INTRO TO COURSE (slides)

what is computer science?

- systematic study of computer systems, algorithms, data, and applications/impacts
- interdisciplinary → math, engineering, empirical science, art, social science

what are algorithms?

- precise sequence of unambiguous steps that effectively compute an output given an input
- intuitive English → precise English → pseudocode → software
    - algorithm design: mathematical, logic of program, problem solving, language independent
    - implementation: semantics and syntax, language dependent, programming on a real machine

theory

- design an algorithm, analyze performance, data structure tradeoffs

practice

- write a Java program, debug/test, measure performance

data structures

- store different kinds of informatists are on in different ways, with algorithmic tradeoffs
- relates to efficiency of accessing or transforming data during operation of a program
- efficiency matters at higher scales (affects program speed)

INTRO TO JAVA (slides)

Java is a compiled language

- compiled
    - studies it all at once
    - **compiler**: program that translates source code into machine code
    - run the executable, the output of the compiler
- interpreted
    - line at a time
    - **interpreter**: program that translates and runs a program line by line
    - Python is an interpreted language
- Java Virtual Machine: write and compile once, run on multiple devices

command line

Command	Meaning	Details
`pwd`	Print Working Directory	Shows the full file path to the directory you are currently in
`ls`	List Files	Shows all files and directories contained in the current directory
`cd`	Change Directory	• `cd` by itself goes to your home directory • `cd directory` goes to the specified directory • `cd ..` goes to the enclosing directory
`mkdir`	Make Directory	• `mkdir directory` creates a directory
`cp`	Copy	`cp source target` Copies the `source` file and names the result `target`.
`rm`	Remove	`rm file` deletes the specified file. No backups!!!

-

compile and run Java

Command	Meaning	Details
`javac`	Compile .java files to .class files	• `javac file.java` compiles and creates `file.class` • `javac *.java` compiles **all** .java files in current directory to .class files.
`java`	Run java class files	`java file` executes the main method of `file.class`. Must have already been compiled from `file.java`.

-

Java basics

- each Java program file contains a single class → <className>.java
- run with public static void main (PSVM) method
    - public: can call outside of class
    - static: belongs to class, not an object
    - void: no return
    - main: starting point for a program to run
    - args: allows for command- line arguments
- strongly typed

primitive types

- byte, short, int, long, float, double, boolean, char
- String objects are not primitive but you can create them without “new”

basic operators

- + - * / % < <= > >= == ! && | |

reference types

- <object type> <name> = new <constructor>();
    - **Constructor**: method that initializes objects
- variable stores a reference to an object (place in memory)
- access instance variables and method calls with “.”

Arrays

- store fixed number of entries of a single type
- Type[] name = new Type[length]

Strings

- hold an Array of characters → can use toCharArray() to char[]
    - characters are ordered, comparable, and correspond to integer values

Loops

- for (int i = 0; i < num; i++) { }
- for (Object object : objects) { }
- while (boolean) { }

Methods

- function defined in a class
- <Object/void> methodName(<parameters>) {   }
- **dynamic methods**: called on a created object, has access to object data and arguments
    - s.equals()
- **static methods**: called on the class, only has access to arguments
    - Math.sqrt()

Java API data structures

- only store reference types (no primitives, int vs Integer)
- brief examples
    - **ArrayList**: list of values of the same type, can grow dynamically
    - **HashSet**: unordered collection, does not store duplicates

style

- comments // one line or /* */ multiple lines
- informative lowerCamelCase variable and method names

- informative CapitalizedCamelCase class names
- javadoc:

```
/**
 * descriptor
 * @param/return/throws
 */
```

OBJECT-ORIENTED PROGRAMMING (slides)

object-oriented: programs in that language are organized by the specification and use of objects

object: has some internal data items plus operations that can be performed on that data

class: blueprint for objects; specifies data and operations for a type of object

- objects are an instance of a class
- instance variables
- constructor: specifies how to create a new object
- . operator accesses instance variable or method of this object (on what it's called)

reasons to call a method

- side effect → what it did to an object
- return value

equals() (not == like for primitives) → @Override in Jdoc

- == → checks if these two are the same object in memory
- equals() without implementation → default Object checks if these two are the same object in memory

Object: default class from which all objects inherit

- you must always allocate memory and create each object

privacy

- public: can be accessed by code outside of class
- private: can only be accessed by code inside of class
- protected: can be access by code inside the package

immutability

- cannot change after creation (vs mutable), Strings are immutable

static

- belong to the class (method → called on a class, usually functional)

INTERFACES & IMPLEMENTATIONS (slides)

ADT (Abstract data type): specifies what a data structure does (functionality) but not how it does it (implementation)

API (Application program interface) perspective: what methods can I call on these objects, what inputs do they take, what outputs do they return?

interface: specify a set of abstract methods that an implementing class must override and define (ADT)

- **collection**: a group of objects
    - **List**: ordered sequence of values ➝ **ArrayList, LinkedList**
    - **Set**: unordered collection of unique values ➝ **HashSet, TreeSet**
    - **Map**: collection that associates keys and values ➝ **HashMap, TreeMap**
- need an implementing class at creation (List<String> l = new ArrayList<>())
- implement list ➝ public class NewList implements List
    - Must have at least all the same methods as the interface
    - As a parameter, you can just have List<Object> since any implementing class can use the same methods
- vs superclass ➝ you get the implementation but not for interfaces

algorithmic tradeoffs

- efficiency of operations on data structures depends on scale

ex: ArrayList

- get() ➝ direct lookup, constant (advantage)
- contains() ➝ loops through Array calling equals(), takes longer with more size
- size() ➝ returns an instance variable tracking size, constant
- add() ➝ depends
    - keeps an Array with space at the end ➝ add to open position or create new copied Array
    - space left ➝ constant time, one Array value assignment
    - no space left ➝ takes N array assignments, copy entire list

- how many times you have to copy an ArrayList:

Geometric growth

$$1 + 2 + 4 + \cdots + N$$

$$= \sum_{i=0}^{\approx \log_2 N} 2^i$$

$$\approx 2N$$

Geometric series formula: $\sum_{i=0}^{n} a\, r^i = a\left(\frac{1 - r^{n+1}}{1 - r}\right)$

Arithmetic growth

$$1 + 101 + 201 + \cdots + N$$

$$= \sum_{i=0}^{\approx N/100} 1 + 100i$$

$$\approx \frac{N^2}{200}$$

Arithmetic series formula: $\sum_{i=1}^{n} a_i = \left(\frac{n}{2}\right)(a_1 + a_n)$

  - geometric is better for larger size, arithmetic is better for smaller size
  - Java API uses geometric growth
- adding to the front of an ArrayList is NOT efficient
  - have to shift everything over
  - worst case: Array is full and must allocate space
- can use Arrays.asList and toArray to convert between List and Array

SETS & MAPS (slides)

Sets

- stores unique elements, not necessarily stored in order
- usage
    - contains() ➝ constant time efficiency
    - add() ➝ constant time efficiency
    - remove()
    - loop with enhanced for (:)
    - addAll() to convert to and from List
- **HashSet**
    - constant time: does not depend on the number of values stored in the Set
    - very efficient add, contains
    - ex: countUniqueWords
        - HashSet ➝ constant time operation, linear complexity
        - ArrayList ➝ must check all the words so far for each word, quadratic complexity
    - HashSet and HashMap implemented with hash table data structure
- **TreeSet**
    - nearly as efficient as HashSet, keeps values sorted
    - TreeSet and TreeMap implemented with binary tree data structure

Maps

- interface that pairs keys with values ➝ lookup the value with the key
- usage
    - put(k, v) ➝ associate value v with key k, constant time efficiency
    - get(k) ➝ gets the value with key k, constant time efficiency
    - containsKey(k) ➝ returns if key k is in the Map, constant time efficiency

- putIfAbsent(k, v)
- check containsKey first because calling get on a key not in the Map crashes the program
- updating Maps
  - single values
    - get() returns a copy of the value
    - must use put() to update → no += directly
  - collection values
    - get() returns reference to collection
    - updates the collection directly
- **HashMap**
  - very efficient put, get, containsKey
- **TreeMap**
  - nearly as efficient as HashMap, keeps keys sorted

HASHING - HASHMAP & HASHSET ([slides]())

hash table concept

- implement HashMap with ArrayList
- calculate hash (int) of key to determine where to store and lookup
- have the same hash for put and get that tells you where to look in the table instead of looping
    - hash = Math.abs(key.hashCode()) % list.size() → gets the hash code and mods from size

HashSet methods in detail

- add() → look up bucket corresponding to hashCode(), check if it's equal to anything in bucket

HashMap methods in detail

- put(k, v) → adds (<k, v>) to list at index hash, if key already there, update value
- get(k) → return value paired with key at index hash position of list
- containsKey(k) → check if key exists at index hash position of list

collisions

- when two different keys hash to the same position in the table
- hash table is an ArrayList of "buckets" (lists) that store multiple <k, v> pairs → chaining
- put(k, v) → add to hash index bucket, update value if key is already in bucket
- get(k) → loop over keys in hash index bucket and return the one that is equal
- ^ amortized constant time to account for rehashing and searching through the ArrayList

correct storage

- need to override equals() for the key type → otherwise cannot detect duplicates
- need hashCode() to work correctly for the key type → equals() have the same hashCode()

hashing efficiency

- runtime of get(), put(), containsKey() → time to get hash, time to search over the hash index bucket (calling equals() on everything inside)
- HashMaps are faster with more buckets bu that takes more memory
- correctness requirement: any equal() keys have the same hashcode() → not efficient

SUHA (simple uniform hashing assumption)

- suppose we hash N pairs to M buckets
- SUHA: any given key $k \in K$ is equally likely to be hashed to one of the M possible values, independently of what values other keys might hash to
- assuming uniform hashing, probability two random keys hash to same bucket is 1/M
- expected number of pairs per bucket is N/M (very likely true)
- time to get the hash is constant, but time to search over hash index "bucket" by calling .equals() on everything will take N/M runtime
- we face memory/runtime tradeoff with N pairs and M buckets, where…
    - if N >> M – too many pairs in too few buckets, runtime N/M is NOT constant
    - if M >> N – too many buckets, constant runtime but NOT memory efficient
    - usually "load factor", or maximum N/M ratio allowed in Java is 0.75, where M is slightly bigger than N, overall constant runtime but reasonable memory usage
    - HashMap/Set resizes if load factor is exceeded

RUNTIME EFFICIENCY (slides)

ex: two methods for repeated concatenation

- using String object and + operator
    - quadratic complexity, asymptotic ➝ $O(N^2)$
    - all the characters in original and concatenated section are copied over
    - grows arithmetically ➝ ex: input size 2N = runtime 2t
- vs using stringBuilder object and append()
    - linear amortized complexity
    - like an ArrayList for characters ➝ oversized Array that copies over when full
    - tradeoff for memory
    - grows geometrically (like ArrayList and HashMap) ➝ ex: input size 2N = runtime 2t
- Strings are an **immutable** (value cannot be changed after creation) Array of characters
- String **buffers** support mutable strings

ex: HashMaps ➝ designing more efficient algorithms

- NM (original) vs N+M (HashMap, only go through once) efficiency

ASYMPTOTIC (BIG-O) ANALYSIS (slides)

runtime and memory

- 2 fundamental resources
    - processor cycles: number of operations per second machine can perform
    - memory: space for storing variables, data, etc

constant time

- runtime does not depend on size of input
- ex:
    - index into an array (ar[1])
    - arithmetic, comparison
    - accessing object attribute (length)
    - ArrayList get, size, add (to end, amortized)
    - HashMap/Set get, put (amortized)
- non constant time usually includes a loop or method call

Big-O notation

- let N be the size of input
- T(N) → number of constant time operations in the code as function of N
- **Definition (big *O notation*).** $T(N)$ is $O(g(N))$ if $\lim_{N\to\infty} \frac{T(N)}{g(N)} \leq c$ for some constant c that does not depend on N.
- T(N) is O(g(N)) if it is at most a constant factor times slower than g(N) for large input N
- general rules
    - 1. Can drop constants
        - 2N+3 → O(N)
        - 0.001N + 1,000,000 → O(N)

      2. Can drop lower order terms
        - $2N^2+3N$ → $O(N^2)$
        - N+log(N) → O(N)
        - $2^N + N^2$ → $O(2^N)$

- hierarchy of complexity class

Big O	Name	Example
$O(2^N)$	Exponential	Calculate all subsets of a set
$O(N^3)$	Cubic	Multiply NxN matrices
$O(N^2)$	Quadratic	Loop over all *pairs* from N things
O(N log(N))	Nearly-linear	Sorting algorithms
O(N)	Linear	Loop over N things
O(log(N))	Logarithmic	Binary search a sorted list
O(1)	Constant	Addition, array access, etc.

- log: cutting in size (division)
- nested loops = multiplication
- side note: consecutive nums until x sum = (x)(x+1)/2

- outer(inner(n)) →big o inner(n) + big o of outer() called on return value of inner(n)