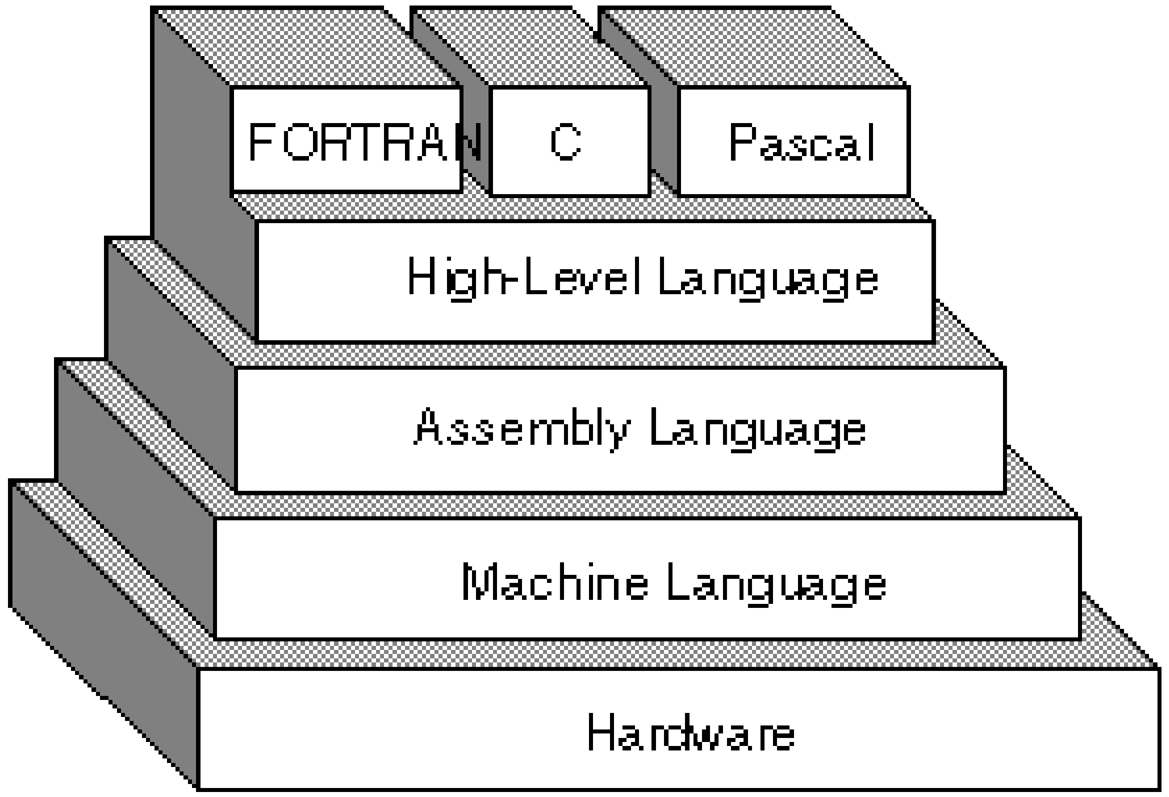
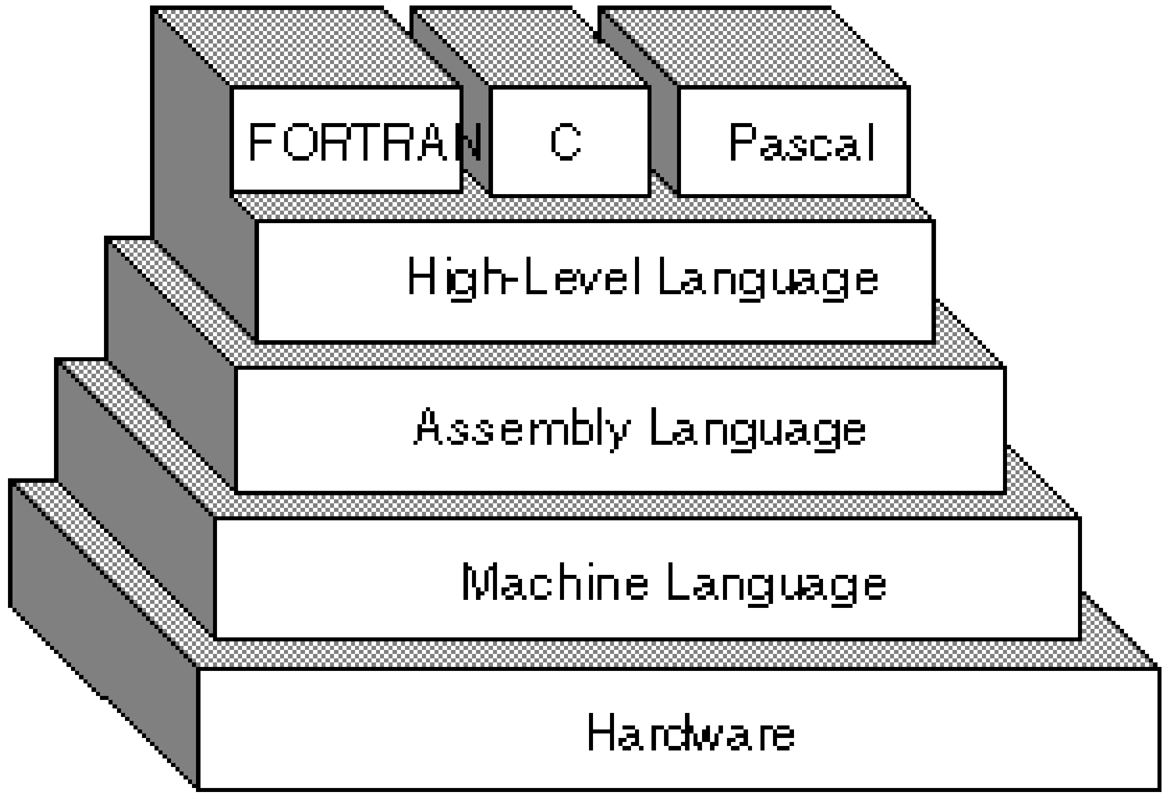
Java Basics

What's going on underneath

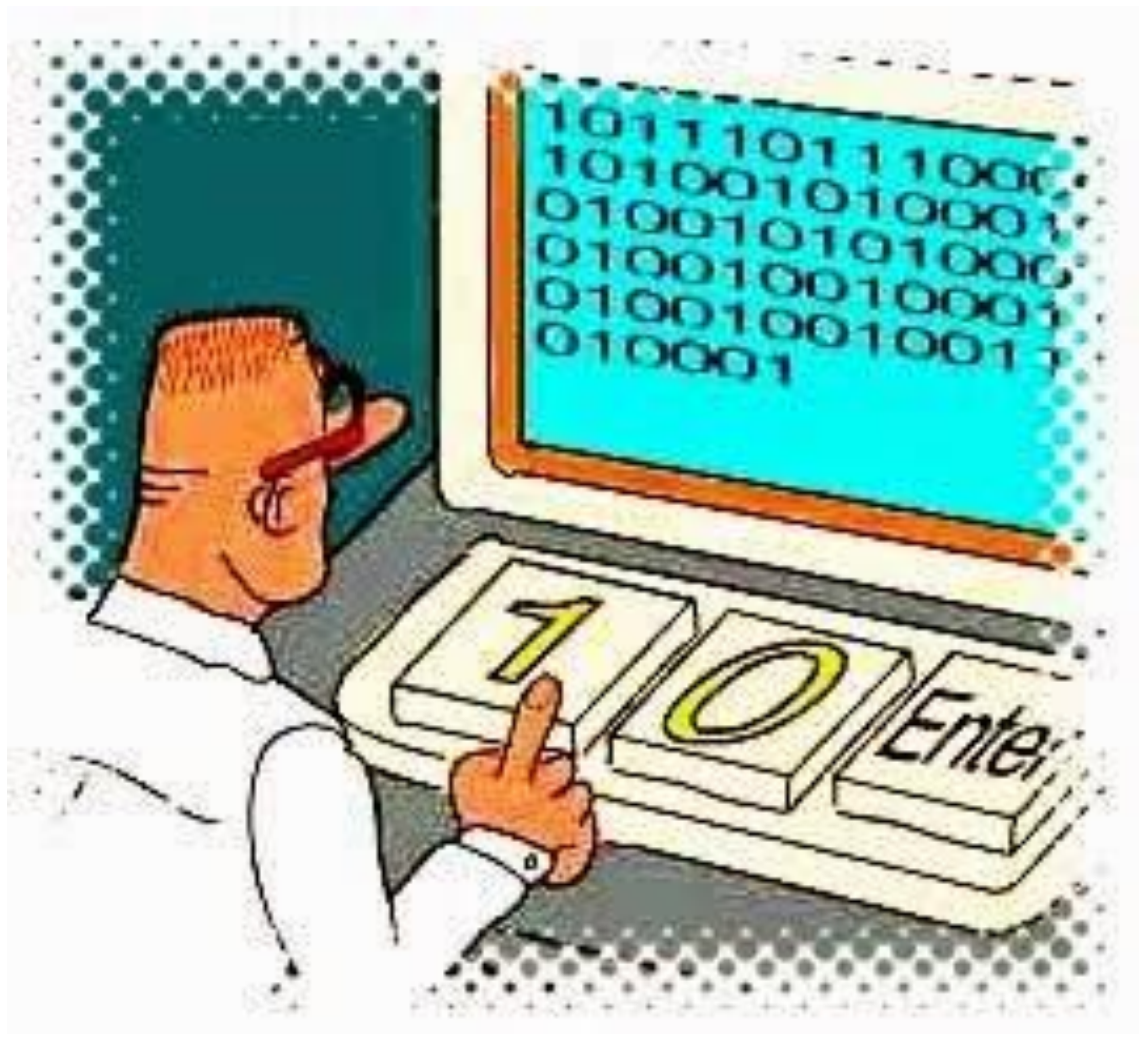


Computers are *digital*, which means everything inside them is stored as 1's or 0's. These *bits* are organized in groups of 8 which are called *bytes*. These bytes are further grouped into *words* (the number of bytes per word depends on the hardware).

What's going on underneath



At the "heart" of the computer is the CPU (Central Processing Unit). Each CPU has a language it understands. This language is all in 1's and 0's and is not at all like Python or Java or C, but, if we want to write a program that runs, we have to have a way to create these "machine programs" or executables.



What our keyboards would look like if we only programmed in machine code.

- Some programs are written in what is called "Assembly" language, which is ALMOST machine code.
- For instance there is a command in 6502 called LDA (Load Accumulator). The machine sees this as 1010 0101.
- An 6502 assembly language programmer would type in LDA for this command into a text file. In that file, the bit pattern is 0100 1100 0100 0100 0100 0001 (The ASCII codes for L, D, and A)
- The computer would not know how to run this! (or at least it would not run as LDA)

- An *Assembler* (a machine language program) reads the programmers file and writes a new file in which it changes the
  0100 1100 0100 0100 0100 0001 (LDA as a human readable ASCII)
  to
  1010 0101   (the machine code for the LDA instruction)

- You will learn much more and write lots of assembly language code in CSc 252!

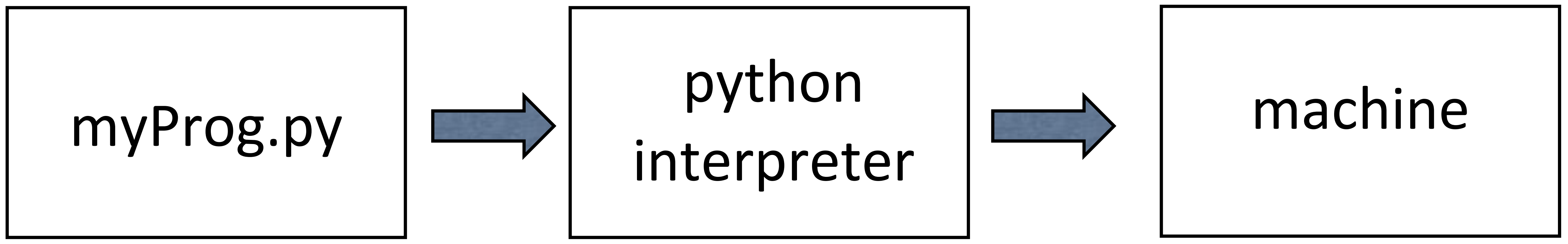
- So how does python work?

Interpreted Languages

- When you write a python program like myCat.py, you are creating a text file of human readable python commands.
  - This is not an executable and can not be run by your computer directly

Interpreted Languages

- When you write a python program like myCat.py, you are creating a text file of human readable python commands.
  - This is not an executable and can not be run by your computer directly
- Instead you run the python *interpreter* (executable program) which reads your human readable text file, interprets the instructions, and carries them out

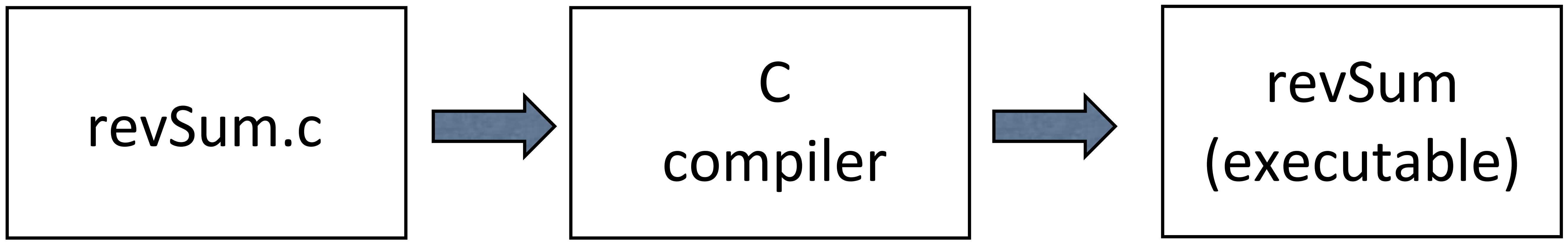


Compiled Languages

- When you write a C program like revSum.c, you are also creating a text file of human readable C statements.
  - Just like the python example, the computer cannot run your C program directly.

Compiled Languages

- When you write a C program like revSum.c, you are also creating a text file of human readable C statements.
  - Just like the python example, the computer cannot run your C program directly.
- In this case we run a C *compiler* (executable program) which reads the program file (source code) and creates a machine executable file.



This executable can be run directly by the computer.

Advantages of Interpreted Languages:

1. Often easier to write
    a. Can test on the fly
    b. Don't need to wait for compiling.
2. Easy to incorporate other peoples code
3. Portable
    a. The python interpreter needs to be ported, but the python program needs no changes.

Advantages of Compiled Languages

1. They are almost always faster when running.

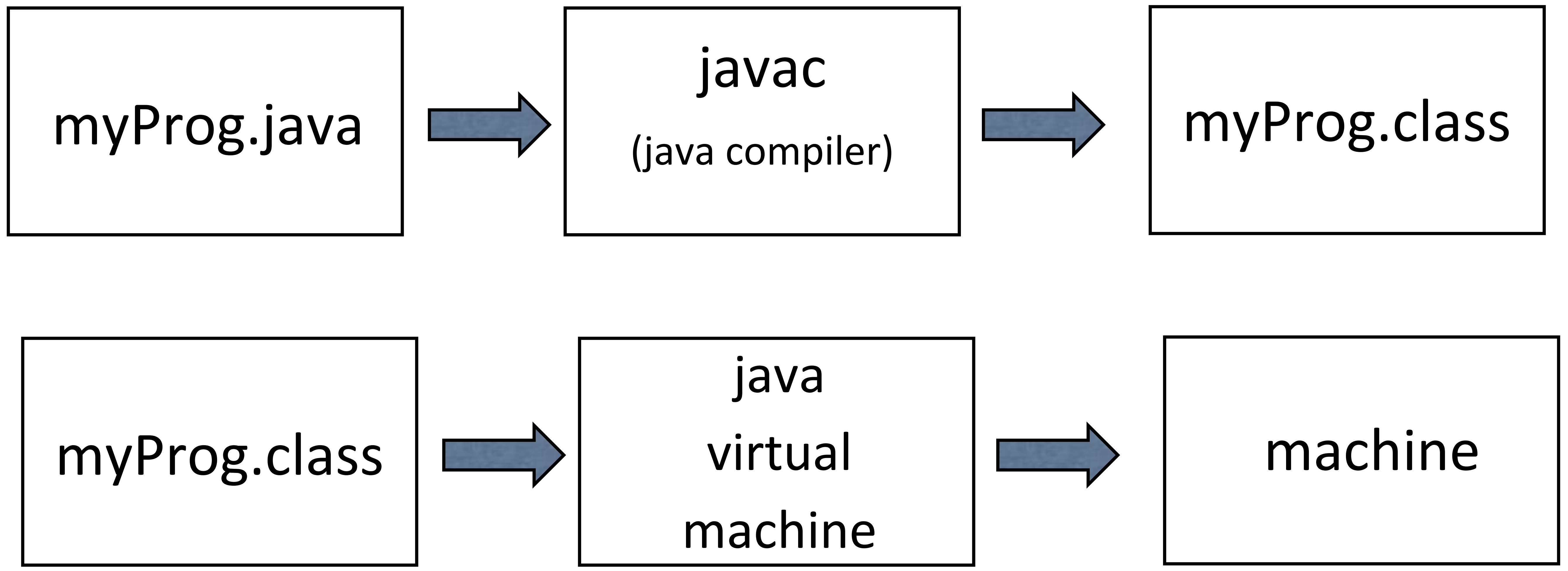
Q. So is Java Interpreted or Compiled?

A. Kind of both.

- A java program is written in human readable form in a java file (myProg.java).
- The java compiler (executable) reads that file and creates another file in java byte code (myProg.class).
- This program is not executable directly by your computer, instead it is run by the Java Virtual Machine (and executable program)

Java

Here's what running a java program looks like:



- **Caveat**: All this said, compiled vs interpreted is really a function of language implementation and not the language itself.
- It is possible to write a compiler for an "interpreted language" or a interpreter for a "compiled language".
- Still, the language is usually implemented and I think (arguably) designed to work one way of the other.

Why Learn Java?

For a one it is widely used.

What languages are most popular today?

The TIOBE index currently has this top eight: [Aug 2017]

1. Java
2. C
3. C++
4. C#
5. Python
6. Visual Basic .NET
7. PHP
8. JavaScript

There are many such lists and they all use different criteria, but everyone I looked at showed Java in the top 3.

Why Learn Java?

Another reason to learn Java is that it is very different than Python.

It is strictly typed (we'll talk about what this means later)

It is much more commonly used (some would say suited) for large projects.

It is a good representative language. Fairly close to C#.

There are 100s (maybe 1000s) of programming languages. Many more will be written before you stop working with computers. A computer scientist must be able to learn to use new languages.

Java was released in 1996 and it's designers said it was designed to be:

1. Simple
2. Object-Oriented
3. Distributed
4. Robust
5. Secure
6. Architecture-Neutral
7. Portable
8. Interpreted
9. High-Performance
10. Multi-threaded
11. Dynamic

Hello World

The simplest possible "Hello, World!" in Python:

```
print("Hello, World!")
```

The simplest possible "Hello, World!" in Java:

```
public class Hello {
    public static void main(String args[]) {
        System.out.println("Hello, World!");
    }
}
```

Note the java program must be saved in a file named Hello.java.

We can run the Python version like this:

```
  % python3 hello.py
  Hello, World!
```

The Java version requires two steps:

```
  % javac Hello.java  ("compiles" Hello.java,
  creating Hello.class)
  % java Hello     (runs Hello.class)
  Hello, World!
```

If **Hello.java** is changed, the two steps must be repeated.

If we just want to run the Java version again, we only need to do "**java Hello**".

Hello World

The Python program is much shorter because the python interpreter starts interpreting the commands at the top of the file.

What would happen if defined a main function in the python program instead

```
def main():
    print("Hello, World!")
```

What would this program output if run?

Hello World

The Python program is much shorter because the python interpreter starts interpreting the commands at the top of the file.

What would happen if defined a main function in the python program instead

```
def main():
    print("Hello, World!")
```

What would this program output if run?

Nothing since main is defined, but not called. Need to add:

```
def main():
    print("Hello, World!")
main()
```

- In Java all code (aside from “imports”) must appear in a class definition
- The file name for a java application must be the name of the class with the extension .java.

Thus we start with the line:

```
public class Hello {
```

- In python execution begins at the top of the file
- In Java, execution begins in the method named *main* defined for the indicated class
  Therefore we have to have:

```
public static void main(String args) {
```

- The main method must always be public static void
- We’ll talk about the **public** and **static** labels later, the **void** label means **main** does not return a value.

Indentation doesn't matter in Java

Python requires that source code be indented to reflect various structural relationships.

- Java has no indentation rules whatsoever. It uses curly brackets to define structure.
- The full contents of **`Hello.java`** could be squashed onto one line.
- This is valid, too:

```
          public
        class Hello {
      public static void main(
    String args[]
      ) { System.out.
        println("Hello, World!"
         );}
          }
```

But HORRIBLE style, don’t ever do this.

Hello World (recap)

```
public class Hello {
    public static void main(String args[]) {
        System.out.println("Hello, World!");
    }
}
```

1. All code wrapped in a class
2. Execution starts in the method named main
3. Boundaries of the class and method are determined by { }
4. The file this lives in should be named Hello.java

Identifiers:

- Symbolic names.
  - Used to name classes, variables, methods.
  - From the example on the previous slide: **`Hello`** and **`main`** are identifiers.
- Identifier Rules:
  - Must start with: a letter, dollar sign, or underscore
  - Followed by more letters, _, $, and/or numbers.
    `baseball   Football   Stalag17`
    - No spaces, no hyphens, no other punctuation.
  - Can be arbitrarily long.
    `sillyLongTakesForeverToTypeIdentifierForThisExample`
  - Case sensitive!
    - The identifiers **`theRailRoad`** and **`therailroad`** are <u>different</u>.
  - Can not be keywords or reserved words.

Statement:

- Performs one action. Examples:
  - Prints a message:
- **`System.out.println("Hello out there!");`**
  - Computes a sum:
- **`result = x + y;`**
- Terminates with a semicolon.
- Can span multiple lines.

```
• result = xray + yoke + 17 +
•                zebra - 42 +
•                63 + alpha;
```

  - *Style point*: Use indentation to indicate the lines go together.

Block:

- Contains 0, 1, 2, or more statements.
- Begins and ends with curly braces: **`{      }`**
- Can be used anywhere a statement is allowed.
- Example:

```
System.out.println("What's up Doc?");
{
   System.out.println("This print statement");
   System.out.println("and this one");
   System.out.println("plus this one are in a Block.");
}
```

Block:

- The **main** method contains a block of code between its **{** and **}**:

```
public static void main( String args[] )
{
   System.out.println("This is inside main's block of code");
} // main method ends
```

White Space:

- Space, tab, newline are *white space* characters.
- At least one white space character is required between a keyword and an identifier.
  - Example:

```
int myNumber;   // correct, white space between int and myNumber

intmyNumber;    // error.

int    myNumber,     myOtherNumber,     myLastNumber;  // correct
```

- Any amount of white space characters are permitted between identifiers, keywords, operators, and literals.

White Space:

Style point: Readability:

- White space makes code easier to read.
- Put white space around operators and operands and identifiers.

```
result = lima + juliet + oscar;  // this one is easier to read!

result=lima+juliet+oscar;        // harder to read
```

- Put blank lines between logical sections of the program.

Comments:

Two types of comments.

- Block comment: The comment can span multiple lines.
    - Begins with **/***
    - Ends with ***/**

```
System.out.println("message here");   /* This is a comment
   that continues here
   and finally ends here */
```

- Line comment: The comment is no longer than one line.
    - Begins with //
    - Ends at the end of the line.

```
System.out.println("message here");   // A line comment
```

Comments (continued):

- *Style point*: Include a block comment at the beginning of <u>each</u> Java source file.
  - Identify yourself as the author of the program and your section letter and Section Leader’s name.
  - Briefly describe what the program does.

```
/* Your first and last name here
 * Your section leader's name here
 * Section: Your section letter here
 *
 * Short description of what your program does.
 * Description can occupy more than one line, as needed.
 */
```

Python is dynamically typed

Python is a *dynamically typed* language.

A fundamental characteristic of a dynamically typed language is that the type of the value held by a variable can differ over time.

A simple example:

```
x = 7
x = "seven"
x = 7.0
```

In the code below, what types of values does **elem** hold over time?

```
for elem in ["one", 2, [3.0, 4]]:
    print(type(elem))
```

Java is statically typed

Java is a *statically typed* language. Two rules:

- Every variable must have a *declaration* that specifies its type.
- A variable can only hold values of its declared type.

Imagine the following Java code inside a **main** method:

```
int i;      // Declares a variable i that can hold only ints.
            // We can also view it as a request for help:
            //   "Stop me if I try to put something else in i!"

i = 5;      // ok
i = "x";    // error

String s; // s is to hold only Strings.  If I foul up, stop me!

s = "x";    // ok
s = i;      // error
```

Costs and benefits

Two costs of static typing:

- Can make functions less flexible
- Often involves complex rules

Code written in a statically typed language has an important characteristic:

- <u>The type of the value produced by every expression can be determined without executing any code.</u>

That characteristic provides some valuable benefits:

- Potential for far higher performance
- Detection of certain types of errors before code is executed
- Can make code easier to read and modify
- Increases ability of IDEs to help when editing code

One performance cost with a dynamically typed language is execution-time type checking.

Evaluating the expression **`x + y`** can involve decision-making like this:

```
if both x and y are integers
    add them
else if both x and y are floats
    add them
else if one is a float and the other an integer
    convert the integer to a float and add them
else if both x and y are strings
    concatenate them
...
```

If that **`x + y`** is in a loop, that decision making is done every time around.

Note: The above "implementation" can be improved upon in many ways.

Primitive types

Java has eight *primitive types*:

- Five are *integral types*: `byte`, `short`, `int`, `long`, and `char`
- Two are *floating-point types*: `float` and `double`
- There is a type for boolean values: `boolean`

Data Types

Integer Types:

Type	Size in Bytes	Minimum Value	Maximum Value
`byte`	1	-128	127
`short`	2	-32,768	32,767
`int`	4	-2,147,483,648	2,147,483,647
`long`	8	-9,223,372,036,854,775,808	9,223,372,036,854,775,807

- Examples:

```
int   testGrade;
int   numPlayers, highScore, diceRoll;
short xCoordinate, yCoordinate;
byte  ageInYears;
long  worldPopulation;
```

Data Types (continued):

Floating-Point Data Types:

3.40 x 10^{+38}

Type	Size in Bytes	Minimum Positive Nonzero Value	Maximum Value
`float`	4	1.4E-45	3.4028235E+38
`double`	8	4.9E-324	1.7976931348623157E+308

- Use **double** when you need:
  - Larger range of values than provided by **float**.
  - Greater precision than provided by **float**.
- Examples:

```
float  courseAverage;
float  battingAverage, sluggingPercentage;
double distanceToAndromeda;
```

"Minimum Positive Nonzero Value"
How close can the **float** or **double** come to zero without changing to zero.

Data Types (continued):

Character Data Type:

Type	Size in Bytes	Minimum Value	Maximum Value
`char`	2	character encoded as 0	character encoded as 65535

- Contains standard Latin alphabet, digits, punctuation.
  - Supports many character sets (i.e., Greek, Cyrillic, Hebrew).
  - Known as Unicode.
- Examples:

```
char myInitial;

char firstLetter, nextLetter, lastLetter;

char newline, tab, quoteMark;
```

Data Types (continued):

Boolean Data Type:

- Only two values possible:
  - **`true`**
  - **`false`**
- Used for decision making or as “flag” variables.
- Examples:

```
boolean done;
boolean programStarted, programFinished;
boolean reachedLastLevel;
boolean hasWizardPowers;
```

Variables

Declaring Variables:

- Variables hold one value at a time.
- That one value can change during execution.

- Syntax:

```
dataType identifier;
```

- Or

```
dataType identifier, anotherIdentifier, andAnother, …;
```

Variables

- *Style point*: Naming convention for variable names:
  - First letter is lowercase.
  - Embedded words begin with uppercase letter; called “camel casing”.
  - Examples:

```
int xray, limeJuice, oneMonth;
float myBankBalance, profitAndLoss;
```

- A *convention* is not required by the language (the Java compiler will not complain).
- *Conventions* generally exist to make code more readable and understandable.
- You are expected to follow these *conventions* in the code you develop in this (and other) courses.

Variables (continued):

Declaring Variables:

- Variable names should be meaningful:
  - The name should tell us how the program will use the value.
  - Makes the logic of the program clearer.
- How long should names be?
  - Abbreviations can be confusing.
  - Long names can also make the code hard to read.
- Avoid names similar to Java keywords.
- Even though you CAN use '$' in variable names, you should not use it as it is intended for names that are generated by the compiler or tools

Initializing Variables

- A declaration can include initialization:

```
int a = 10, b, c = 20;
```

- An initializing expression can use previously initialized variables:

```
int d = a + c * 5;
```

- Note: Initializations are not considered to be assignments.

- It is an error to use an uninitialized variable.

Variables

Declare a Variable exactly Once per Block:

```
double myTwoCents;
double myTwoCents = 0.02;
```

- Generates this compile error:

```
myTwoCents is already defined in main(java.lang.String[])
```

Once a variable is declared, its data type cannot be changed:

```
double cashInMyPocket = 5.0;
int cashInMyPocket;
```

- Generates this compile error:

```
cashInMyPocket is already defined in main(java.lang.String[])
```

Literals

- Literals for integer types.
  - **int**, **short**, **byte**.
    - Optional initial sign (**+** or **-**) followed by digits **0**-**9** in any combination.
  - **long**.
    - Optional initial sign (**+** or **-**) followed by digits **0**-**9** in any combination, terminated with **L** or **l**.
      - *Style point*: Use the capital **L**, not the lowercase **l**. The lowercase **l** (ell) can be confused with **1** (one).
  - Note: there are no commas or periods in an integer literal.
    - However, you can use underscores.
      1_000_000 is legal

Literals

- Examples:

```
short myAge = 62;

int   numSongs = +4406, nitrogenLiquid = -210;

//The next one will give a compile time error

long  ageOfUniverse = 15000000000;  //literal is treated as an int

// The following is the fix

long  ageOfUniverse = 15000000000L;

// This next one will generate a compile error:

short smallValue = 103467;    // value too big to go into a short
```

Literals

- Literals for floating-point types.
  - **float**.
    - Optional initial sign (**+** or **-**) followed by a floating-point number in fixed or scientific format, terminated by an **F** or **f**.
  - **double**.
    - Optional initial sign (**+** or **-**) followed by a floating-point number in fixed or scientific format.

Literals

- Examples:

```
float pi = 3.1415926536F;           // fixed format

float small = 0.00000000456f;       // fixed format

double lightSpeed = 2.99792E+8;     // scientific format

float tinyValue = 1.837E-12F;       // value close to zero

double negValue = -1.837E-12; // negative value close to zero
```

Literals

- Literals for **char** type.
  - **char**.
    - Any printable character enclosed in single quotes: **'Q'**
    - A decimal value in the range **0 - 65535**.
    - Certain escape sequences:
      - **'\t'** represents a tab, **'\n'** represents a newline.
    - Examples:

```
char myInitial = 'P';

char aTab = '\t', percent = '%';

char alsoPercent = 37;
```

Literals

- Literals for **boolean** type.
  - **boolean**.
    - Use the value **true** or **false**.
    - Examples:

```
boolean afternoon = true;

boolean doneYet = false
```

Assigning Values of Other Variables:

- Syntax:

```
variable2 = variable1;
```

- Rules:
  - **variable1** needs to be defined earlier in the code.
  - **variable1** and **variable2** need to be compatible data types.
    - Compatible data types: the precision of **variable1** must be lower than or equal to that of **variable2**.

Assigning Values of Other Variables

- Any type in the right column can be assigned to the type in the left column:

Data Type	Compatible Data Types
byte	**byte**
short	**byte, short**
int	**byte, short, int, char**
long	**byte, short, int, long, char**
float	**float, byte, short, int, long, char**
double	**float, double, byte, short, int, long, char**
boolean	**boolean**
char	**char**

- Hints:
  - Small items fit into larger items.
  - Integers fit into floats or doubles.

Assigning Values of Other Variables

- Example assignments:

```
short smallValue = 42;
int largerItem = smallValue;
long biggerStill = largerItem;
biggerStill = 2787L;
biggerStill = smallValue;
float myBankBalance = 273.42f;
double largeBalance = myBankBalance;
myBankBalance = 2738.2F;
largBalance = biggerStill;
```

Assigning Values of Other Variables

- Beware! What is wrong with the following?

```
int numBooks = 173;
short howMany = numBooks;
short againBooks = 173;

double milesToGo = 435.7F;
float howFar = milesToGo;
```

Constants:

- A constant is a "variable" whose value cannot change during program execution.
- In Java, we use the keyword final to denote a constant
- Example:

```
final double CM_PER_INCH = 2.54;
```

- The keyword final means you can only assign a value to a variable once.
- *Style Point*: The convention is to use CAPITAL LETTERS for constants and separate words with an underscore.
  - Declare constants at the beginning of the method so their values can easily be seen.
  - Declare as a constant any data that should not change during execution.

Constants (example):

```
/* Constants Class */
public class Constants {

  public static void main( String[] args ) {
    final double KM_TO_MILE = 1.609;
    double speedLimit = 65;

    System.out.println(speedLimit + " in mph is " +
                       speedLimit * KM_TO_MILE +
                       " in km/hr");

  } // main

} // class Constants
```

String Literals:

- *String* is actually a class, not a basic data type (and, not a reserved word).
  - *String* variables are objects.
  - More on classes and objects a bit later.
- *String literal*: text contained within double quotes.
  - Note you may use single quotes for strings in Python, NOT in Java
- Examples of String literals:

```
"The answer is "
"Four-score and seven years ago"
"Gallia est omnis divisa in partes tres"
"To boldly go where no man has gone before."
"Please Sir, can I have some more?"
```

String Concatenation:

- Can combine two String literals together.
- Can combine one String literal with another data type for printing.

```
String howdy = "Howdy";
String buddy = "Pardner!";
String greeting = howdy + ' ' + buddy;
System.out.println( greeting );
String greeting2 = howdy + buddy;
System.out.println( greeting2 );
String greeting3 = howdy + "         " + buddy;
System.out.println( greeting3 );
```

```
  Prints:
          Howdy Pardner!
          HowdyPardner!
          Howdy         Pardner!
```

String (continued):

- Common error (!!!):

```
System.out.println("Four-score and seven years ago,
    Our Fathers brought forth on this continent");
```

  - Generates these errors:

```
unclosed string literal
')' expected
```

  - String <u>literals</u> must start and end on the <u>same</u> line.

The Fix

- Break long strings into shorter strings and use the concatenation operator:

```
System.out.println("Four-score and seven years ago,\n" +
    "Our Fathers brought forth on this continent");
```

- Or, use two separate print statements:

```
System.out.println("Four-score and seven years ago,");
System.out.println("Our Fathers brought forth on this continent");
```

Escape Sequences in String literals:

Character	Escape Sequence
Newline	**\n**
Tab	**\t**
Double quotes	**\"**
Single quote	**\'**
Backslash	\
Backspace	**\b**
Carriage return	**\r**
Form feed	**\f**

Use **`\n`** when you need a newline or a blank line.

We will ***not*** use **`\r`** or **`\f`**.

Escape Sequences in String literals (continued):

```
  /* Literals Class */
public class Literals {
  public static void main( String[] args ) {
    System.out.println( "How to get multiple\nlines from one statement\n" );

    System.out.println( "When you want to indent a line" );
    System.out.println( "\tUse a tab" );
    System.out.println( "\tFor each indented line" );

    System.out.println( "It started: \"Four-score and seven years ago\"");
    System.out.println( "He said \"NO!\" and meant it!" );
  } // end of method main
}  // end of class Literals
```

Expressions and Arithmetic Operators:

- The Assignment Operator.
- Arithmetic Operators.
- Operator Precedence.
- Integer Division and Modulus.
- Division by Zero.
- Mixed-Type Arithmetic and Type Casting.
- Shortcut Operators.

Assignment Operator:

- In Java, = is an assignment operator that has a side effect of assigning a value.
- Syntax:

```
target = expression;
```

- The operator returns the value of the expression BUT
- It also assigns the value of the expression to the target (called a side effect).
- Note: The value of an expression must be compatible with the target's data type.

Assignment Operator:

- Because assignment is an operator, the following is an expression that returns the value of 10

  ```
  i = 10
  ```

- This is what allows multiple assignments in a statement.

  ```
  i = j = 0;  // j = 0 returns 0 which is then assigned to i
  ```

- It is also legal to write something like:

  ```
  if ((i = j) == 10) {
  ```

  Although it can make the code cofusing to read

Arithmetic Operators:

Operator	Operation
*	multiplication
/	division
%	modulus (remainder after division)
+	addition
-	subtraction

Arithmetic operators

Java has five binary (two-operand) arithmetic operators:

```
    +   -   *  /   %
```

and two unary (one-operand) arithmetic operators:

```
    -  +
```

The operators behave as they do in Python 3, except for **`/`** and **`%`**.

Like Python, mixing integers and floating-point values produces a floating-point result:

```
    double ans = 2 + 3.4;  //produces a double

    int badExp = 2 + 3.4;  //gives an error
```

Operator Precedence:

Operator	Order of evaluation	Operation
`()`	left to right	parenthesis for explicit grouping
`* / %`	left to right	multiplication, division, modulus
`+ -`	left to right	addition, subtraction
`=`	right to left	assignment

Operator Precedence (continued):

- Operator precedence follows normal arithmetic rules. Makes the following work correctly:

```
int singles = 43, doubles = 17, triples = 2, homeRuns = 28;
int hits, totalBases;
hits = singles + doubles + triples + homeRuns;

totalBases = singles + 2 * doubles + 3 * triples + 4 * homeRuns;
```

Operator Precedence (continued):

- Batting average in baseball is computed as

$$\frac{\text{number of hits}}{\text{plate appearances - walks}}$$

  - The following does not work (why?):

```
float hits = 153f, appearances = 482f, walks = 27f;

float average = hits / appearances - walks;
```

    - Where are parentheses needed?

Operator Precedence (continued):

- Batting average in baseball is computed as $\dfrac{\text{number of hits}}{\text{plate appearances - walks}}$
  - The following does not work (why?):

```
float hits = 153f, appearances = 482f, walks = 27f;
float average = hits / appearances - walks;
```

    - Where are parentheses needed?

```
float average = hits / (appearances – walks);
```

Division

Python 3 has two division operators: / and // (floor division)

Java has one division operator (/).

IMPORTANT: If both operands of / are integers, the quotient is an integer, rounded towards 0.

```
    int i = 3 / 2;  //i has value 1

    i = -10 / 3;   //i has value -3
```

Inadvertent integer division is a common source of bugs, especially for Python programmers!

Integer division by zero is an error:

```
    i = 1 / 0;     //throws an error
```

Integer Division

- What value does the following code store in percentAttended?

```
int classSize = 175;
int attendance = 137;
int percentAttended = attendance / classSize;
```

- The value stored in percentAttended is 0
- What happens if we change the type of percentAttended?

```
double percentAttended = attendance / classSize;
```

- The value stored is STILL 0, since
  - **`attendance / classSize`** produces the integer 0
  - The integer 0 is then converted to a double, also with value 0

- Modulus example: (Furlong.java)

```
public class Furlong {
  public static void main(String[] args) {
    int numFeet = 23487;
    System.out.println("length in feet = " + numFeet);

    // Calculate number of miles and feet left over
    final int FEET_PER_MILE = 5280;
    int miles = numFeet / FEET_PER_MILE;
    int leftOver = numFeet % FEET_PER_MILE;

    System.out.print("miles = " + miles + '\t' );
    System.out.println("leftOver = " + leftOver);

    // Calculate number of furlongs and feet left over
    final int FEET_PER_FURLONG = 660;
    int furlongs = numFeet / FEET_PER_FURLONG;
    leftOver = numFeet % FEET_PER_FURLONG;

    System.out.print("furlongs = " + furlongs + '\t');
    System.out.println("leftOver = " + leftOver);

  } // end of main method
} // end of class Furlong
```

Division by Zero:

- Integer division by **0**:
  - Example:

    ```
    int answer = 173 / 0;
    ```

    - No error from the compiler.
    - At runtime, generates an *ArithmeticException* and program stops executing.

Floating-point division by **0**:

- Two possibilities:
  - Dividend is not zero; Divisor is zero.

```
double resultAlpha = 173.628 / 0;
System.out.println("resultAlpha = " + resultAlpha);
```

  - Prints:

```
 resultAlpha = Infinity
```

  - Both dividend and divisor are zero.

```
double resultBeta = 0.0 / 0.0;
System.out.println("resultBeta = " + resultBeta);
```

  - Prints:

```
resultBeta = NaN
```

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

      double resultAlpha = 173.628 / 0;
      System.out.println("resultAlpha = " + resultAlpha);

      double resultBeta = 0.0 / 0.0;
      System.out.println("resultBeta = " + resultBeta);

      double tinyNumber = 1.0E-310;
       System.out.println("tinyNumber = " + tinyNumber);

       resultAlpha = 173.628 / tinyNumber;
      System.out.println("resultAlpha = " + resultAlpha);

   } // end of main method

} // end of class DivideZero
```

Mixed-Type Arithmetic:

- Calculations can be performed with operands of different data types.

```
int sample = 17;
float price = 37.95F;
double cost = sample * price;
```

- There are rules:
  - Lower-precision operands are *promoted* to higher-precision data types before the operation is performed.
  - *Promotion* is only done while evaluating the expression. It is not a permanent change.
  - Called “Implicit type casting”.

Mixed-Type Arithmetic (continued):

- Rules of Promotion, apply the first rule that fits:
    1. If either operand is a **double**, the other operand is converted to a **double**.
    2. If either operand is a **float**, the other operand is converted to a **float**.
    3. If either operand is a **long**, the other operand is converted to a **long**.
    4. If either operand is an **int**, the other operand is promoted to an **int**.
    5. If neither operand is a **double**, **float**, **long**, or an **int**, both operands are promoted to an **int**.

```
double cost = 730 * 4.95;    // 730 converted to double, rule 1
float height = 16.75F;
float stories = height * 7;  // 7 converted to float, rule 2
byte months = 7;
short days = 31;
long hours = days * 24L;     // days converted to long, rule 3
int seconds = days * 3600 * 24;  //days converted to int, rule 4
```

Explicit Type Casting:

- Want to change the type of an expression.
- Syntax:

```
(datatype) expression
```

- This can be used to force to get decimals or higher precision.

```
int classSize = 175;
int attendance = 137;
float percentAttended = (float) attendance / classSize;
```

- Would the following work?

```
float percentAttended = attendance / (float) classSize;
```

What about?

```
float percentAttended = (float) (attendance / classSize);
```

Run the example on the next page.

```
  public class Weeks {
     public static void main(String[] args) {
       final int DAYS_IN_WEEK = 7;
       int days = 2873;
       float weeks;

       weeks = days / DAYS_IN_WEEK;
       System.out.println("weeks = " + weeks);

       weeks = days / (float) DAYS_IN_WEEK;
       System.out.println("weeks = " + weeks);

       weeks = (float) days / (float) DAYS_IN_WEEK;
       System.out.println("weeks = " + weeks);

       weeks = (float) days / DAYS_IN_WEEK;
       System.out.println("weeks = " + weeks);

       weeks = (float) (days / DAYS_IN_WEEK);
       System.out.println("weeks = " + weeks);


   } // end of main method
} // end of class Weeks
```

- You can also use explicit casts to "downgrade" a value.
  - Decimal numbers cast as integral numbers truncate their value

```
double x = 6.87;
int i = x;    // will give you a compile time error

int j = (int) x; // is legal and will store 6 in j
```

  - You may also cast from integral to integral or decimal to decimal

```
int i = 5;
byte j = i;   // will give a compile time error

byte k = (byte) i; // is legal and will store 5 in k

double x = 456.98;
float g = x;    // will give compile time error

float f = (float) x; // is legal
```

- Warning:
- If the value is too large for the type being cast, the result will be a truncated number

```
int i = 300;

byte b = (byte) i; // legal but stores the value 44 in b
```

- Why?

```
300 = 00000000 00000000 00000001 00101100
```

Last byte is:

```
00101100
```

Which evaluates to 44

Shortcut Operators:

- Often want to add 1   or subtract 1.
- `++` lets us add 1   `--` lets us subtract 1.
- Examples:
  - Can be used in a postfix form, where the operator appears after the variable:

```
int xray = 17, sample = 24;

xray++;    // xray now contains 18, same as: xray = xray + 1;

sample--;  // sample now contains 23

sample--;  // sample now contains 22
```

  - Can be used in a prefix form, where the operator appears before the variable:

```
System.out.println("xray becomes " +  --xray);
 // xray now contains 17
```

Shortcut Operators:

- `++` and `--` are operators which return a value and have a side effect of incrementing or decrementing the value of the variable they're applied to.
- For both postfix and prefix forms they increment or decrement, but the value they return depends on which form is used.
  - In a postfix form, the value returned is the value before the increment or decrement:

```
int xray = 17;
System.out.println("xray becomes " + xray++);
```

  - Stores 18 in xray, but prints the following:

```
xray becomes 17
```

Shortcut Operators:

- In a prefix form, the value returned is the value after the increment or decrement:

```
int xray = 17;
System.out.println("xray becomes " + ++xray);
```

- Stores 18 in xray (just like in the prior slide), but prints the following:

```
xray becomes 18
```

- This can be confusing to programmers and many (especially Java programmers) choose to only use the increment or decrement as a statement by itself:

```
++xray;
System.out.println("xray becomes " + xray);
```

Shortcut Operators (continued):

- There are additional operators that combine assignment with one operation.

Operator	Example	Equivalent
`+=`	`alpha += 3;`	`alpha = alpha + 3;`
`-=`	`bravo -= 17;`	`bravo = bravo - 17;`
`*=`	`charlie *= 8;`	`charlie = charlie * 8;`
`/=`	`delta /= 9;`	`delta = delta / 9;`
`%=`	`echo %= 60;`	`echo = echo % 60;`

Shortcut Operators (continued):

- No spaces are allowed between the arithmetic operator and the equals sign.

```
xray +  =  12;
```

  - Causes a compiler error.

- The order of the arithmetic operator and the equals sign is important:

```
int foxtrot = 30;

foxtrot =+ 42;
```

  - The compiler does not find fault with this.
  - The statement will also execute correctly.
  - What will be the value of **`foxtrot`** after the statement executes?

Operator Precedence (updated):

Operator	Order of evaluation	Operation
`( )`	left to right	parenthesis for explicit grouping
`++ --`	right to left	preincrement, predecrment
`++ --`	right to left	postincrement, postdecrement
`* / %`	left to right	multiplication, division, modulus
`+ -`	left to right	addition or *String* concatenation, subtraction
`= += -= *= /= %=`	right to left	assignment

Formatted Printing:

- When you print numbers:

```
double foxtrot = 30.7 / 7.3;

System.out.println( "The answer is " + foxtrot );

The answer is 4.205479452054795
```

  - We don't usually need so many decimal places!!

Formatted Printing:

- Use the **printf** method instead:

```
System.out.printf( "The answer is %4.2f\n", foxtrot);
```

```
The answer is 4.21
```

  - **printf** uses C-style formatting. The **%** marks the start of a formatting instruction.
  - In the example above, **f** means “floating-point”, and is used with float or double types.
  - The **4.2** part means:
    - Use 2 decimal places; value will be rounded.
    - Use a total of 4 characters.
      - 2 for the decimal places
      - 1 for the decimal point
      - 1 for the value in front

Formatted Printing:

```
foxtrot = 1020.29876;
System.out.printf( "The answer is 6.3f\n", foxtrot);
The answer is 6.3f

String first = "Charlie";
String last = "Brown";
System.out.printf("The name is %10s", last);
System.out.println();
System.out.printf("%6s, %9s\n", last, first);
The name is      Brown
 Brown,   Charlie
```

Formatted Printing (continued):

- Most commonly used format codes:

Code	Formats	Example Use	Corresponding Output
`d`	`Integers (Base 10)`	`("%5d", 29)`	`29`
`x`	`Integers (Base 16)`	`("%x %x", 29, 32)`	`1D 20`
`o`	`Integers (Base 8)`	`("%o", 29)`	`35`
`f`	`Floating-point`	`("%8.2f", 874.9163)`	`874.92`
`e`	`Exponential Floating-point`	`("%.2e", 874.9163)`	`8.749163E+02`
`c`	`Character`	`("%c", 'Y')`	`Y`
`s`	`Strings`	`("%10s", "Hi")`	`Hi`

- There are more. See the Java API: http://docs.oracle.com/javase/8/docs/api/
  - **`System`** is part of java.lang
  - **`System.out`** returns a **`PrintStream`**.
    - **`PrintStream`** is part of java.io
      - look at the **`printf`** method.

- See also (on lectura and the lab machines):
  - **`man -s 3 printf`**
    - under the heading "Format of the format string"

Formatted Printing (continued):

- To help you in understanding how these work, try the following (and then explain why it turns out that way):

```
public class Printing {
  public static void main( String [] args ) {
    double foxtrot = 30.7 / 7.3;
    System.out.printf( "The answer is %4.2f\n", foxtrot);
    System.out.println( "The answer is " + foxtrot );

    int value = 1782;
    System.out.printf( "value =%2d\n", value);
    System.out.printf( "value =%4d\n", value);
    System.out.printf( "value =%7d\n", value);
    System.out.printf( "value =%d\n",  value);

    float small = 0.00000029872F;
    System.out.printf( "small =%f\n", small);
    System.out.printf( "small =%4.2f\n", small);
    System.out.printf( "small =%.5f\n", small);
    System.out.printf( "small =%.8f\n", small);
    System.out.printf( "small =%10.8f\n", small);
    System.out.printf( "small =%11.8f\n", small);
    System.out.printf( "small =%15.8f\n", small);
  } // main
} // Printing
```