AP Computer Science A

Compiled from miscellaneous sources by Sanvi Pal

Using A Main Method

```
public class SomeClass
{
  public static void main(String[] args)
  {
         System.out.println("I run!");
  }
}
```

Console Output

```
System.out.print("I print out!  ");
System.out.println("I'm on the same line!");
System.out.println("I'm on the next line!");
```

Reading From System.in

```
Scanner scanner = new Scanner(System.in);
...
String text = scanner.nextLine();
```

Reading From Files

```
Scanner fileIn = new Scanner(new File("somefile.txt"));
while(fileIn.hasNextLine())
{
  String line = fileIn.nextLine();
  ...
}
```

Writing To Files

```
PrintWriter fileOut = new PrintWriter(new FileWriter("somefile.txt"));
...
fileOut.println("some text");
...
fileOut.close();
```

Using Strings

- Strings are objects that represent character sequences.
- Create new instances by enclosing text in quotes.
  `String name = "Dave";`
- Write \" to include a quotation mark in a string literal, \\ to include a backslash, or \n to include a newline character.
  "Say\n\"Hi\"" creates a string with 8 characters: S a y *newline* " H i "
- Different instances may contain the same characters, so don't compare with ==.
- Strings are immutable--no methods will change the sequence of characters.
- "Da" + "ve" will create the new string "Dave".

Data Types

`boolean`	Boolean type, can be `true` or `false`
`byte`	1-byte signed integer
`char`	Unicode character (i.e. 16 bits)
`short`	2-byte signed integer
`int`	4-byte signed integer
`long`	8-byte signed integer
`float`	Single-precision fraction, 6 significant figures
`double`	Double-precision fraction, 15 significant figures

Operators

+ - * / %	Arithmetic operators (% means *remainder*)
++ --	Increment or decrement by 1 result = ++i; means increment by 1 first result = i++; means do the assignment first
+= -= *= /= %= etc.	E.g. i += 2 is equivalent to i = i + 2
&&	Logical AND, e.g. if (i > 50 && i < 70) The second test is only carried out if necessary - use & if the second test should *always* be done
||	Logical OR, e.g. if (i < 0 || i > 100) The second test is only carried out if necessary - use | if the second test should *always* be done
!	Logical NOT, e.g.if (!endOfFile)
== != > >= < <=	Relational operators
& | ^ ~	Bitwise operators (AND, OR, XOR, NOT)
<< >> >>>	Bitwise shift operators (shift left, shift right with sign extension, shift right with 0 fill)
instanceof	Test if an object is an instance of a class, e.g. if (anObj instanceof BankAccount) System.out.println("$$$");

Operator Precedence

Level	Operator	Description	Associativity

16	`[]`
	`.`
	`()`
		access array element
		access object member
		parentheses
			left to right

15	`++` `--`	unary post-increment unary post-decrement	not associative

14	`++`
	`--`
	`+`
	`-`
	`!`
	`~`
		unary pre-increment
		unary pre-decrement
		unary plus
		unary minus
		unary logical NOT
		unary bitwise NOT
			right to left

13	`()` `new`	cast object creation	right to left
12	`* / %`	multiplicative	left to right
11	`+ -` `+`	additive string concatenation	left to right
10	`<< >>` `>>>`	shift	left to right

9	`< <=`
	`> >=`
	`instanceof`
		relational	not associative

8	`==` `!=`	equality	left to right
7	`&`	bitwise AND	left to right
6	`^`	bitwise XOR	left to right
5	`|`	bitwise OR	left to right
4	`&&`	logical AND	left to right
3	`||`	logical OR	left to right
2	`?:`	ternary	right to left

1	`=  +=  -=`
	`*=  /=  %=`
	`&=  ^=  |=`
	`<<=  >>=  >>>=`
		assignment	right to left

Control Flow— if … else

`if` statements are formed as follows (the `else` clause is optional). The braces {} are necessary if the if-body exceeds one line; even if the if-body is just one line, the braces {} are worth having to aid readability:

```
String dayname;
…
if (dayname.equals("Sat") || dayname.equals("Sun")) {
   System.out.println("Hooray for the weekend");
}
else if (dayname.equals("Mon")) {
   System.out.println("I don't like Mondays");
}
else {
   System.out.println("Not long for the weekend!");
}
```

Control Flow— switch

`switch` is used to check an integer (or character) against a fixed list of alternative values:

```
int daynum;
…
switch (daynum) {
   case 0:
   case 6:
      System.out.println("Hooray for the weekend");
      break;

   case 1:
      System.out.println("I don't like Mondays");
      break;

   default:
      System.out.println("Not long for the weekend!");
      break;
}
```

Control Flow— Loops

Java contains three loop mechanisms:

```
int i = 0;
while (i < 100) {
   System.out.println("Next square is: " + i*i);
   i++;
}
```

```
for (int i = 0; i < 100; i++) {
   System.out.println("Next square is: " + i*i);
}
```

```
int positiveValue;
do {
   positiveValue = getNumFromUser();
}
while (positiveValue < 0);
```

Defining Classes

When you define a class, you define the data attributes (usually `private`) and the methods (usually `public`) for a new data type. The class definition is placed in a `.java` file as follows:

```
// This file is Student.java. The class is declared
// public, so that it can be used anywhere in the

program public class Student {

   private String name;
   private int    numCourses = 0;

   // Constructor to initialize all the data members
   public Student(String n, int c) {
      name = n;
      numCourses = c;
   }

   // No-arg constructor, to initialize with defaults
   public Student() {
      this("Anon", 0);       // Call other constructor
   }

   // finalize() is called when obj is garbage collected
   public void finalize() {
      System.out.println("Goodbye to this object");
   }

   // Other methods
   public void attendCourse() {
      numCourses++;
   }

   public void cancelPlaceOnCourse()
      { numCourses--;
   }

   public boolean isEligibleForChampagne() {
      return (numCourses >= 3);
   }
}
```

Using Classes

To create an object and send messages to the object:

```
public class MyTestClass {

   public static void main(String[] args) {

      // Step 1 - Declare object references
      // These refer to null initially in this example
      Student me, you;


      // Step 2 - Create new Student objects
      me = new Student("Andy", 0);
      you = new Student();


      // Step 3 - Use the Student objects
      me.attendCourse();
      you.attendCourse();


      if (me.isEligibleForChampagne())
         System.out.println("Thanks very much");

   }
}
```

Arrays

An array behaves like an object. Arrays are created and manipulated as follows:

```
// Step 1 - Declare a reference to an array
int[] squares;          // Could write int squares[];


// Step 2 - Create the array "object" itself
squares = new int[5];  // Creates array with 5 slots


// Step 3 - Initialize slots in the array
for (int i=0; i < squares.length; i++) {
   squares[i] = i * i;
   System.out.println(squares[i]);
}
```

Note that array elements start at [0], and that arrays have a length property that gives you the size of the array. If you inadvertently exceed an arrays' bounds, an exception is thrown at run time and the program aborts.

Note: Arrays can also be set up using the following abbreviated syntax:

```
String[] cities = {
      "San Francisco",
      "Dallas",
      "Minneapolis",
      "New York",
      "Washington,
      D.C."
};
```

2D arrays using initializer:

```
int[][] bob = {{7,8}, {5,6}, {4,2}};
```

2D arrays traversal using an enhanced for-each loop:

```
for (int[] row: arrayName){
        for(int element : row){
                System.out.println("Hi");
        }
}
```

Using Arrays

- Arrays can hold either reference types or primitive types.
- Arrays can never change size.
- Arrays are not classes, and are therefore accessed using special syntax.

Code	Behavior
`String[] names;`	Declares names to be of type String[]. *names is currently null*, but is capable of referring to an array of Strings.
`names = new String[3];`	names refers to a new array of size 3. *The array contains 3 nulls*, but it is capable of referring to 3 Strings.
`names.length`	Evaluates to 3--the length of the names array
`names[0] = "Dave";`	The *first* position in the array names now refers to the String "Dave".
`copy = names;`	Does *not* create a new array. copy refers to the *same* array as names
`names[1] = "fberg";`	The *second* position in the array names, *and therefore copy*, now refers to the String "fberg".
`copy[1]`	Evaluates to "fberg"
`double[] roots = {1.0, 1.4142, 1.7321, 2.0};`	Declares and initializes roots to refer to an array of the given 4 doubles.

ArrayList

```
import java.util.ArrayList;

public class Students {
  public static void main(String[] args) {

     // create an ArrayList called studentList, which initially
holds []
        ArrayList<String> studentList = new ArrayList<String>();

    // add students to the ArrayList
    studentList.add("John");
    studentList.add("Lily");
    studentList.add("Samantha");
    studentList.add("Tony");

    // remove John from the ArrayList, then Lily
    studentList.remove(0);
    studentList.remove("Lily");

    // studentList now holds [Samantha, Tony]

    System.out.println(studentList);
  }
}
```

Enhanced For Loop

The enhanced for loop (sometimes called a "for each" loop) can be used with any class that implements the Iterable interface, such as ArrayList.

```
import java.util.*;
public class IteratorExampleTwo
{
  public static void main ( String[] args)
  {
    ArrayList<String> names = new ArrayList<String>();

    names.add( "Amy" );    names.add( "Bob" );
    names.add( "Chris" );  names.add( "Deb" );
    names.add( "Elaine" ); names.add( "Frank" );
    names.add( "Gail" );   names.add( "Hal" );

    for ( String nm : names )
      System.out.println( nm );
  }
}
```

For Loop vs. For Each Loop

For Loop	For Each Loop

class Main {
 public static void main(String[] args) {
 	
  char[] vowels = {'a', 'e', 'i', 'o', 'u'};
 	
  // iterating through an array using a for loop
  for (int i = 0; i < vowels.length; ++ i) {
 	
System.out.println(vowels[i]);
  }
 }
}
	class Main {
	 public static void main(String[] args) {
	
	  char[] vowels = {'a', 'e', 'i', 'o', 'u'};
	
	  // iterating through an array using the for-each loop
	  for (char item: vowels) {
	  System.out.println(item);
	  }
	 }
	}

Inheritance and Polymorphism

A class can inherit all of the data and methods from another class. Methods in the *superclass* can be over-ridden by the subclass. Any members of the superclass that you want to access in the subclass should be declared `protected`. The `protected` access specifier allows subclasses, plus any classes in the same package, to access that item.

```
public class Account {
   private double balance = 0.0;

   public Account(double initBal) {
      balance = initBal;
   }

   public void deposit(double amt) {
      balance += amt;
   }

   public void withdraw(double amt) {
      balance -= amt;
   }

   public void display() {
      System.out.println("Balance is: " + balance);
   }
}

public class CheckAccount extends Account {
   private int maxChecks = 0;
   private int numChecksWritten = 0;

   public CheckAccount(double initBal, int maxChk) {
      super(initBal);             // Call superclass
      ctor maxChecks = maxChk;    // Initialize our data
   }

   public void withdraw(double amt) {
      super.withdraw(amt);        // Call superclass
      numChecksWritten++;         // Increment chk.
      num.
   }

   public void display() {
      super.display();            // Call
      superclass
      System.out.println(numChecksWritten);
   }
}
```

Abstract Classes

An abstract class is one that can never be instantiated; in other words, you cannot create an object of such a class. Abstract classes are specified as follows:

created and

```
// Abstract
superclass public abstract class
Mammal {
   …
}


// Concrete subclasses
public class Cat extends Mammal {
   …
}

public class Dog extends Mammal {
   …
}

public class Mouse extends Mammal {
   …
}
```

Abstract Methods

An abstract method is one that does not have a body in the superclass. Each concrete subclass is obliged to override the abstract method and provide an implementation; otherwise, the subclass is itself deemed abstract because it does not implement all its methods.

```
// Abstract superclass
public abstract class Mammal {

   // Declare some abstract methods
   public abstract void eat();
   public abstract void move();
   public abstract void
   reproduce();

   // Define some data members if you like
   private double weight;
   private int age;

   // Define some concrete methods too if you like
   public double getWeight{} {
      return weight;
   }

   public int getAge() {
      return age;
   }
}
```

Interfaces

An interface is similar to an abstract class with 100% abstract methods and no instance variables. An interface is defined as follows:

```
public interface Runnable {
   public void run();
}
```

A class can implement an interface as follows. The class is obliged to provide an implementation for every method specified in the interface, otherwise the clas must be declared `abstract` because it doesn't implement all its methods..

```
public class MyApp extends Applet implements Runnable {
   public void run() {
      // This is called when the Applet is kicked off
      // in a separate thread
      …
   }


   // Plus other applet methods
   …
}
```

Static Variables

A `static` variable is like a global variable for a class. In other words, you only get one instance of the variable for the whole class, regardless of how many objects exist. `static` variables are declared in the class as follows:

```
public class Account {
   private String accnum;                // Instance
   var private double balance = 0.0;     // Instance
   var

   private static double intRate = 5.0; // Class var

   …
}
```

Static Methods

A `static` method in a class is one that can only access `static` items; it cannot access any non-`static` data or methods. `static` methods are defined in the class as follows:

```
public class Account {

   public static void setIntRate(double newRate) {
      intRate = newRate;
   }

   public static double getIntRate()
      { return intRate;
   }
   …
}
```

To invoke a `static` method, use the name of the class as follows:

```
public class MyTestClass {

   public static void main(String[] args) {
      System.out.println("Interest rate is" +
                         Account.getIntRate());
   }
}
```

Packages

Related classes can be placed in a common package as follows:

```
// Car.java
package mycarpkg;

public class Car {
   …
}


// Engine.java
package mycarpkg;

public class Engine {
   …
}


// Transmission.java
package mycarpkg;

public class Transmission {
   …
}
```

Importing Packages

Anyone needing to use the classes in this package can *import* all or some of the classes in the package as follows:

```
import mycarpkg.*;      // import all classes in package
```

or

```
import mycarpkg.Car;    // just import individual classes
```

The `final` Keyword

The `final` keyword can be used in three situations:

`final` classes (for example, the class cannot be inherited from)
`final` methods (for example, the method cannot be overridden in a subc lass)
`final` variables (for example, the variable is constant and cannot be changed)

Here are some examples:

```
// final classes
public final class Color {
   …
}


// final methods
public class MySecurityClass {
   public final void validatePassword(String password) {
      …
   }
}


// final variables
public class MyTrigClass {
   public static final double PI = 3.1415;
   …
}
```

Exception Handling

Exception handling is achieved through five keywords in Java:

`try`	Statements that could cause an exception are placed in a 'try' b lock
`catch`	The block of code where error processing is placed
`finally`	An optional block of code after a 'try' block, for unconditional execution
`throw`	Used in the low-level code to generate, or 'throw' an exception
`throws`	Specifies the list of exceptions a method may throw Here

are some examples:

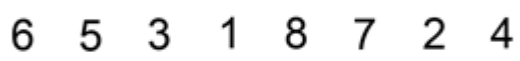
```
   public class MyClass {

         public void anyMethod() {
                  try {
                     func1();
                     func2();
                     func3();
                  }
                  catch (IOException e) {
                     System.out.println("IOException:" + e);
                  }
                  catch (MalformedURLException e) {

System.out.println("MalformedURLException:" + e);
                  }
                  finally {
System.out.println("This is always displayed");
                  }
         }

         public void func1() throws IOException {
                  …
         }

         public void func2() throws MalformedURLException {
                  …
         }

         public void func3() throws IOException,
                                    MalformedURLException {
                  …
         }
   }
```

Java Quick Reference

Accessible methods from the Java library that may be included in the exam

Class Constructors and Methods	Explanation
String Class	
`String(String str)`	Constructs a new `String` object that represents the same sequence of characters as `str`
`int length()`	Returns the number of characters in a `String` object
`String substring(int from, int to)`	Returns the substring beginning at index `from` and ending at index `to - 1`
`String substring(int from)`	Returns `substring(from, length())`
`int indexOf(String str)`	Returns the index of the first occurrence of `str`; returns `-1` if not found
`boolean equals(String other)`	Returns `true` if `this` is equal to `other`; returns `false` otherwise
`int compareTo(String other)`	Returns a value `<0` if `this` is less than `other`; returns zero if `this` is equal to `other`; returns a value `>0` if `this` is greater than `other`
Integer Class	
`Integer(int value)`	Constructs a new `Integer` object that represents the specified `int` value
`Integer.MIN_VALUE`	The minimum value represented by an `int` or `Integer`
`Integer.MAX_VALUE`	The maximum value represented by an `int` or `Integer`
`int intValue()`	Returns the value of this `Integer` as an `int`
Double Class	
`Double(double value)`	Constructs a new `Double` object that represents the specified `double` value
`double doubleValue()`	Returns the value of this `Double` as a `double`
Math Class	
`static int abs(int x)`	Returns the absolute value of an `int` value
`static double abs(double x)`	Returns the absolute value of a `double` value
`static double pow(double base, double exponent)`	Returns the value of the first parameter raised to the power of the second parameter
`static double sqrt(double x)`	Returns the positive square root of a `double` value
`static double random()`	Returns a `double` value greater than or equal to `0.0` and less than `1.0`
ArrayList Class	
`int size()`	Returns the number of elements in the list
`boolean add(E obj)`	Appends `obj` to end of list; returns `true`
`void add(int index, E obj)`	Inserts `obj` at position `index` (`0 <= index <= size`), moving elements at position `index` and higher to the right (adds 1 to their indices) and adds 1 to size
`E get(int index)`	Returns the element at position `index` in the list
`E set(int index, E obj)`	Replaces the element at position `index` with `obj`; returns the element formerly at position `index`
`E remove(int index)`	Removes element from position `index`, moving elements at position `index + 1` and higher to the left (subtracts 1 from their indices) and subtracts 1 from size; returns the element formerly at position `index`
Object Class	
`boolean equals(Object other)`	
`String toString()`	

Insertion Sort




```
void insertionSort(int arr[])
{
    int n = arr.length;
    for (int i = 1; i < n; ++i) {
        int key = arr[i];
        int j = i - 1;

        /* Move elements of arr[0..i-1], that are
           greater than key, to one position ahead
           of their current position */
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

Selection Sort


```
void selectionSort(int arr[])
{
    int n = arr.length;

    // One by one move boundary of unsorted subarray
    for (int i = 0; i < n-1; i++)
    {
        // Find the minimum element in unsorted array
        int min_idx = i;
        for (int j = i+1; j < n; j++)
            if (arr[j] < arr[min_idx])
                min_idx = j;

        // Swap the found minimum element with the first
        // element
        int temp = arr[min_idx];
        arr[min_idx] = arr[i];
        arr[i] = temp;
    }
}
```

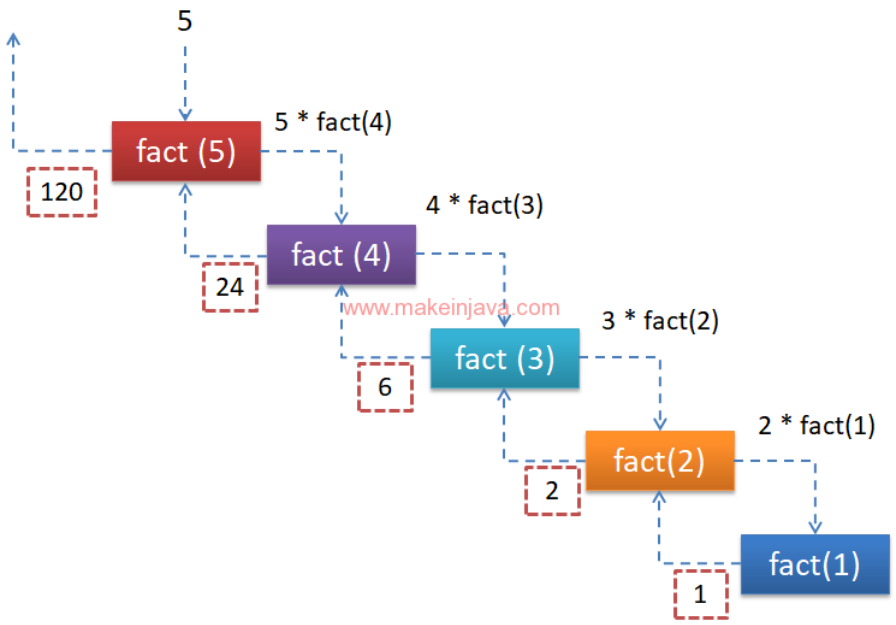
Insertion vs. Selection Sort

Insertion Sort	Selection Sort
Inserts the value in the presorted array to sort the set of values in the array.	Finds the minimum / maximum number from the list and sort it in ascending / descending order.
It is a stable sorting algorithm.	It is an unstable sorting algorithm.
The best-case time complexity is O(N) when the array is already in ascending order.	There is no best case the time complexity is $O(N^2)$ in all cases.
The number of comparison operations performed in this sorting algorithm is less than the swapping performed.	The number of comparison operations performed in this sorting algorithm is more than the swapping performed.
It is more efficient than the Selection sort.	It is less efficient than the Insertion sort.
Here the element is known beforehand, and we search for the correct position to place them.	The location where to put the element is previously known we search for the element to insert at that position.

Bubble Sort


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

Recursion



- Function is calling itself somewhere (start, end, middle) in its implementation with reduced parameter values.
    - For example fact() is calling fact() while reducing input value by 1 on each invocation
- Eventually function reaches an invocation with the final parameter value for which it performs some operation and returns instead of calling itself.
    - For example fact(1) returns 1
    - Stack unwinding towards the caller starts from here.
- Key is to follow stack winding and unwinding.
    - A pictorial view might help to follow the trail.

Top 5 Tips for the MC Section

1. Always test the algorithm. Don't make assumptions about what you think is happening based off of just looking at the code. Oftentimes there may be a NullPointerException error or "off by one" error that you might miss.
2. This section is where recursion is tested. You don't need to write recursive code but need to know how to test it and determine output. There are two types of recursion problems. The first is where the output is linear, it's in order of the order the method is called. The second type is when the output is in reverse order.
3. Sorting and searching is also tested on the MC section, you don't need to write most Sorting and Searching algos on the FRQ. You only need to write sequential search for the FRQs. Make sure you know the pseudocode for Selection Sort, Merge Sort, and Insertion Sort. Make sure you know Binary Search and of course, sequential search.
4. Timing gets better with practice. But if you can't finish the whole test, work on increasing your accuracy in the problems you do have time for solving
5. If you don't get the answer to a problem in about 2 minutes, skip it and come back to this (THIS ADVICE DOES NOT APPLY TO DIGITAL TEST)

Top 5 Tips for the FRQ section

1. If you can't visualize an algo, write the pseudocode or a flowchart.
2. Don't ever "freeze" as in get stuck thinking about a problem for too long. Type/write as you think as this ensures potential for partial credit OR you cracking the algorithm.
3. Do as many practice frqs as you can before the test. This will make you write more efficient code. Also Collegeboard isn't expecting you to make up most algos. It is expecting you to manipulate existing ones so by practicing, you get exposure to code that you can cross apply in multiple contexts.
4. Some existing "standard" algos to know are:
   For arrays~
   - determine min or max value in an array
   - compute sum, average, or mode
   - determine whether one/all element(s) has certain property
   - access all consecutive pairs of elements
   -determine presence of duplicate elements
   -determine how many elements meet certain criteria
   For arraylist~
   -inserting elements in traversal
   -deleting elements in traversal

5. Common mistakes on the FRQ include "off - by - one" errors where your loop does not access all elements of there is an IndexOutOfBounds exception thrown at runtime. Another common mistake is using a for-each loop to add or remove elements. Another common mistake is increasing indexes in a for or while loop after removing an element, causing the array to skip over an element. The last common mistake is forgetting that when a formal parameter is primitive, a copy of its value is made but when it is reference type, a copy of it's reference is made and you have the power to manipulate the original object's value.