Cheatography

CS201 Midterm 1 Cheat Sheet

by Paloma via cheatography.com/55343/cs/14725/

Primitive Data Types

Type	Description	Default	Examples
Boolean	true/false	false	boolean b = true;
char	character	0	char c = 'A';
int- takes up less space than a double	integer	0	int i = 0;
double	floating point	0.0	double vel = 85.4;

Memory

Type	Bit/Bytes	Range
boolean	1 bit	True or false
char	16 bit/ 2 bytes	0 to 65535
byte	8 bit/ 1 byte	-128 to 127
short	16 bit/ 2bytes	-32768 to 32767
int	32 bits/4 bytes	-2147483648 to 2147483647
long	64 bits/ 8 bytes	Huge To huge
float	32 bits/ 4 bytes	varies
double	64 bits /8 bytes	varies

Array: Length of array * memory of type it contains

Primitives vs Objects

- Java is an object oriented language --> object oriented mean organized in terms of classes
- Primitive: can create them without using the new word
int x= 5;
- Objects: represented by a class and contain variables and methods
- Java has built-in objects, like String and ArrayList, but can also write your own

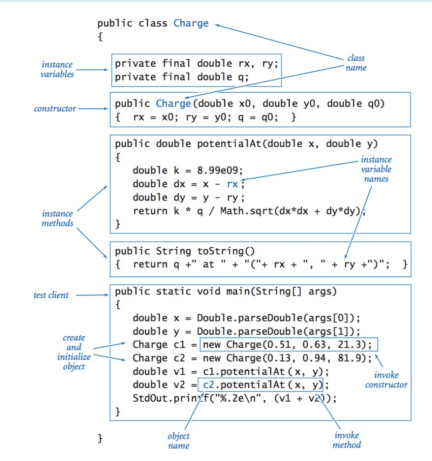
Primitives vs Objects (cont)

- Objects are created by writing a class to represent them
String s = new String ("Hello world");
- Object versions of all the primitive types because many data types require
you to say what is inside them
ArrayList<Integer> goodList = new ArrayList<Integer>();
- One except are arrays. They are created by having the type the array
contains, followed by square brackets.

Static Methods

You can write code like Math.exp(-x*x/2) / Math.sqrt(2*Math.PI) and Java knows what you mean.) Static methods are associated with the class. They can be called from static or dynamic methods.
No object
Static: associated with class
Dynamic: associated with instances of the class (objects) and have access to instance variables

Creating Data Types



NBody

Question 2: For what values of timeStep, does the simulation no longer behave correctly?
With a large totalTime and dt,the planets move in a spiral and some of the planets knock each other around during the simulation.
Large values don't work in the simulation because, the time step is too large. The planets don't
move fluidly and take large jumps from one position to the next.

Sets

```
/*
  Union: Set of members that belong to set A "or" set B.
*/
Set<Integer> union = new HashSet<Integer>();
union.addAll(A);
union.addAll(B);

/*
  Intersection: Set of members that belong to set A "and" set B.
*/
Set<Integer> intersection = new HashSet<Integer>();
intersection.addAll(A);
intersection.retainAll(B);

/*
  Difference: Set of members that belong set A "and not" set B.
*/
Set<Integer> difference = new HashSet<Integer>();
difference.addAll(A);
difference.removeAll(B);

/*
  Complement: Set of members that belong to set B "and not" set A.
*/
Set<Integer> complement = new HashSet<Integer>();
complement.addAll(B);
complement.removeAll(A);
```

s1.containsAll(s2) — returns true if s2 is a subset of s1. (s2 is a subset of s1 if set s1 contains all of the elements in s2.)

Arrays

```
type [] name = new type[size];
    Student [] studentList = new Student[3];
Length
   nameOfArray.length
Access value at a specific index
   nameOfArray[index]
Convert List to array:
Arrays.asList(ArrayList)
```

Arrays (cont)

Homogenous collections: once created, don't grow

Constructors

A class contains constructors that are invoked to create objects from the class blueprint and have same name as class. Constructor declarations look like method declarations—except that they use the name of the class and have no return type. For example, Bicycle has one constructor:

```
public Bicycle(int startCadence,
int startSpeed, int startGear) {
    gear = startGear;
    cadence = startCadence;
    speed = startSpeed;
}
Bicycle myBike = new Bicycle(30, 0,
8);
```

the call to new :

- calls the constructor, reference to the object
- all non primitive variables are pointers

-Calling constructor creates new object

To create a new Bicycle object called myBike, a constructor is called by the new operator:

Scanner Example Code

nextInt(), nextDouble(), and next() methods in the Scanner

```
List<Set<String>>
attendeeList(Scanner in) {
ArrayList<Set<String>> result = new
ArrayList<Set<String>>();
while (in.hasNext()) {
```

Scanner Example Code (cont)

```
TreeSet<String> words = new
TreeSet<String>();
Scanner line = new
Scanner(in.nextLine());
while (line.hasNext())
words.add(line.next());
result.add(words);
}
return result;
}
```

Trade-offs

Tree Maps and Tree Sets are slower but ordered

Hash Maps and Hash Sets are faster but unordered

ArrayList is slower and takes up more memory, but you can change the size

Array is faster and takes up less memory but you can't change the size

Comparing and Sorting

Arrays.sort

Collections.sort

Strings are comparable lexicographically: zebra>aardvark but Zebra<aardvark

yak.compareTo(s) returns <0, ==0, >0

Equals Method Example

A. Implement the equals method for the ABCChapter class. Two ABCChapters are equal if they have the same state, the same region, and all the members are the same. Assume the members are stored in alphabetical order in myMembers.

```
public boolean equals(Object o){
   ABCChapter chap = (ABCChapter) o;
   if (chap == null) return false;
   if (myState.equals(chap.myState) && myRegion.equals(chap.myRegion) &&
       myMembers.length == chap.myMembers.length) {
     for (int k = 0; k < myMembers.length; k++)
       if (!myMembers[k].equals(chap.myMembers[k]))
         return false;
     return true;
   }
   return false;
}
```

compareTo method example

B. Implement the compareTo method for the ABCChapter class. An ABCChapter is less than another ABCChapter if they have fewer members. If both chapters have the same number of members, then a chapter is less than another chapter if its state comes before it in alphabetical order. If the states are the same then a chapter is less than another chapter if its region comes before the other region in alphabetical order. If both the regions and the state are the same, then one chapter is less than another chapter if the list of member names comes before the list of the other chapter member names in alphabetical order.

```
public int compareTo(ABCChapter chap) {
  if (myMembers.length != chap.myMembers.length)
    return myMembers.length - chap.myMembers.length;
  if (!myState.equals(chap.myState))
    return myState.compareTo(chap.myState);
  if (!myRegion.equals(chap.myRegion))
    return myRegion.compareTo(chap.myRegion);
  for (int k = 0; k < myMembers.length; k++)
    if (!myMembers[k].equals(chap.myMembers[k]))
      return myMembers[k].compareTo(chap.myMembers[k]);
  return 0;
}
```

Markov

- **EfficientMarkov** based on using maps rather than rescanning the training text// Implement a new version of getFollows so that it's a constant time operation that uses myMap to return an ArrayList using the parameter to getFollows as a key. If the key isn't present in the map, throw a new NoSuchElementException with an appropriate String.

- **WordGram** that will use a word-markov model rather than the character-markov model you start with

- **EfficientWordMarkov**, modeled on EfficientMarkov and with the same structure in terms of methods --- but using WordGram objects rather than Strings//This class uses a map (either a TreeMap or a HashMap) to create a more efficient version of WordMarkovModel.

The order of the Markov Model, size of the N gram, does not have more impact than the size of the text on the run time.

Map Interface

A Map is an object that maps keys to values. A map cannot contain duplicate keys: Each key can map to at most one value. Keys in a map need to be comparable

Adding Values to Maps

```
Value is a counter
m.put(key, m.get(key) + 1)
value is arrayList// set
if (map.get(key) == null) {
    map.put(key, new
ArrayList<Integer>());
}
map.put(key,
map.get(key).add(number));
```

Maps API

Method	return	purpose
Map.size()	int	# keys
Map.keySet()	Set<K>	Set of keys
Map.values()	Collection<V>	All values
Map.containsKey(K)	boolean	Is key in Map?
Map.put(K,V)	V (ignored)	Insert (K,V)
Map.entrySet()	Set<Map.Entry>	Get (K,V) pairs
Map.clear()	void	Remove all keys

value = map.get(key)
Collections.max

ArrayList

Does not have fixed sized/ no primitive types

```
ArrayList<String> list = new
ArrayList<String>();
```

add(Object o) - adds an object to the ArrayList, must be of the correct type

remove(Object o) - removes that object from the ArrayList, if it exists

get(int index) - returns object at that index

contains(Object o) - returns true/false

size() - returns the length of the ArrayList

Object Values and Variables

== is usually used for primitive types / do they point to the same location in memory while .equals() is used for objects.

Strings

```
String name = "word";
```

String syntax

length() - returns length of String

charAt(int index) - returns character at that index

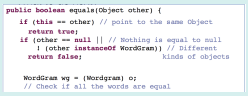
concat(String str) - cocatenates String str to the end of the String

compareTo(String stre) - compares the two Strings lexicographically

equals (Object obj) - compares the String to the object

indexOf(char c) - returns first index of the character c in the String

Hashing Markov Example



Instance of checks to see if the object is an instance of a class. Returns true if it is and returns false if it isn't. Hashing is important because it helps with efficiency (can access in O(1) time). While HashMaps already have a hash code, we overwrite it to make it more effective and so that ordering matters.

Inheritance: Implements vs. Etends.

Object-oriented programming allows classes to inherit commonly used state and behavior from other classes.Implements you have to override all the methods. Extends is you add some methods to existing class.

Hash Code Example

```
public int hashCode(){
int h = 0;
h += myState.hashCode() +
myRegion.hashCode()*3;
for (int k=0, fact=9; k <
myMembers.length; k++, fact *= 3) {
h += myMembers.hashCode()*fact;
return h;
}
```

Hashing

- Hashing is a search method: average case is O(1) search
- Associate a number with every key, use the number to store the key
- A hash function generates the number from the key
- Hash table is an array of fixed size with a key to each location and each key is mapped to an index in the table ArrayList<ArrayList<Type>>()
- Every object has a hashCode which is an integer value
- Two objects can have the same hashCode when two keys have the same value

Hashing (cont)

- you can look for the next spot
- two equal objects should hash to the same place because they have the same hash code and key
- if x.equals(y) then x.hashCode() == y.hashCode()

By **Paloma**
cheatography.com/paloma/

Not published yet.
Last updated 30th April, 2018.