01 - misc (unfinished)

Saturday, May 3, 2025 1:35 PM

Class vs Object

- defined w/ class or enum
- instances of classes (created w/ new)

all classes are types that can be used to creat objects

but not all classes are objects

enums are classes w/ fixed set of instances (constants)

String s = "hi"
class ref var
obj

Overriden vs overloaded

public, private, protected, static

Errors

type error

reference error

MIN_VALUE = -2^{31} = -2147483648

MAX_VALUE = $2^{31}-1$ = 2147483647

MIN_VALUE - 1 = 2147483647 (MAX_VALUE)

MAX_VALUE + 1 = -2147483648 (MIN_VALUE)

Primitive Types

byte

int

char

bigDecimal

double

float - has suffix -f } both decimal numbers

boolean - true or false, cant be null, false if uninitialized

Wrapper Classes

Byte

-

Boolean - null if uninitialized, use if need generics (eg. List<Boolean>)

collections

02 - loops

Thursday, May 1, 2025 9:35 AM

for loop

initiate ↓ — bool expression used before loop ↓

```
for (init; guard; update) {   ← statement ran after
    ~~~~~~
}
```

```
for (int i = 0; i < 10; i++) {
    system.out.println(i);
}
```

for each loop
access ea iterator (ex. array)

```
for ( SomeType someIdentifier : iteratorOfSomeType ) {
    ~~~~
}
```

```
int[] xs = {2, 4, 6, 1, 3};
for (int i = 0; i < xs.length; i++) {   ← i is the index of the value
    system.out.println(xs[i]);
}
for (int val : xs) {   ← array of ints so must use type int for val
    system.out.println(val);   val - val inside array @ ea location, not index
}
```

- continue - skip current iteration
- break - exit loop entirely
- return - exit method

while loop

```
while (thisIsStillTrue) {
    ~~~~
}
```

do while loop

```
do {
    ~~~~   ← will always run once bc didn't test the until after
} while (thisIsStillTrue);
```

03 - arrays & lists

Thursday, May 1, 2025 9:59 AM

arrays - ordered sequence of vals, can only hold 1 type

→ import java.util.Arrays

length cannot change sizes, to create longer array must copy over

```
int[] xs;
double[] sample;
String[] names;
```

instantiation – if you know specific vals, only @ declaration time ↓

```
int[][] yss = {{1,2,3}, {5,6,7}, {4}};
```

if no vals initially, specify dimension & get default vals

```
xs = new int[5];          // xs holds {0,0,0,0,0}
ys[][] = new int[2][3];   // ys holds {{0,0,0}, {0,0,0}}
bs[] = new boolean[3];    // bs holds {false, false, false}
```

lists - ordered collections that store elements sequentially & allow duplicates

→ import java.util.List

dynamic sizing, can change

create list

```
List<type> name = new ArrayList<>();
List<Integer> nums = new LinkedList<>();
List<String> fruits = List.of("Apples", "Banana");
```

add elems

```
names.add("Bob");          → append to end
names.add(0, "Alice")      → insert @ index 0
names.addAll(List.of("Charlie", "Dave"));   → add multiple
```

access elements

```
String s first = names.get(0);   → return elem @ index 0
int index = names.indexOf("Alice");   → find 1st occurrence
boolean exists = names.contains("Bob");   → check exists
```

remove elements

```
names.remove(0);          → remove elem @ index 0
names.remove("Alice");    → removes 1st occurence & returns true if removed
names.clear();            → remove all elements
```

loops & arrays

```
int[] xs = new int[15];   ← array w/ 15 zeros
for (int i=0; i<xs.length; i++){   loop until i is greater than len of array
    xs[i] = i*i;   updated w/ square of ea index
}
System.out.println("Some square nums: ");
for(int i=0; i<xs.length; i++){
    System.out.print(xs[i]+" ");   prints val (squared index) @ ea index
}
```

iterating over a loop

```
for(int i=0; i<names.size(); i++){
    System.out.println(names.get(i));
}
for (String name : names){
    System.out.println(name);
}
names.forEach(System.out::println);   → java 8+
```

04 - methods

Thursday, May 1, 2025 10:20 AM

methods

include to declare static method

```
modifier  static  return-type  method-name ( parameter list ){
}
```

```
public int max (int x, int y){
}
```

Static class
- can have both static & nonstatic vars
- can use obj of class to access a static thing in the class (obeying its visibility)
  - → objName . StaticMember but usually use ClassName.StaticMember

Static method

*cannot use this & super

can be called all by itself (no obj of enclosing needed)

```
public class Test {
    public static void math (String[] args) {
        int x = 10;
        String s = "hello";
        int res = doTheTask (x, s);
        System.out.println ("result: " + res);
    }
    public static int doTheTask (int a, String t) {
        int max = 0;
        int i;
        for (i = 0; i < a; i++) {
            System.out.println (t);
            if (i > max) {
                break;
            }
        }
        return i;
    }
}
public class OtherTest {
    public static void main (String[] args) {
        Test.main ()
        Test.doTheTask (5, "yo");
    }
}
```

class / method / creates Test obj

class / method name

doTheTask method is called twice (Test.main & OtherTest's main method calls it directly)

Features of static methods

- static method is part of a class rather than an instance of the class
- every instance of a class has access to the method
- have access to class variables (static variables) w/out using class's obj (instance)
- only static data may be accessed by a static method – unable to access data that is not static (instance variables)

*both static & nonstatic methods can access static methods

↳ static methods cant use nonstatic method

instance method

- req. obj of the class
- access all attributes of a class
- methods accessed via obj reference
- Syntax: Obj . methodName ()

static methods

- doesn't req. obj of a class
- access only static attribute of a class
- method accessed by class name
- Syntax: className.methodName ()

main method

```
public static void main (String[] args)
```

access advice

no objects exist yet & method belongs to class itself, not an instance

main() method doesn't return a val

how command line args are passed

↳ run:

java MyApp Hello World

args array recieves:

args[0] = "Hello"

args[1] = "World"

non-static method

```
public class Square {
    public int side;
    public Square (int side){
        this.side = side;
    }
    public int perimeter(){
        return (side * 4);
    }
}
```

to call perimeter method, must have Square obj (ex. sq.perimeter())

```
Square sq = new Square (5);
System.out.println ("square's perimeter: " + sq.perimeter() );
```

← instance of class generated

static variables

class members – indicate 1 shared val for entire class

initialize @ declaration (if don't, default val given)

to access a class member → ClassName.memberName

modifiers	type	identifier	instantiation
public static	int	idNum	= 1;
public static final int		favNum	= 49;

obj – actual val stored in mem (complex & accessed via references)

reference – "handle" / "address" where an object is

variable – created to store a reference

obj are instances of classes

```
String s = "hi";
```

class / ref var / obj

```
int x = 2
```

primitive type / var / primitive val

Calling convention

primitive vals – simple & immutable vals (byte, short, int, long, float, double & char)
var contains val, stored in stack

reference vals – vary wildly in size & complex vals (objects)
var contains address of var & references the val, stored in heap

```
public static void main (String[] args){
    int i = 15;
    char c = "a";
}
```

		random address
i	15	300
c	"a"	604

```
public static void main (String[] args){
    int i1 = 5;
    int i2 = i1;
    String s1 = "hello";
    String s2 = s1
}
```

i1	5	100
s2	1008	500
i2	5	200
s1	1008	300
"hello"		1008

```
public static void main (String[] args){
    String s = "hello";
}
```

		address
s	1004	800
"hello"		1004

- s1 & s2 are diff but ref same vals
- i1 & i2 are diff & contain distinct vals

What if you change the obj the ref vars are referencing?
↳ modify obj of either ref will affect both vars

```
Person p1 = new Person ("Al");   → creates new object
Person p2 = p1;                  → points to same obj (Al)
p2.name = "Bob";                 → modifies p1 obj via p2
p2 = new Person ("Cat");         → p2 points to new obj
```

Scope & visibility

Scope – the region of a program in which a def is available

locations & scope – where something is defined & where it is accessible

- local var – def in a block or statements (ex. for loop, in a method).
  Accessible from declaration to end the statement block
- parameter – var def in a method signature (definition). Access through entire body of its method
- field (instance var or class var) – def in class. Accessible throughout class def, except static methods cant access non static fields

```
public class Bar {
    public int a;
    private int b;
    public static int c;
    int d) {
```

- local var - def in a block of statements (ex. for loop, in a method).
  Accessible from declaration to end the statement block
- parameter - var def in a method signature (definition). Access through entire body of its method
- field (instance var or class var) - def in class. Accessible through ent class def, except static methods cant access non static fields
- method - defined in a class. accessible through the entire class def
- class - defined in a package. accessible through the entire package

lifetimes - when def begins its existance & when it ceases to exist
- local var - when declared → exiting the block of code
- parameter - defined @ start of a method → destroyed when method returned
- instance var - obj is in begins it → dies when obj is no longer accessible
- class var - exist when entire program begins running → when prog quits entirely
- method - entire duration of the class containing it
- class - entire program's duration

```
for (int i=0; i < x.length; i++){
    int sum = 0;
    sum += x[i];
}
System.out.prin        → error, fails to compile
                         since sum isn't in scope
```

```
public class Bar {
    public int a;
    private int b;
    public static int c;

    public int doStuff (int x,                    int y) {
        int count = 0;
        for (int i = 0; i < x; i++) {
            int temp = i;
            temp += y;                                    nt;}
            if (temp % 10 == 0) { return cou
            if (temp % 2 == 0) {
                String s = "eaa";
                System.out.println ("looking" + s);
                count++;
            }
            else{count--;}
        }
        return count;
    }
    public void otter(){
        return;
    }
}
```

a, b, c, doStuff, otter
x, y, i, count
temp, s

visibility
- public - anyone w/ reference
- private - only object can access its fields & methods
- protected - used during inheritance (private but child classes can access)
- default < package - private > - public inside package but private outside

Selectively restore reading & writting privileges through
public methods in same class as private field
- getters - pub method that returns a copy of field (restore reading)
- setters - pub method that accepts parameter & update a field (restore writing)

*good practice to set every field
as private & then set if could read/write
selectively.
Exception would be constants - since
cant be written & safe to have read

```
public class Square {
    private int side;  → revoke reading & writting privleges
    public Square (int side) {
        this.side = side;
    }
    public int getSide(){return side;}  → restore reading
    public void setSide(int side){      → restore writting &
        if (side > 0) {this.side = side;}    perform checks
        else          {this.side = 1;}
    }
}
```

05 - file input & output

Saturday, May 3, 2025 12:57 PM

File I/O - reading & writting the contents of files

- Scanner - read textual data
- printWriter - write textual data
- ObjectInputStream & ObjectOutputStream - read & write binary files

Reading Text w/ a Scanner

```
import java.util.Scanner;
import java.io.File;          } before class

...
Scanner sc1 = new Scanner (System.in);
Scanner sc2 = new Scanner ("really long \n string \n
Scanner sc3 = new Scanner (new File ("local.txt"))  →needs exception handling
sc3.close();  → close file when done
```

```
import java.util.*;
import java.io.*;
public class FIO {
    public static void main (String[] args) {
        Scanner sc3 = null;
        try {
            SC3 = new Scanner (new File ("help.txt"));
        }
        catch (FileNotFoundException e) {
            System.err.println ("file not found");   →print error msg to System.err
            System.exit (0);   →end prog w/ catch-block
        }
        String s = sc3.nex();
        system.out.println ("s = `" + s "'");
        sc3.close();
    }
```

```
        sc3.close();
    }
}
```

Writting to Text Files w/ PrintWriter

```
import java.util.*;
import java.io.*;
public class F10 {
    public static void main (String[] args){
        PrintWriter pw = null;
        try { pw = new PrintWriter (new File ("help.txt")); }
        catch (FileNotFoundException e){
            System.err.print ("couldnt open file for writting");
            System.exit(0);
        }
        pw.print ("partial line");
        pw.println ("complete line");
        pw.printf ("with %s \n", "subs");
        pw.close();
    }
}
```

06 - classes & objects

Saturday, May 3, 2025 2:19 PM

type - representing a set of values

primitive types

- byte (-128 ~ 127)
- int (-2147483648 ~ 2147483647)
- boolean (true, false)
- char
- short
- long
- float
- double

objects

- Integer
- Double (wrapper class for double)

class - defines a new type of vals by grouping diff types of vals together & define behaviors by defining methods

subvalues - (data components) that make up their state

- attributes - general term for var that stores data of obj/class
  - field members - vars declared directly in class
    - static fields - use when data is shared across all obj (eg wheels for car)
    - final fields - for constants (eg. MAX_SPEED = 120)
  - instance vars - subset of field members, belong to individual objects, not class (eg. name, age) → non static

object - specific value of class, has its own val for ea attribute in the class

constructor - method of taking brand new obj & setting up all the instance vars & returning that obj

```
public class Person {
    public int age;
    public String name;

    public Person (int age, String name) {
        this.age = age ;
        this. name = name ;
    }
}
```

(constructor)

```
Public class TestLab2 {
    public static void main (String[] args) {
        Person p1 = new Person();
        p1.age = 18 ;
        p1. name = "Bob";
        System.out.println ("name" +  p1.name + "age" + p1.age);
    }
}
```

When compiling:

```
javac TestLab2.java
java TestLab2
```

07 - inheritance

Saturday, May 3, 2025 3:56 PM

inheritance - new subclass / child can reuse, extend, or modify properties of existing parent class / superclass

Cat is an animal → Dog inherits from Animals

```
class Animal {
    protected String name;   → accessible only to subclasses only
    public Animal (String name) {   (can use private String name
    this.name = name;                + setter & getters instead)
    }
    void sound () {   → use void method to perform action but not return
        System.out.println ("Animal makes sound");   anything
    }
}
```

```
Public class Main {
    public static void main (String[] args) {
        Cat myCat = new Cat ("Whiskers");
        myCat.meow();
    }
}
```

encapsulation principles - data (fields) & methods (behaviors) in one class (→ vars) & restrict access to internal details

```
Class Cat extends Animal {
    public cat (String name, String color) {
        super(name);   → pass name to Animal constructor
        this.color = color;
    }
    @Override   → override's parents (Animal) sound
    void sound () {
        System.out.println (name + " says Meow");   → direct access
    }
}
```

this is somehow an acted would write like?

protected field → only subclasses can access
@Override → overrides parent's method
extends → class becomes child / subclass to parent / superclass
super(field) → used to call the constructor of parent / super class from child / subclass

Abstract Classes - cannot be instantiated
can contain both concrete & abstract methods
all abstract methods must be implemented by concrete subclasses

- package-private access
- class is visible w/in own package
- use for small tests

```
abstract class Shape {   ← no public / private modifiers used bc default (package-private) access being applied (class visible in its own package)
    abstract double area();   ← no implementation
}

class Circle extends Shape {
    double radius;
    @Override
    double area() {   ← implementation req
        return Math.PI * radius * radius;
    }
}
```

Interfaces - abstract type
multiple inheritance
provides abstraction w/out implementation details
methods are public abstract
fields are public static final (constants)
cannot be instantiated directly
no constructors

```
interface Rentable {
    double startRental (int days, float dailyRate);
}

abstract Vehicle {
    protected String Model;
    public Vehicle (String model) {
        this.model = model;
    }
    abstract void startEngine ();
}
```

```
class Car extend Vehicle implements Rentable {
    public Car (String model) { super(model); }
    @Override
    void startEngine () {
        System.out.println ("Car engine started");
    }
    @Override
    public double startRental (int days, float dailyRate) {
        return days * dailyRate;
    }
}
```

Comparable & Iterator - 2 fundamental interfaces
comparable (sorting) - allow obj to define their natural ordering (eg. # by val, string alphabetically)
uses Collections.sort () method sort elements in ascending order

```
import java.util.*;   → imports all classes from java.util package
class Number implements Comparable <Number> {
    int value;
    public Number (int value) {
        this.value = value;
    }
    @Override
    public int compareTo (Number other) {   → defines ordering (ascending)
        return this.value - other.value;
    }
    @Override
    public String toString () {
        return String.valueOf (value);
    }
}
```

```
public class Main {
    public static void main (String[] args) {
        List<Number> numbers = Arrays.asList (   → converts array into fixed sized List
            new Number (5),
            new Number (1),
            new Number (3)
        );
        Collections.sort (numbers);   → uses compareTo()
        System.out.println (numbers);   → [1, 3, 5]
    }
```

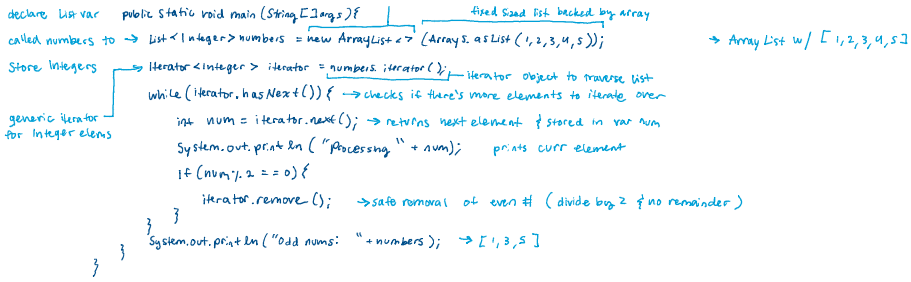
array vs. list
{1, 3, 5} → [1, 3, 5]
↑ size is fixed ↑ can remove & add elements
listName.remove( ) .add( )

compareTo() returns:
- Neg - obj is smaller
- Zero - equal
- pos - obj is larger

Iterator - way to move through collections (eg ArrayList) safely, allowing element removal during iteration

```
import java.util.*;
public class Main {
    public static void main (String[] args) {
        List<Integer> numbers = new ArrayList<> (Arrays.asList (1, 2, 3, 4, 5));
        Iterator<Integer> iterator = numbers.iterator();
        while (iterator.hasNext()) {
```

declare List var called numbers to → List<Integer> numbers
Store Integers
resizeable arraylist
fixed sized list backed by array
→ Array List w/ [1, 2, 3, 4, 5]
→ Iterator<Integer> iterator
iterator object to traverse list
→ checks if there's more elements to iterate over


declare list var
called numbers to
Store Integers
generic iterator for Integer elems
public static void main (String[] args) {
List<Integer> numbers = new ArrayList<> (Arrays.asList(1,2,3,4,5));
fixed sized list backed by array
→ ArrayList w/ [1,2,3,4,5]
Iterator<Integer> iterator = numbers.iterator();
iterator object to traverse list
while (iterator.hasNext()) { → checks if there's more elements to iterate over
int num = iterator.next(); → returns next element & stored in var num
System.out.println("Processing " + num); prints curr element
if (num % 2 == 0) {
iterator.remove(); →safe removal of even # (divide by 2 & no remainder)
}
}
System.out.println("Odd nums: " + numbers); → [1,3,5]
}
}

09 - enumeration

Sunday, May 4, 2025 1:19 AM

enumeration – type that restrict values to a predefined set (named constants) all caps

```
public enum Day { SUN, MON, TUE, WED, THURS, FRI, SAT }
Day today = Day.MON;
System.out.println(today);
```

Key features: type safety - only enum vals allowed
immutability - enum vals are constants

```
String style = "";
switch (ghost) {
    case BLINKY:
        style = "chase";
    case PINKY:
        style = "ambusher";
}
```

enums can have fields, constructors & methods (like classes)

```
public enum Ghost {
    BLINKY ("Blinky", "attacker"),
    PINKY ("Pinky", "ambusher");
    private final String name;
    private String movementStyle;

    private Ghost (String name, String style){
        this.name = name;
        this.movementStyle = style;
    }
    public String getName() {return name;}
    public void setMovementStyle (String style){
        this.movementStyle = style;
    }
}

Ghost blinky = Ghost.BLINKY;
System.out.println (blinky.getName());
```

10 - exceptions

Sunday, May 4, 2025 4:38 PM

exceptions - events that disrupt normal program flow when error occurs

exception hierarchy
- all exceptions inherit from java.lang.Throwable
  - Error - unrecoverable sys errors (eg. NullPointerException)
  - Exception - recoverable issues
    - RuntimeException - unchecked (e.g. NullPointerException)
    - CheckedException - must be handled (eg. IOException)

arithmeticException - divide by zero

FileNotFoundException - accessing a non-existent file

NullPointerException - using null as an object

NumberFormatException - failed String to num conversion (Integer.parseInt("abc")

Try - Catch Blocks — handles exceptions

risky code in try block

can have multiple catch blocks

order of exceptions matter

```
try {
    int[] arr = {1,2,3}
    system.out.println(arr[5]);
}catch (ArrayIndexOutOfBoundsException e){
    System.out.println("Index out of bounds");
}catch (.... e){
    ...
} catch (Exception e){
    system.out.println("General error: " +e);
    ↓ (same.)
```

Finally Block - code always runs regardless of exceptions

```
} finally {
    System.out.println("Cleanup done");
}
```

	checked exceptions	vs	unchecked exceptions
handling	must be caught/declared		optional (no compiler enforcement)
examples	IOException, SQLException		NullPointerException, ArithmeticException
usage	External factors (eg. file I/O)		Programming Errors (eg. Bugs)

Declare Checked Exceptions by using throws

```
public void readFile(String path) throws FileNotFoundExceptin {
    Scanner sc =new Scanner(new File(path));
}
```

Creating Custom Exceptions

```
class NegativeNumberException extends RuntimeException {
    public NegativeNumberException(String msg){
        super(msg);
    }
}
if (num<0){
    throw new NegativeNumberException("Neg nums not allowed");
}
```

14 - generics

Sunday, May 4, 2025 5:47 PM

generics – enable types (classes & interfaces) to be parameters when defining classes, interfaces, & methods

Allows to:
- create classes that work w/ diff data types
- provide stronger type checks @ compile time
- eliminate many explicit type casts

```
public class Pair<T, U> {          ↪ <T, U> declares type parameters
    private T first;                  T & U can be used throughout the class as types
    private U second;
    public Pair (T first, U second) {
        this.first = first;
        this.second = second;
    }
    public T getFirst() { return first; }
    public U getSecond() { return second; }
    public void setFirst (T first) { this.first = first; }
    public void setSecond (U second) { this.second = second; }
```

```
Pair<String, Integer> nameAge = new Pair<> ("Alice", 30);
Pair<Double, Double> coordinates = new Pair<> (40.7128, -74.0060);
String name = nameAge.getFirst();
int age = nameAge.getSecond();
```

Bounded Type Parameters

You can restrict the types that can be used w/ generics

```
public class NumberBox<T extends Number> {
    private T value;
    public NumberBox (T value) {
        this.value = value;
    }
    public double sqrt() {
        return Math.sqrt (value.doubleValue());
    }
}
```

can only be instantiated w/ Number types (Integer, Double etc)

Wildcards (?)

provide flexibility when working w/ generic types

```
public static void printList (List<?> list) {
    for (Object elem: list)
        System.out.println (elem);
    }
```

bounded wildcards

- `<? extends T>` → accepts T or any of its subclasses
- `<? super T>` → accepts T or any of its superclasses

Java Collections Framework use generics

```
List<String> names = new ArrayList<>();
names.add ("Bob");

Map<String, Integers> ages = new HashMap<>();
ages.put ("Bob", 25);
```

practical example

```
public class Box<T extends Comparable<T>> implements Comparable<Box<T>> {
    public T value;
    public B (T value) {
        this.value = value;
    }
    @Override
    public int compareTo (Box<T> other) {
        return this.value.compareTo (other.value);
    }
}
```

→ generic class Box w/ type parameter T

<T extends Comparable<T>> → T can compare itself to other obj of same type; T has a compareTo(T other) method

implements Comparable<Box<T>> → Box class can be compared to other boxes of same T

compareTo overrides compareTo from Comparable<Box<T>>; compares itself w/ objs w/ same type

returns
- Neg num if this.value < other.value
- zero if equal
- pos num if this.value > other.value

15 - recursion

Monday, May 5, 2025 12:10 AM

recursion - where method calls itself

$$n! = \begin{cases} 1 & \text{if } n = 0 \quad \text{(Base Case)} \\ n \times (n-1)! & \text{if } n > 0 \quad \text{(Recursive Case)} \end{cases}$$

must have

- base case(s)
  - method returns result w/out recursion
  - prevents ∞ recursion (eg. if (n == 0) return 1; for factorial)
- recursive case(s)
  - method calls itself w/ a smaller or simpler input
  - progresses toward base case (eg. factorial (n-1))

```
public static int factorial (int n) {
    if (n == 0) return 1;
    else return n * factorial (n-1);
}
```

16 - searching & sorting

Monday, May 5, 2025 1:05 AM

Searching Algorithms

Linear Search - checks ea element sequentially until target is found (use for small or unsorted datasets)

```
public static int linearSearch (int key, int [] arr ) {
    for (int i = 0; i < arr.length ; i ++ ) {
        if (arr [i ] == key ) return i ;
    }
    return -1 ;     → not found
}
```

Binary Search - efficiently searches sorted arrays by repeatedly dividing the search interval by half

* recurssive, use for large sorted data sets (eg. dictionaries, phonebooks)

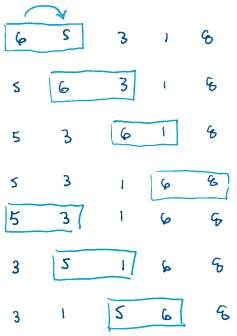
```
public static int binarySearch (int key, int [] arr, int left, int right ) {
    if (left > right ) return -1 ; → base case not found
    int mid = (left + right ) / 2 ;
    if (arr [mid ] = key ) return mid ;
    else if (arr [mid ] > key ) return binarySearch (key , arr, left, mid - 1 );
    else return binarySearch (key, arr, mid + 1 , right );
}
```

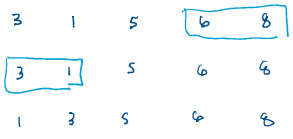
Sorting Algorithms

Bubble Sort - repeatedly swaps adjacent elements if they are in wrong order

for educational only, not practical

```
public static void bubbleSort (int [ ] arr ) {
    for (int i = 0 ; i < arr. length - 1 ; i ++ ) {
        for (int j = 0 ; j < arr. length - i - 1 ; j ++ ) {
            if (arr [j ] > arr [j + 1 ] ) {
                int temp = arr[j ];
                arr [j ] = arr [j + 1 ];
                arr [j + 1 ] = temp ;
            }
        }
    }
}
```





Insertion Sort - builds a sorted array one element @ a time by inserting ea elem into its correct position

for small or nearly sorted datasets

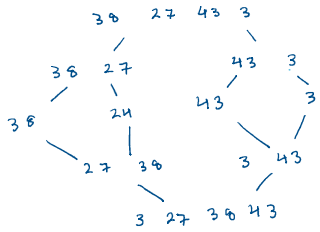
```
public static void insertionSorted (int [] arr ) {
    for ( int i = 1 ; i < arr.length ; i ++ ) {
        int key = arr [i ];
        int j = i - 1 ;
        while ( j >= 0 && arr[j ] > key ) {
            arr [j + 1 ] = arr [j ]
            j -- ;
        }
        arr [j + 1 ] = key
    }
}
```

merge Sort - divide & conquer algorithm that splits the array into halves, sorts them, & merges them

for large datasets where stability is important

```
public static void mergeSort (int [ ] arr ) {
    if (arr. length < = 1 ) return ;
    int mid = arr. length / 2 ;
    int [ ] left = Arrays . copyOfRange ( arr, 0, mid );
    int [ ] right = Arrays. copyOfRange (arr , mid , arr. length) ;
    mergeSort (left );
    mergeSort (right );
    merge (arr , left , right );
}

private static void merge ( int[ ] arr , int [ ] left , int [ ] right ) {
    int i = 0 , j = 0 , k = 0 ;
    while (i < left. length && j < right. length ) {
        if ( left [i ] < = right [ j ] ) arr [k ++ ] = left [i ++ ];
```



```
    int i = 0, j = 0, k = 0;
    while (i < left.length && j < right.length) {
        if (left[i] <= right[j]) arr[k++] = left[i++];
        else arr[k++] = right[j++];
    }
    while (i < left.length) arr[k++] = left[i++];
    while (j < right.length) arr[k++] = right[j++];
}
```

quick Sort - divide & conqure using a pivot element to partition the array

use for large datasets (generally faster than Merge Sort in practice) → general purpose sorting

```
public static void quickSort(int[] arr, int low, int high) {
    if (low < high) {
        int pivot = partition(arr, low, high);
        quickSort(arr, low, pivot - 1);
        quickSort(arr, pivot + 1, high);
    }
}

private static int partition(int[] arr, int low, int high) {
    int pivot = arr[high];
    int i = low - 1;
    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }
    int temp = arr[i+1];
    arr[i+1] = arr[high];
    arr[high] = temp;
    return i+1;
}
```