
    double radius; // Declare radius
    double area; // Declare area

    // Assign a radius
    radius = 20; // New value is radius

    // Compute area
    area = radius * radius * 3.14159;

    // Display results
    System.out.println("The area for the circle of radius " +
      radius + " is " + area);
  }
}
```



EXAMPLE: COMPUTING THE AREA

Special symbol = means assignment, or giving a value to a variable

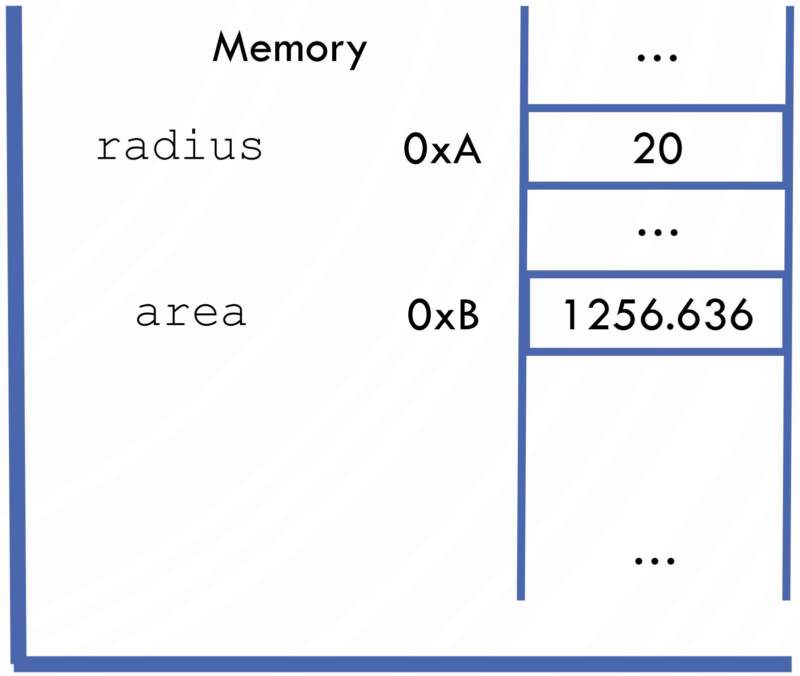
```java
public class ComputeArea {
  public static void main(String[] args) {
    double radius; // Declare radius
    double area; // Declare area

    // Assign a radius
    radius = 20; // New value is radius

    // Compute area
    area = radius * radius * 3.14159;

    // Display results
    System.out.println("The area for the circle of radius " +
      radius + " is " + area);
  }
}
```



Certain symbols, e.g., *, compute new values. It is like a calculator!

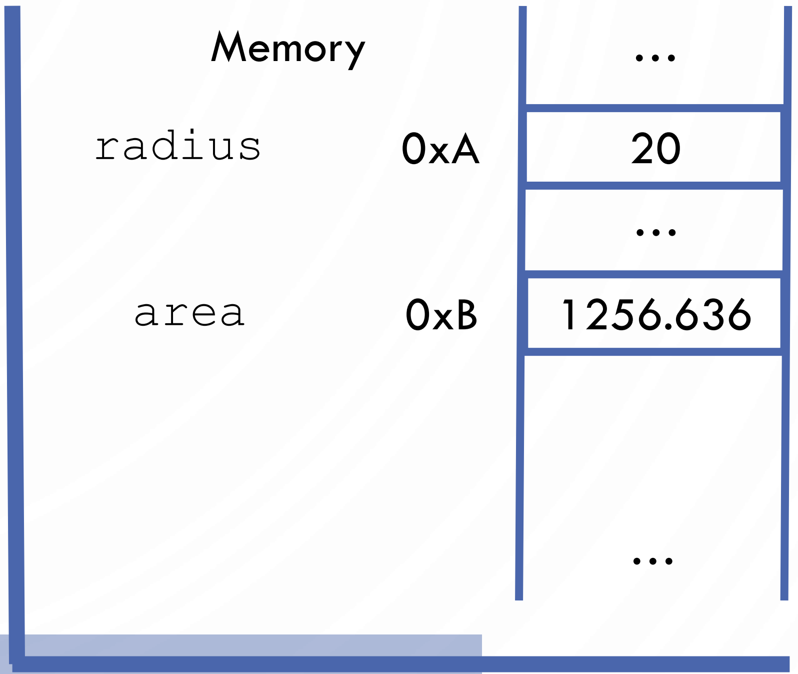
EXAMPLE: COMPUTING THE AREA

```
public class ComputeArea {
  public static void main(String[] args) {
    double radius; // Declare radius
    double area; // Declare area

    // Assign a radius
    radius = 20; // New value is radius

    // Compute area
    area = radius * radius * 3.14159;

    // Display results
    System.out.println("The area for the circle of radius " +
      radius + " is " + area);
  }
}
```



EXAMPLE: COMPUTING THE AREA

```java
public class ComputeArea {
  public static void main(String[] args) {
    double radius; // Declare radius
    double area; // Declare area

    // Assign a radius
    radius = 20; // New value is radius

    // Compute area
    area = radius * radius * 3.14159;

    // Display results
    System.out.println("The area for the circle of radius " +
      radius + " is " + area);
  }
}
```



Print values to terminal

READING INPUT FROM THE CONSOLE

```
import java.util.Scanner; //Tell java you want to use a Scanner
public class ComputeArea {
  public static void main(String[] args) {
    System.out.print("Enter a radius: "); //Message to "prompt" the user
    Scanner in = new Scanner(System.in);  //Scanner gathers input

    double radius = in.nextDouble(); // Declare radius and assign a new value is from user!

    double area = radius * radius * 3.14159; // Declare and Compute area

    System.out.println("The area for the circle of radius " +
      radius + " is " + area); // Display results
  }
}
```

- Use a Scanner to gather input from a user!

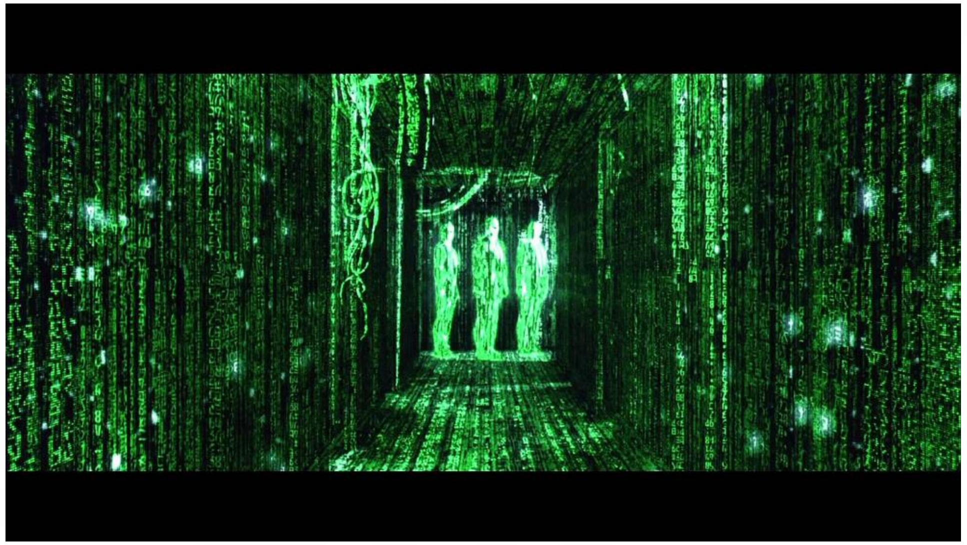
PROGRAMMING WITH DATA

THE DETAILS

HOLD ON A SEC!
I THOUGHT COMPUTERS WERE ALL 0'S AND 1'S?

0'S AND 1'S

- Yes, computers operate in 0's and 1's. So first off, javac (our compiler) converts our text into commands of 0's and 1's the computer can understand
- java (executing) interprets the 0's and 1's
- Memory also stores entirely 0's and 1's
- So what we need to know is how computers do this



INTEGER REPRESENTATION

- First, a look at our number system. It is base 10, meaning we use 10 different symbols (the digits). Lets look at an example number: 1037.

1	0	3	7
$10^3 = 1000$	$10^2 = 100$	$10^1 = 10$	$10^0 = 1$
$1 * 10^3 +$	$0 * 10^2 +$	$3 * 10^1 +$	$7 * 10^0 = 1037$

- And adding we use carry-and-add

	1	0	3	7
$+$	0	0	4	9
	1	0	8	6

INTEGER REPRESENTATION

- Synonymously, binary numbers work the same way. Except instead of base 10, it is base 2. A digit can only be 0 or 1. Example: 0010 0101

0	0	1	0	0	1	0	1
$2^7 = 128$	$2^6 = 64$	$2^5 = 32$	$2^4 = 16$	$2^3 = 8$	$2^2 = 4$	$2^1 = 2$	$2^1 = 1$
$0 * 2^7 +$	$0 * 2^6 +$	$1 * 2^5 +$	$0 * 2^4 +$	$0 * 2^3 +$	$1 * 2^2 +$	$0 * 2^1 +$	$1 * 2^1 = 37$

- And adding 0010 1010 + 0000 0101

	0	0	1	0	0	1	0	1
+	0	0	0	0	0	1	0	1
	0	0	1	0	1	0	1	0

- Note there are other common number systems: Octal (base 8, digits 0-7) and Hexadecimal (base 16, digits 0-9 and A-F, used for memory addresses)

INTEGER REPRESENTATION

- By limiting the number of bits, we limit the expressiveness of the data type
  - Means that if we only have 2 bits, we can only represent 4 numbers: 00, 01, 10, 11
- In programming, we must make conscious decisions about this otherwise there can be severe consequences

NUMERICAL DATA TYPES

We will use `int` and `double` most often in this class.

Name	Range	Storage Size
`byte`	-2^7 to $2^7 - 1$ (-128 to 127)	8-bit signed
`short`	-2^{15} to $2^{15} - 1$ (-32768 to 32767)	16-bit signed
`int`	-2^{31} to $2^{31} - 1$ (-2147483648 to 2147483647)	32-bit signed
`long`	-2^{63} to $2^{63} - 1$ (i.e., -9223372036854775808 to 9223372036854775807)	64-bit signed
`float`	Negative range: -3.4028235E+38 to -1.4E-45 Positive range: 1.4E-45 to 3.4028235E+38	32-bit IEEE 754
`double`	Negative range: -1.7976931348623157E+308 to -4.9E-324 Positive range: 4.9E-324 to 1.7976931348623157E+308	64-bit IEEE 754

ACTIVITY

- With a partner
- Convert the binary number 1001 1001 to decimal
- Add the binary number 0101 0101 to 1001 1001 (DO NOT DO THIS IN DECIMAL) and then convert to a decimal number
- Bonus: 0xA1 to decimal, add 0x0E to it and convert to decimal. Hint: 0x means that the number is a hexadecimal (base 16)

VARIABLES AND NAMING

IDENTIFIERS (NAMES)

- An **identifier** is a sequence of characters that consist of letters, digits, underscores (_), and dollar signs ($).
- An identifier must start with a letter, an underscore (_), or a dollar sign ($). It cannot start with a digit.
- An identifier cannot be a reserved word. (See Appendix A, “Java Keywords,” for a list of reserved words). An identifier cannot be true, false, or null.
- An identifier can be of any length.

DECLARING VARIABLES

- A **variable** is a named piece of data (memory). It stores a **value**!
- It has a **type** that defines how the memory is interpreted and what operations are allowed
- A **declaration** states the existence of a variable for use in the program. It defines the type and an identifier (name) for a variable.

```
int x;          // Declare x to be an integer variable
double radius;  // Declare radius to be a double variable
char a;         // Declare a to be a character variable
```

In pseudocode (just x, to say a variable) we often exclude the type for numeric variables

ASSIGNMENT STATEMENTS

- Assignment statements give values to a variable
- The type defines what possible values are allowable
- `x = 1;              // Assign 1 to x;`
- `radius = 1.0;      // Assign 1.0 to radius;`
- `a = 'A';           // Assign 'A' to a;`

In contrast to pseudocode, java uses =, not ← for assignment!

DECLARING AND INITIALIZING IN ONE STEP

- **Initialization** is how we refer to the very first assignment of a variable's value
- Declarations and initialization are allowed in one step
- `int x = 1;`
- `double d = 1.4;`

NAMED CONSTANTS

- Often, we need constants in programs, e.g., π. Use the keyword `final` to specify a variable can ONLY be initialized.

- `final double PI = 3.14159;`
- `final int SIZE = 3;`

NAMING CONVENTIONS

- Choose meaningful and descriptive names.
- Variables and method names: Use lowercase. If the name consists of several words, concatenate all in one, use lowercase for the first word, and capitalize the first letter of each subsequent word in the name, called **camelCasing**.
  - radius
  - area
  - computeArea

NAMING CONVENTIONS

- Class names: Capitalize the first letter of each word in the name
    - ComputeArea.
- Constants: Capitalize all letters in constants, and use underscores to connect words
    - PI
    - MAX_VALUE

LITERALS

- A **literal** is a constant value that appears directly in the program. For example, 34, 1,000,000, and 5.0 are literals in the following statements:

- `int i = 34;`
- `long x = 1000000;`
- `double d = 5.0;`

INTEGER LITERALS

- An integer literal can be assigned to an integer variable as long as it can fit into the variable. A compilation error would occur if the literal were too large for the variable to hold. For example, the statement byte b = 1000 would cause a compilation error, because 1000 cannot be stored in a variable of the byte type.
- An integer literal is assumed to be of the int type, whose value is between -2^{31} (-2147483648) to $2^{31} - 1$ (2147483647). To denote an integer literal of the long type, append it with the letter L or l. L is preferred because l (lowercase L) can easily be confused with 1 (the digit one).

FLOATING-POINT LITERALS

- Floating-point literals are written with a decimal point. By default, a floating-point literal is treated as a double type value. For example, 5.0 is considered a double value, not a float value. You can make a number a float by appending the letter f or F, and make a number a double by appending the letter d or D. For example, you can use 100.2f or 100.2F for a float number, and 100.2d or 100.2D for a double number.

SCIENTIFIC NOTATION

- Floating-point literals can also be specified in scientific notation, for example, 1.23456e+2, same as 1.23456e2, is equivalent to 123.456, and 1.23456e-2 is equivalent to 0.0123456. E (or e) represents an exponent and it can be either in lowercase or uppercase.

READING NUMBERS FROM THE KEYBOARD

1.
```
Scanner input =
    new Scanner(System.in);
```
2.
```
int value =
    input.nextInt();
```

Method	Description
`nextByte()`	reads an integer of the `byte` type.
`nextShort()`	reads an integer of the `short` type.
`nextInt()`	reads an integer of the `int` type.
`nextLong()`	reads an integer of the `long` type.
`nextFloat()`	reads a number of the `float` type.
`nextDouble()`	reads a number of the `double` type.

EXPRESSIONS

NUMERIC OPERATORS

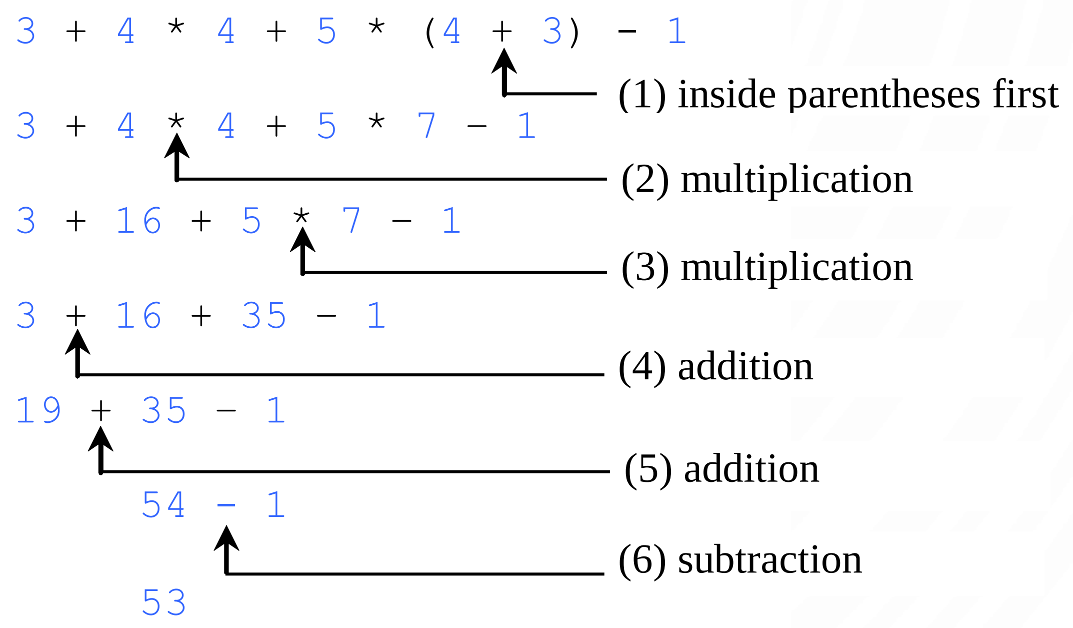
Name	Meaning	Example	Result
`+`	Addition	`34 + 1`	`35`
`-`	Subtraction	`34.0 – 0.1`	`33.9`
`*`	Multiplication	`300 * 30`	`9000`
`/`	Division	`1.0 / 2.0`	`0.5`
`%`	Remainder	`20 % 3`	`2`

ARITHMETIC EXPRESSIONS

- **Expressions** are combinations of literals, variables, operations, and method calls that generate new values
- $\frac{3+4x}{5} - \frac{10(y-5)(a+b+c)}{x} + 9\left(\frac{4}{x} + \frac{9+x}{y}\right)$
- is translated to
- `(3+4*x)/5 – 10*(y-5)*(a+b+c)/x + 9*(4/x + (9+x)/y)`

HOW TO EVALUATE AN EXPRESSION

- Though Java has its own way to evaluate an expression behind the scene, the result of a Java expression and its corresponding arithmetic expression are the same. Therefore, you can safely apply the arithmetic rules for evaluating a Java expression.



EXERCISE

- Write a program to compute the average of three numbers
    - Make the numbers integers to see what happens?
    - Make the numbers floats to see what happens?
- Trace the execution in memory if the user enters 3, 5, 7
- Write a program to draw a circle based on user input!

INTEGERS DIVISION

- Integers do not store decimals
- Division computes how many times a divisor evenly goes into a number
- Remainder (modulus) computes what is left over
- 5 / 2 yields an integer 2
- 5 % 2 yields 1 (the remainder of the division)
- Practice: What is 3456421 % 2? 25%3? 87%4?

FLOATING-POINT NUMBERS (DOUBLE)

- Calculations involving floating-point numbers are approximated because these numbers are not stored with complete accuracy. For example,
- `System.out.println(1.0 - 0.1 - 0.1 - 0.1 - 0.1 - 0.1);`
- displays 0.5000000000000001, not 0.5, and
- `System.out.println(1.0 - 0.9);`
- displays 0.09999999999999998, not 0.1. Integers are stored precisely. Therefore, calculations with integers yield a precise integer result.

DOUBLE VS. FLOAT

- The double type values are more accurate than the float type values. For example,

```
System.out.println("1.0 / 3.0 is " + 1.0 / 3.0);
```

displays $\texttt{1.0 / 3.0 is 0.}\underbrace{\texttt{3333333333333333}}_{\text{16 digits}}$

```
System.out.println("1.0F / 3.0F is " + 1.0F / 3.0F);
```

displays $\texttt{1.0F / 3.0F is 0.}\underbrace{\texttt{3333333}}_{\text{7 digits}}\texttt{4}$

AUGMENTED ASSIGNMENT OPERATORS

Operator	*Name*	*Example*	*Equivalent*
`+=`	Addition assignment	`i += 8`	`i = i + 8`
`-=`	Subtraction assignment	`i -= 8`	`i = i - 8`
`*=`	Multiplication assignment	`i *= 8`	`i = i * 8`
`/=`	Division assignment	`i /=  8`	`i = i / 8`
`%=`	Remainder assignment	`i %= 8`	`i = i % 8`

INCREMENT AND DECREMENT OPERATORS

Operator	*Name*	*Description*	*Example (assume i = 1)*

`++var`	preincrement	Increment `var` by `1`, and use the new `var` value in the statement	`int j = ++i;`
			`// j is 2, i is 2`

`var++`	postincrement	Increment `var` by `1`, but use the original `var` value in the statement	`int j = i++;`
			`// j is 1, i is 2`

`--var`	predecrement	Decrement `var` by `1`, and use the new `var` value in the statement	`int j = --i;`
			`// j is 0, i is 0`

`var--`	postdecrement	Decrement `var` by `1`, and use the original `var` value in the statement	`int j = i--;`
			`// j is 1, i is 0`

INCREMENT AND DECREMENT OPERATORS

```
int i = 10;
int newNum = 10 * i++;
```

Same effect as

```
int newNum = 10 * i;
i = i + 1;
```

```
int i = 10;
int newNum = 10 * (++i);
```

Same effect as

```
i = i + 1;
int newNum = 10 * i;
```

INCREMENT AND DECREMENT OPERATORS

- Using increment and decrement operators makes expressions short, but it also makes them complex and difficult to read. Avoid using these operators in expressions that modify multiple variables, or the same variable for multiple times such as this:
- `int k = ++i + i++ - --i;`

ASSIGNMENT EXPRESSIONS AND ASSIGNMENT STATEMENTS

- `variable = expression;`
- `variable op= expression; // Where op is +, -, *, /, or %`
- `++variable; //preincrement`
- `variable++; //postincrement`
- `--variable; //predecrement`
- `variable--; //postdecrement`

TYPE CONVERSION

NUMERIC TYPE CONVERSION

- Consider the following statements:
- `int i = 5;`
- `double d = i;`
- `double f = i*d;`

CONVERSION RULES

- When performing an operation involving two operands of different types, Java can automatically converts the operand based on the following rules:
- 1. If one of the operands is double, the other is converted into double.
- 2. Otherwise, if one of the operands is float, the other is converted into float.
- 3. Otherwise, if one of the operands is long, the other is converted into long.
- 4. Otherwise, both operands are converted into int.

TYPE CASTING

- **Implicit casting** is done when the range of the data increases, e.g., int to double. Implicit means automatic by Java
  - `double d = 3; //type widening`
- **Explicit casting** is done when the range of the data decreases, e.g., double to int. Explicit must be stated by the programmer
  - `int i = (int)3.9;` (type narrowing, fraction will be truncated not rounded!)
- What is wrong?
  - `int x = 5 / 2.0;`

$$\xrightarrow{\texttt{range increases}}$$

`byte, short, int, long, float, double`

TEMPORARIES – UNSEEN “VARIABLES”

- Temporarily computed values are also stored in memory, but do not have a name
- Important to note – they have a type. It is the type of the final result of the (sub)expression
- Example

```
1. double x = 1.5;
2. int y = 2;
3. double z = y*y/3 + 1.5;
```

- Determine the type of each subcomponent of the exression:
- `y*y?`
- `y*y/3?`
- `y*y/3 + 1.5?`

- Practice labeling the following for any line of code: kind of (sub)statement, type of (sub)expressions

EXERCISE

- Write a program to compute sales tax for a purchase.
- How could you alter your program to only store 2 decimal places? Try it!

COMMON ERRORS AND PITFALLS

- Common Error 1: Undeclared/Uninitialized/Unused Variables
- Common Error 2: Integer Overflow
- Common Error 3: Round-off Errors
- Common Error 4: Unintended Integer Division

- Common Pitfall 1: Redundant Input Objects

COMMON ERROR 1
UNDECLARED/UNINITIALIZED/UNUSED VARIABLES

```
double interestRate = 0.05;

double interest = interestrate * 45;
```

- Here, `interestrate` is undeclared to the compiler and `interestRate` is unused. The programmer just made a “spelling” mistake!
- Following naming conventions help

COMMON ERROR 2
INTEGER OVERFLOW

```
int value = 2147483647 + 1;

// value will actually be -2147483648
```

COMMON ERROR 3
ROUND-OFF ERRORS

```
System.out.println(1.0 - 0.1 - 0.1 - 0.1 - 0.1 - 0.1);

System.out.println(1.0 - 0.9);
```

COMMON ERROR 4
UNINTENDED INTEGER DIVISION

```
int number1 = 1;
int number2 = 2;
double average = (number1 + number2) / 2;
System.out.println(average);
```

(a)

```
int number1 = 1;
int number2 = 2;
double average = (number1 + number2) / 2.0;
System.out.println(average);
```

(b)

COMMON PITFALL 1
REDUNDANT INPUT OBJECTS

```java
Scanner input = new Scanner(System.in);
System.out.print("Enter an integer: ");
int v1 = input.nextInt();

Scanner input1 = new Scanner(System.in);
System.out.print("Enter a double value: ");
double v2 = input1.nextDouble();
```