Spring 2024 CS 284 Midterm Review

1.
Java Basics

Primitive Data Types

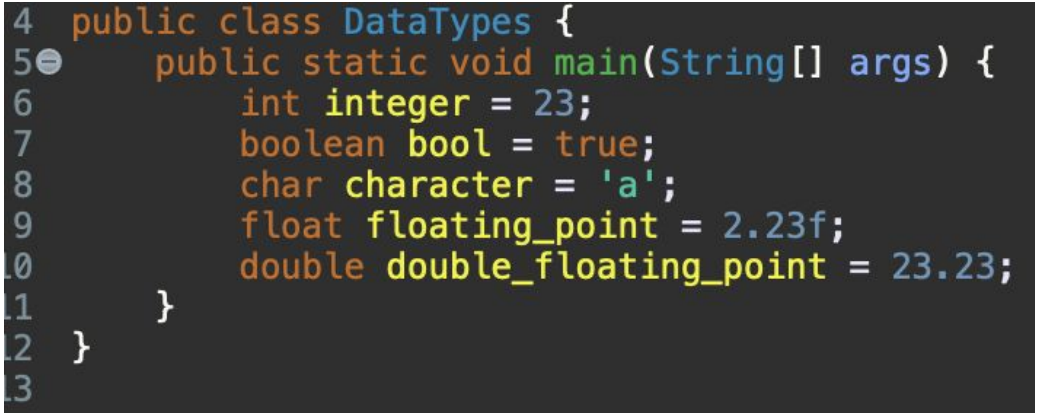
int → (1, 2, 3, 4, 5, etc.)

boolean → (true or false)

char → (‘a’, ‘L’, ‘2’, ‘|’, etc.)

float → (2.23f) ← note the “f”!

double → (45.45, 2.3333, etc.)



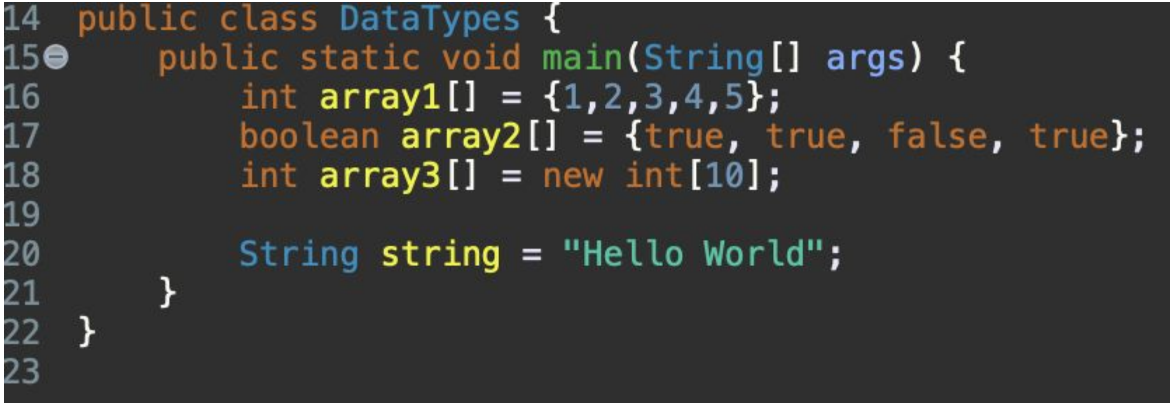
```
public class DataTypes {
    public static void main(String[] args) {
        int integer = 23;
        boolean bool = true;
        char character = 'a';
        float floating_point = 2.23f;
        double double_floating_point = 23.23;
    }
}
```

Non-Primitive Data Types

Strings → (“Hello”, “I love CS284”)

Arrays → ([1,2,3,4,5,], [‘a’,’b’,’c’], etc.)

Classes → (Scanner, ArrayList, etc.)



```java
public class DataTypes {
    public static void main(String[] args) {
        int array1[] = {1,2,3,4,5};
        boolean array2[] = {true, true, false, true};
        int array3[] = new int[10];

        String string = "Hello World";
    }
}

```

Access Modifiers

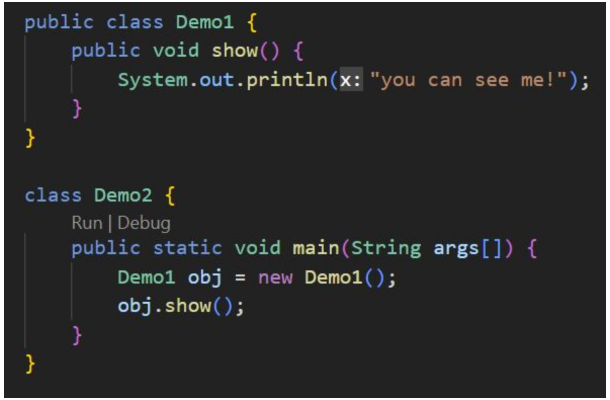
Public

Data can be accessed by any class

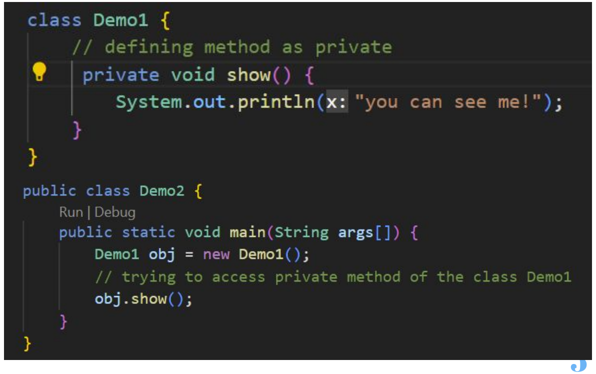
Private

Data can ONLY be accessed from within the class it is defined in

What is printed?



```
public class Demo1 {
    public void show() {
        System.out.println(x: "you can see me!");
    }
}

class Demo2 {
    Run | Debug
    public static void main(String args[]) {
        Demo1 obj = new Demo1();
        obj.show();
    }
}
```



```
class Demo1 {
    // defining method as private
    private void show() {
        System.out.println(x: "you can see me!");
    }
}
public class Demo2 {
    Run | Debug
    public static void main(String args[]) {
        Demo1 obj = new Demo1();
        // trying to access private method of the class Demo1
        obj.show();
    }
}
```

Conditionals And Loops

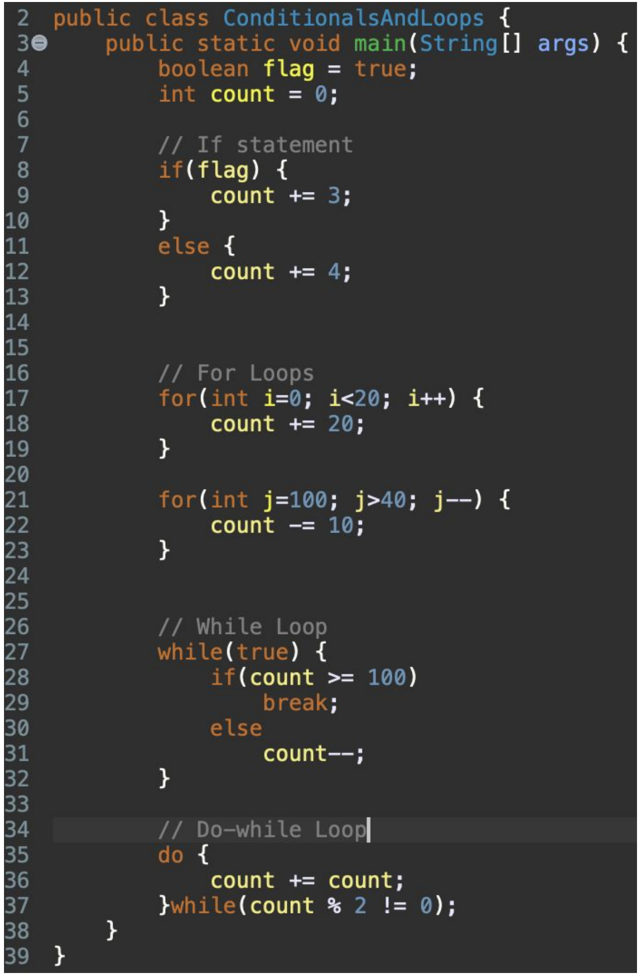
&& - and

|| - or

! - not

Types of Loops

- for
- while
- do-while



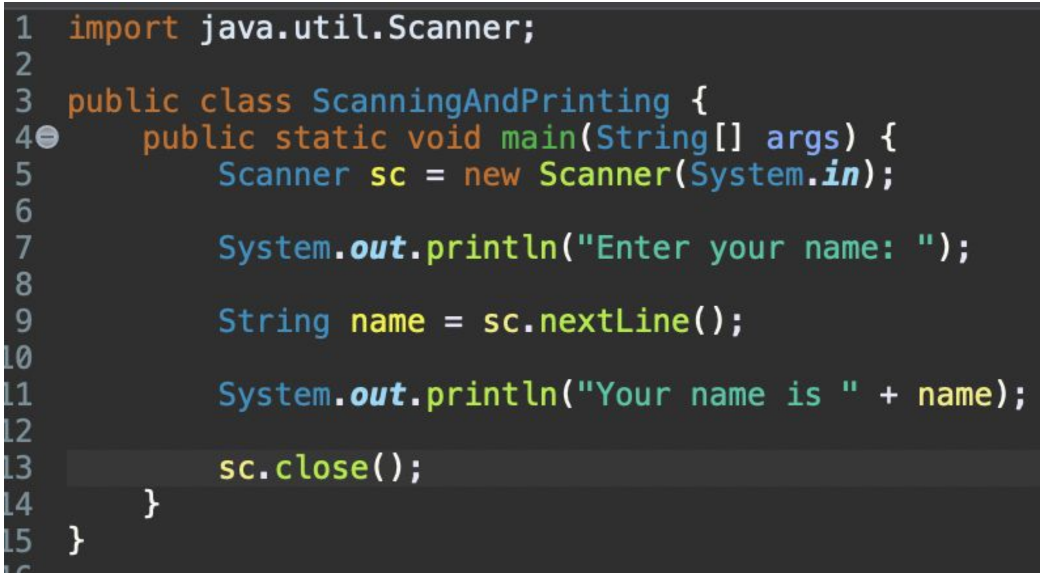
```java
public class ConditionalsAndLoops {
    public static void main(String[] args) {
        boolean flag = true;
        int count = 0;

        // If statement
        if(flag) {
            count += 3;
        }
        else {
            count += 4;
        }


        // For Loops
        for(int i=0; i<20; i++) {
            count += 20;
        }

        for(int j=100; j>40; j--) {
            count -= 10;
        }


        // While Loop
        while(true) {
            if(count >= 100)
                break;
            else
                count--;
        }

        // Do-while Loop
        do {
            count += count;
        }while(count % 2 != 0);
    }
}
```

Scanning and Printing

Scanner - Reads input from console/file

- System.in → console
- new File(filename) → file

System.out.println() - Prints/writes to console



```java
import java.util.Scanner;

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

        System.out.println("Enter your name: ");

        String name = sc.nextLine();

        System.out.println("Your name is " + name);

        sc.close();
    }
}
```

2.
Classes

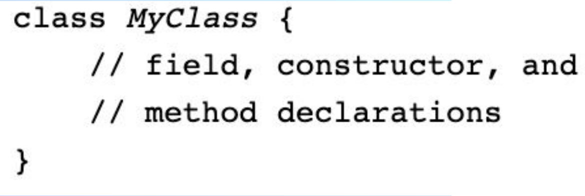
Polymorphism

- Backbone of *Object Oriented Programming (OOP)*
- Different instances of the same objects with the same attributes
- Method overriding: methods in subclasses with same name, parameters type, and return type
- Allows for multiple implementations of the same interfaces!
- Helps remove duplicate code by using the base class rather than checking the `instanceof` each object

Objects & Inheritance

keyword: extends

code reusability avoids redundancy!



```
class MyClass {
    // field, constructor, and
    // method declarations
}
```

Is-a

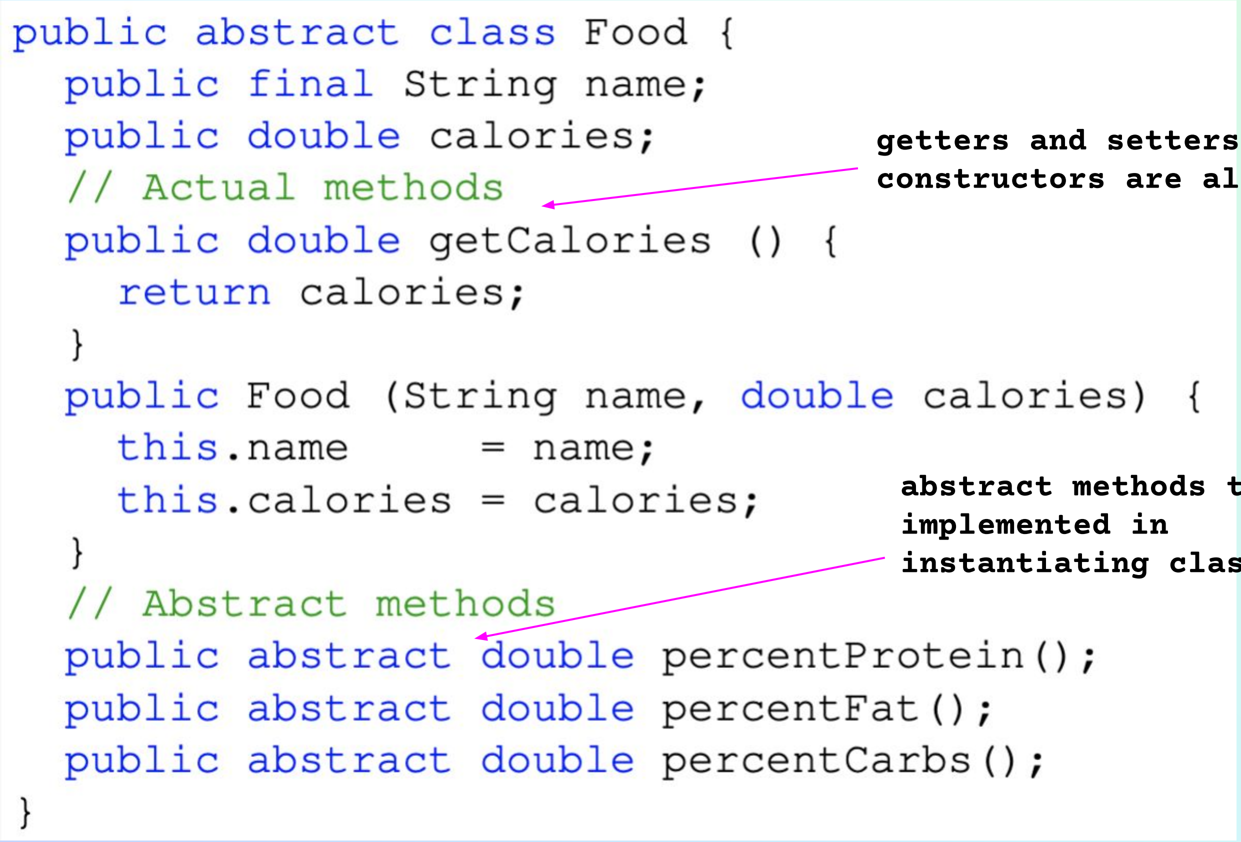
```
class Animal {
      int numOfLegs
      String color
}
class Pet extends Animal
{
      String name
}
class Toucan extends
Animal {
      //numOfLegs = 2
      String habitat
}
class Dog extends Pet{
      bool needsWalk
      //name = Rocco
}
```

Shape

```
      //numCorners
circle extends shape
      //numCorners = 0
rectangle extends shape
      //numCorners = 4
square extends rectangle
```

all squares are rectangles and all rectangles are shapes, but not all shapes are rectangles and not all rectangles are squares

multiple implementations of the same classes



```
public abstract class Food {
  public final String name;
  public double calories;
  // Actual methods
  public double getCalories () {
    return calories;
  }
  public Food (String name, double calories) {
    this.name     = name;
    this.calories = calories;
  }
  // Abstract methods
  public abstract double percentProtein();
  public abstract double percentFat();
  public abstract double percentCarbs();
}
```

getters and setters and constructors are allowed

abstract methods to be implemented in instantiating class

Interfaces

*is-a

*blueprint

*no constructor

*can implement more than one

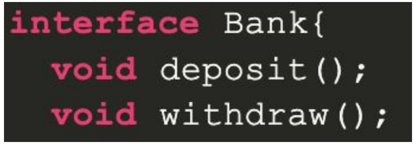
***keyword: implements**

*Hold many method declarations that are empty internally

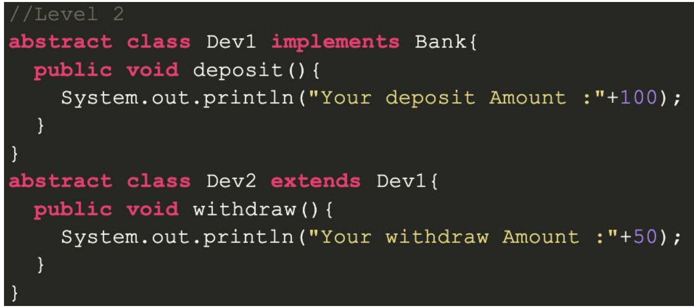
*Cannot be instantiated

*The implementing classes must remain consistent with declared types & provide a method for each declaration

*Abstract class can have instance variables

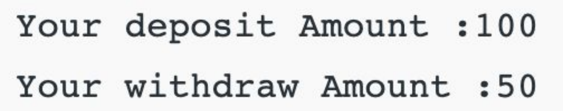


```
interface Bank{
  void deposit();
  void withdraw();
```



```
//Level 2
abstract class Dev1 implements Bank{
  public void deposit(){
    System.out.println("Your deposit Amount :"+100);
  }
}
abstract class Dev2 extends Dev1{
  public void withdraw(){
    System.out.println("Your withdraw Amount :"+50);
  }
}
```

```
class banking {
     psv main(String[] args ){
     Dev2 d = new Dev2();
     d.deposit();
     d.withdraw();
}
```



```
Your deposit Amount :100
Your withdraw Amount :50
```

Interfaces

*is-a

*blueprint

*no constructor

*can implement more than one (_,_)

***keyword: implements**

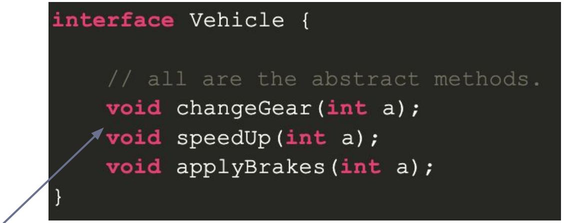
*Hold many method declarations that are empty internally

***Cannot be instantiated**

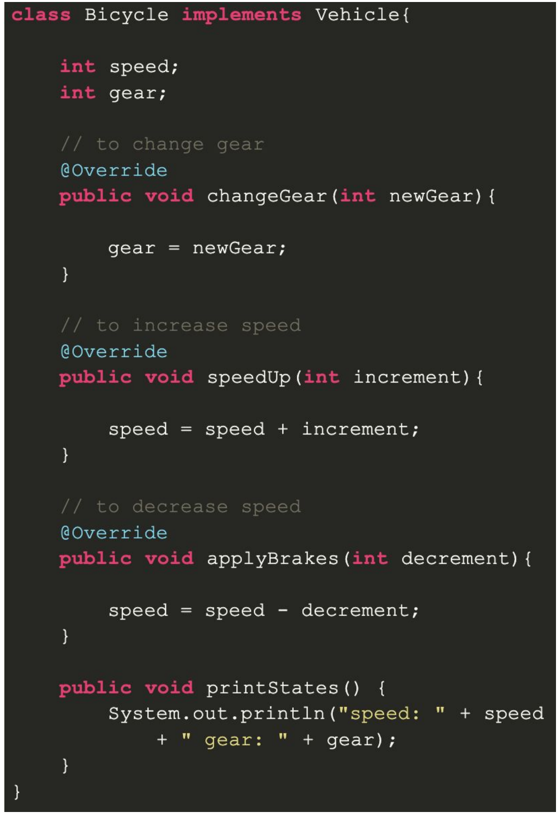
*The implementing classes must remain consistent with declared types & provide a method for each declaration

*Abstract class can have instance variables

*implementing classes add data, methods, instance variables, constructors



```java
interface Vehicle {

   // all are the abstract methods.
   void changeGear(int a);
   void speedUp(int a);
   void applyBrakes(int a);
}
```



```java
class Bicycle implements Vehicle{

    int speed;
    int gear;

    // to change gear
    @Override
    public void changeGear(int newGear){

        gear = newGear;
    }

    // to increase speed
    @Override
    public void speedUp(int increment){

        speed = speed + increment;
    }

    // to decrease speed
    @Override
    public void applyBrakes(int decrement){

        speed = speed - decrement;
    }

    public void printStates() {
        System.out.println("speed: " + speed
            + " gear: " + gear);
    }
}
```

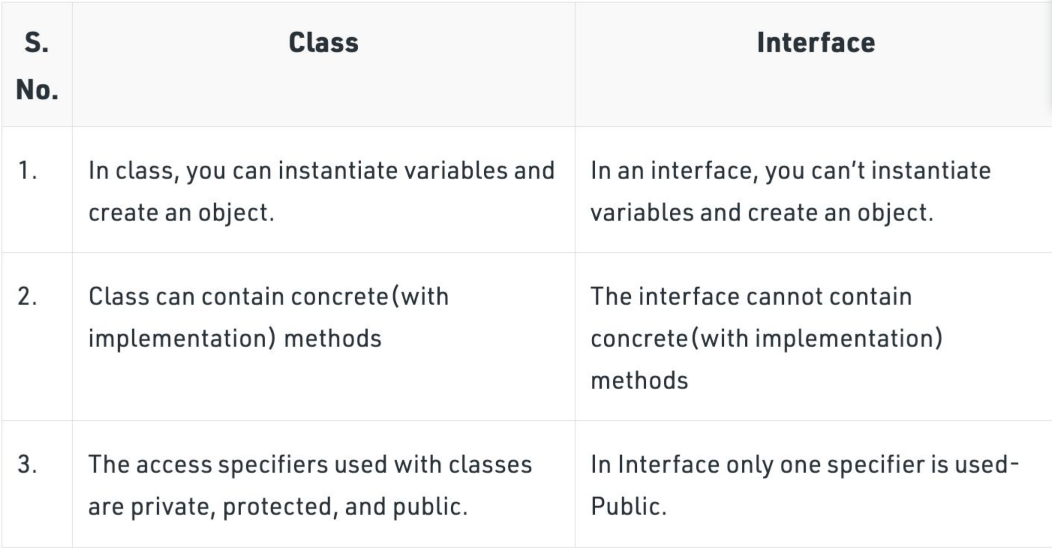
Comparison

Abstract Classes:

*cannot be instantiated

*can declare abstract methods

*the class will declare the concrete



S. No.	Class	Interface
1.	In class, you can instantiate variables and create an object.	In an interface, you can’t instantiate variables and create an object.
2.	Class can contain concrete(with implementation) methods	The interface cannot contain concrete(with implementation) methods
3.	The access specifiers used with classes are private, protected, and public.	In Interface only one specifier is used- Public.

Overloading VS Overriding

Abstract Classes:

*cannot be instantiated

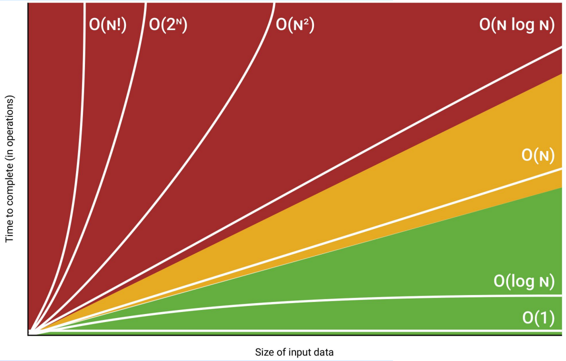
*can declare abstract methods

*the class will declare the concrete

	Overloading	**Overriding**
Method Name	Same! The method names between two methods will be the same.	Same! The method names between two methods will be the same.
Inherited?	No! Overloading is not related to inheritance.	Yes! Overriding is when the child class's method has the same signature as the parent class's method!
Same Parameters?	Never	Always! he method names between two methods will be the same.
:D	:D I love CS284! 	:D I ALSO love CS284!

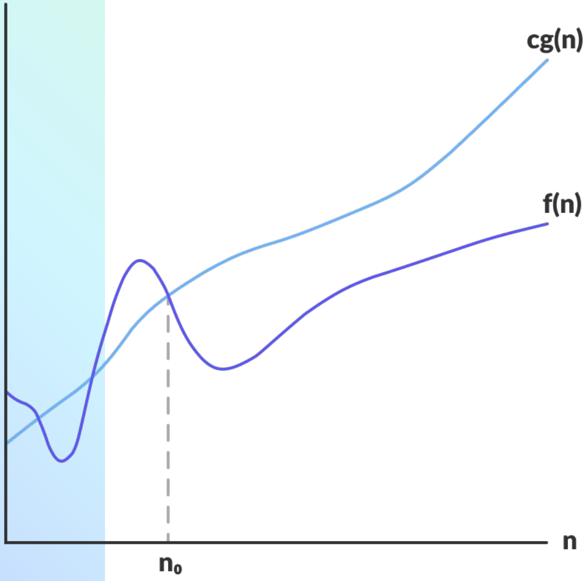
3.
Runtime Analysis

What is Big-O Notation?



- Used to measure the efficiency of an algorithm
- Think of it as the "worst case scenario" – what will the run time be of the code if it is given the worst possible input?
- Remember, this is different from the number of lines code will execute
- O(g(n)) is the set of all functions that could be an upper bound for g(n)
  - For Big-O, we ignore coefficients and only consider the fastest-growing term.
  - You do NOT do O(6n^3 + n^2)
  - Instead, do O(n^3)

More Big O

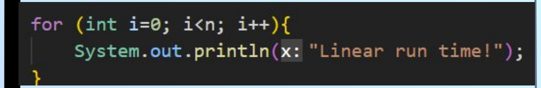


f(n) = O(g(n))

- You might wonder how O(n^3) could represent an upper bound for something like f(n) = 6n^3 + n^2
- For our function **g(n) = n^3**, there is an **n_0** (value of n) and a coefficient **c** for which

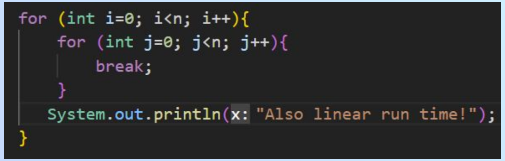
cn^3 >= 6n^3 + n^2 for **n >= n_0**.

Linear - O(n)



```
for (int i=0; i<n; i++){
    System.out.println(x: "Linear run time!");
}
```

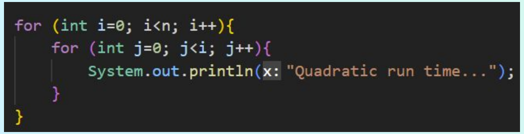
# of prints = n, O(n)



```
for (int i=0; i<n; i++){
    for (int j=0; j<n; j++){
        break;
    }
   System.out.println(x: "Also linear run time!");
}
```

# of prints = n, O(n)

Quadratic - O(n^2)



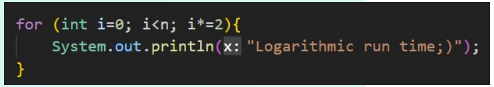
```
for (int i=0; i<n; i++){
    for (int j=0; j<i; j++){
        System.out.println(x: "Quadratic run time...");
    }
}
```

# of prints = n*((n+1)/2)), O(n^2)

The inner loop runs (n+1)/2 times, because of this known (very common) summation:

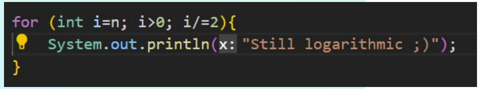
1 + 2 + 3 + … + n = n*(n+1)/2

Logarithmic - O(log(n))



```
for (int i=0; i<n; i*=2){
    System.out.println(x: "Logarithmic run time;)");
}
```

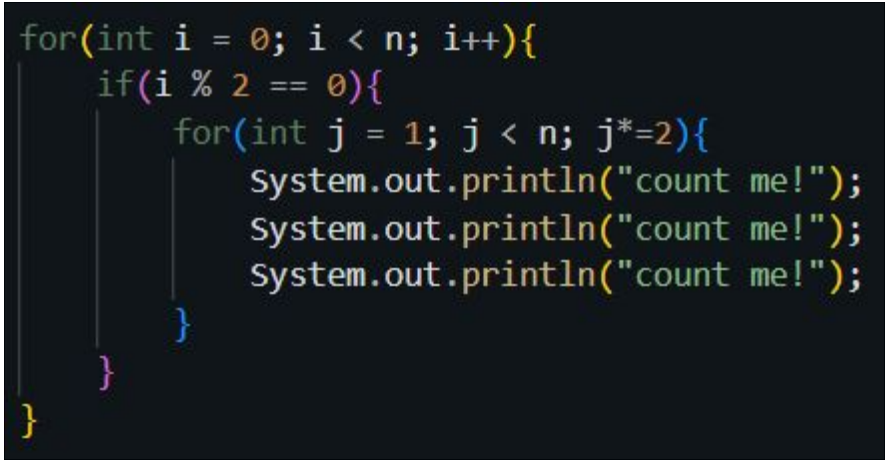
# of prints = log_2(n), O(log(n))



```
for (int i=n; i>0; i/=2){
    System.out.println(x: "Still logarithmic ;)");
}
```

# of prints = log_2(n), O(log(n))

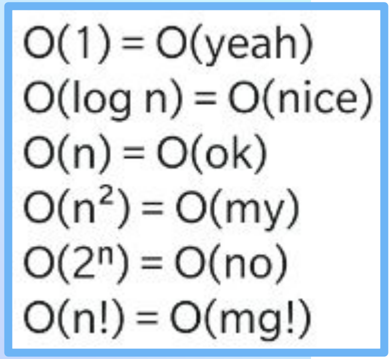
Analyzing a code snippet! (Example)



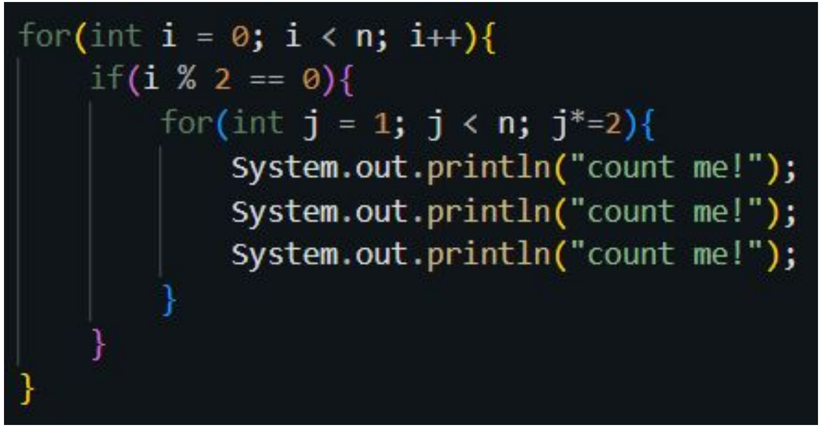
```
for(int i = 0; i < n; i++){
    if(i % 2 == 0){
        for(int j = 1; j < n; j*=2){
            System.out.println("count me!");
            System.out.println("count me!");
            System.out.println("count me!");
        }
    }
}
```

What is the number of times it prints and complexity using Big-O notation?

Solution
+
Explanation



O(1) = O(yeah)
O(log n) = O(nice)
O(n) = O(ok)
O(n^2) = O(my)
O(2^n) = O(no)
O(n!) = O(mg!)



```
for(int i = 0; i < n; i++){
    if(i % 2 == 0){
        for(int j = 1; j < n; j*=2){
            System.out.println("count me!");
            System.out.println("count me!");
            System.out.println("count me!");
        }
    }
}
```

- The outer loop runs n times
- The if statement makes it so that the inner loop is called (n+1)/2 times (we include 0)
- The inner loop runs log_2(n) times, since we multiply by 2 (cutting iterations in half each time)
- When the inner loop runs it prints 3 times
- # of times it prints: 3*((n+1)/2)*log_2(n)
- O(n*log(n))

How can I prepare more?

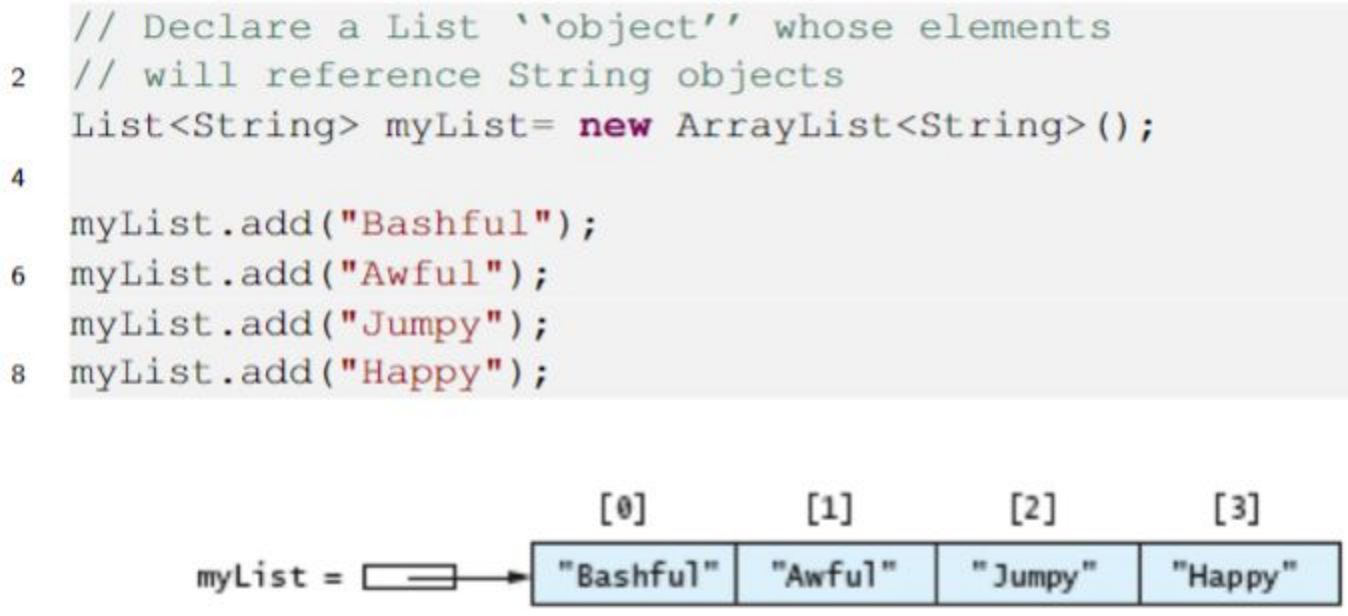
- Do the booklet exercises - they will be very helpful!
- When you are confused about how a function behaves, play around with it in your IDE with different inputs.
- If you are unsure during the test, also plug in inputs to see a pattern.

4.
Lists

literally me →

ArrayLists

- ArrayLists are dynamically sized
- Constant time access to elements
    - Memory is all stuck together!
- Removal is linear
    - Shifting elements!
- Insertion is linear
    - ALSO shifting elements!



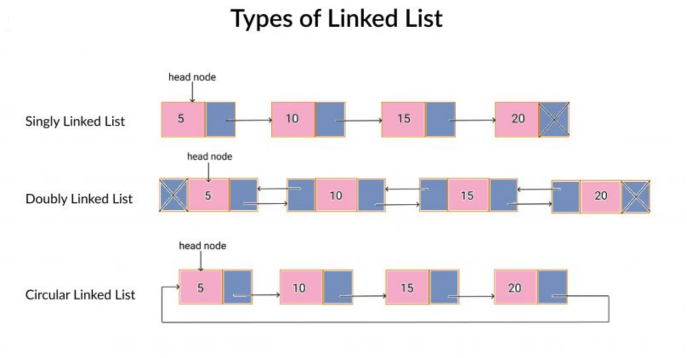
```
// Declare a List ``object'' whose elements
// will reference String objects
List<String> myList= new ArrayList<String>();

myList.add("Bashful");
myList.add("Awful");
myList.add("Jumpy");
myList.add("Happy");
```

```
ArrayList<Integer> intList = new ArrayList<Integer>();
ArrayList<Boolean> boolList = new ArrayList<Boolean>();
```

Linked Lists

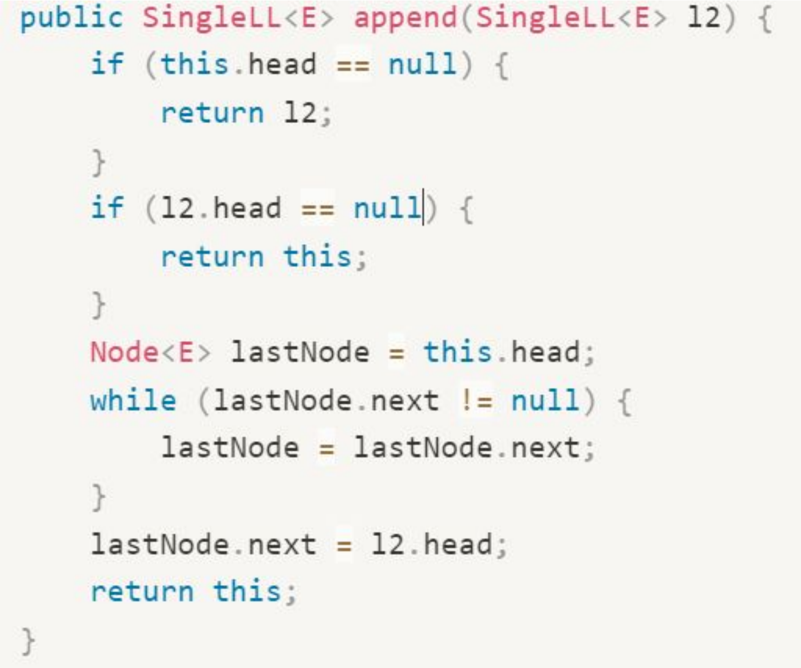
- A list made up of nodes
  - SLL: data, next
  - DLL: data, next previous

- Linked lists may grow and shrink
- Linear time access
- Linear time insertion and removal (except if previous element supplied, then constant)
  - Is constant if adding to head or tail!



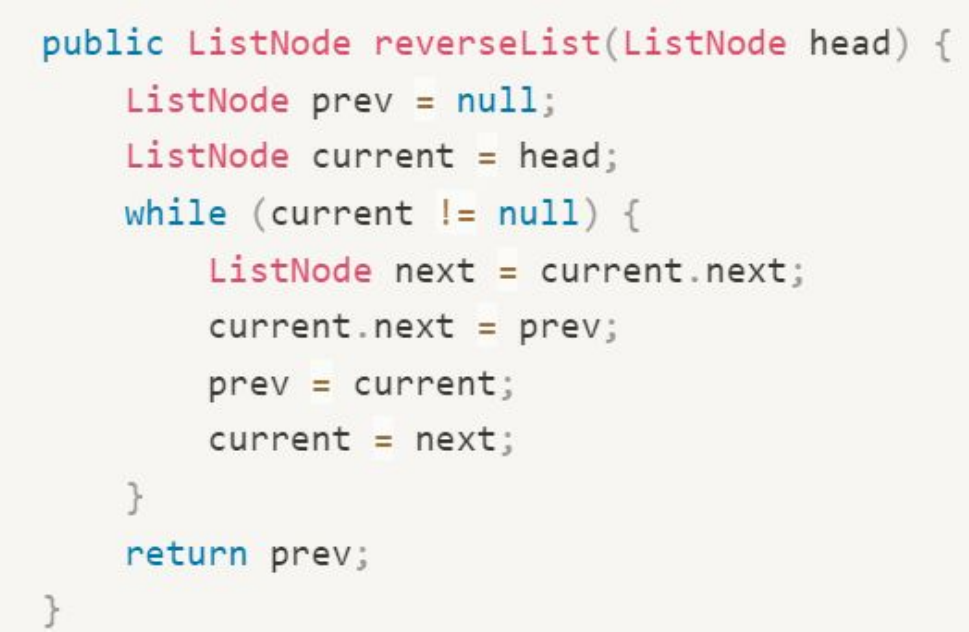
SingleLL<E> append(SingleLL<E> l2) that appends the two lists.

Eg. Given [1,2,3] and [4,5] returns [1,2,3,4,5].



```
public SingleLL<E> append(SingleLL<E> l2) {
    if (this.head == null) {
        return l2;
    }
    if (l2.head == null) {
        return this;
    }
    Node<E> lastNode = this.head;
    while (lastNode.next != null) {
        lastNode = lastNode.next;
    }
    lastNode.next = l2.head;
    return this;
}
```

Given the head of a singly linked list, reverse the list, and return the reversed list.



```
public ListNode reverseList(ListNode head) {
    ListNode prev = null;
    ListNode current = head;
    while (current != null) {
        ListNode next = current.next;
        current.next = prev;
        prev = current;
        current = next;
    }
    return prev;
}
```

“ 5.

Stacks

Stacks

- Stack's storage policy is **L**ast **I**n, **F**irst **O**ut or LIFO
  - The last thing put in will be on the top of the stack
- Push: Puts an element into the stack
- Peek: Looks at the top of the stack
- Pop: Looks and removes the element at the top of the stack

Stack Example

- Push: Puts an element into the stack
- Peek: Looks at the top of the stack
- Pop: Looks and removes the element at the top of the stack

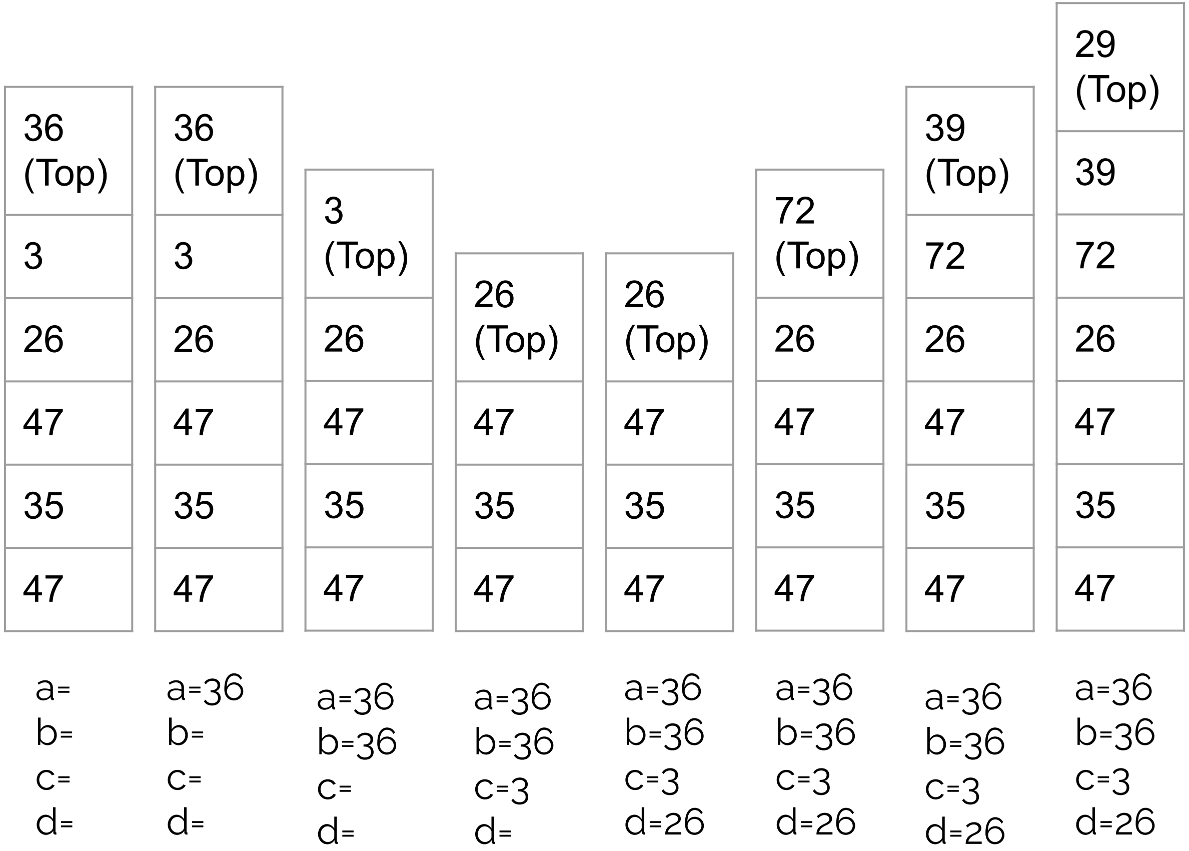
numStacks (name)
36 (Top)
3
26
47
35
47

```
a=numStacks.peek();
b=numStacks.pop();
c=numStacks.pop();
d=numStacks.peek();
numStacks.push(a+b);
numStacks.push(b+c);
numStacks.push(c+d);
```

What does the final stack look like?

Lines for reference!

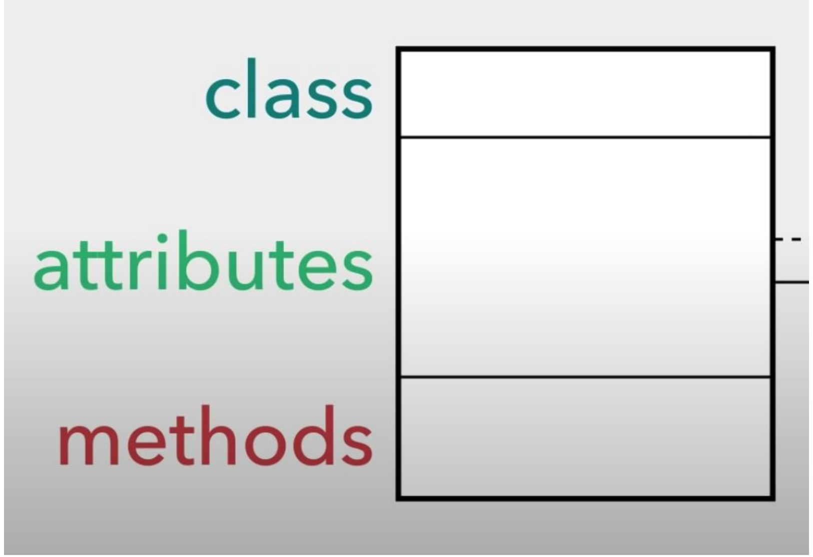
```
a=numStacks.peek();
b=numStacks.pop();
c=numStacks.pop();
d=numStacks.peek();
numStacks.push(a+b);
numStacks.push(b+c);
numStacks.push(c+d);
```



							29 (Top)
36 (Top)	36 (Top)					39 (Top)	39
3	3	3 (Top)			72 (Top)	72	72
26	26	26	26 (Top)	26 (Top)	26	26	26
47	47	47	47	47	47	47	47
35	35	35	35	35	35	35	35
47	47	47	47	47	47	47	47
a= b= c= d=	a=36 b= c= d=	a=36 b=36 c= d=	a=36 b=36 c=3 d=	a=36 b=36 c=3 d=26	a=36 b=36 c=3 d=26	a=36 b=36 c=3 d=26	a=36 b=36 c=3 d=26

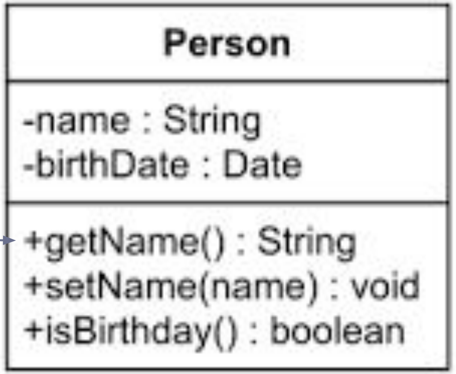
6.
UML Diagrams

A class UML Diagrams describes attributes and methods



“-” → private

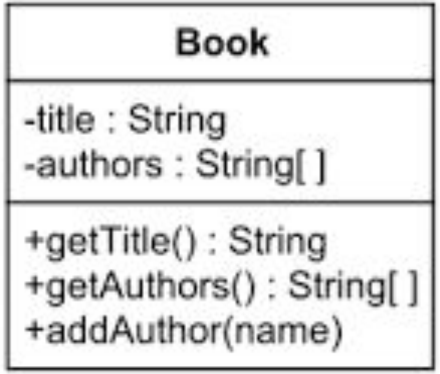
“+” → public



Person

-name : String
-birthDate : Date

+getName() : String
+setName(name) : void
+isBirthday() : boolean



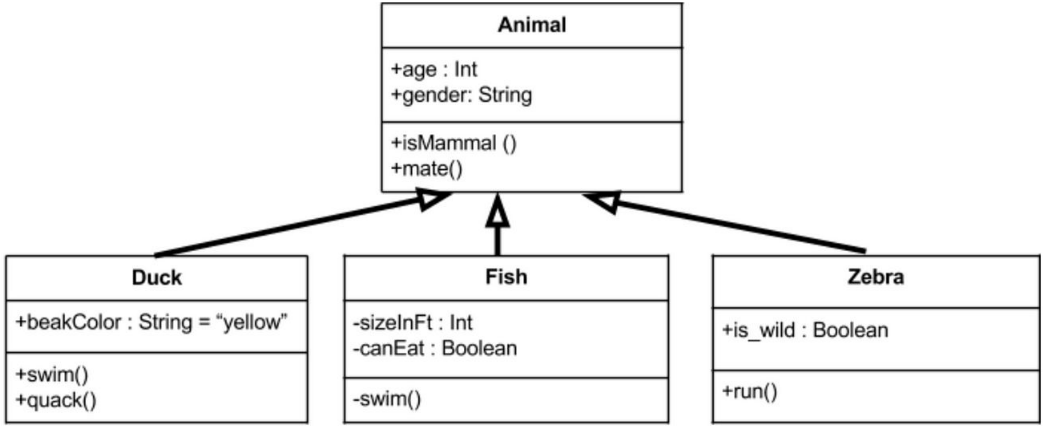
Book

-title : String
-authors : String[ ]

+getTitle() : String
+getAuthors() : String[ ]
+addAuthor(name)

They also describe connections between classes

Parent Class



Child Classes

7. Exceptions

Checked vs. Unchecked Exceptions

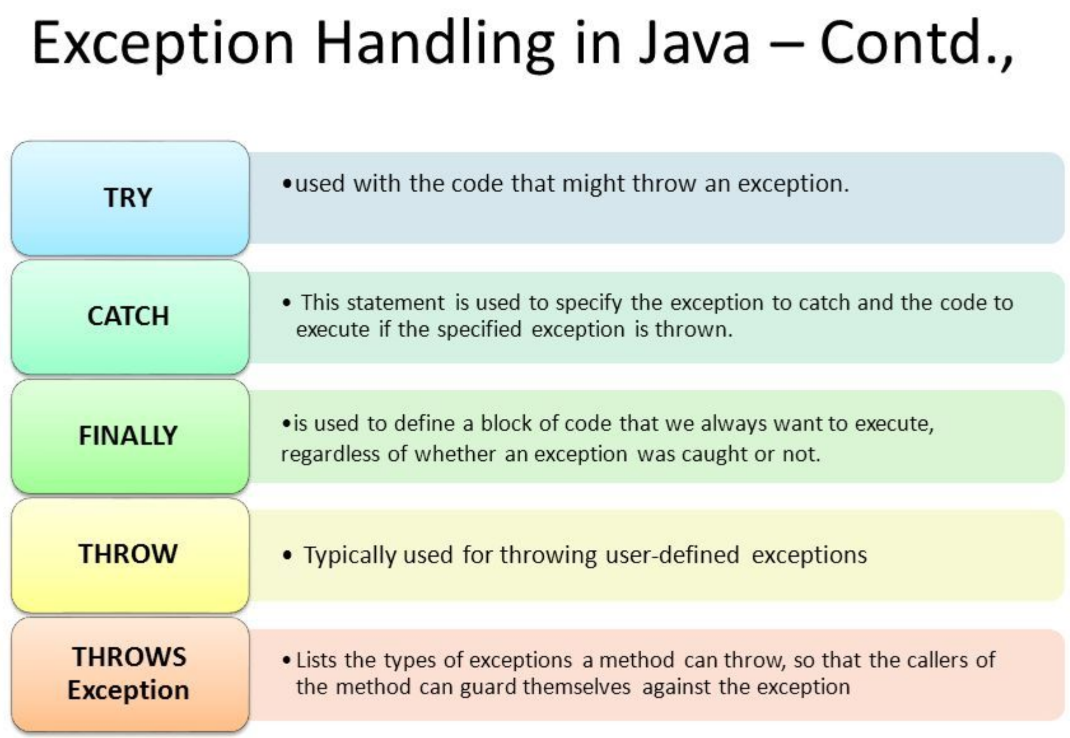
Checked Exceptions

- Not due to programmer error
- Mostly due to input/output
- Examples: IOException, FileNotFoundException

Unchecked Exceptions

- Due to programmer error
- Examples: NullPointerException, ArrayOutOfBoundsException

How to handle exceptions



Exception Handling in Java – Contd.,

TRY	•used with the code that might throw an exception.
CATCH	• This statement is used to specify the exception to catch and the code to execute if the specified exception is thrown.
FINALLY	•is used to define a block of code that we always want to execute, regardless of whether an exception was caught or not.
THROW	• Typically used for throwing user-defined exceptions
THROWS Exception	•Lists the types of exceptions a method can throw, so that the callers of the method can guard themselves against the exception

Helpful resources!

- https://medium.com/@amejiarosario/8-time-complexity-examples-that-every-programmer-should-know-171bd21e5ba
  - Don't worry so much about O(2^n) or O(n!)
- https://swapnil-mishra.github.io/Time-Complexity/
  - Stick your code in, and it'll calculate the time complexity!

How to study!

- Review code from class, labs, homework
    - Skim over what you know
    - Extra practice problems on Canvas!
- Concepts + Syntax
    - Draw out the data structures while practicing
    - Check out geeksforgeeks