ICMC Summer Coding

Table of contents on the next slide →

Table of Contents

Today is: **July 14** (at time of export)

July 28

Agenda

- Introductions
- Resources
- Schedule
- Coding Setup
- Hello World
- *Break*
- About Java and Python
- Variables and Basic I/O
- Common Errors and Debugging
- *Worksheet*

Introductions

Hello!

Message us on Discord or send an email to the following addresses:

- `iowacitymathcircle@gmail.com `

Resources

Bookmark this slide presentation to following along!

Helpful cheatsheets to reference:

- Java Cheatsheet

Schedule

Week 1 - Foundations - July 28 to August 1

July 28	July 29	July 30	July 31	August 1
Variables, Data Types, I/O	Using Data Types and Variables	Arrays, Strings, Functions, Intro to Classes	Control Flow, Data Structures	Imports / Packages, Exception Handling

Week 2 - Projects - August 4 to August 8

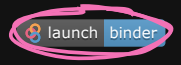
August 4	August 5	August 6	August 7	August 8
Time & Random, ANSI Escape Sequences	File I/O	Turtle, Lambdas	Multithreading	Project Presentations

Worksheets

- Worksheets will be frequently assigned, but are not graded.
- You may check your answers against the answer keys.
- Both will be posted on these slides.

Setup

- Click on the button below to open up a **Binder**.



Always use this button to open Binder.

- We will use Binders during camp. *The Binder service is generously provided for free, but we don't want to overuse it.*
- When programming outside of class, here are some other options:
  1. Install an editor + Java and/or Python on a non-Chromebook device.

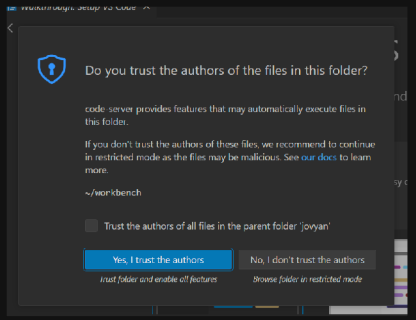
     You can ask us for guidance.
  2. Replit
  3. JavaFiddle (Java)
  4. JupyterLite (Python)

How to save your files from Binder

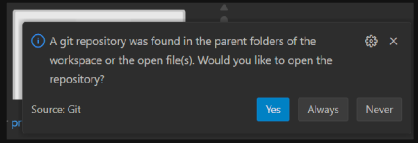
Using Binder

Instructor Guided

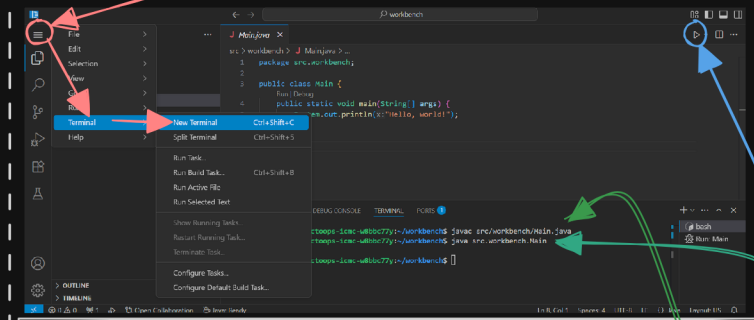
1. Click "Yes, I trust the authors"



2. Click "Never"



Create a new terminal to run commands.



Compile Command: javac src/workbench/Main.java

Run Command: java src.workbench.Main

You must always compile before running!

Use the run button to compile and run in one step!

Tip: Python Run Command: python main.py

Break

Have a break!

About Java and Python

Java

- Java 1.0 was released in 1996
  - Not that old, still very popular
- Originally designed for TV
- Versions: 1.0, 1.1, ..., 1.4, 5.0, 6, 7, ...
- Versions you should care about (LTS): 8, 11, 17, 21
- Java and JavaScript JS are different!

FIRST Robotics

There are many high school robotics teams around Iowa City that use **Java**!
Consider joining one!

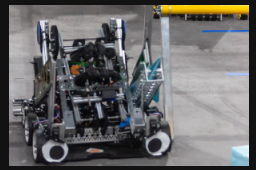
FTC Teams:

- West High: Trobotix 8696
- City High: Raw Bacon 8743
- Liberty High: ThunderBots 22064

FRC Team:

- Iowa City: Children of the Corn 167


8696

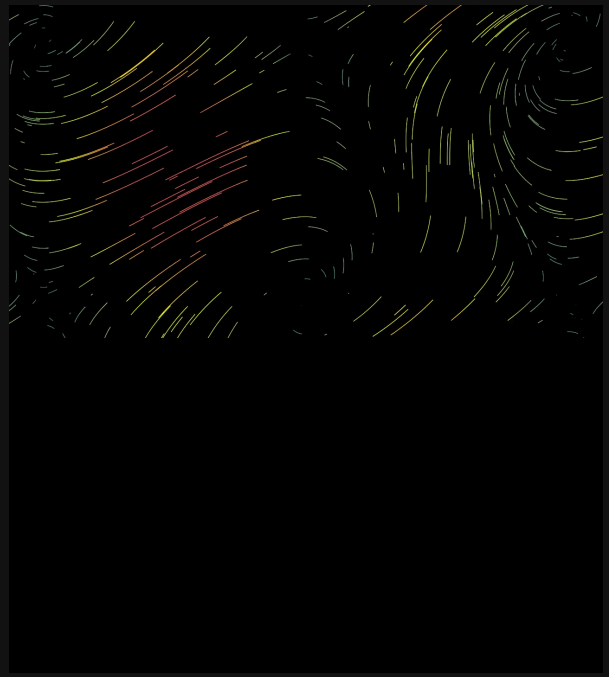



Children of the Corn

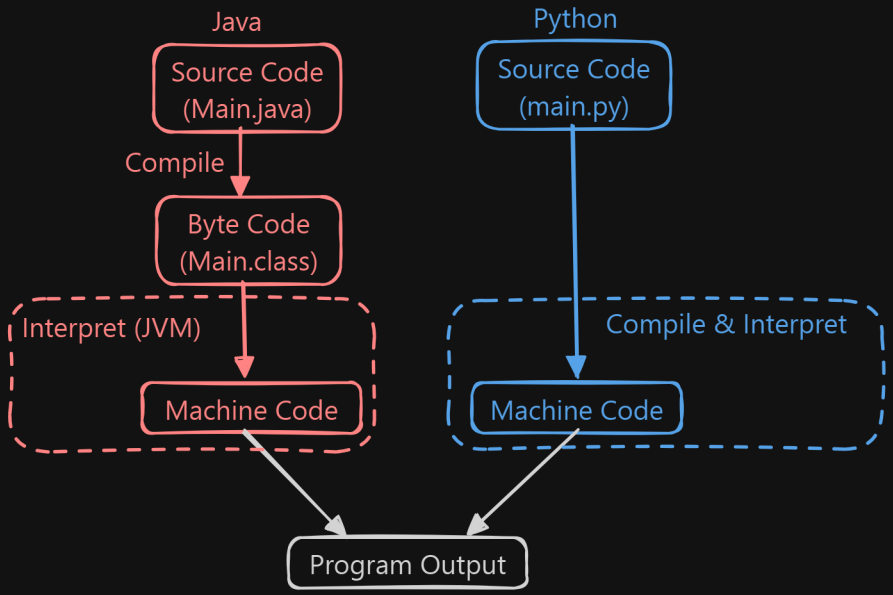
Python

- Python 0.9.0 was released in 1991
- Two variants: Python 2 & Python 3
  - Don’t use 2, use 3
- As of writing this slide, the latest version is 3.13.5
- We will mainly be using Java (it’s more straight forward)
  - Python is handy for some things

Animations made with Python! →



Explanation



Explanation - Cont.

```java
package src.workbench; // Main.java is in a folder called workbench, which is in a folder called src.

public class Main { // You can think of the Main class as the command center of the program.
    public static void main(String[] args) { // The main function is where your code starts executing.
        System.out.println("Hello, world!"); // Outputs: Hello, world!
    }
}

// This is a comment. It will be ignored by the program.
/*
This comment
can be on multiple lines.
*/
```

```python
# Python starts running immediately from line 1.
print("Hello, world!") # In Python, any file can run on its own.

# This is a comment.
"""
This comment
can be on multiple lines.
"""
```

Keyboard Map

Helpful keyboard map.

Variables

- A variable represents a piece of data
- Every piece of data has a **type**
  - The most basic types are called **primitive types**

Groups	Integers	Floating-Point	Boolean	Characters
Types	`byte`, `short`, `int`, `long`	`float`, `double`	`boolean`	`char`
Examples	0, -1, 32	3.14, -100.0, 0.0	`true` or `false`	a, b, c, *, /

Visit this website to learn more: docs.oracle.com

- Every variable has a name, names follow specific rules
  - Example valid names: `num`, `myNum`, `NUM`, `_num_`, `num1`

    (`_` is the only special character that can be used)
  - Example invalid names: `int`, `my num`, `😳`, `num`, `%num%`, `1num`, `my-num`

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

        int exampleVariable = 5;

        System.out.println("Value of exampleVariable: " + exampleVariable);
    }
}
```

Running...

```python
example_variable = 5 # Notice that Python automatically infers the type of the variable.

print("Value of exampleVariable:", example_variable)
```

Running...

- More complex data requires more complex types
  - These are called **reference types**
- Two commonly used **reference types** are *strings* and *arrays*

```java
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        // `String` must be capitalized! Value inside "".
        String videoGame = "Minecraft: Java Edition";
        int[] ratings = {1, 3, 2, 4, 5};
        // int[] ratings = new int[10];
        System.out.println("Video Game: " + videoGame + " Ratings: " + Arrays.toString(ratings));
    }
}
```

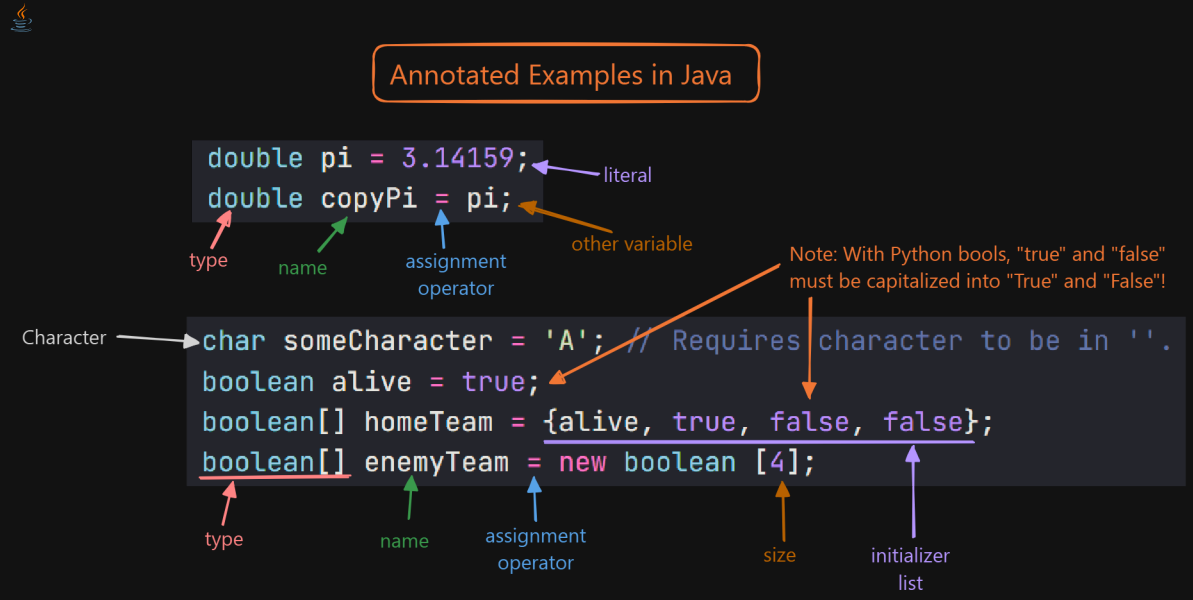
Running...

 (Python actually calls this a *list*, which is similar to *arrays*.)

```python
name = "Alice"
friends = ["Bob", "Charlie", "Dave"]
print("Name:", name, "Friends:", friends)
```

Running...

Summary


Annotated Examples in Java
double pi = 3.14159;
double copyPi = pi;
literal
other variable
type
name
assignment operator
Note: With Python bools, "true" and "false" must be capitalized into "True" and "False"!
Character
char someCharacter = 'A'; // Requires character to be in ''.
boolean alive = true;
boolean[] homeTeam = {alive, true, false, false};
boolean[] enemyTeam = new boolean [4];
type
name
assignment operator
size
initializer list

Basic I/O

- I/O means Input / Output
- Simplification:
  - By default, programs use the *terminal* to output
  - Also use the terminal to input text

```java
import java.util.Scanner; // Required to use the scanner.

public class Main {
    public static void main(String[] args) {
        Scanner scan = new Scanner(Systen.in); // Initialize a scanner.

        int num = scan.nextInt();        // `nextInt` gets an integer.
        double num2 = scan.nextDouble(); // `nextDouble` gets a floating-point.
        String s = scan.nextLine();      // `nextLine` gets a string.

        System.out.println("Values: " + num + " " + num2 + " " + s);
        scan.close() // Remember to close the scanner.
    }
}
```

```python
num = int(input("Enter a number: "))          # Get an integer.
num2 = float(input("Enter another number: ")) # Get a floating-point.
s = input("Enter a string: ")                 # Get a string.
```

Errors

- You will encounter many errors while programming
  - **Compile-time errors**: occur before you run the program
  - **Run-time errors**: occur when the program is running

Find the error:

```
System.out.println("Hello, world!");
```

Syntax Error ↑

```
int number = 123;
```

Type Error ↑

```
  int a = 5;
  int b = 0;
  int result = a / b;
```

Runtime Error: Divide by Zero ↑

Worksheet

Click here to access the worksheet.

Click here to access the answer key.

July 29

launch binder

Agenda

- Review basic data types
- Number bases, computer memory, binary, hexadecimal
- Rounding Errors
- Characters, ASCII
- *Break*
- Using Debuggers
- Understanding the scanner buffer
- Operators and Casting
- *Worksheet*

- More complex data requires more complex types
  - These are called **reference types**
- Two commonly used **reference types** are *strings* and *arrays*

```java
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        // `String` must be capitalized! Value inside "".
        String videoGame = "Minecraft: Java Edition";
        int[] ratings = {1, 3, 2, 4, 5};
        // int[] ratings = new int[10];
        System.out.println("Video Game: " + videoGame + " Ratings: " + Arrays.toString(ratings));
    }
}
```

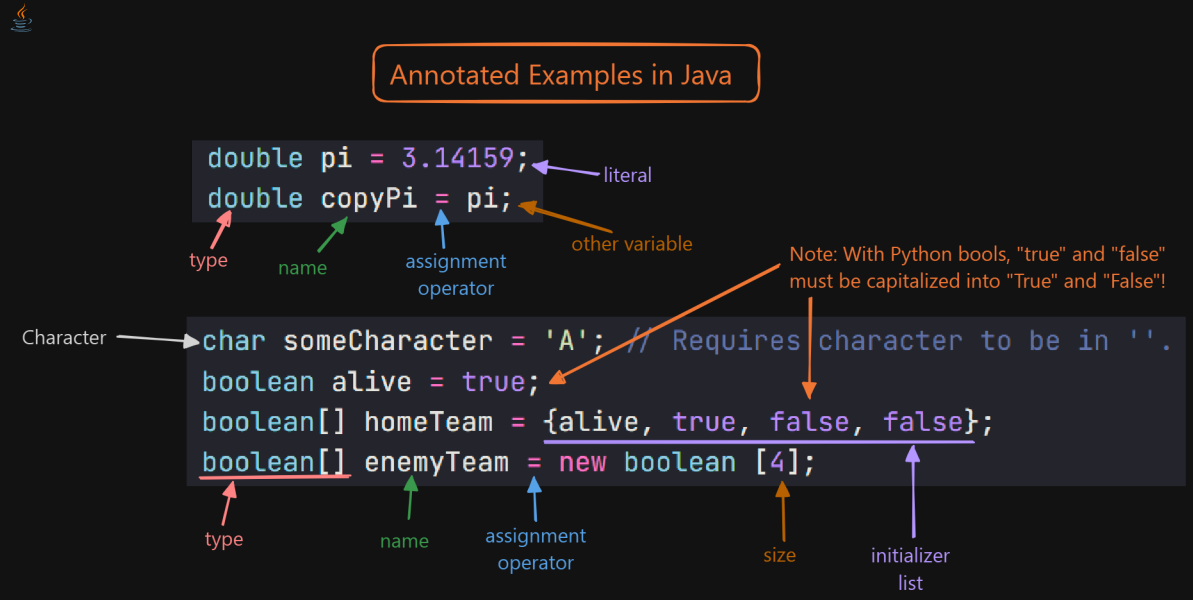
Running...

 (Python actually calls this a *list*, which is similar to *arrays*.)

```python
name = "Alice"
friends = ["Bob", "Charlie", "Dave"]
print("Name:", name, "Friends:", friends)
```

Running...

Summary


Annotated Examples in Java
double pi = 3.14159;
double copyPi = pi;
literal
other variable
type
name
assignment operator
Note: With Python bools, "true" and "false" must be capitalized into "True" and "False"!
Character
char someCharacter = 'A'; // Requires character to be in ''.
boolean alive = true;
boolean[] homeTeam = {alive, true, false, false};
boolean[] enemyTeam = new boolean [4];
type
name
assignment operator
size
initializer list

Basic I/O

- I/O means Input / Output
- Simplification:
  - By default, programs use the *terminal* to output
  - Also use the terminal to input text

```java
import java.util.Scanner; // Required to use the scanner.

public class Main {
    public static void main(String[] args) {
        Scanner scan = new Scanner(Systen.in); // Initialize a scanner.

        int num = scan.nextInt();        // `nextInt` gets an integer.
        double num2 = scan.nextDouble(); // `nextDouble` gets a floating-point.
        String s = scan.nextLine();      // `nextLine` gets a string.

        System.out.println("Values: " + num + " " + num2 + " " + s);
        scan.close() // Remember to close the scanner.
    }
}
```

Number Bases

- Most of the time, we use the **base 10** (decimal) number system
- For a number like 123, we can see that $1 * 10^2 + 2 * 10^1 + 3 * 10^0 = 1 * 100 + 2 * 10 + 3 * 1 = 123$
  - Each digit is multiplied by powers of 10
  - Start with $10^0 = 1$ on the right, and move to the left
  - Increase the power for each digit until reaching the leftmost digit
- Other number bases exist!

Computer Memory

- Computers are made of tiny circuits called **transistors**
  - A transistor is controlled by *electrical voltage*:
    - High voltage → `1`
    - Low voltage → `0`
    - If it's not `1` or `0`, then something is *very* wrong in the computer
- Computers use billions of transistors to represent data

Integers

Binary represents integers in **base 2**.

Base 2 to Base 10

Convert the binary number 11110110010_2 from base 2 to base 10.

$$
\begin{gathered}
1 \to 1 * 2^{10} \\
1 \to 1 * 2^{9} \\
1 \to 1 * 2^{8} \\
1 \to 1 * 2^{7} \\
0 \to 0 * 2^{6} = 0 \\
1 \to 1 * 2^{5} \\
1 \to 1 * 2^{4} \\
0 \to 0 * 2^{3} = 0 \\
0 \to 0 * 2^{2} = 0 \\
1 \to 1 * 2^{1} \\
0 \to 0 * 2^{0} = 0 \\
2^{10} + 2^{9} + 2^{8} + 2^{7} + 2^{5} + 2^{4} + 2^{2} = 1970
\end{gathered}
$$

```
11111101001 = 2025
```

```
1111011 = 123
```

Base 10 to Base 2

- To go from base 10 to base 2, we need to use the remainder when dividing.
  - $10 \div 3 = 3 \text{ remainder } 1$
- We only want the remainder. We have an operation for this called the **modulo**.
  - We use the symbol `%` to represent the modulo. This symbol is often used to represent percentages, but we are not using it for that.

Convert the number 1970_{10} from base 10 to base 2. Take the number and mod it by 2. Then, divide the number by 2 and repeat until you reach zero. Remove any digits past the decimal point after dividing by 2.

$$1970\%2 = 0$$
$$985\%2 = 1$$
$$492\%2 = 0$$
$$985/2{=}492.5, \text{ but we don't care about the } 0.5$$
$$246\%2 = 0$$
$$123\%2 = 1$$
$$61\%2 = 1$$
$$30\%2 = 0$$
$$15\%2 = 1$$
$$7\%2 = 1$$
$$3\%2 = 1$$
$$1\%2 = 1$$
$$0\%2 = 0$$

Then, starting from the bottom of the column of numbers, write down each modulo result.

$$\Rightarrow 011110110010$$
$$\Rightarrow \ 11110110010$$

```
456 = 111001000
```

This method only works with positive integers or zero.

Visit this page to learn how to represent negative integers in binary: en.wikipedia.org

Base 16 - Hexadecimal

- Binary (base 2) is hard to read
  - Programmers need a number system that's easy to use like base 10, but easier to convert to base 2
- In base 10, a digit can be from 0 to 9
  - In base **16**, a digit can be from 0 to 15
  - We use the characters A, B, C, D, E, and F to represent digits with values from 10 to 15
  - Base 16 is also called **hexadecimal**
- Example: $1970_{10} = 7B2_{16}$
  - $7 * 16^2 + B * 16^1 + 2 * 16^0 = 7 * 16^2 + 11 * 16^1 + 2 * 10^0 = 1970$

Decimal	Hexadecimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

To convert from base 2 to base 16 (and base 16 to base 2), simply use the table above!

```
1970 =
0111 1011 0010
   7    B    2
```

```
1000 = 3E8 = 0011 1110 1000
```

```
2025 = 0111 1110 1001 = 7E9
```

- Other number bases that are commonly used in programming are base 8 (octal) and base 64
- We can use Python to help us with converting between various number bases

```
decimal = 1234567890

binary = bin(decimal)
hexadecimal = hex(decimal)
octal = oct(decimal)

print(f"""
decimal: {decimal}
binary: {binary}
hex: {hexadecimal}
octal: {octal}
""")
```

Running...

```python
some_number = "F4240"
number_base = 16

decimal = int(some_number, number_base)

print(f"{some_number} in base {number_base} is {decimal} in base 10!")
```

Running...

Integer Limits

How big of a number can we have?

```python
big_number = 99999999999999999999999999999999999999999999999999999999999999999999
print(big_number)
```

Running...

In Python, numbers can be practically as big as you want them to be.

What about Java?

```java
public class Main {
    public static void main(String[] args) {
        int bigNumber = 99999999999999999999999999999999999999999999999999999999999999999999;
        // int bigNumber = 2147483647;
        // int bigNumber2 = bigNumber + 1;

        System.out.println("bigNumber: " + bigNumber + ", bigNumber2: " + bigNumber2);
    }
}
```

Running...

- In Java, integers declared with `int` are *32-bit signed integers*.
- This means, for positive integers, we only have **31** binary digits!
- 2147483647_{10} is $1111111111111111111111111111111_{}$ in binary How many `1` s are there?
  - There are exactly 31 `1` s.
  - If we try to make the number any bigger, even by adding 1, we will **overflow**
- The 32nd binary digit is a special digit that indicates if the integer is negative. That's why we only have 31 digits to use!

Floating-Point

Representing floating-point numbers like 123.456 is too complicated to cover here.
If you're curious, you can visit this site: en.wikipedia.org

```
num = 0.1 + 0.2
print(num)
```

Running...

- Computers use *scientific notation* to represent floating-point numbers
  - Accurate enough, but not perfect!
- This small difference from the correct value is called a **rounding error**

Characters

- Computers can only store data in binary
  - We've seen how integers are represented in binary
- What about characters? (Guesses?)
- Assign each character a number
  - This is called a **character encoding**
- ASCII is the base of most other character encodings

The ASCII Table

```
cook@pop-os:~$ ascii -d
   0 NUL    16 DLE    32      48 0    64 @    80 P    96 `   112 p
   1 SOH    17 DC1    33 !    49 1    65 A    81 Q    97 a   113 q
   2 STX    18 DC2    34 "    50 2    66 B    82 R    98 b   114 r
   3 ETX    19 DC3    35 #    51 3    67 C    83 S    99 c   115 s
   4 EOT    20 DC4    36 $    52 4    68 D    84 T   100 d   116 t
   5 ENQ    21 NAK    37 %    53 5    69 E    85 U   101 e   117 u
   6 ACK    22 SYN    38 &    54 6    70 F    86 V   102 f   118 v
   7 BEL    23 ETB    39 '    55 7    71 G    87 W   103 g   119 w
   8 BS     24 CAN    40 (    56 8    72 H    88 X   104 h   120 x
   9 HT     25 EM     41 )    57 9    73 I    89 Y   105 i   121 y
  10 LF     26 SUB    42 *    58 :    74 J    90 Z   106 j   122 z
  11 VT     27 ESC    43 +    59 ;    75 K    91 [   107 k   123 {
  12 FF     28 FS     44 ,    60 <    76 L    92 \   108 l   124 |
  13 CR     29 GS     45 -    61 =    77 M    93 ]   109 m   125 }
  14 SO     30 RS     46 .    62 >    78 N    94 ^   110 n   126 ~
  15 SI     31 US     47 /    63 ?    79 O    95 _   111 o   127 DEL
```

```python
# Convert a character into its ASCII code.
ascii_code = ord('J') # The quotes '' are needed.
print(ascii_code)

# Convert an ASCII code into its character.
c = chr(80)
print(c)
```

Running...

Exercise: ASCII values for...
'A': 65
'a': 97
'=': 61
Which character corresponds with the value 32?

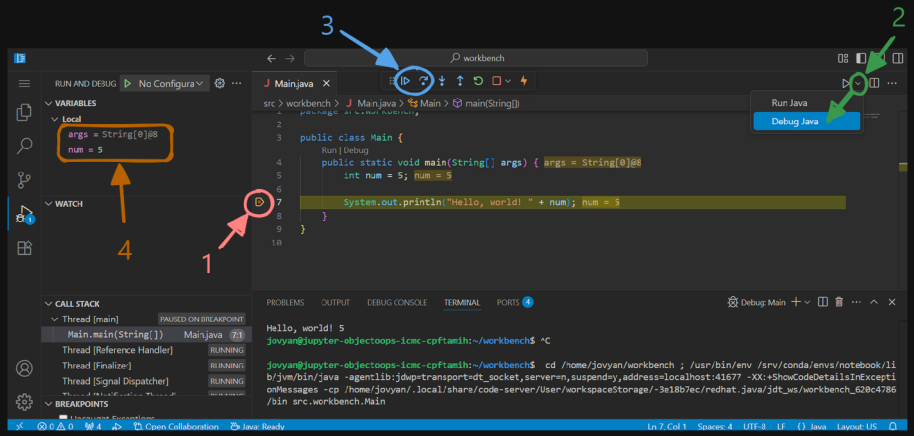
→ A space!

Break

Have a break!

Using Debuggers

Instructor Guided



The Scanner Buffer

What does the following code output?

(only applies to Java)

```java
Scanner scan = new Scanner(System.in);

int num = scan.nextInt();
String someString = scan.nextLine();

System.out.println("num: " + num + ", someString: " + someString);
```

Sample Output: `num: 5, someString: `

The variable `someString` is empty because of **buffering**.

- What character is at ASCII code 10?
  - The `LF` character is called the line feed or **newline** character
  - It is an *invisible character*, so the table labels it as "LF".
  - In this case, it represents that you have hit the enter key
- A **buffer** stores characters before they can be used by the program
  - When we entered 5 in the example above, the buffer looked like this: `5<LF>`
  - After `scan.nextInt()`, the buffer looked like this: `<LF>` (the character '5' was *consumed*)
  - When we entered a string like "hello" afterwards, the buffer became: `<LF>hello<LF>` ("hello<LF>" was inserted after the first "<LF>")
  - When calling `scan.nextLine()`, the scanner gets characters until the first `<LF>`
  - Finally, the buffer looks like this: `hello<LF>` (only the first "<LF>" was consumed) And the value of `someString` is an empty String, because there were no characters before the first LF

To fix this, add a `scan.nextLine()` after reading the integer:

```
Scanner scan = new Scanner(System.in);

int num = scan.nextInt();
scan.nextLine(); // Add this.
String someString = scan.nextLine();
```

Operators

- We have already seen the assignment operator (`=`) and the addition operator (`+`)
- There are many other operators
  - We will gradually learn more about them
  - For full descriptions, visit this site: docs.oracle.com
- The *order of operations* generally applies in programming the same way as in algebra
  - Multiplication (`*`) has higher precedence over addition (`+`)
  - Put your expressions inside parenthesis `(1 + 2) * 3` to given them higher precedence

Operators	Precedence
postfix	*expr*++ *expr*--
unary	++*expr* --*expr* +*expr* -*expr* ~ !
multiplicative	* / %
additive	+ -
shift	<< >> >>>
relational	< > <= >= *instanceof*
equality	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	|
logical AND	&&
logical OR	||
ternary	? :
assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

(mostly the same between Python and Java)

```python
num1 = 5
num2 = 6
num3 = 2 + num1 * num2
print(num3)
```

Running...

```java
public class Main {
    public static void main(String[] args) {
        String s = "Test";
        int n = 32;
        System.out.println(s + n);
    }
}
```

Running...

Casting

- To convert one data type to another data type, we can often use **casting**
- Casting is different between Java and Python

Example - Python

The same examples above, but in Python.
Similar to Java, but casting takes the form `type(value)`.

```
num = 3.14159
# Print the original `num`, but also casted to an integer.
print("num:", num, int(num))

# Casting to a String is a bit different:
num_string = str(num) # `str` is short for "string".
print("string:", num_string)

# Casting from a String to other data types:
num_string_2 = "123"
num2 = int(num_string_2)
print("num2:", num2)
```

Running...

Example - Java

Put `(type)` (where `type` is a data type) before a variable to cast it to another type.

```java
public class Main {
    public static void main(String[] args) {
        double num = 3.14159;
        // Print the original `num`, but also casted to an integer.
        System.out.println("num: " + num + " " + (int)num);

        int num2 = (int)num;
        // Casting to a String is a bit different:
        String numString = Double.toString(num); // `Double` is uppercase!
        String numString2 = Integer.toString(num2);
        System.out.println("strings: " + numString + " " + numString2);

        // Casting from a String to other data types:
        String numString3 = "123";
        int num3 = Integer.valueOf(numString3);
        System.out.println("num3: " + num3);
    }
}
```

Running...

Integer Division

In **Java**, when using the division operator (`/`) between **two integers**, integer division is performed.

Integer division is the same as regular division, but any digits after the decimal point are discarded.

> *This is called "flooring".*

```java
public class Main {
    public static void main(String[] args) {
        double num = 10 / 3;
        System.out.println(String.valueOf(num));
    }
}
```

Running...

Worksheet

Click here to access the worksheet.

Click here to access the answer key.

July 30

launch binder

Agenda

- Review basic I/O
- Using Arrays
- String Manipulation
- Basic Escape Sequences
- *Break*
- Function Scope
- Defining Functions
- Intro to Classes
- *Worksheet*

1. Basic I/O
2. Arrays
    1. Arrays in Computer Memory
3. Multidimensional Arrays
    1. Multidimensional Arrays in Computer Memory
4. String Manipulation
5. Basic Escape Sequences
6. Break
7. Functions / Methods
    1. Global Variables
    2. Summary
8. Intro to Classes
    1. Example
9. Worksheet

Basic I/O

- I/O means Input / Output
- Simplification:
  - By default, programs use the *terminal* to output
  - Also use the terminal to input text

```java
import java.util.Scanner; // Required to use the scanner.

public class Main {
    public static void main(String[] args) {
        Scanner scan = new Scanner(Systen.in); // Initialize a scanner.

        int num = scan.nextInt();        // `nextInt` gets an integer.
        double num2 = scan.nextDouble(); // `nextDouble` gets a floating-point.
        String s = scan.nextLine();      // `nextLine` gets a string.

        System.out.println("Values: " + num + " " + num2 + " " + s);
        scan.close() // Remember to close the scanner.
    }
}
```

Arrays

- An array is a collection of multiple pieces of data
  - An array is a kind of **data structure** , because it structures other pieces of data
  - We call these pieces of data **elements**
  - An array is *fixed-size*, meaning that it always has the same number of elements
    - Python uses **lists**, which are similar to arrays
    - For now, we will treat them as the same thing
  - Most of the time, every **element** in an array will be of the same **data type**

- Defining an array:

```java
String[] fruits = {"Apple", "Banana", "Cantaloupe"};
String[] fiveEmptyStrings = new String[5];
int[] fiveZeroes = new int[5];
```

 (Technically a list.)

```python
fruits = ["Apple", "Banana", "Cantaloupe"]
```

- Arrays can be **subscripted** to access their **elements**
  - Each element in an array can be referred to by a number called an **index**
  - In Java and Python, indices (plural for index) start from 0

```java
String firstFruit = fruits[0]; // The string "Apple" is at index 0.
String secondFruit = fruits[1]; // The string "Banana" is at index 1.
```

```python
first_fruit = fruits[0] # "Apple", same for Python.
second_fruit = fruits[1] # "Banana"
```

- Getting the size / length of an array:

```java
int fruitsCount = fruits.length;
```

```python
fruits_count = len(fruits) # `len` is short for "length".
```

- Things to not do with arrays:
  - Mix elements of different types
  - Attempt to access an element that does not exist (such as with an index that's too big or negative)

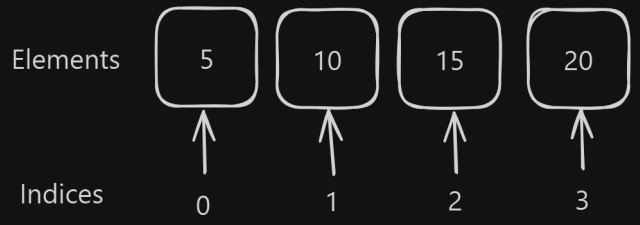
```java
public class Main {
    public static void main(String[] args) {
        String[] fruits = {"Apple", "Banana", "Cantaloupe"};

        String someFruit = fruits[3]; // Invalid index.
        System.out.println(someFruit);
    }
}
```

Running...

Arrays in Computer Memory

Elements in an array are sequentially stored with incremental indices.

```
1   int[] nums = {5, 10, 15, 20};
```



Multidimensional Arrays

You can have arrays as elements of other arrays.
The Multidimensional Array

```java
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        int[][] matrix = {
            {2, 4, 6},
            {8, 10, 12},
            {14, 16, 18}
        };
        System.out.println(Arrays.toString(matrix[1]));
        System.out.println(matrix[1][2]);
    }
}
```

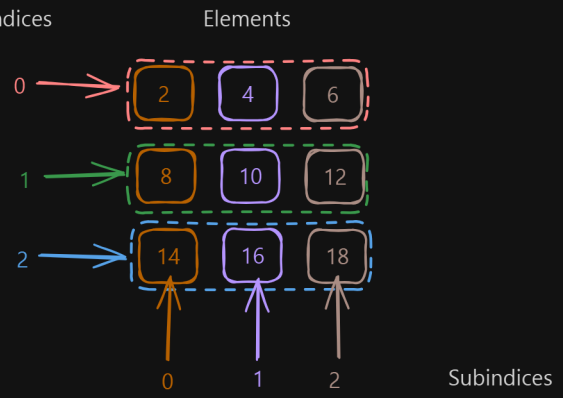
Running...

```python
# Same array in Python.
matrix = [[2, 4, 6], [8, 10, 12], [14, 16, 18]]
```

Multidimensional Arrays in Computer Memory

Consider the matrix from the previous slide.

```
int[][] matrix = {
    {2, 4, 6},
    {8, 10, 12},
    {14, 16, 18}
};
```



String Manipulation

- Inside a string is an **array of characters**

```java
// Kind of like this:
char[] insideString = {'h', 'e', 'l', 'l', 'o'};
```

  - We can use indices to access each character in a string →

```java
public class Main {
    public static void main(String[] args) {
        String s = "Hello, world!";
        char c = s.charAt(7); // Get character at index 7.
        System.out.println(c);
    }
}
```

Running...

```python
s = "Hello, world!"
c = s[7] # Get character at index 7.
print(c)
```

Running...

- We can also use indices to get a specific part of a string

```java
public class Main {
    public static void main(String[] args) {
        //          0123456789...
        String s = "Hello, world!";
        // Get a substring from string `s`, starting at index 0 and stopping right before index 5.
        String s2 = s.substring(0, 5);
        System.out.println(s2);
    }
}
```

Running...

```python
#    0123456789...
s = "Hello, world!"
# Get a substring starting at index 0 and stopping right before index 5.
s2 = s[0:5]
print(s2)
```

Running...

- The addition operator `+` works with strings. It's fairly obvious what it does

```java
String s1 = "Hello, ";
String s2 = "world!";
String s3 = s1 + s2; // ⟶ s3 is "Hello, world!"
```

```python
s1 = "Hello, "
s2 = "world!"
s3 = s1 + s2 # ⟶ s3 is "Hello, world!"
```

- Getting the size / length of a string

```java
String s = "...";
int stringLength = s.length(); // Notice that the empty parenthesis "()" are required.
```

```python
s = "..."
str_len = len(s)
```

- There are more ways to manipulate strings (like replacing parts of a strings with `.replace`).

  Learn more here:

  Java: docs.oracle.com

  Python: docs.python.org

Basic Escape Sequences

- When defining *strings* and *characters* in Java and Python, some characters cannot be usde directly.

```
fail_string = "This is an "invalid string""
```

Running...

- In the previous example, the double quote character (`"`) is already used to indicate the start and end of a string
  - Using more double quote characters within the string "confuses" the programming language as to where the string starts, and where it ends.
- To use double quotes inside a string, we can use a backslash (`\`)

```
fail_string = "This is a \"valid string\"" # The \ character "escapes" the double quote.
print(fail_string) # Notice that the \ character is not printed, but the double quotes are printed.
```

Running...

- The sequence of characters `\"` is called an **escape sequence**
- Inside a string `\"` is replaced by `"`
- Characters to **not** use inside strings / characters and their **escape sequence**:
  - `"` → `\"`
  - `'` → `\'`
  - `\` → `\\`
  - See the keyboard map to locate the `\` character
- Escape sequences can also insert invisble characters
  - The newline (LF) character → `\n`
  - The tab character → `\t`
  - See the ASCII Table for more

```
print("This string prints \" \' \n \t escape sequences.")
```

Running...

Break

Have a break!

Functions / Methods

- A **function**, also called a **method**, is a block of code that can be reused in varying contexts
  - This definition is unclear; let's see some examples
- Say that we have a math function $f(x) = 2^x * 100$
  - $f(0) = 100$
  - $f(1) = 200$
  - $f(5) = 3200$
- If we have a complicated formula, we can simplify it with our function
  - $2^5 * 100 + 2^1 * 100 + 2^0 * 100 \Rightarrow f(5) + f(1) + f(0) = 3200 + 200 + 100 = 3500$
- We can do the same thing in languages like Java and Python
- Functions have four parts:
  - *return type*
  - *name*
  - *parameters*
  - *definition*
    - Might include: *return value* (if *return type* is not `` `void` ``)
    - **Function scope**: Variables declared inside a function can only be used inside that function!

Example 1 - Java

Our goal: Ask the user for two names using the same prompt and output them together.

```java
import java.util.Scanner;
import java.util.Arrays;

public class Main {

    private static Scanner scan = null; // This is a global variable.

    public static void main(String[] args) {
        scan = new Scanner(System.in);

        String[] manyNames = {prompt(1), prompt(2), prompt(3), prompt(4)};

        System.out.println(Arrays.toString(manyNames));

        scan.close();

        // We can use the same `prompt` many times!
        // Less typing, more coding!
    }

    private static String prompt(int num) {
        System.out.println("Enter name #" + num);
/* Important!: `name` has "function scope". It can only be used inside the `prompt` function! */
        String name = scan.nextLine();
        return name;
    }
}
```

Example 2 - Java

This example shows how to define a function with **multiple parameters** and **no return value**.
Goal: Calculate and output the *area* and *circumference* of two circles.

$$A(r) = \pi * r^2$$
$$P(r) = 2 * \pi * r$$

Please view the full file here: FunctionsExample2.java

```java
public class FunctionsExample2 {
    private static final double pi = 3.14159; // Global variable.

    public static void main(String[] args) {
        double radius1 = 3;
        double radius2 = 9;

        double a1 = area(radius1);
        double a2 = area(radius2);
```

Running...

Examples - Python

These are the same as the two prior examples, but written using Python.
Python automatically infers the types for the *return type* and *parameters*.

Example 1

```python
def prompt(num): # ←— notice that Python uses a single colon ":" instead of curly braces "{}".
    return input(f"Enter name #{num}\n")

names = [prompt(1), prompt(2), prompt(3), prompt(4)]

print(names)
```

Continued →

Example 2

```
pi = 3.14159

# All Python functions start with `def`.
def area(radius):
    return pi * radius * radius

def circumference(radius):
    return 2 * pi * radius

def output_circle_info(r, a, c):
    print("Circle Info:")
    print(f"Radius: {r}")
    print(f"Area: {a}")
    print(f"Circumference: {c}")

radius1 = 3
radius2 = 9

a1 = area(radius1)
a2 = area(radius2)

c1 = circumference(radius1)
c2 = circumference(radius2)

output_circle_info(radius1, a1, c1)
output_circle_info(radius2, a2, c2)
```

Global Variables

In the past few Java examples, we saw the use of **global variables** or "class-scope variables".

```
1   private static Scanner scan = null;
```

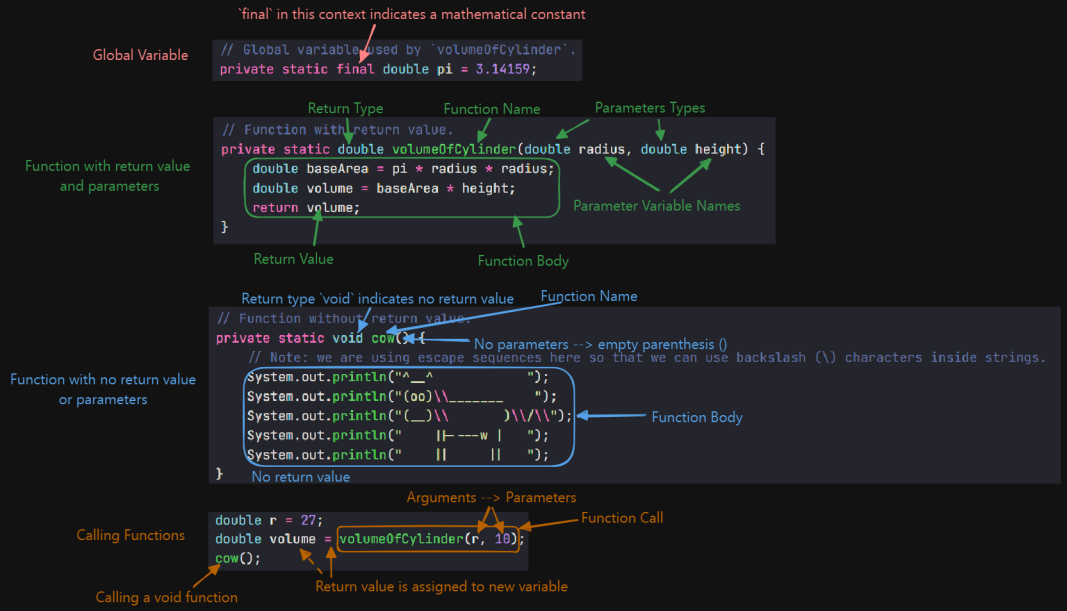
```
1   private static final double pi = 3.14159;
```

And with Python...

```
1   pi = 3.14159
```

- Global variables are available to all functions!
- Useful for values which don't change (*constants*)
- The `final` keyword used when declaring the global variable `pi` tells Java that the value of `pi` cannot be changed to something other than 3.14159.
- The `static` keyword will be discussed later

Summary


`final` in this context indicates a mathematical constant
Global Variable
// Global variable used by `volumeOfCylinder`.
private static final double pi = 3.14159;
Return Type
Function Name
Parameters Types
Function with return value and parameters
// Function with return value.
private static double volumeOfCylinder(double radius, double height) {
double baseArea = pi * radius * radius;
double volume = baseArea * height;
return volume;
}
Parameter Variable Names
Return Value
Function Body
Return type `void` indicates no return value
Function Name
Function with no return value or parameters
// Function without return value.
private static void cow() {
No parameters --> empty parenthesis ()
// Note: we are using escape sequences here so that we can use backslash (\) characters inside strings.
System.out.println("^__^ ");
System.out.println("(oo)_____ ");
System.out.println("(__)\\ )\\/\\");
System.out.println(" ||---w | ");
System.out.println(" || || ");
}
Function Body
No return value
Arguments --> Parameters
Function Call
Calling Functions
double r = 27;
double volume = volumeOfCylinder(r, 10);
cow();
Calling a void function
Return value is assigned to new variable

Intro to Classes

- Let's say that we want to store information about multiple people
- We can use arrays:

```
String[] names  = {   "Alice",      "Bob", "Charlie"};
int[]    ages   = {        20,         21,        25};
String[] cities = {"Shanghai", "New York",  "London"};
```

- To get an individual person:

```
String name = names[0]; // Alice
int age = ages[0]; // 20
String city = cities[0]; // Shanghai
```

- This is an inefficient way of storing and retrieving data!
- **Classes** allow us to *abstract* our data
  - A **class** is like a blueprint for how to construct something
  - An **object** is like a thing made from a **class**'s "blueprint"
- A **class** has components called **members**
  - There are multiple kinds of members
  - **attributes** - variables contained within a class
  - **methods** - functions which are a part of a class
  - **constructors** - *special methods* which define how to create objects from a class
    - When we create an *object* from a *class*, we call it *instantiation*
  - **static members** - if an **attribute** or **method** is *static*, it means that the member is associated with the class, and not an object
    - Useful for defining *constants* and extra functions
- We will only be looking at classes in Java

Example

Please view the full file here: ClassExample.java

```java
public class ClassExample {
    public static void main(String[] args) {
        // We can still use arrays to store multiple people, but here we only need one array.
        Person[] people = {
            // Use the `new` keyword when creating objects from the Person class.
            new Person("Alice", 20, "Shanghai", "Aly"), // This a person object.
            new Person("Bob", 21, "New York", "B"), // Another person object.
            new Person("Charlie", 25, "London", "Chip") // ...
        };

        // Get a person.
        Person person = people[0]; // `person` is an object instantiated from `Person`.
        Person samePerson = people[0];

        person.sayHi();
        samePerson.sayHi();

        // Both `person` and `samePerson` refer to the same object of `Person`.
```

Running...

Worksheet

Click here to access the worksheet.

Click here to access the answer key.

Bonus worksheet (one question, if you're willing).

Bonus worksheet answer key.

July 31

launch binder

Agenda

- Review arrays and functions
- Boolean Expressions
- Block Scope
- Conditionals
- Loops
- *Break*
- Pass by value vs. pass by reference
- Basic Runtime Complexity
- Data structures beyond arrays
- *Worksheet*

- Defining an array:

```java
String[] fruits = {"Apple", "Banana", "Cantaloupe"};
String[] fiveEmptyStrings = new String[5];
int[] fiveZeroes = new int[5];
```

 (Technically a list.)

```python
fruits = ["Apple", "Banana", "Cantaloupe"]
```

- Arrays can be **subscripted** to access their **elements**
  - Each element in an array can be referred to by a number called an **index**
  - In Java and Python, indices (plural for index) start from 0

```java
String firstFruit = fruits[0]; // The string "Apple" is at index 0.
String secondFruit = fruits[1]; // The string "Banana" is at index 1.
```

```python
first_fruit = fruits[0] # "Apple", same for Python.
second_fruit = fruits[1] # "Banana"
```

- Getting the size / length of an array:

```java
int fruitsCount = fruits.length;
```

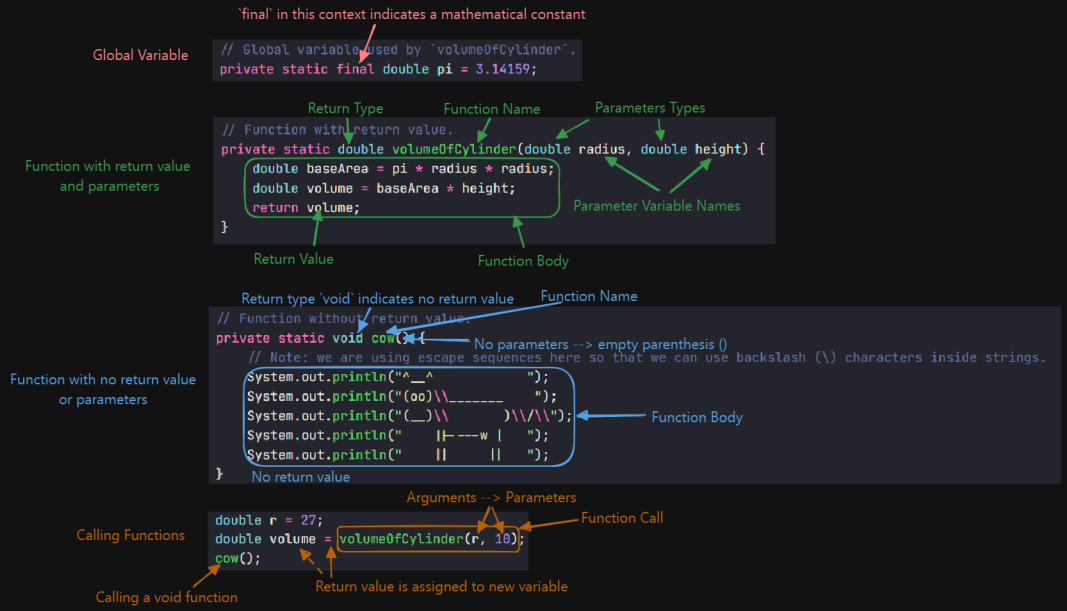
```python
fruits_count = len(fruits) # `len` is short for "length".
```

- Things to not do with arrays:
  - Mix elements of different types
  - Attempt to access an element that does not exist (such as with an index that's too big or negative)

```java
public class Main {
    public static void main(String[] args) {
        String[] fruits = {"Apple", "Banana", "Cantaloupe"};

        String someFruit = fruits[3]; // Invalid index.
        System.out.println(someFruit);
    }
}
```

Running...

Summary


`final` in this context indicates a mathematical constant
Global Variable
// Global variable used by `volumeOfCylinder`.
private static final double pi = 3.14159;
Return Type
Function Name
Parameters Types
Function with return value and parameters
// Function with return value.
private static double volumeOfCylinder(double radius, double height) {
double baseArea = pi * radius * radius;
double volume = baseArea * height;
return volume;
}
Parameter Variable Names
Return Value
Function Body
Return type `void` indicates no return value
Function Name
Function with no return value or parameters
// Function without return value.
private static void cow() {
No parameters --> empty parenthesis ()
// Note: we are using escape sequences here so that we can use backslash (\) characters inside strings.
System.out.println("^__^ ");
System.out.println("(oo)_____ ");
System.out.println("(__)\\ )\\/\\");
System.out.println(" ||----w | ");
System.out.println(" || || ");
}
Function Body
No return value
Arguments --> Parameters
Function Call
Calling Functions
double r = 27;
double volume = volumeOfCylinder(r, 10);
cow();
Calling a void function
Return value is assigned to new variable

Boolean Expressions

Comparison Operators

- This is the multiplication operator: `*`
  - What needs to be on either side of it to work? *Two integers or floating-point values*
  - What does it return? *An integer if both values are integers, a floating-point if at least one value is an integer*
- This is the equality operator: ==
  - What needs to be on either side of it to work (guess)? *Two variable of compatible primitive types, like an integer and an integer, or a double and an integer*
  - What does it return (guess)? **A boolean value**

- List of comparison operators:
  - `==` → equality operator, tests if two values are equal
    - The single equals sign `=` is already used as the assignment operator!
  - `!=` → inequality operator, tests if two values are not equal
  - `<` → less than
  - `>` → greater then
  - `<=` → less than or equal to
  - `>=` → greater than or equal to

```python
num1 = 5
num2 = 7.0

bool1 = True
bool2 = False

print(f"num1 == num2: {num1 == num2}")
print(f"num1 != num2: {num1 != num2}")
print(f"num1 + 2 == num2: {num1 == num2}")
print(f"num1 + 2 != num2: {num1 != num2}")
print(f"num1 < num2: {num1 < num2}")
print(f"num1 > num2: {num1 > num2}")

print(f"bool1 == bool2: {bool1 == bool2}")
print(f"bool1 != bool2: {bool1 != bool2}")
print(f"not bool1 == bool2: {not bool1 == bool2}")
print(f"not bool1 != bool2: {not bool1 != bool2}")
# Note that in Java, the not operator is `!`, so replace "not" above with `!`.
# Example: !bool1 == bool2
```

Running...

Logical Operators

- A logical operator takes boolean value(s) and outputs a boolean value
- Operators:
  - Java ☕:
    - `&&` → AND operator
    - `||` → OR operator
    - `!` → NOT operator
  - Python 🐍:
    - `and` → AND operator
    - `or` → OR operator
    - `not` → NOT operator

AND Operator

Given two boolean values (*A*, *B*), the AND operator returns *true* if and only if *A* AND *B* are both *true*.

- `1` is *true*
- `0` is *false*

The following shows the **truth table** for AND.
A **truth table** shows all possible cominations of inputs, and the corresponding outputs.

AND	A=0	A=1
B=0	0	0
B=1	0	1

OR Operator

Given two boolean values (*A*, *B*), the OR operator returns *true* if *A* is *true* OR *B* is *true*.

OR	A=0	A=1
B=0	0	1
B=1	1	1

NOT Operator

Given **a single** boolean value (*A*), the NOT operator returns the opposite of *A*.

	NOT	
	A=0	1
	A=1	0

Let's say that Mr. Dijkstra (a very famous computer scientist) wants to get from the city of Rotterdam to Groningen, but he needs to pass through some other cities inbetween.

These are the distances between the cities:

Distances (km)	Rotterdam	Amsterdam	Utrecht	Groningen
Rotterdam	0	65	55	?
Amsterdam	65	0	40	180
Utrecht	55	40	0	180
Groningen	?	180	180	0

However, he only wants to know which routes meet **at least one** of these criteria:

- If the route is the shortest
- The distance of the route **is not** divisible by 7 or 11

Example - Java

Please view the full file here: Distances.java

```java
public class Distances {
    public static void main(String[] args) {
        int r_to_a = 65;
        int r_to_u = 55;
        int a_to_u = 40;
        int u_to_a = 40;
        int a_to_g = 180;
        int u_to_g = 180;

        int path_1 = r_to_a + a_to_g;
        int path_2 = r_to_u + u_to_g;
        int path_3 = r_to_a + a_to_u + u_to_g;
        int path_4 = r_to_u + u_to_a + a_to_g;

        boolean path_1_valid = isPathValid(path_1, path_2, path_3, path_4);
        boolean path_2_valid = isPathValid(path_2, path_1, path_3, path_4);
        boolean path_3_valid = isPathValid(path_3, path_1, path_2, path_4);
        boolean path_4_valid = isPathValid(path_4, path_1, path_2, path_3);
```

Running...

Example - Python

Please view the full file here: distances.py

```
r_to_a = 65
r_to_u = 55
a_to_u = 40
u_to_a = 40
a_to_g = 180
u_to_g = 180

path_1 = r_to_a + a_to_g
path_2 = r_to_u + u_to_g
path_3 = r_to_a + a_to_u + u_to_g
path_4 = r_to_u + u_to_a + a_to_g

def is_path_valid(main_path, other_path_a, other_path_b, other_path_c):
    # This is the main boolean expression that return `True` if the path meets Dijkstra's criteria.
    return (
        (main_path <= other_path_a and main_path <= other_path_b and main_path <= other_path_c)
        or (main_path % 7 != 0 and main_path % 11 != 0)
    )
```

Running...

Block Scope

- Earlier, we covered **function scope**
  - Variables declared in a function can only be used inside that function
- In programming, we use **blocks** to separate code
  - In Java, each set of curly braces (`{}`) defines its own **block**
  - In Python, the colon (`:`) and indentation (empty space that appears before code on a line of code) defines a **block**
- **block scope**: Variables declared inside a block can only be used inside that block, or subblocks
- When a variable is no longer accessible because its **block** has been completely executed, we say that the variable is "*out-of-scope*"

Example

```java
public class Main {
    public static void main(String[] args) {
        {
            int num = 5;
        }
        System.out.println("num: " + num); // Fails: `num` is "out-of-scope". It is in a different block

        {
            int num = 6; // We can re-declare `num` since the old one is now "out-of-scope".
            {
                System.out.println("num: " + num);
                // Succeeds: The print statement is inside a subblock, which can access its "parent block"
            }
        }
    }
}
```

Running...

Conditionals

- Conditionals allow us to control the flow of execution in our code
  - Use boolean expressions to determine which code should run
  - Made of **if-else statements**
  - Each **if-else statement** is its own **block**
- For instance: play an animation **if** *the user presses a button*, **else** tell the user to press the button

Example - Java

```java
public class Main {
    public static void main(String[] args) {
        int num1 = 3, num2 = 5;
        boolean bool1 = true, bool2 = false;

        // These outer parenthesis "(expr)" are necessary.
        if (num1 < num2) { // Case 1: Runs if num1 < num2.
            System.out.println("num1 < num2");
        } else if (num1 == num2) { // Case 2: Runs if Case 1 did not run and num1 == num2.
            System.out.println("num1 == num2");
        } else { // If both Case 1 and Case 2 did not run, then num1 > num2.
            System.out.println("num1 > num2");
        }

        if (bool1 || bool2) { // Case 1: Runs if bool1 is true OR bool2 is true.
            System.out.println("OR");
        }
        if (bool1 && bool2) { // This is a separate case! It doesn't depend on Case 1!
            System.out.println("AND");
        } else {
            System.out.println("At least one is false.");
        }
    }
}
```

Running...

Example - Python

```python
num1 = 3
num2 = 5
bool1 = True
bool2 = False

# Python does not need the outer parenthesis "(expr)" and uses a colon ":".
if num1 < num2: # Case 1: Runs if num1 < num2.
    print("num1 < num2")
# Instead of "else if" Python uses a single word "elif".
elif num1 == num2: # Case 2: Runs if Case 1 did not run and num1 == num2.
    print("num1 == num2")
else: # If both Case 1 and Case 2 did not run, then num1 > num2.
    print("num1 > num2")

if bool1 or bool2: # Case 1: Runs if bool1 is true OR bool2 is true.
    print("OR")

if bool1 and bool2: # This is a separate case! It doesn't depend on Case 1!
    print("AND")
else:
    print("At least one is false.")
```

Running...

Loops

- Loops allow us to run a block of code multiple times
- Very similar to conditionals
  - A loop will run if a specific boolean expression is *true*
- There are four kinds of loops:
  - `while` loops
  - `for` loops
  - *for-each* loops
  - *do-while* loops
- Loops can be stopped by any point by using a `break` statement
- Loops can jump back to their starting point by using a `continue` statement
- Each loop is its own **block**

`while` Loop

A while loop will run as long as a boolean expressionis *true*.

The following examples print the integers 0 to 9.

```java
public class Main {
    public static void main(String[] args) {
        int num = 0;

        while (num < 10) {
          System.out.print(num + " ");
          ++num; // This is the prefix increment operator.
          // It does the same thing as num = num + 1.
        }
    }
}
```

Running...

```python
num = 0
while num < 10:
    print(num, end=' ')
    num += 1 # Python does not have the prefix increment operator.
    # However, num += 1 is also the same thing as num = num + 1.
    # The += operator also works in Java.
print()
```

Running...

`for` Loop

- A `for` loop has three parts:
  - *initialization statement* → initializes a variable
    - Often used during the loop
  - *condition* → a condition that indicates if the body of the for loop should run (like the `while` loop)
  - *update statement* → a statement that does something at the end of *each loop*
  - Often used to update variables that control the loop

The following examples print the *even integers* from 0 to 20.

```java
public class Main {
    public static void main(String[] args) {
        // int i = 0;

        // Notice that we use i <= 20 so that i == 20 will be true.
        for (int i = 0; i <= 20; i += 2) { // i += 2 ensures that only even numbers will be calculated.
            System.out.print(i + " ");
        }
    }
}
```

Running...

```python
# for-loops in Python are weird. You can just pay attention to how to use them.
for i in range(21):
    if i % 2 == 0: # If i is even.
        print(i, end=' ')
print()
```

Running...

for-each loops

- These loops are useful when going through the elements of an array **in order**

The following examples print the elements in an array of strings.

```java
public class Main {
    public static void main(String[] args) {
        String[] arr = {"Alice", "Bob", "Charlie"};
        for (String s : arr) {
            System.out.print(s + " ");
        }
    }
}
```

Running...

```python
arr = ["Alice", "Bob", "Charlie"]
for s in arr:
    print(s, end=' ')
print()
```

Running...

do-while loops

- *do-while* loops are rarely used
  - Checks if the condition is true **after** running the body / block of code (a `while` loop check the condition **before** running)
  - Can help in some specific cases
- Java has *do-while* loops, but Python does not

The following example continuously asks the user to enter "confirm" before executing the rest of the program.

```java
public class Main {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);

        String input;
        do {
            System.out.print("Please enter \"confirm\": ");
            input = scan.nextLine();
        } while (!input.equals("confirm")); // We will discuss why comparing Strings is tricky later.

        scan.close();
    }
}
```

`break` statement

To stop and break out of a loop, use the `break` statement.

```java
public class Main {
    public static void main(String[] args) {
        int num = 0;
        while (true) {
            ++num;
            if (num == 9) {
                break; // Breaks the loop.
            }
            System.out.print(num);
            System.out.print(" | ");
        }
        // Execution jumps to here when `break` executes.
        System.out.println("\nDone");
    }
}
```

Running...

```python
num = 0
while True:
    num += 1
    if num == 9:
        break
    print(num, end=' ')
    print(" | ", end=' ')
print("\nDone")
```

Running...

`continue` statement

To instruct a loop to jump back to its starting point, use the `continue` statement.

```java
public class Main {
    public static void main(String[] args) {
        int num = 0;
        while (num < 20) { // This is the "Starting Point".
            ++num;
            if (num == 9) {
                continue; // Causes the loop to jump back to "Starting Point".
            } // If `continue` executes, then any line below this line will not be run.
            System.out.print(num);
            System.out.print(" | ");
        }
        // Execution jumps to here when `break` executes.
        System.out.println("\nDone");
    }
}
```

Running...

```python
num = 0
while num < 20:
    num += 1
    if num == 9:
        continue
    print(num, end=" | ")
print("\nDone")
```

Running...

Break

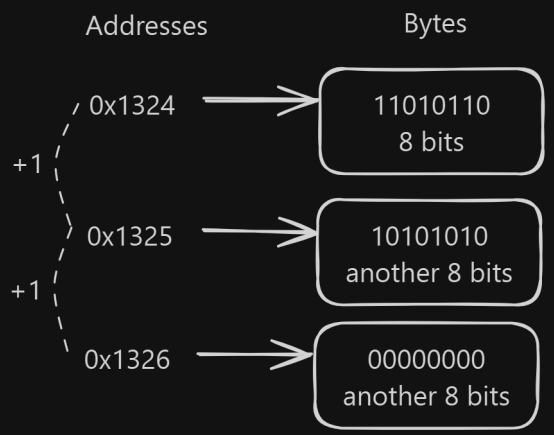
Have a break!

Pass by reference vs. pass by value

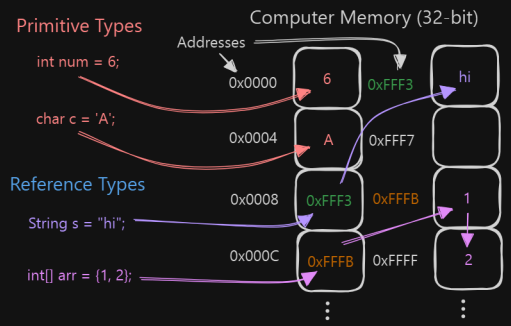
- Notice testing if `input` was not equal to "confirm" in the previous example
  - `!input.equals("confirm")`
  - *not input string equals "confirm"*
- The `String` type is a *reference type*
  - Other reference types include *arrays*, the `Scanner`, and any *classes*
  - We cannot use the equality operator == on reference types

Variables in Computer Memory

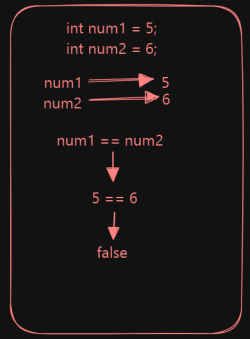
- Many modern computers break data into **bytes**
  - A **byte** is 8 **bits**
  - A **bit** is a single binary state state, `1` or `0`
  - Recall: A 32-bit integer has "31 binary digits and 1 bit to indicate the sign"
    - Link to slide
- In these systems, each **byte** has an **address** that represents where it is in computer memory



- *primitive types* like `int`, `double`, `char`, `boolean`:
  - Variable contains data in bytes
- *reference types* like `String`, `Scanner`, *arrays*, any *classes*:
  - Variable contains address to data in bytes (simplification)
- If we use the equality operator `==` on reference types, it compares the addresses, not the data

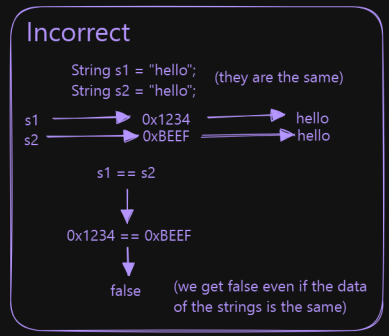


Comparing Primitive Types

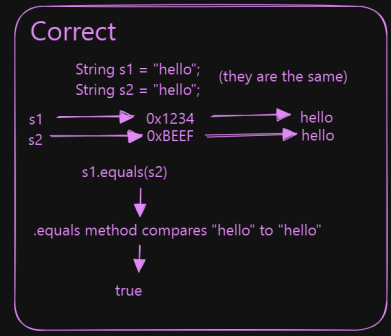


Comparing Reference Types

Incorrect



Correct



Example 1 - Java

```java
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        // Comparing Strings
        String s1 = new String("hello");
        String s2 = new String("hello");
        // Incorrect
        System.out.println(s1 == s2);
        // Correct
        System.out.println(s1.equals(s2));

        // Comparing Arrays
        int[] arr1 = {1, 2, 3};
        int[] arr2 = {1, 2, 3};
        // Incorrect
        System.out.println(arr1 == arr2);
        // Correct
        System.out.println(Arrays.equals(arr1, arr2));
        // System.out.println(Arrays.deepEquals(arr1, arr2)); // Use for multidimensional arrays.
    }
}
```

Running...

Example 2 - Java ☕

- When passing *arguments* into the *parameters* of a *function*,
  - The data of *primitive types* is copied
  - The data of *reference types* is **not** copied
- Example on next slide →

```java
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        int[] nums = {1, 2, 3};
        int num = 4;
        String myString = "Test 1";
        System.out.println("nums before: " + Arrays.toString(nums));
        System.out.println("num before: " + num);
        System.out.println("myString before: " + myString);
        doSomething(nums, num, myString);
        System.out.println("nums after: " + Arrays.toString(nums));
        System.out.println("num after: " + num);
        System.out.println("myString after: " + myString);
    }

    private static void doSomething(int[] arr, int n, String s) {
        arr[1] = 3;
        n = 5;
        s = "Test 2"; // Although strings are a reference type, Java treats them more like primitive type in this context.
    }
}
```

Running...

- Both `nums` and `arr` are references to the same data. Changing one of them will change the other.
- `num` and `n` each have their own copy of the data originally in `num`.
  - Changing one of them **will not** change the other.
  - Same with `myString` and `s` (but this is a special case with strings in Java)

Examples - Python

In Python, things are very strange and it's difficult to explain.
The example below shows how primitive and reference types are treated.

```python
# Comparing Strings
s1 = "hello"
s2 = "hello"
print(s1 == s2) # This is correct in Python!
# The Python interpreter is designed to make things easier for programmers.
# It knows to compare the data in the strings when using the `==` operator.

arr1 = [1, 2, 3]
arr2 = [1, 2, 3]
print(arr1 == arr2) # This also works. The Python interpreter helps us again.

arr3 = [[1, 2, 3], [4, 5, 6]]
arr4 = [[1, 2, 3], [4, 5, 6]]
print(arr3 == arr4) # This also works. The Python interpreter helps us again.

def do_something(arr, n, s):
    arr[1] = 3
    n = 5
    s = "Test 2"

nums = [1, 2, 3]
num = 4
my_string = "Test 1"
print(f"Before: {nums=} {num=} {my_string=}")
do_something(nums, num, my_string)
print(f"After: {nums=} {num=} {my_string=}")
```

Running...

Basic Runtime Complexity

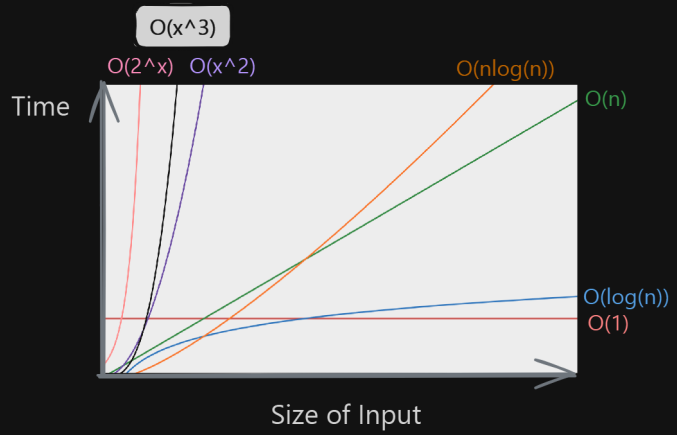
The following program is an **algorithm** (steps to solve a program) that finds the smallest integer in a list of integers.

```
nums = [-100, 400, 600, -200, 300, 0, -1, 500, -600, 800]
smallest = nums[0]
for i in nums:
    if i < smallest:
        smallest = i
print(f"{smallest} found in {len(nums)} integers")
```

Running...

How fast is the program?
How fast would it be if we used a million integers? A billion?

- The **runtime complexity** of an algorithm describes how the amount of time for an algorithm to run changes as the size of the input to the algorithm changes.
  - You can refer back to this slide once later examples make it more clear what this means
- We use Big-O notation to describe this behavior:
  - $O(1)$ → constant time
  - $O(\log(n))$ → logarithmic time
  - $O(n)$ → linear time
  - $O(n \log(n))$ → linearithmic time
  - $O(n^2)$ → quadratic time
  - $O(n^3)$ → cubic time
  - $O(2^n)$ → exponential time
- Example: Accessing an element from an array using an index takes the same amount of time regardless of how big or small the index is
  - $O(1)$ / constant time


O(x^3)
O(2^x)
O(x^2)
O(nlog(n))
O(n)
Time
O(log(n))
O(1)
Size of Input

More Data Structures

- Arrays have some disadvantages:
  - Cannot add or insert new elements
  - Cannot remove existing elements
  - Indices can be difficult to work with
- Other *data structures* are available to solve these issues
- There are many data structures, we will look at these:
  - `ArrayList`
  - `LinkedList`
  - `HashMap`

`ArrayList`

An `ArrayList` is a variable-length array:

- Add / insert elements
- Remove elements

Learn more here: docs.oracle.com

```java
import java.util.Arrays;
import java.util.ArrayList; // Import the `ArrayList` class!

public class Main {
    public static void main(String[] args) {
        // Specify the data type being stored inside angle brackets "<>".
        // Notice that the class `Integer` is used here instead of simply `int`.
        // This is a quirk of Java.
        ArrayList<Integer> nums = new ArrayList<>(Arrays.asList(2, 4, 6, 8));
        // Initialize with some data using `Arrays.asList`.

        // Get an element at a specific index in the array-list.
        int element = nums.get(2); // Gets the integer at index 2 --> 6
        System.out.println(element);

        // Add an element to the end of the array-list.
        nums.add(5);

        // Insert an element at index 3.
        nums.add(3, 6);

        // Remove an element at index 1.
        nums.remove(1);

        // Get the number of elements in the linked-list.
        int count = nums.size();
        System.out.println("Final contents: " + nums + ", size: " + count);
    }
}
```

Running...

- Additional notes:
  - Using an invalid index will cause an error
  - `get` runs on $O(1)$ / constant time
  - `add` and `remove` run on average $O(n)$ / linear time
    - Adding / removing is a comparatively slow operation

Learn more here: docs.python.org

```python
# Lists in Python are already variable-length arrays.
# No need to import anything.
nums = [2, 4, 6, 8]

# Get an element at a specific index.
e = nums[2]

# Add an element to the end of the list.
nums.append(5)

# Insert an element at index 3.
nums.insert(3, 6)

# Remove an element at index 1.
nums.pop(1)

print(f"Final contents: {nums}, size: {len(nums)}")
```

Running...

`LinkedList`

A `LinkedList` behaves like an `ArrayList`, but with certain advantages and disadvantages.
Python does not have linked-lists, so we will only look at Java.

Learn more here: docs.oracle.com

```java
import java.util.Arrays;
import java.util.LinkedList; // Import the `LinkedList` class!

public class Main {
    public static void main(String[] args) {
        // Specify the data type being stored inside angle brackets "<>".
        // Notice that the class `Integer` is used here instead of simply `int`.
        // This is a quirk of Java.
        LinkedList<Integer> nums = new LinkedList<>(Arrays.asList(2, 4, 6, 8));
        // Initialize with some data using `Arrays.asList`.

        // Get an element at a specific index in the linked-list.
        int element = nums.get(2); // Gets the integer at index 2 --> 6
        System.out.println(element);
        // Other methods.
        element = nums.getFirst();
        System.out.println(element);
        element = nums.getLast();
        System.out.println(element);

        // Add an element to the beginning and end of the linked-list.
        nums.addFirst(5);
        nums.addLast(5);

        // Insert an element at index 3.
        nums.add(3, 6);

        // Remove an element at index 1.
        nums.remove(1);

        // Remove the first and last elements.
        nums.removeFirst();
        nums.removeLast();

        // Get the number of elements in the linked-list.
        int count = nums.size();
        System.out.println("Final contents: " + nums + ", size: " + count);
    }
}
```

Running...

- Additional notes:
  - Using an invalid index will cause an error
  - `get` runs on $O(n)$ / linear time
    - This is **slower** than `ArrayList`!
  - `add` and `remove` run on average $O(n)$ / linear time
  - `addFirst`, `addLast`, `removeFirst`, `removeLast`, `getFirst`, and `getLast` run on $O(1)$ / constant time
    - This is **faster** than `ArrayList`!
- Prefer to use `LinkedList` over `ArrayList` when frequently adding / removing elements at the start / end of the data structure.

`HashMap`

- An array is a data structure that relates indices to values.
- A `HashMap` is a data structure that relates *keys* to *values*
  - We can get a values associated with a key
    - But not a key asociated with a value!
  - A key does not have to be an integer!
  - Keys are often useful as strings, but can be anything
  - Keys do not have to be in-order
  - A single key is associated with a single value
    - Stored as a *key-value pair*
  - **There are no duplicate keys**

Learn more here: docs.oracle.com

Let's say that a group of friends are playing the board game Monopoly, and they want to use a program to keep track of their cash balance in the game.

```java
import java.util.Arrays;
import java.util.HashMap; // Import the `HashMap` class!

public class Main {
    public static void main(String[] args) {
        // Key type: `String`
        // Value type: `Integer` / `int`
        HashMap<String, Integer> map = new HashMap<>();

        // Add / change a key-value pair.
        map.put("Alice", 2000);
        map.put("Bob", 1900);
        map.put("Charlie", 1950);

        map.put("Alice", 3000); // Changes the Alice's existing balance.

        // Check if the map has a specific key.
        System.out.println("Contains Alice: " + map.containsKey("Alice"));
        System.out.println("Contains Dave: " + map.containsKey("Dave"));

        // Get a value associated with a key.
        System.out.println("Alice's Balance: " + map.get("Alice"));
        System.out.println("Dave's Balance: " + map.get("Dave")); // `null` since Dave is not in the map.

        // Get the number of key-value pairs in the hashmap.
        int count = map.size();
        System.out.println("Final contents: " + map + ", size: " + count);
    }
}
```

Running...

- Additional notes:
  - `put`, `constansKey`, and `get` all run on average $O(1)$ / constant time
  - The hashmap is a very efficient data structure

In Python, hashmaps are called *dictionaries*.
Learn more here: docs.python.org

```python
players = dict()

# Or, create a dictionary with some initial data.
# players = {
#     "Alice": 100,
#     "Bob": 200,
#     "Charlie": 150
# }

# Add / change a key-value pair.
players["Alice"] = 2000
players["Bob"] = 1900
players["Charlie"] = 1950

players["Alice"] = 3000 # Changes the Alice's existing balance.

# Check if the players has a specific key.
hasAlice = "Alice" in players
hasDave = "Dave" in players
print("Contains Alice:", hasAlice)
print("Contains Dave:", hasDave)

# Get a value associated with a key.
print("Alice's Balance: ", players.get("Alice"))
print("Dave's Balance: ", players.get("Dave")) # `None` since Dave is not in the players.
# `None` is the equivalent of `null` in Python.

print(f"Final contents: {players}, key-value pairs: {len(players)}")
```

Running...

Worksheet

Click here to access the worksheet.

Click here to access the answer key.

Bonus worksheet (one question, if you're willing).

Bonus worksheet answer key.

August 1

Agenda

- Imports / Packages
- *Break*
- Exception Handling
- *No worksheet today*
- Projects Overview
- Saving progress from Binder
- Replit
- Additional Topics Overview

1. Packages
    1. Creating / Using Custom Packages
2. Break
3. Exceptions
4. Handling Exceptions
5. Worksheet
6. Projects
7. Saving progress from Binder
8. Replit
9. Additional Topics Overview

Packages

- Large programs can be split into separate files
  - Improves organization
  - **Modularize** code
    - Use the same classes / functions across multiple projects
    - Share your code with others
- Related classes are grouped into **packages**
  - Only **top-level public** classes in a package can be accessed outside the package
    - Only one top-level public class per file. Must match file name
    - This will be clarified later
  - Classes in other packages can be **imported**

```java
// We use an `import` statement to import classes from other packages.
import java.util.Scanner; // Importing the `Scanner` class from the `java.util` package.
import java.util.ArrayList; // Importing the `ArrayList` from the same package.

// This import all of the classes in a package.
import java.util.*;
// We do not recommend importing all classes.
// It can make your code less clear. You may forget which classes you are using.

public class Main { // Top-level public class.
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);

        // ...
        ArrayList<String> arr = new ArrayList<>();
        LinkedList<String> list = new LinkedList<>();
        // ...

        scan.close();
    }
}
```

Creating / Using Custom Packages

- In Binder, the `Main` class is already inside a package called `src.workbench`
  - The file `Main.java` is under the folder `workbench` inside the folder `src`
  - The name of a package should match the **directory** that the class is in
    - The directory in this case is `src/workbench/`

Example 1 - Java

demo/MyClass.java

```java
package demo;

public class MyClass { // `MyClass` is the top-level public
    public static void foo() {
        System.out.println("Foo");
    }
    public void bar() {
        System.out.println("Bar");
    }
}

class SecondClass { // `SecondClass` is top-level, but not p
    public static void baz() {
        System.out.println("Baz");
    }
}
```

Running...

demo/Main.java

```java
package demo;

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

        MyClass.foo();
        MyClass myObject = new MyClass();
        myObject.bar();

        // We can also access this class because it is in th
        SecondClass.baz();
    }
}
```

Running...

Example 2 - Java

package1/MyClass.java

```java
package package1; // Package #1

// `MyClass` is the top-level public class of
// `MyClass.java`
public class MyClass {
    public static void foo() {
        System.out.println("Foo");
    }
    public void bar() {
        System.out.println("Bar");
    }
    public void baz() {
        SecondClass.baz();
    }
}

// `SecondClass` is top-level, but not public.
// It cannot be accessed outside the `package1` package.
class SecondClass {
    public static void baz() {
        System.out.println("Baz");
    }
}
```

Running...

package2/Main.java

```java
package package2; // Package #2

import package1.*;

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

        MyClass.foo();
        MyClass myObject = new MyClass();
        myObject.bar();

        // We cannot access this class
        // because it is not top-level public.
        // SecondClass.baz();
    }
}
```

Running...

Example 3 - Python

We will not cover creating packages in Python, which follows its own rules.
But, here is an example of how to import packages in Python:

```
import cowsay # Import the cowsay package.

cowsay.cow("MOO") # Call a function from the package.
```

Running...

Break

Have a break!

Exceptions

- An **exception** is a type of event that disrupts the execution of a program
- **Exceptions** are often used to indicate that a problem occurred in the program

  For example:

  - Array index out of bounds
  - Divide by zero
  - Trying to use an object with value `` `null` ``
  - Incorrect usage of a function
  - etc.
- *By default*, when an **exception** occurs, it will stop a program immediately (*termination*)
  - We can **catch** exceptions to prevent the program from *terminating*
  - We can create / **throw** our own exceptions
- We will only be covering exceptions in Java, but note that Python also has exceptions

Example

Examples of common exceptions in Java.

```java
public class Main {
    public static void main(String[] args) throws Exception {
        // ArrayIndexOutOfBoundsException
        int[] nums = {1, 2, 3};
        System.out.println(nums[10]); // Index 10 is out-of-bounds.

        // ArithmeticException
        int n = 10 / 0; // Dividing by 0 is illegal.

        // NullPointerException
        String s = null;
        s.length(); // Don't use an object which is `null`.

        // Your own exception!
        throw new Exception("MY EXCEPTION");
    }
}
```

Running...

Handling Exceptions

- By *handling* an exception, you are instructing the program to do something else instead of *terminating*
- Put the code that can throw exception(s) inside a `` `try` `` block
- If an exception is occurs / *thrown*, execution will jump to the start of a `` `catch` `` block
- There is also a `` `finally` `` block, but we will not cover it

Example

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

        // Start of the `try` block!
        try {

            // ArrayIndexOutOfBoundsException
            int[] nums = {1, 2, 3};
            System.out.println(nums[10]); // Index 10 is out-of-bounds.

            // ArithmeticException
            int n = 10 / 0; // Dividing by 0 is illegal.

            // NullPointerException
            String s = null;
            s.length(); // Don't use an object which is `null`.

            // Your own exception!
            throw new Exception("MY EXCEPTION");

        // End of the `try` block. Start of the `catch` block!
        } catch (Exception e) {
            System.out.println("A problem occurred!");

            // You can also print the details of the exception.
            System.out.println(e.toString());
            System.out.println(e.getMessage());
        }
    }
}
```

Running...

Worksheet

No worksheet today!

Projects

- The next week of content will cover niche topics
- There will be no more worksheets
- Get into a group to brainstorm project ideas, or *solo it*

Saving progress from Binder

Binder does not save your data for you. Follow these steps to save your files.

Downloading

To download your files from Binder:

1. In the terminal (the box where your program outputs to), run the command `` `zip -r workbench.zip .` ``

   Don't forget the ".", it's a part of the command
2. In the file explorer (the list of files on the left), right-click / two-finger-click on the file `` `workbench.zip` ``. Then, click "Download..." to download the file.

Uploading

To upload your files to Binder:

1. Right-click in an empty space inside the file explorer. Then click "Upload..." and upload your `` `workbench.zip` `` file that you previously downloaded.
2. In the terminal, run the command `` `unzip -o workbench.zip` ``. Your files have been restored!
3. Delete the file `` `workbench.zip` ``.

Replit

Create a Replit account at replit.com.

Replit gives you 1200 minutes per month of usage. **Use it wisely.**
You also have a limit of 1 collaborator.

Additional Topics Overview

- Time & Random - Work with time and generate random numbers
- Basic ANSI Escape Sequences - Give your text-based programs colors, other *styles*, clear the screen, and move the cursor
- File I/O - Read and write to files to store data
- Turtle - Use Python to draw colorful spirals
- Lambdas / Anonymous Functions - Assign functions to variables and call variables as functions
- Multithreading - Run multiple functions at the same time

August 4

Agenda

- Time & Random
- *Break*
- Basic ANSI Escape Sequences

1. Time
    1. Pausing Execution
    2. Getting the Time
2. Basic ANSI Escape Sequences

Time

- Pausing execution
- Getting the time

Pausing Execution

The following example prints "Pausing...", waits for 3 second (3000 milliseconds), and then finally prints "Done".

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Pausing execution for 3 seconds...");
        pause(3000);
        System.out.println("Done.");
    }

    // Pauses execution for some amount of milliseconds.
    // 1 second ⟶ 1000 milliseconds
    private static void pause(int milliseconds) {
        try {
            Thread.sleep(milliseconds);
        } catch (InterruptedException e) {
            System.out.println("Thread was interrupted.");
        }
    }
}
```

Getting the Time

The following example prints the current year, month, day, and time.
In Binder, it may not match the time on your computer. This is supposed to happen.

```java
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;

public class Main {
    public static void main(String[] args) {
        LocalDateTime now = LocalDateTime.now();
        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
        String formattedTime = now.format(formatter);

        System.out.println("Current time: " + formattedTime);
    }
}
```

Running...

Basic ANSI Escape Sequences

ANSI escape sequences allow us to give our program output colors, styles, and perform other operations on the terminal.
Moving the cursor: en.wikipedia.org
Adding colors and styles: en.wikipedia.org

```java
public class Main {
    public static void main(String[] args) {
        String CSI = "\033[";

        // Clear the terminal / screen.
        System.out.print(CSI + "H" + CSI + "J");

        // Move the cursor to a position.
        int row = 5, column = 5;
        System.out.print(CSI + row + ";" + column + "H");

        // Set text to bold.
        System.out.println(CSI + "1m" + "This is some bold text.");

        // Set text to italic.
        System.out.println(CSI + "3m" + "This is some bold and italic text.");

        // Set text to a color.
        System.out.println(CSI + "95m" + "This is some bold, italic, and colorful text.");

        // Set background / highlight color.
        System.out.println(CSI + "44m" + "Colorful text 2.");

        // Clear all colors and styles.
        System.out.println(CSI + "0m");

        // Set text to a color (advanced).
        System.out.println(CSI + "38;2;25;210;75m" + "Colorful text 3."); // Text
        System.out.println(CSI + "48;2;20;225;225m" + "Colorful text 4."); // Background / Highlight

        System.out.println(CSI + "0m");
    }
}
```

August 5

Agenda

- File I/O
- *Break*

1. File I/O

File I/O

The following example demonstrates creating a file, writing to the file, adding to the file, and finally reading from the file.

→

```java
import java.io.FileWriter;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.FileReader;

public class Main {
    public static void main(String[] args) {
        String fileName = "example.txt";

        // Creating and writing to the file.
        try {
            FileWriter writer = new FileWriter(fileName);
            writer.write("This is the first line.\n");

            System.out.println("File created and initial content written.");
        } catch (IOException e) {
            System.out.println("Error writing to file: " + e.getMessage());
        }

        // Appending to the file.
        try {
            FileWriter writer = new FileWriter(fileName, true);
            writer.write("This is an appended line.\n");

            System.out.println("Additional content appended.");
        } catch (IOException e) {
            System.out.println("Error appending to file: " + e.getMessage());
        }

        // Reading from the file.
        try {
            BufferedReader reader = new BufferedReader(new FileReader(fileName));

            String line;
            System.out.println("Reading from file:");

            while ((line = reader.readLine()) != null) {
                System.out.println(line);
            }
        } catch (IOException e) {
            System.out.println("Error reading from file: " + e.getMessage());
        }
```

August 6

Agenda

- Turtle
- *Break*
- Lambdas

1. Turtle
2. Lambdas

Turtle

The turtle library (a kind of package) for Python can be used to draw colorful spirals.
This example **cannot be run in Binder**. Create a Replit account at replit.com to get started. Ask an instructor for help.

```
import turtle

# Setup screen.
screen = turtle.Screen()
screen.bgcolor("black")

# Create turtle.
spiral = turtle.Turtle()
spiral.speed(0)
spiral.width(2)

# Draw spiral with RGB colors.
for i in range(255):
    # Set RGB color manually
    r = i % 255
    g = (255 - i) % 255
    b = (i * 2) % 255
    spiral.pencolor(r / 255, g / 255, b / 255)

    spiral.forward(i)
    spiral.left(59)

spiral.hideturtle()
turtle.done()
```

Lambdas

Lambdas are also known as anonymous functions.
The following Java example shows how to use lambdas.

```java
public class Main {
    public static void main(String[] args) {
        MyRunnableLambda[] arrayOfLambdas = {
            (f, l, a) -> { System.out.println(l + ", " + f + " is " + a + " years old."); },
            (f, l, a) -> { System.out.println(f + " " + l + " is " + a + " years old."); },
            (f, l, a) -> { System.out.println("Somebody is " + a + " years old."); }
        };

        int lambdaToUse = 0;

        arrayOfLambdas[lambdaToUse].run("Jane", "Doe", 20);
    }
}

@FunctionalInterface
interface MyRunnableLambda {
    void run(String firstName, String lastName, int age);
}
```

Running...

August 7

Agenda

- Multithreading
- *Break*

1. Multithreading
2. Tomorrow

Multithreading

The following example creates two threads.
You can think of a thread as a task the program tries to do at the same time as other tasks.
The threads in this example each print numbers independently of each other.

```java
public class Main {
    public static void main(String[] args) {
        System.out.println("Starting threads...");

        MyThread t1 = new MyThread("Thread A");
        MyThread t2 = new MyThread("Thread B");

        t1.start(); // The Java JVM will start Thread A,
        t2.start(); // then proceed to start Thread B while Thread A is running.

        System.out.println("Threads started!");

        try {
            t1.join(); // Wait for Thread A to complete.
            t2.join(); // Wait for Thread B too.
        } catch (InterruptedException e) {
            System.out.println(e.getMessage());
        }
    }
}

class MyThread extends Thread {
    private String name;

    public MyThread(String name) {
        this.name = name;
    }

    public void run() {
        for (int i = 1; i < 10; ++i) {
            System.out.println(name + " - " + i);
        }
    }
}
```

Tomorrow

Share your projects with us!
We will be presenting them tomorrow.
If you don’t want to share or aren’t done yet, that’s okay too.

August 8

Agenda

- Project Presentations
- *Break*
- Project Presentations

Thank You!

Thank you for attending ICMC Summer Coding! 😊

Sorry!

This resource is currently unavailable.