AP CS Java Sparky Notes

Hip, Hip, Array!!!

BR2016

~ Primitive data ~
(int, double, boolean, char)
*Passed by "value" not "reference"

Examples:

```
int x;          //has value of 0
int y=5;       // integers have a range  -2^31 to 2^31 - 1
double a;     //has value 0.0
double b = 4.2;
char topGrade = 'A';  //single quote marks for char
boolean checker = true;
int min = Integer.MIN_VALUE;     //-2^31
int max = Integer.MAX_VALUE   //2^31
```

if/else/else if:

remember, the "else" goes with the closest "if" clause

Order of Operations:

! is always first , then ( ) *, /, %, +,-

(% returns remainder. Use for even/odd check.)

Equality and Relational

==	equal to
!=	not equal to
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to

Logical Operators

&& AND – both parts must be true

|| OR – at least must be true

! NOT – negate

DeMorgan's Laws:

!(p && q) = !p || !q

!(p || q) = !p && !q

Short Circuiting:

&& - once a false is reached, all is false

|| - once a true is reached, all is true

enumerated type – "category" data

```
enum season {winter, spring, summer, fall};
```

~ Objects ~ Passed by "reference" not "value."

anything that is not Primitive (String, Arrays, any object you create) is an object

Be careful of **"aliasing"** with **all** objects.

~ Strings ~

Key Points:

- index from 0 to .length()-1
- to compare strings - use .equals .equalsIgnoreCase or .compareTo (do NOT use != or ==)
- Strings is an **immutable** object - its methods cannot change the content of a string object.
- Be Aware/Careful of aliasing - when two variables point to the same memory location.

Examples:

```
String str1 = "compsci";
String str2 = new String ("hi");  // notice two ways to instantiate a String object
String str3;          //creates a null string of size 0
str2 = str3           //causes aliasing - str2 now points to the same memory location as str1
if (str2 == str3)   // == test for alias and it will return true. == is NOT the same as .equals
```

Frequently Used Methods:

.length()	returns the length (or number of characters) of the string
.equals(s)	returns true or false when comparing two strings
.equalsIgnoreCase()	returns true or false when comparing two strings and ignores case difference
.substring (x,y)	returns a section of the string from the beginning of x to the beginning of y
.substring (x)	returns a section of the string from x to length-1
.charAt()	returns the character at index x
.indexOf(s)	returns the index of the String s in the string, searching from index 0
.indexOf(s,x)	returns the index of the String s in the string, searching from index x
.trim()	removes leading and trailing whitespaces
.replaceAll(x,y)	returns a new String with all x changed to y
.toUpperCase()	returns a new String with all uppercase characters
.toLowerCase()	returns a new String with all lowercase characters

Iterative Loops ~

~ for ~

Test condition at beginning of loop
Good for specific # of iterations

for loop syntax & example:

```
for (initial ; test;  change)

for (int i=0; i<str.length(); i++){
  //do something; }
```

~ for each ~

Good with arrays

for each loop syntax & example:

```
for (type variable: someList)

String [] someList;
for (String item: someList) {
      System.out.println(item); }
```

~ while ~

Test condition at beginning of loop
Good for unknown # iterations

while loop syntax:

```
while (someConditionIsTrue){
//do something
}
```

while loop example:

```
int count=0;
while (grade >=0){
   sum += grade;
   count++; }
```

~ do - while ~

Test condition at end of loop
Good for unknown # iterations
Runs at least once

do - while syntax:

```
do {
//do something
} while(condition);
```

do - while example:

```
do{
   sum += grade;
   count++; }
while (grade >=0);
```

~ Arrays ~

Arrays - can contain any data type (primitive or object) but all must the same data type (i.e. no mixing)

to find length of array: nameOfArray.length

to print elements of an array: Arrays.toString(nameOfArray) or a for-loop, or your own toString

~ 1D-Array~

int [] example1 = new int[10];

creates an array of size 10, index 0 to 9, filled with zeros.

int [] example2 = {1,2,5,6};

creates an array with specific values

print 1D-array: either use a for loop or toString method.

```
Arrays.toString(example2);

for (int cnt=0; cnt < example2.length; cnt++)
    System.out.prinln (example2[cnt] + " ");
```

~ 2D-Array ~

int [][] example3 = new int[10][4];

creates a 2D array of size 10 rows and 4 columns, filled with zeros.

int [][] example4 ={ {1,2,3},{4,5,6}};

creates a 2D array 2rows x 3col with specific values.

print 2D-array in row-major order

// (row control variable is in outer loop)

```
for (int row=1; row<myArray.length; row++) {
   for (int col=1; col<myArray[row].length; col++) {
      System.out.print( myArray[row][col] + " ");
   }
System.out.println();
}
```


SOME ARRAY ACCESS EXAMPLES

~ to find min (in 1D array) ~

```
// setting min to largest int number possible; search 0 to end
// pre-condition: array contains more than 0 elements
public static double findMin (int[] a) {
    int  aMin = Integer.MAX_VALUE;
    for (int i=0; i<a.length; i++){
       if (a[i] <aMin)
          aMin = a[i];
    }
    return aMin;
}
```

~ to find max (in 1D array) ~

```
// setting max to 0th element; search element #1 to end
// pre-condition: array contains more than 0 elements
public static double findMax (double[] a) {
    double aMax = a[0];
    for (int i=1; i<a.length; i++){
       if (a[i] > aMax)
          aMax = a[i];
    }
    return aMax;
}
```

~ Generate random integer from 1 to n ~

```
int r = (int)(n * Math.random()) + 1;
```

~ random numbers (used in array) ~

//code example will fill an array with random integer values in range 50-100 inclusive

```
Random generator = new Random();
 int [] numArray = new int [100]
 for (int index= 0; index<=numArray.length-1; index++)
      numArray[index] = (int)generator.nextInt(51) + 50;
```

~ to find average (in 2D array) ~

```
// pre-condition: array contains more than 0 elements
public static double findDoubleAverage (double[][] a) {
  int count=0;
 double sum = 0;
  for (int row=1; row<a.length; row++) {
     for (int col=1; col<a[row].length; col++) {
        sum +=a [row][col];
        count++;
      } }
 return sum/count; }
```

~ Searching Array – Sequential or Linear ~

- sequential search compares every element for a key.
- Not good for a large data set

```
public static int search(int [ ] num, int key)
{
    for (int index = 0; index < num.length; index++)
    {
        if ( num[index] = = key )
            return index;          //We found it!!!
    }
    return -1;                     //We did not find!!!
}
```

~ Searching Array - Binary~

- **binary** search is a "divide and conquer" approach.
- Array **must** be in order (See Sorting on Page 7)
- import java.util.Arrays.*;

- Frequently used methods that search a specified array for a key value:

public static int binarySearch (int[] a, int key)
public static int binarySearch (double[] a, double key)

  - The array must be sorted **before** making this call.
  - If not found, returns a -1.
  - If found, returns the index where "key" is located.
  - If it is not sorted, the results are undefined.

AP CS – must know

A. Operations on data structures
   1. Traversals
   2. Insertions
   3. Deletions

B. Searching
   1. Sequential
   2. Binary

C. Sorting
   1. Selection
   2. Insertion
   3. Mergesort

~ Code Example – Array sort & search ~

```
int arr1[] = {30, 20, 5,12,55};
Arrays.sort (arr1);  //from standard library to sort array
// now arr1 = {5,12,20,30,55}

int searchVal = 12;
int retVal = Arrays.binarySearch(arr1,searchVal);
//will return a 1 (index 1)
// if not found, binarySearch returns a -1
System.out.println ("The index of element 12 is : "+ retVal);
```


~ Sorting Algorithms ~

- **Arrays.sort (arrayToBeSorted)** method that sorts an array in ascending/descending order
- There are many algorithms but ALL require swapping elements.
- Swapping elements in an array requires 3 assignment statements.
- **Efficiency (big O notation)**: classify algorithms efficiency is based on input size (n) and identify best and worst case:
  Selection (best/worst: $O(n^2)$) Insertion (best: O(n) worst: $O(n^2)$) MergeSort (best/worst: O(n log n))
  Bubble (best: O(n) worst: $O(n^2)$)

Selection Sort

•select smallest element, put in 0^{th} position. Select next smallest element, put in 1^{st} position, etc.

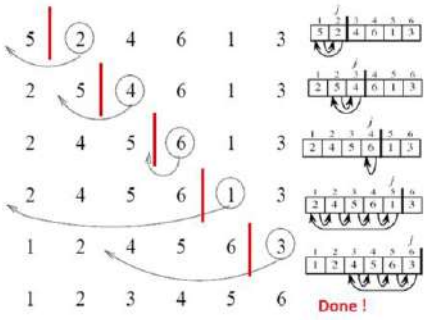
• **Inefficient on large lists.**

89	45	68	90	29	34	**17**
17 |	45	68	90	**29**	34	89
17	29 |	68	90	45	**34**	89
17	29	34 |	90	**45**	68	89
17	29	34	45 |	90	**68**	89
17	29	34	45	68 |	90	**89**
17	29	34	45	68	89 |	90

Insertion sort

start with two elements, put in order. Add another element and insert it into the proper location of the "subset", continue until done.

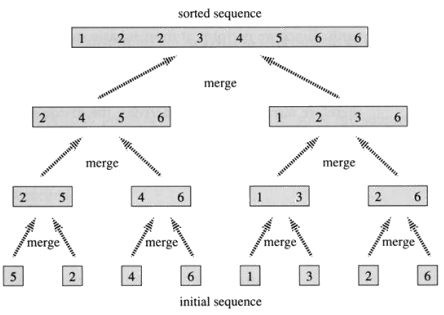
More efficient than selection.



MergeSort

• Split array in half, recursively sort the first half and the second half, then merge two sorted halves.

• Invented by John vonNeumann



~ArrayList ~

The **java.util.ArrayList** class provides resizeable-array and implements the **List** interface.

- You must **import java.util.ArrayList;**
- ArrayList **index starts at 0** and **ends at .size()-1**
- **ArrayLists hold objects.** Java will automatically convert primitive types to an object using the **Wrapper** class.

- **Syntax** to build an array list of integers:

ArrayList<Integer> nameOfList = new ArrayList<Integer>();

- **Frequently used ArrayList methods:**

Name	Use
add(item)	adds item to the end of the list
add (index, item)	adds item at index and shifts other items
set (index, item)	puts item at index
get (index)	returns the item at index
size()	returns the # of items in the list
remove()	removes an item from the list **
clear()	removes all items from the list

**** Warning** - removing items from an ArrayList if you process items right to left (low index to high index) and remove an element of an ArrayList, you can miss processing an item.

Initialize and add elements to ArrayList:

```
ArrayList<String> example5 = new ArrayList<String>();
example5.add ("java");
example5.add(0,"C");
example5.add(2,"run");
// example5 now contains:  C  java  run
```

3 ways to print an ArrayList:

```
//1. Using an  iterator
 ListIterator iterator = example5.listIterator();
 while (iterator.hasNext())
      System.out.println(iterator.next());

//2. Using a for loop with the .get() method
 for int(i=0; i<example5.size();i++)
       System.out.println(example5.get(i);

//3. Using a  print output statement
 System.out.println(example5);
```


~ Compare double, int and objects ~

1. Compare **integers**, use == != < > <= >=
   if (5 > 3) System.out.println ("5 is greater than 3");
2. Compare **doubles**, use < > <= >= **with a tolerance and absolute value**
   double d1 = 5.3, d2 = 5.31;
   if (Math.abs(d1-d2) <= .01) System.out.println("within tolerance");
3. Compare **objects**
   - **Strings** – use the **.equals**() or **.equalsIgnoreCase()** method.
     Returns true if the strings contain the same contents, false if not
     The .equals() method is of the class Object: public boolean equals (Object other)
     ```
     String favorite = "Comp Sci";
     if favorite.equals("Comp Sci") System.out.println("the strings are the equal");
     ```
     Note: Using == with Strings tests for aliases
   - **a.compareTo(b)** method : public int compareTo (T other)
     compareTo is part of the **Comparator interface** and can be used to compare two strings lexicographically (alphabetical order) and returns
     0 if "a" and "b" are equal in their order
     + integer if "a" comes after "b" in their order
     - integer if "a" comes before "b" in their order
   - **compare(a,b)** method: public int compare (T obj1, T obj2)
     compare compares values of two objects. It is **implemented as part of the Comparator interface.** You define what is compared and returned. compare(a,b) is an example of **implementation of an abstract method.** (see page 12)
     compare(a,b) returns a
     0 if obj1 and obj2 are equal
     + integer if obj1 >obj2,
     - integer if obj1<obj2

Method Header format: visibility static returnType name (parameters)

~Visibility: Public, Private, and Protected ~

The concepts of public and private apply to the class as a whole, not to the individual objects of the class.

- **private** features of a class can be directly accessed only within the class's own code.
- **public** features can be accessed in client classes using appropriate name-dot prefix.
- Instance variables are almost always **private.**

~ static (optional) ~

static modifier - an attribute belongs to a variable or class as a whole, not to an individual object instance of that class.

~ returnType ~

return statement returns a reference to an object of a class. The data type must match the return type.

~ void ~

If there is no returnType, use the reserved word **void**

~ final ~

final is a fixed value for a variable.

*File name and class name MUST match in name and case.
* main or runner is in a separate file/class

~ Class Order ~

```
// 1. class name
public static Account {
// 2.  instance variable definition
int  accountNumber;
double balance;
String name;
// 3.  constructor (like method header but no return type)
// constructor name matches class name
public Account (int acct, double bal, String nm)
{
accountNumber = acct;
balance = bal;
name = nm;
}
// 4. Methods – accessor, mutator
public double getBalance()
{ return balance;}}
```


~ Math Class ~

Math.pow(double base, double power) - returns double
Math.sqrt(double num) - returns double
Math.random() - returns double from [0,1)
Math.min (a, b) - returns minimum of a and b. if a and be are both int, returns int. If a and b are both double, returns double.
Math.max (a, b) - same as above but max
Math.abs(a) - returns absolute value of a. If a is int, return int. If a is double, returns double.
Math.PI – final value for pi

Math Class methods can be combined in one line of code.

To find the min of three numbers a,b,c:

```
int min = Math.min ( Math.min(a,b) , c);
```

Steps to generate a random integer from 1 to 6 (die)

```
int die1 = (int) (Math.random() *6) + 1;
```

1) Math.random returns a double [0,1).
2) Multiply both end values by 6 → possible values [0,6).
3) Convert to an integer → 0,1,2,3,4,5
4) Add one to every possible value → 1,2,3,4,5,6

~ Data Formatting ~

1. Escape Sequences (some)

\t	Insert a tab in the text at this point
\n	Insert a new line in the text at this point
\'	Insert a single quote in the text
\"	Insert a double quote in the text at this point
\\	Insert a \ in the text at this point

2. printf

Begins with a % and ends with a converter. The converter is a character indicating the argument type to be formatted.

import java.util.Formatter;

d	decimal (integer)
f	float
n	new line

```
double pi = Math.PI;
System.out.printf ("%f%n", pi);     // 3.141593
System.out.printf ("%.3f%n", pi);  // 3.142
```

3. DecimalFormat class

User specified "mask" for data formatting.
0 will force a 0, # will round the output to specified.

```
import java.text.DecimalFormat;
DecimalFormat fmt = new DecimalFormat("0.###");
double  someNumber = Math.random()*100;
System.out.println (fmt.format(someNumber));
NumberFormat money = NumberFormat.getCurrencyInstance();
double  someMoney = 20.16;
System.out.println(money.format(someMoney));  //$20.16
NumberFormat percent = NumberFormat.getPercentInstance();
double somePercent = 10.6;
System.out.println(percent.format(somePercent));   //10.6%
```

Interface & Abstract methods

- An **Interface** is a collection of abstract methods that you must implement in your class.
- An **abstract method** is a method that is declared but is not implemented (no code) in the interface but must be implemented in the class that "implements" the interface.

```
// interface is in one file (ExampleInterface.java)
public interface ExampleInterface {
        public int setSomething(); }  //abstract method

// implementation of interface in your class
public class Example implements ExampleInterface {
//instance variables and constructor here
public  int setSomething() {return 10;}
```

Recursion – the process of a method calling itself.

Always identify a base case and make sure you will reach it!

Example1: Factorial

Factorial: 5! = 5 * 4 * 3 * 2 * 1

n! = n * (n-1) * (n – 2) * … * 2 * 1

```
public static int factorial (int n) {
  if (n == 1)  return 1;              //base case
  return n * factorial (n-1);       //recursive call
}
```

Example2: sum 1 to n

```
public int sum (int n) {
  int result;
  if ( n == 1) result = 1;            //base case
  else  result = n + sum(n-1);   //recursive call
return result; }
```

Direct recursion is when a method calls itself (like above examples). **Indirect recursion** is when a method calls another method, eventually resulting in the original method being called again.

Uses of Recursion: solving maze, solving Towers Of Hanoi, **Sorting** (Merge Sort and Quick Sort), graphics.

- **Subclass (or child)**- A class that is derived from another class (parent) and inherits all fields and public and protected methods from its super class.
- **Java** only allows for **single inheritance** (a child can have only one parent)
- **All** classes in Java are descendants of Object.
- **extends** is the keyword used to inherit properties of a class
- **super** keyword is similar to **this** keyword. It is used to differentiate the members of superclass from members of the subclass if they have the same name.
- **this** is a keyword that references the currently executing object.

~ PARENT CLASS ~

```
public class Bicycle {

// the Bicycle class has three fields
   public int cadence;
   public int gear;
   public int speed;

 // the Bicycle class has one constructor
   public Bicycle(int startCadence, int startSpeed, int startGear) {
      gear = startGear;
      cadence = startCadence;
      speed = startSpeed;   }

// the Bicycle class has four methods
   public void setCadence(int newValue) {cadence = newValue;}
   public void setGear(int newValue) { gear = newValue;  }
    public void applyBrake(int decrement) { speed -= decrement; }
   public void speedUp(int increment) { speed += increment;  }    }
```

~ CHILD CLASS ~

```
public class MountainBike extends Bicycle  {

 // the MountainBike subclass adds one field
   public int seatHeight;

 // the MountainBike subclass has one constructor
   public MountainBike(int startHeight,  int startCadence,
                       int startSpeed,  int startGear) {
      super(startCadence, startSpeed, startGear);
      seatHeight = startHeight;  }

   // the MountainBike subclass adds one method
   public void setHeight(int newValue) {
                       seatHeight = newValue; }}
Instantiation:  public Bicycle MomBike = new Bicycle(2,4,6);
                public MountainBike myBike = new MountainBike();
```

- **Polymorphism** is the ability of an object to take on many forms.
- The most common use of **polymorphism** in OOP occurs when a parent class reference is used to refer to a child class object.
- Any **Java** object that can pass **more than one** IS-A test is considered to be **polymorphic.**

Example

public interface Vegetarian {}
public class Animal {}
public class Deer extends Animal implements Vegetarian{}

The Deer class is to be polymorphic since it has multiple inheritances.

Deer IS-A Animal
Deer IS-A Vegetarian
Deer IS-A Deer
Deer IS-A Object

The following are legal:

```
Deer d = new Deer();
Animal a = d;
Vegetarian v = d;
Object o = d;
```

All reference variables d,a,v,o refer to the same Deer object.

- An **exception** is an event, which occurs during the execution of a program, that interrupts the normal flow of the program. It is an error thrown by a class or method reporting an error in code
- The ***'Throwable'*** class is the superclass of all errors and exceptions in the Java language.
- Exceptions can be handled by using ***'try-catch'*** block. Try block contains the code which is under observation for exceptions. The catch block contains the remedy for the exception. If any exception occurs in the try block then the control jumps to catch block.

Exceptions to watch out for:

- A NullPointerException is thrown when an application is trying to use or access an object whose reference equals to null..
- IndexOutOfBoundsException - indicate that an index of some sort (such as to an array, to a string, or to a vector) is out of range.
- ArrayIndexOutOfBoundsException – indicates and index of an array is out or range.
- ArithmethicException – indicates a divide by zero.
- IllegalArgumentException - indicate that a method has been passed an illegal or inappropriate argument
- InputMismatchException – is thrown by a Scanner to indicate that the token retrieved does not match the pattern for the expected type, or that the token is out of range for the expected type

Exception Handling Syntax Rules:

1. The statements in the try{} block can include:
   Statements that work.
   Statements that might throw an exception
2. One or several catch{} blocks follow the try block
   Sometimes there can be no catch{} block
3. Each catch{} block says which type of Exception it catches.

Code Example:

```
Scanner scan = new Scanner (System.in);
int num;
System.out.println("enter an integer: ");
try
  {  num = scan.nextInt();
   System.out.println("your number is: "+num); }
catch (InputMismatchException ex)
  { System.out.println ("You entered bad data. "); }
```

Javadoc is a tool that generates html documentation (similar to the reference pages at java.sun.com) from Javadoc comments in the code.

Javadoc Comments

- Javadoc recognizes special comments **/** .... */** which are highlighted blue by default in Eclipse (regular comments **//** and **/* ... */** are highlighted green).
- Javadoc allows you to attach descriptions to classes, constructors, fields, interfaces and methods in the generated html documentation by placing Javadoc comments directly before their declaration statements.

Javadoc Tags

- ***Tags*** are keywords recognized by Javadoc which define the type of information that follows.
- Common pre-defined tags:
  - ***@author [author name]*** - identifies author(s) of a class or interface.
  - ***@version [version]*** - version info of a class or interface.
  - ***@param [argument name] [argument description]*** - describes an argument of method or constructor.
  - ***@return [description of return]*** - describes data returned by method (unnecessary for constructors and void methods).
  - ***@exception [exception thrown] [exception description]*** - describes exception thrown by method.
  - ***@throws [exception thrown] [exception description]*** - same as ***@exception***.

Javadoc code Example Shell:

```
/** Description of MyClass
 *
 * @author Favorite TeacherOne
 * @author Favorite TeacherTwo
 * @version 1.2a  January 2016
 */
public class MyClass
{       /** Description of input1 */
          public int input1;
          /** Description of MyClass()
           *
           * @throws myException
          Description of myException
           */
          public MyClass() throws myException
          {     // code would be here for myException   }

          /** Description of myMethod(int a, String b)
           *
           * @param a                          Description of a
           * @param b                          Description of b
           * @return                           Description of c
           */
          public Object myMethod(int a, String b)
          {
                    Object c;
                    // code would be here for myMethod
                    return c;
          }
       }
```