

AP Computer Science Manual

Brandon Waller | brandonw@live.com

August 12, 2017

Contents

1 Purpose and Notes	2
1.1 Purpose	2
1.2 Notes	2
2 Introduction	3
2.1 Running Java Programs	3
2.1.1 BlueJ	3
2.1.2 Linux Terminal	3
2.1.3 The .java file	4
3 Java Basics	4
3.1 Introduction	4
3.1.1 bits	4
3.2 Data Types	5
3.2.1 Primitives	5
3.2.2 Declaring and Converting Primitives	5
3.2.3 Objects	6
3.2.4 Declaring Objects	6
3.2.5 String Operations	6
3.2.6 Arrays and the ArrayList	7
3.3 Operations	8
3.3.1 System.out.print(String)	8
3.3.2 Mathematical Operations	9
3.3.3 Modulo	10
3.4 Control Structure	10
3.4.1 Boolean Operations	10
3.4.2 If, If-Else, Else	11
3.4.3 Loops	11
3.4.4 Nested Loops	13
3.5 Questions	14

4	Methods	15
4.1	Structure of a Method	15
4.2	Calling Methods	16
4.3	Variable Scope	16
4.4	Global Variables	16
4.5	Return Type and Return Statement	16
4.6	Recursive Methods	18
4.7	Overloading Methods	20
5	Classes and Objects	21
5.1	Why use objects?	22
5.2	this	22
5.3	Example Object	22
5.4	Inheritance	24
5.4.1	Implementation	24
6	AP Exam Free Response Questions	25
6.1	2012 q4	25
6.1.1	Part A	26
6.1.2	Part B	27
6.1.3	Analysis and Solution	28
6.2	2017 11	29
6.2.1	Analysis and Solution	29
6.3	2017 q2	29
6.3.1	Analysis and Solution	29
6.4	2017 q3	30
6.4.1	Analysis and Solution	30
7	2017 q4	31
7.0.1	Analysis and Solution	31

1 Purpose and Notes

1.1 Purpose

This is mainly an excuse for me to practice L^AT_EX. Also, if you understand everything in this, you should get a 5.

1.2 Notes

- 1: Don't print this in case I update it
- 2: APCS is not an accurate representation of computer science. It is an introduction to computer science, upper level classes are more theory, and applications than coding.

- 3: Read **everything** carefully. Especially on the free response, the questions are long, pay attention to everything in them.
- 4: I didn't proof read this at all, try to ignore spelling and grammar errors.

2 Introduction

to score well on the AP exam, you will need to have a grasp of the basics of Java, a programming language that is rarely used any more, but the college board for some reason won't switch to Python.

2.1 Running Java Programs

note: Skip this section until you are ready to run Java programs

To run Java programs, you will either need to install a Java IDE (Integrated Development Environment) on PC/Linux/Mac , or use a terminal compiler on Linux or Mac.

2.1.1 BlueJ

When I took the class, we all used BlueJ, a free Java IDE. You can download it from this [Link](#). you may also need to download a [JDK](#) (Java Development Kit)

2.1.2 Linux Terminal

Run the following commands in the Terminal to run and compile .Java files:

- 1: `sudo apt-get install openjdk-8-jre`
- 2: `sudo apt-get install default-jdk`

To run and compile a file with the name **helloworld.java**:

- 1: `javac helloworld.java`
- 2: `java helloworld`

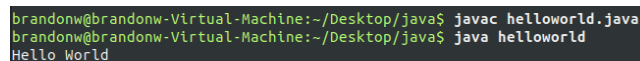
A terminal window with a dark background and light green text. The prompt is 'brandonw@brandonw-Virtual-Machine:~/Desktop/java\$'. The first command is 'javac helloworld.java'. The second command is 'java helloworld', which outputs 'Hello World'.

Figure 1: Running a .java file from the terminal

Additionally, you may want to learn how to use a terminal text editor such as vim or emacs to edit files.

```

1 import java.util.*;
2
3 public class helloworld
4 {
5     public static void main(String[] args)
6     {
7         System.out.println("Hello World");
8     }

```

Figure 2: Hello World Java program

2.1.3 The .java file

Java programs are written in a .java file that is interpreted by the java compiler to execute the instructions. Features of this file

- 1: line 1: importing java.util.*, a file that include additional Java functions
- 2: line 3: class declaration, public class followed by the file name
- 3: line 5: The main method. The set of instructions that are ran when by the executable after compiling
- 4: line 7: Java functions

There are many more things that go into .java files. These are just a few.

3 Java Basics

3.1 Introduction

Java is an object-oriented programming language developed by Sun Microsystems in 1995, and is the main language for Android app development. This section will cover the basics of Java including data types, operations, loops, and control structures.

3.1.1 bits

In computers, data is stored in bits that can either have a value of 1 or 0. Each data type is defined by the number of bits consumed by the data type, and by the way the bits are interpreted by the compiler. For example, the number 23 in 8-bit binary is 00010111 and the number -23 is stored as 11101001. This will be discussed later, but when converted to decimal 11101001 is 233. Depending on the data type, the compiler will either decide to make that number -23 or 233.

3.2 Data Types

Data types are the classifications given to variables/data. Unlike other languages such as Python and MATLAB, java requires the programmer to explicitly state the data type for a variable, making this section more important to know when programming in Java.

3.2.1 Primitives

Primitives are the basic Java data types used to store variables in memory. They are as follows:

- 1: byte: 8-bit signed data type (value $[-127, 127]$) used for applications where memory space is limited
- 2: int: 32-bit integer value that can positive and negative value
- 3: float: 32-bit decimal value
- 4: double: 64-bit decimal value, can hold more data than a float
- 5: char: 16-bit single character
- 6: boolean: stores either true or false

3.2.2 Declaring and Converting Primitives

Declaring a primitive in Java:

```
int x = 3;
byte y = 1271
double z = -12.34567;
boolean bool = true;
char a = 65;
char b = 'A';
//both a and b have the same value. Each character is
//mapped to a number
```

The left side of the = sign is the data type followed by the variable's name. On the right, is the value of the data type. Sometimes, you will want to convert one primitive to another, for example rounding a double to whole number. This is how you would do it. This uses if statements which will be discussed later:

```
double x = 2.6;
int x_whole = (int) a;//converts a to an int by removing the portion
    after the decimal point
if(x - x_whole >= 0.5)//if the decimal is greather than or equal to 0.5
    x = x_whole + 1;//round up by removing the decimal and adding 1
else
    x = (int) x;//rounding down, simply remove the decimal
```

3.2.3 Objects

Objects are other data types that are either defined by the Java language, or can be created by the user (new primitive types cannot be created by the user). Additionally, unlike primitives, functions can be called on objects, potentially modifying their values.

Some common objects defined by the Java language are

- 1: String: A group of characters
- 2: Integer
- 3: Double

note: Objects start with capital letters while primitives start with lower case

3.2.4 Declaring Objects

The syntax is slightly different for the declaration of objects

```
Integer j = 20;
String str = "the is the text for the string";
String s = new String();
s = "this is the text for String s";
```

The syntax for creating an instance for a user defined object will be a little different One of the defining features that distinguishes primitives from objects is that you can call functions on objects:

```
String s = "abcd";
System.out.println(s.length()); //output = 4
```

Because String is an object, you can call the function **.length()** on it.

3.2.5 String Operations

```
String str = "String";
```

substring(int startIndex, int endIndex): Creates a shorter string from a longer string, starting from startIndex and ending **one before** endIndex:

index	0	1	2	3	4	5
str	S	t	r	i	n	g
str.substring(0,2)	S	t				
str.substring(0,5)	S	t	r	i	n	
str.substring(0,str.length())	S	t	r	i	n	g
str.substring(3,3)						
str.substring(3,2)	error					

.equals() When strings are created, they are mapped to a certain location in memory, when a new string is created, the compiler first checks if there already is a string with the same value, if there is, they are both mapped to the same location. This causes the `==` operator to have inconsistent behavior with strings. Rather than check for string equality, the `==` operator checks if the strings are stored in the same location, if they are, the result is true, if not, the result is false. Here are some examples:

```
String a = "abc";
String b = new String("abc");
String c = new String ("abc");
String d = a;
a == b;//false
b == c;//false
a == d;//true
b == a;//false
```

To avoid this, use the **.equals(String2)** operator, which tests each character for equality.

```
a.equals(b);//true
b.equals(c);//true
c.equals(d);//true
```

3.2.6 Arrays and the ArrayList

Arrays are a group of variables that can be individually addressed and have a set size. They are useful for storing data of fixed length. Here is how you declare and use an array:

```
//Methods for declaring a new array
int[] i = new int[10];
int[] j = {1,2,3};
String[] s = new String[3];
String[] t = {"a", "b", "c"};
//Assigning values to an array
i[0] = 1;
i[1] = 2;
i[9] = 10;
s[0] = "abc";
//j and t already have values and are both size 3
//getting the length of an array
s.length;
i.length;
//accessing data from an array
i[1];
t[2];
j[1] + i[9];
```

note: Java, like any well written programming language starts its arrays at 0.

The ArrayList, unlike the array, does not have a define length. However, primitives cannot be stored in an ArrayList

```
//Basic syntax for creating an ArrayList: ArrayList<object type> name =
    new ArrayList<object type>();
ArrayList<String> strings = new ArrayList<String>();
ArrayList<Integer> integers = new ArrayList<Integer>();
//adding data to ArrayList
strings.add("abc");//add "abc" to index 0
strings.add("def");//add "def" to index 1
strings.add("ghi");
integers.add(1);
integers.add(2);
integers.add(3);
integers.add(0,4);//add 4 to index 0, shifting each element in the
    ArrayList forward
//accessing the data in an ArrayList
strings.get(0);//abc
strings.get(2);//ghi
integers.get(3);//3
//removing from an array list
strings.remove(0);//remove index 0 from the ArrayList and shift the rest
    of the list back
strings.get(0);//def
```

3.3 Operations

For the early stages of the class, you will mainly be staying within the **main()** function. This section will cover some of the basic operations you can do.

3.3.1 System.out.print(String)

System.out.print (and System.out.println()) are functions for displaying information to the console window. when using println(), subsequent text will be placed on a new line.

```
System.out.print("Hello");
System.out.print("World");
System.out.println("Hello ");
System.out.println("World");
```

Variables can be included in print statements. Each variable is separated by a +. When the variables are replaced by their value, they are included without a space.

```
String a = "a";
```

```
HelloWorldHello
World
brandonw@brandonw-Virtual-Machine:~/Desktop/java$
```

Figure 3: Console Window Output

```
String b = "c";
String c = "c";
int x = 123;
System.out.println(a + b + c + x);
System.out.println("abc123");
System.out.println(a + " " + b + " " + c + x);
System.out.println("a b c 123");
```

```
abc123
abc123
a b c 123
a b c 123
```

Figure 4: Console window output

Adding a " " between variables will but a space in the output.

3.3.2 Mathematical Operations

Java includes many mathematical operations, and many more can be added with the `java.math` package.

```
int x = 3, y = 5;
x + y; //8
x - y; //-2
x * y; //15
x / y; //0
sqrt(4); //2
sin(0); //0
```

Why is the result of x/y 0?

When two integers are divided, the result is stored as an integer. The result of dividing 3 and 5 is 0.6, which is stored as the integer 0. These are correct ways to divide integers and store the result as a double:

```
int x = 3, y = 5;
double z;
z = x/(double)y; //casting y as a double will result in a double result
z = 3/5.; z = 3/5.0;
//These implementations do not work:
```

```
z = (double)x/y; z = 3.0/5;  
//The denominator must be a double for the result of division to be a  
double
```

3.3.3 Modulo

The modulo is an operations similar to division, but the output is the remainder. The modulo operation is represented by the % sign. For example, $22 \% 10 = 2$ because $\frac{22}{10} = 2r2$

3.4 Control Structure

Control Structure is using boolean expressions to control if/when code is ran.

3.4.1 Boolean Operations

The boolean operations are as follows:

> : Greater than
>= : Greater than or equal to
< : Less than
<= : Less than or equal to
== : Equal to
!= : Not equal to
! : Not/Negation. Converts a boolean to the opposite value.
&& : and. Comparison of two booleans. True if and only if both conditions are true.
|| : or. Comparison of two booleans. If either is true, the or operation results in true

These operations are used to compare values and other booleans and result in a boolean. Here are some examples:

```
true == false;//false  
true == true;//true  
false == false;//true  
1 >= 0;//true  
1 > 1;//false  
!true;//false  
!false;//true  
!(true || false);//false  
!(!(true)||false) && true;//true
```

3.4.2 If, If-Else, Else

These three operations are the main three for controlling functions using boolean conditions. The basic syntax for this type of control is:

```
if(/*condition*/)
    //do something
//optional
else if(/*some other condition*/)
    //do something else
.....
else
    //If none of the conditions are true, run this code
```

In this type of statement, only one set of code will run regardless of how many are true. For example:

```
int x = 8;
if(x == 10)
    System.out.println("The value of x is 10");
else if(x % 2 == 0)
    System.out.println("x is an even number");
else if(x >= 5)
    System.out.println("x is greater than or equal to 5");
else
    System.out.println("None of the conditons are true");
```

The result of this code will be **x is an even number**. Although it is true that $x \geq 5$, that line will not be executed because the rest of the if-else statement is passed after one of the conditions is true. A way to implement this function so that all the applicable lines are executed is:

```
int x = 8;
if(x == 10)
    System.out.println("The value of x is 10");
if(x % 2 == 0)
    System.out.println("x is an even number");
if(x >= 5)
    System.out.println("x is greater than or equal to 5");
if(!(x == 10 || x % 2 == 0 || x >= 5))
    System.out.println("None of the conditons are true");
```

If the last line was kept as **else** then the code would be executed if $x < 5$ is true, even if x is even, or equal to 10.

3.4.3 Loops

Loops are a control structure that allows lines of code to be repeated many times, saving time when writing code, and allowing programs to be more flexible. There

are two types of loops: **for** and **while** loops. They are both fundamentally the same, but have different syntax.

The first type of loop is the **for** loop. It consists of declaring a variable, a condition, and the increment. When a for loop is created, a variable is given a variable, and it is compared against the condition. If the condition is true, then the code inside the loop runs. After, the variable is changed by the increment, and the process is repeated. Here is a for loop and a description of each step:

```
for(int i = 0; i < 3; i++)
{
    System.out.print("*");//The College Board loves printing *'s
}
```

- 1: declare the variable **i** with the value 0
- 2: evaluate $i < 3$ which results in true
- 3: output: *
- 4: evaluate: $i++$ (the $++$ and $--$ operation increase/decrease the variable by 1 respectively)
- 5: evaluate: $i < 3$ which results in true
- 6: output: **
- 7: evaluate: $i++$
- 8: evaluate: $i < 3$ which results in true
- 9: output: ***
- 10: evaluate: $i++$
- 11: evaluate: $i < 3$ which results in false ($3 < 3$ is false)

In a while loop, the variable is declared outside the loop, and the increment is inside the loop, meaning that you can change the increment and any time, not just at the end of the body of the loop. Here is an equivalent implementation using a while loop:

```
int i = 0;
while(i < 3)
{
    System.out.println("*");
    i++;
}
```

3.4.4 Nested Loops

Loops can appear inside other loops:

```
for(int i = 0; i < 3; i++)
{
    for(int j = i; j < 3; j++)
    {
        System.out.print("*")
    }
    System.out.println();
}
```

Output:

```

* * *
  * *
    *
```

Loops with arrays and ArrayLists

```
ArrayList<Integer> integers = new ArrayList<Integer>();
int[] ints = new int[10];
//Using a for loop to set values for in ints and integers
for(int i = 0; i < ints.length; i++)
{
    integers.add(i);
    ints[i] = i;
}
//integers and ints have the values 0...9
//increase each value by 1
for(int i = 0; i < integers.size(); i++)
{
    integers.set(i, integers.get(i) + 1); //set the element at index i to
        the value at index i + 1
    ints[i] += 1;
}
/*
**There is also the for-each loop that can access the data within an
    array list or array, but cannot change it
*/
for(int x : ints)
    System.out.print(x);
//this is equivalent to the following
for(int i = 0; i < ints.length; i++)
{
    int x = ints[i];
    System.out.print(x);
}
```

3.5 Questions

Create the following pattern by editing the code provided:

```
* * * *
* * * *
* *
* *
* * * *
* * * *
```

```
int i = 0;
while(i < 6)
{
    for(int j = 0; /*implemnt condition*/; j++)
    {
        /*
        Implement body
        */
    }
    i++;
}
```

Evaluate the following given that $t = \text{true}$ and $f = \text{false}$:

- 1: $!((t \parallel f) == f)$
- 2: $!((t \&\& f) == t)$
- 3: $((f == f) != (5 > 4)) \parallel (3 < 5)$
- 4: $(f == (!f \parallel (5 < (8 \% 3))))$
- 5: $5/3. == 5/3$
- 6: $5./3 == 5/3$

Create a for loop that will reverse the order of an array before: $i = [1,2,3,4,5]$ after: $i = [5,4,3,2,1]$

```
//code for swapping data without creating a new variable
int[] x = new int[2];
x[0] = 1;x[2] = 2;
x[0] = x[0] + x[1];//3
x[1] = x[0] - x[1];//3 - 2 = 1, x[1] is now equal to the old value of
    x[0]
x[0] = x[0] - x[1];//3 - 1 = 2, x[0] is now equal to the old value of
    x[1]
```

4 Methods

A method is a collection of statements that are grouped together to perform an operation¹. Methods allow for easy reuse of code without having to rewrite functions multiple times.

4.1 Structure of a Method

Here are declarations and implementation for three different methods:

```
public static String combineStrings(String s1, String s2)
{
    //Create a string that is the combination of s1 and s2
    return s1 + s2;
}
public static void count(int n)
{
    //print all of the numbers from 0 to n
    for(int i = 0; i <= n; i++)
        System.out.print(i + " ");
}
public static int three()
{
    //return the number 3
    return 3;
}
```

Components of these methods:

- 1: public: methods that are public can be accessed by any other method. Private methods won't be used in APCS.
- 2: static: static methods can be called without an object (covered in section 5). This section will focus on static methods.
- 3: return type: Data type for the variable output of the function
- 4: name: how a function is called. A function name cannot start with a number or symbol other than _
- 5: parameters: variables that are passed into and used by a function
- 6: body: where the operations within a function take place
- 7: return statement: terminating statement for a function resulting in a variable output

¹https://www.tutorialspoint.com/java/java_methods.htm

4.2 Calling Methods

This program shows a declaration for the methods `add1()` and `add2()`, and calling them from the main method: Public methods can be called from any-

```
1 import java.util.*;
2 public class methods{
3 {
4     public static int add1(int x)
5     {return x+1;}
6     public static int add2(int x)
7     {return add1(x)+1;}
8     public static void main(String[] args)
9     {
10         int y = 3, z = 5;
11         add1(y);
12         z = add2(z);
13         System.out.println("y+1= "+y+" z+2= "+z);
14     }
15 }
```

```
U: **- methods.java All L2 <I> (Java/l Abbrev)
172 brandon@brandon-Virtual-Machine:~/Desktop/java$ javac methods.java
173 brandon@brandon-Virtual-Machine:~/Desktop/java$ java methods
174 y+1= 3 z+2= 7
```

Figure 5: Output: $y+1 = 3$, $z+2 = 7$

where within the class braces with the call `name(param1, param2,...)`. The next section will cover why `y` is equal to 3.

4.3 Variable Scope

When a method is called that takes in parameters, a copy of the variable is made by the method that is then modified, potentially returned by the function, then removed. The same is true with variables created within a method body. Variables are scoped to the function that are created within, meaning that they cannot exist outside of them. For that reason [objects](#) and global variables are used.

4.4 Global Variables

Global variables are variables that exist within the class itself that can be accessed by all static (for static global variables) or all non static (for non static global variables) methods. They are declared in the same way as all other variables, except static global variables must be explicitly labeled as static. Here is an example class that uses a global variable: Because `globalInt` is a global static variable, its value can be changed by any static method. However, if `globalInt` was passed in as a parameter, its value could not be changed.

4.5 Return Type and Return Statement

Each method can return one variable (you can return more data by using an array or array list), and the data type of that variable is decided in the heading


```

1 import java.util.*;
2
3 public class global
4 {
5     static int counter = 0; // gloabl static int
6     public static int add(int x, int y)
7     {
8         count();
9         return x+y;
10    }
11    private static void count() // count number of addition operations
12    { counter++; } // increment counter
13    public static void main(String[] args)
14    {
15        for(int i = 0; i < 100; i++)
16            add(0,0);
17        System.out.println(counter);
18    }
19 }

```

U:--- gloabl.java All L1 <N> (Java/L Abbrev)

```

221 brandon@brandon-Virtual-Machine:~/Desktop/java$ java global
222 100

```

Figure 6: Output:100

of the method. In addition, there are **void** methods that do not return any variable. In addition, the returns statement terminates the method and removes all variables scoped to the method from memory. A function may have more than one return statement, but only one can be called for a given method call. Here are three different implementations for the function **greaterThan3(int x)**). They are all equivalent:

```

public static boolean greaterThan3(int x)
{
    if(x > 3)
        return true;
    else
        return false;
}
public static boolean greaterThan3(int x)
{
    if(x > 3)
        return true;
    return false;
}
public static boolean greaterThan3(int x)
{
    return x > 3;
}

```

Both the first and the second implementation have more than one return statement, but only one can be reached because the method exits when a return statement is called. For that reason, the **else** is not needed, and the second implementation is better than the first. However, in this case, the third is the best implementation. In this example:

```
public static String numberProperties(int x)
{
    if(x % 2 == 0)
        return "x is an even number";
    if(x > 10)
        return "x is greater than 10";
    if(Math.sqrt(x) == (int)Math.sqrt(x))
        return "x is a perfect square";
    return "none of these properties are true for x";
}
```

only one of these properties can be output because the method terminates as soon as any of these properties is true. One benefit of this is that you don't have to explicitly write **else** or **else if**. One work around is adding these strings to an ArrayList of properties:

```
public static ArrayList<String> numberProperties(int x)
{
    ArrayList<String> props = new ArrayList<String>();
    if(x % 2 == 0)
        props.add("x is an even number");
    if(x > 10)
        props.add("x is greater than 10");
    if(Math.sqrt(x) == (int)Math.sqrt(x))
        props.add("x is a perfect square");
    if(props.isEmpty())
        props.add("none of these properties are true for x");
}
```

4.6 Recursive Methods

In addition to calling other methods, a method can call itself; these methods are known as **recursive methods**. They consist of a general case, and a base case. The base case comes after condition, and causes the recursion to stop. The general case is what occurs if that condition is not true. This is a recursive implementation of an exponent method:

```
public static int exponent(int x, int y)
{
    //recursively compute x^y
    //assume y > 0
}
```

```

//base case
if(y == 1)
    return x;

//general case
return x * exponent(x, y - 1);
}

```

Tracing **exponent(2,4)**:

- 1: $\text{exponent}(2,4) = 2 * \text{exponent}(2,3)$
- 2: $\text{exponent}(2,3) = 2 * \text{exponent}(2,2)$
- 3: $\text{exponent}(2,2) = 2 * \text{exponent}(2,1)$
- 4: $\text{exponent}(2,1) = 2$, base case
- 5: $\text{exponent}(2,2) = 2 * 2 = 4$
- 6: $\text{exponent}(2,3) = 2 * 4 = 8$
- 7: $\text{exponent}(2,4) = 2 * 8 = 16$

If there is no base case (or the base case cannot be reached), the recursion will never stop, and the program will not terminate. The Fibonacci Series is the most common example for recursive functions. The value of the Fibonacci Series at n is given by:

$$\begin{cases} F_n &= F_{n-1} + F_{n-2} \\ F_1 &= 1 \\ F_0 &= 1 \end{cases}$$

From this equation, you can easily see that the top line represents the general case, and the bottom two are the base cases, thus, this can easily be converted into a recursive method:

```

public static int fibbo(int n)
{
    if(n == 1 || n == 0)
        return 1;
    return fibbo(n-1) + fibbo(n - 2);
}

```

Tracing **fibbo(5)**

- 1: $\text{fibbo}(5) = \text{fibbo}(4) + \text{fibbo}(3)$
- 2: $\text{fibbo}(4) = \text{fibbo}(3) + \text{fibbo}(2)$
- 3: $\text{fibbo}(3) = \text{fibbo}(2) + \text{fibbo}(1)$

- 4: $\text{fibbo}(2) = \text{fibbo}(1) + \text{fibbo}(0)$
- 5: $\text{fibbo}(1) = 1$, base case
- 6: $\text{fibbo}(0) = 1$, base case
- 7: $\text{fibbo}(2) = 1 + 1 = 2$
- 8: $\text{fibbo}(3) = 2 + 1 = 3$
- 9: $\text{fibbo}(4) = 3 + 2 = 5$
- 10: $\text{fibbo}(5) = 5 + 3 = 8$

4.7 Overloading Methods

A method can be defined multiple times with different parameter **types** or **numbers** (overloaded methods with differently named parameters of the same type are invalid). Here are four overloaded methods for getting the average of integers:

```
public static double avg(int x, int y)
{
    return (x + y) / 2.0;
}
public static double avg(int x, int y, int z)
{
    return (x + y + z) / 3.0;
}
public static double avg(ArrayList<Integer> list)
{
    int sum = 0;
    for(int i : list)
        sum += i;
    return sum / (double)list.size();
}
public static double avg(int[] list)
{
    int sum = 0;
    for(int i : list)
        sum += i;
    return sum / (double)list.length;
}
```

Each of these is valid, and they can all exist in the same class because they have different types and/or numbers of parameters. This method cannot exist in the same file:

```
public static double avfg(int x, int z)
{
```

```
    return (x + z) / 2.0;
}
```

because there already is an implementation that takes two integers. The compiler would not know which of the two to call. Overloads are useful when you either do not know the exact parameters for input, or you want to make some parameters optional.

5 Classes and Objects

Classes allow you to create your own object types that can be changed by user defined functions. Each .java file can define one object. Here is a .java file for an on object called **Obj**:

```
import java.util.*;

public class Obj
{
    private int x,y;//private variables
    public obj(int x, int y)//constructor
    {
        this.x = x; this.y = y;
    }
    //get methods
    public int getX()
    {
        return x;
    }
    public int getY()
    {
        return y;
    }
    //set methods
    public void setX(int x)
    {
        this.x = x;
    }
    public void setY(int y)
    {
        this.y = y;
    }
    //main
    public static void main(String[] args)
    {
        Obj o = new Obj(3, 5);
        o.getX();//3
        o.getY();//5
        o.setX(10);
    }
}
```

```
        o.getX();//10
    }
}
```

components of this class:

- 1: Private variables: Variables that cannot be accessed by normal means. Methods can be written to access private variables
- 2: Constructor: A function that is called when creating a new instance of the object. An object may have many different constructors
- 3: Get methods: Methods that return the value of private variables
- 4: Set methods: Methods that set the value of private variables

5.1 Why use objects?

- 1: Code reuse: Several instances of the same object can be created to save time and make more organized code.
- 2: Variable scope: Variables outside of objects cannot have their value changed by other functions. Variables inside objects can through set and other methods
- 3: Control: Set ways for accessing the data within an object

5.2 this

In a method, functions can be called on objects. The object the function is being called on can be accessed using **this**. It is almost never required to use **this** but it can make code more readable.

5.3 Example Object

Here is an object of a bank account with the functions: **withdraw**, **deposit**, **check balance**, and **interest**.

```
import java.util.*;

public class bankAccount
{
    private String id;
    private double balance;
    private double interest;
    public bankAccount(String id, float startingBalance, float interest)
    {
        this.id = id;
        this.balance = startingBalance;
        this.interdest = interest;
    }
}
```

```

    }
    public bankAccount(String id)//starting balance 0 and interest to 2%
    {
        this.id = id;
        this.balance = 0;
        this.interest = .02;
    }
    public String accountID()
    {
        return id;
    }
    public int checkBalance()
    {
        return balance;//can also use this.balance
    }
    public void withdraw(double amount)
    {
        if(amount > balance)//to prevent negative balance
        {
            System.out.println("Cannot withdraw more than $" + balance);
        }
        else
        {
            balance -= amount;
        }
    }
    public void deposit(double amount)
    {
        balance += amount;
    }
    public double interest(int periods)
    {
        //calculate account balance after a certain number of
        //interest periods using the compound interest formula
        return balance * Math.pow((1 + interest), periods);
    }
}

```

Within the main function, you can create instances of **bankAccount** and perform operations on **bankAccount** objects.

```

public static void main(String[] args)
{
    bankAccount b1 = new bankAccount("b1");
    bankAccount b2 = new bankAccount("b2", 1000, .02);
    b1 = new bankAccount("bank account 1", 0, .02)//changing the id
        without a changeId function
    ArrayList<bankAccount> accounts = new ArrayList<bankAccount>();
    //create 10 bank accounts within accounts
}

```

```

for(int i = 0; i < 10; i++)
{
    accounts.add(new bankAccount("account" + i, i * 100, i/100.0));
}
System.out.println("The balance of b1 is $" + b1.ckeckBalance());
b1.withdraw(1000000);
b1.deposite(1000);
b1.withdraw(100);
for(int i = 0; i < 10; i++)
{
    System.out.println("After " + i + " interest periods, account, " +
        accounts.get(i).accountID() + ", will have a balance of $" +
        accounts.get(i).interest(i));
    //...
    //After 3 interest periods, account, account 3, will have a
        balance of $...
}
}

```

5.4 Inheritance

Inheritance allows one class to use functions from another class to create hierarchies. In the hierarchy, the lower classes have all the functions/properties of the parent, and possibly more. Here is an example hierarchy:

5.4.1 Implementation

Each .java file can only hold one class, when the files are stored in the same folder, the relationship will be created automatically

```

public class vehicle
{
    private String name;
    public vehicle(int name)
    {this.name = name;}
    public void setName(String name)
    {this.name = name;}
    public void getName()
    {return name;}
}
public class car extends vehicle
{
    private String company; private int seats;
    public car(String name, Sring company, int seats)
    {
        super(name);
        this.company = company;
    }
}

```



```

        this.seats = seats;
    }
    //get set methods
}
public class bicycle
{
    public bicycle(String name, String company, int gears)
    {
        super(name);
        this.company = company;
        this.gears = gears;
    }
    //get set methods
}
public class sportsCar extends car
{
    public sportsCar(String name, String company, int seats, int
        topSpeed, boolean convertible)
    {
        super(name, company, seats);
        //other variables not shown
    }
}

```

Inheritance is created by adding **extends** into the class declaration followed by the **super** class. The constructor of the lower class includes all of the parameters of the lower class and possibly any additional parameters. The first line in the body must be **super(param1, param2, ...)** with all of the parameters of the higher class' constructor. This allows all of the super class' methods to be called.

6 AP Exam Free Response Questions

[Link](#)

note: For these questions, do only what is asked. Additional operations, such as printing, can actually take away points.

6.1 2012 q4

A grayscale image is represented by a 2-D rectangular array of pixels. A pixel is an integer value that represents a shade of gray. In this question, pixel values can be in the range from 0 through 255, inclusive. A black pixel is represented by a 0, and a white pixel is represented by 255.

The declaration of the **GrayImage** class is shown below. You will write two unrelated methods for the **GrayImage** class.

```

public class GrayImage

```

```

{
    public static final int BLACK = 0;
    public static final int WHITE = 255;

    //2-D array of values in the range [0,255]. Guaranteed to not be null
    private int[] [] pixelValues;

    //return: the total number of white pixels in the image
    //postcondition: this image has not been changed
    public int countWhitePixels()
    { /*to be implemented in part (a)*/}

    //process the image in row-major order and decreases the value of
    //each pixel by pixelValues[row + 2][col + 2]
    //where possible, if that location does not exist
    //do nothing. Any negative values should be set to BLACK
    public void processImagea()
    { /*to be implemented in part (b)*/}
}

```

6.1.1 Part A

Write the methods **countWhitePixels** that returns the number of pixels in the image that contain the value WHITE. For example, assume that **pixelValues** contains the following image.

255	184	178	84	129
84	255	255	130	84
78	255	0	0	78
84	130	255	130	84

A call to **countWhitePixels** method would return 5 because there are 5 entries that have the value WHITE.

Complete method **countWhitePixels** below.

```

/** @return the total number of white pixels in the image.
 * Postcondition: this image has not been changed.
 */
public int countWhitePixels()
{

```

}

6.1.2 Part B

write the method **processImage** that modifies the image by changing the values in the instance variable **pixelValues** according to the following description. The pixels in the image are processed one at a time in row-major order. Row-major order process the first row in the array from left to right and then processes the second row from left to right, continuing until all rows are processed from left to right. The first index of **pixelValues** represents the row number, and the second index represents the column number.

The pixel value at (row,col) is decreased by the value at (row+2,col+2) if such a position exists. If the result of subtraction is less than BLACK, the pixel is assigned the value of BLACK. You may assume that all the original values in the array are within the range [BLACK, WHITE] inclusive.

The following diagram shows the contents of the instance variable **pixelValues** before and after a call to **processImage**. The values shown in boldface represent the pixels that could be modified in a grayscale image with 4 rows and 5 columns.

before					after				
221	184	178	84	135	221	184	100	84	135
84	255	255	130	84	0	125	171	130	84
78	255	0	0	78	78	255	0	0	78
84	130	255	130	84	84	130	255	130	84

Information repeated from beginning of the question

```
public class GrayImage
public static final int BLACK = 0
public static final int WHITE = 255
private int[][] pixelValues
public void processImage()
```

Complete method **processImage** below

```
/* process this image in row-major order and decrease
the value of each pixel at position (row,col) by the
value of the pixel at (row+2,col+2) if it exists.
Resulting values that would be less than BLACK are
replaced by BLACK. Pixels for which there is no pixel
at position (row+2, col+2) are unchanged.
*/
public void processImage()
```

{

}

6.1.3 Analysis and Solution

Part A

This part is a very simple traversal through a 2-D array with a comparison to WHITE and an increment when the condition is true.

Part B

This part isn't very different than (a). The for-loop should either be modified to stop early, or a line should be added to make sure that you don't try to access values that don't exist in the 2-D array. The first method is faster (not that it matters for the AP exam). Also make sure to check for values less than BLACK after subtraction.

Solution

note: there are many possible solutions to every question. You are only responsible for writing what is in these two methods. In part (b) the for-loop stops too early so that time isn't wasted using if statements to reject values.

```

1 public class GrayImage
2 {
3     public static final int BLACK = 0;
4     public static final int WHITE = 255;
5     private int[][] pixelValues;
6     public GrayImage(int[][] pixelValues)
7     {
8         this.pixelValues = pixelValues;
9     }
10    public int countWhitePixels()
11    {
12        int whitePixels = 0;
13        for(int i = 0; i < pixelValues.length; i++)
14        {
15            for(int j = 0; j < pixelValues[0].length; j++)
16            {
17                if(pixelValues[i][j] == WHITE)
18                    whitePixels++;
19            }
20        }
21        return whitePixels;
22    }
23    public void processImage()
24    {
25        for(int i = 0; i < pixelValues.length - 2; i++)
26        {
27            for(int j = 0; j < pixelValues[0].length - 2; j++)
28            {
29                pixelValues[i][j] += pixelValues[i+2][j+2];
30                if(pixelValues[i][j] < BLACK)
31                    pixelValues[i][j] = BLACK;
32            }
33        }
34        //does not need to be included
35        //print array
36        for(int i = 0; i < pixelValues.length; i++)
37        {
38            for(int j = 0; j < pixelValues[0].length; j++)
39            {
40                System.out.print(pixelValues[i][j] + " ");
41            }
42            System.out.println();
43        }
44    }
45    public static void main(String[] args)
46    {
47        int[][] exampleA = new int[][]{
48            {255, 184, 178, 84, 129},
49            {84, 255, 255, 130, 84},
50            {78, 255, 0, 0, 78},
51            {84, 130, 255, 130, 84}
52        };
53        int[][] exampleB = new int[][]{
54            {221, 184, 178, 84, 135},
55            {84, 255, 255, 130, 84},
56            {78, 255, 0, 0, 78},
57            {84, 130, 255, 130, 84}
58        };
59        GrayImage g1 = new GrayImage(exampleA);
60        GrayImage g2 = new GrayImage(exampleB);
61        System.out.println(g1.countWhitePixels());
62        g2.processImage();
63    }
64 }

```

Figure 7: (a) 11-20 (b) 21-40

6.2 2017 11

6.2.1 Analysis and Solution

You can find this question from the link at the top of the section.

(a)

The first part requires setting the value of the ArrayList using the integer division operation. ArrayList.add(index, num) will cause num to be added at the specific index, and all of the other numbers to be pushed up. Use this behavior to put all the digits in the correct order even though you will add from the end of the number to the start. Also, remember to declare the ArrayList as a new ArrayList at the start. Finally, the example shows a case for 0, add a specific line for that case.

(b)

Part (b) is a simple traversal, either from the back to the front, or front to back. Make sure that you don't try to access an index that doesn't exist.

6.3 2017 q2

6.3.1 Analysis and Solution

This question is simply asking if you know how to use an interface. For the purpose of this class, they are pretty much useless except for the fact that the College Board wants you to know how to use them. To use an interface, add **implements** into your class declaration, and implement all of the functions of the interface in your class. The body of the class would have not been different if the interface did not exist. Other than that, this question is really easy. The first methods is concatenating a string (my solution uses **String.format()**, but

```

1 import java.util.*;
2
3 public class Digits
4 {
5     private ArrayList<Integer> digitList;
6
7     public Digits(int num)
8     {
9         digitList = new ArrayList<Integer>();
10        if(num == 0)
11        {
12            digitList.add(0);
13            return;
14        }
15        while(num > 0)
16        {
17            digitList.add(0, num % 10);
18            num = num / 10;
19        }
20    }
21
22    public boolean isStrictlyIncreasing()
23    {
24        int i = digitList.size() - 1;
25        while(i > 0)
26        {
27            if(digitList.get(i) <= digitList.get(i - 1))
28                return false;
29            i--;
30        }
31        return true;
32    }
33
34    public void print()
35    {
36        for(Integer i : digitList)
37            System.out.format("%d ", i);
38        System.out.println(isStrictlyIncreasing());
39    }
40
41    public static void main(String[] args)
42    {
43        new Digits(77).print();
44        new Digits(1356).print();
45        new Digits(1336).print();
46    }
47 }

```

```

94 brandonw@brandonw-Virtual-Machine:~/Desktop/java/apcs$ java Digits
95 7 7 true
96 1 3 5 6 true
97 1 3 3 6 false
98 1 5 3 6 false
99 6 5 3 1 0 false
100 brandonw@brandonw-Virtual-Machine:~/Desktop/java/apcs$

```

Figure 8: (a) 7-21 (b) 22-38

you can use the + method as well). The second part is simply changing an object's variable.

6.4 2017 q3

6.4.1 Analysis and Solution

This question is testing your knowledge of string operations. For this question, assume that **findNthOccurrence** works, and use it in your answers.

(a)

This part entails using the substring operation to remove part of the string and replace it with the input. For this part, test your code with the provided examples after you answer to make sure that you are starting at the correct index for the start of the second part of the string.

```

//solution:
index = findNthOccurrence(str, n);
CurrentPhrase =
    currentPhrase.substring(0, index) + // part before replacement
    repl + //inserted string
    currentPhrase.substring(index + str.length(), currentPhrase.length());

```

for the string str = "abc_abc_abc":

0	1	2	3	4	5	6	7	8	9	10
a	b	c	_	a	b	c	_	a	b	c

```

1 import java.util.*;
2
3 public interface StudyPractice
4 {
5     String getProblem();
6     void nextProblem();
7 }
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25 }

```

```

137 brandonw@b
138 brandonw@b
139 7 TIMES 3
140 7 TIMES 4
141 7 TIMES 5
142 brandonw@b

```

```

--:-- StudyPractice.java All L1 <N> (Java/l Abbrev)
1 public class MultPractice implements StudyPractice
2 {
3     private int x,y;
4     public MultPractice(int x, int y)
5     {
6         this.x = x; this.y = y;
7     }
8     public String getProblem()
9     {
10         return String.format("%d TIMES %d",x,y);
11     }
12     public void nextProblem()
13     {
14         y++;
15     }
16     public static void main(String[] args)
17     {
18         StudyPractice p1 = new MultPractice(7,3);
19         System.out.println(p1.getProblem());
20         p1.nextProblem();
21         System.out.println(p1.getProblem());
22         p1.nextProblem();
23         System.out.println(p1.getProblem());
24     }
25 }

```

```

Phrase p = new Phrase("abc_abc_abc");
//operations in the solution
int index = p.findNthOccurrence("abc", 2); //4
string s1 = currentPhrase.substring(0, 4 /*index*/); //"abc_"
String s2 = repl;
Setring s3 = currentPhrase.substring(4 + 3, 11); //"_abc"
return s1 + s2 + s3; //"abc_cde_abc"

```

(b)

This part is simply using the provided function until there are no matches (-1) returned. The easiest method to implement this is with a while loop:

```

while(there is a match for given 'n')
    n++
\\last call to findNthOccurrence returns -1
return the value of the call before -1 was returned

```

Solution

7 2017 q4

7.0.1 Analysis and Solution

This is the 2D array question for the 2017 test; each test has one. For the first part, you simply have to traverse the array, make a comparison, and create and

```

import java.util.*;

public class Phrase
{
    private String currentPhrase;

    public Phrase(String p)
    { currentPhrase = p; }

    public int findNthOccurrence(String str, int n)
    {
        int matches = 0;
        for(int i = 0; i <= currentPhrase.length() - str.length(); i++)
        {
            if(currentPhrase.substring(i, i + str.length()).equals(str))
            {
                matches++;
                if(matches == n)
                    return i;
            }
        }
        return -1;
    }

    public void replaceNthOccurrence(String str, int n, String repl)
    {
        int index = findNthOccurrence(str, n);
        if(index == -1) //if there is no match
            return;
        currentPhrase = currentPhrase.substring(0, index) + repl + currentPhrase
            .substring(index + str.length(), currentPhrase.length());
        System.out.println(currentPhrase);
    }

    public int findLastOccurrence(String str)
    {
        //this is a pretty stupid way to solve this part
        int n = 1;
        while(findNthOccurrence(str, n) != -1)
        {
            n++;
        }
        return findNthOccurrence(str, n - 1);
    }

    if(index == -1) //if there is no match
    {
        return;
    }
    currentPhrase = currentPhrase.substring(0, index) + repl + currentPhrase
        .substring(index + str.length(), currentPhrase.length());
    System.out.println(currentPhrase);
}

public int findLastOccurrence(String str)
{
    //this is a pretty stupid way to solve this part
    int n = 1;
    while(findNthOccurrence(str, n) != -1)
    {
        n++;
    }
    return findNthOccurrence(str, n - 1);
}

public String toString()
{
    return currentPhrase;
}

public static void main(String[] args)
{
    Phrase p = new Phrase("abc_abc_abc");
    System.out.println(p.findLastOccurrence("abc"));
    p.replaceNthOccurrence("abc", 2, "cde");
    System.out.println(p.findLastOccurrence("cdefg"));
    p = new Phrase("aaaa");
    p.replaceNthOccurrence("aa", 1, "xx");
    p = new Phrase("aaaa");
    p.replaceNthOccurrence("aa", 2, "bbb");
}

```

Figure 9: On your solution, do not include any print operations

return an object. In the second part, you can assume that your answer to the first is correct. In (b) you have to call **findPosition** on each element of the 2D array to fill a 2D array of positions.

Solution


```

public static Position findPosition(int num, int[][] arr)
{
    for(int i = 0; i < arr.length; i++)
        for(int j = 0; j < arr[0].length; j++)
            if(arr[i][j] == num)
                return new Position(i,j);
    return null;
}
public static Position[][] getSuccessorArray(int[][] arr)
{
    Position[][] ret = new Position[arr.length][arr[0].length];
    for(int i = 0; i < arr.length; i++)
    {
        for(int j = 0; j < arr[0].length; j++)
        {
            ret[i][j] = findPosition(arr[i][j] + 1, arr);
        }
    }
    printPositionArr(ret);
    return ret;
}

public static void main(String[] args)
{
    int[][] i ={
        {15,5,9,10},
        {12,16,11,6},
        {14,8,13,7}
    };
    findPosition(7, i);
    findPosition(3,i);
    findPosition(5, i);
    findPosition(8,i);
    getSuccessorArray(i);
}

public static void printPositionArr(Position[][] arr)
{
    for(int i = 0; i < arr.length; i++)
        for(int j = 0; j < arr[0].length; j++)
            System.out.print("(" + arr[i][j].i + "," + arr[i][j].j + "); ");
    System.out.println();
}
}

```

--- Position.java 12% L31 <N> (Java/l Abbrev)
 3 brandonw@brandonw-Virtual-Machine:~/Desktop/java/apcs\$ java Position
 4 (1,1); (1,3); (0,3); (1,2);
 5 (2,2); null; (1,0); (2,3);
 6 (0,0); (0,2); (2,0); (2,1);