UNIT 1
Primitive Types

16. (C)

A common mistake for a question like this is to simply write the numerator followed by a division slash (`/`), followed by the denominator. The problem with that is that according to *Order of Operations* you are simply dividing the last value of the numerator by the first value of the denominator. This is what happens in answer choice **D**, which only divides `b` by `3`. What we want is to divide the entire numerator by the entire denominator.

This requires putting both the entire numerator and the entire denominator in parentheses. Remember also that there is no *implied multiplication* in Java. Anytime you want to multiply, you must use the asterisk (*) operator.

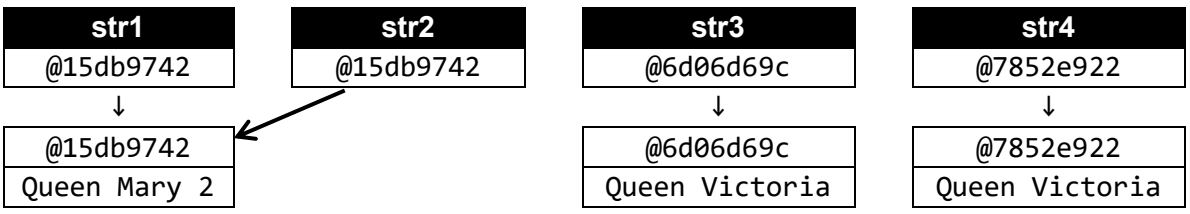
UNIT 2
Using Objects

30. (A)

This question may look simple but there is actually quite a bit going on here and unless you have a very good understanding of strings and objects, you can easily get tricked into choosing a wrong answer. First, you need to understand that the *is equal to* (==) operator does not do the same thing as the `equals` method. The *is equal to* (==) operator is normally used to compare *primitive data types* (`int`, `double`, `boolean`) like the 3 integers below. In this case, the statement `x == y` would have a value of `false` and the statement `x == z` would have a value of `true`.

x	y	z
100	200	100

Unlike primitive data type, objects do not store the data directly. Instead, they store the location or memory address of the data. In the diagram below we see all four string variables from this program. When strings are simply assigned to *string literals*, as is done with `str1` and `str2` in the program, they become part of the *string pool*. Any string variables that store the same string value will actually use the same address. When strings are created with the `new` operator, a new string object is created at a new memory address.



The statement `str3 == str4` would have a value of `false` and the statement `str3.equals(str4)` would have a value of `true`. The reason is the *is equal to* (==) operator compares the *shallow values* (memory addresses) while the `equals` method compares the *deep values* (actual data). When comparing str1 and str2, both `str1 == str2` and `str1.equals(str2)` have a value of `true` because they are actually sharing the same string object.

UNIT 3
Boolean Expressions and if Statements

25. (E)

De Morgan's Law states that:

NOT(A AND B) is equivalent to **NOT(A) OR NOT(B)**

According to De Morgan's Law, expressions `x` and `y` are logically equivalent, which means that the value of `z` must be true.

UNIT 4
Iteration

07. (D)

In this program, the `while` condition is `n < 1000`. At the beginning of the program, `n` is assigned a value of `0`, so the program does enter the `while` loop; however, there is nothing in the `while` loop that changes the value of `n`. Since `n` will always be `0`, the condition `n < 1000` will always be `true` and the `while` loop will never end.

UNIT 5
Writing Classes

25. (E)

The `Fraction` class has two attributes: `num` is used for the fraction's numerator and `den` is used for the fraction's denominator. `Fraction` object `f1` stores 3/4; `Fraction` object `f2` stores 2/3; and `Fraction` object f3 initially stores 1/1. Then `f3` calls the `multiply` method with `f1` and `f2` as parameters.

The `multiply` method multiplies the `numerators` of `f1` and f2. The product becomes the new `numerator` of `f3`. It also multiplies the `denominators` of `f1` and `f2`. This product becomes the new denominator of `f3`. Essentially, `Fraction` object `f3` is the product of `f1` and `f2`.

When 3/4 is multiplied by 2/3, the result is 6/12. Be careful here. The temptation may be to select answer choice **D** because `6/12` reduces to `1/2`; however, there is nothing in the Fraction class that reduces the fraction. This means `6/12` is the final answer.

UNIT 6
Array

13. (E)

First, we need to remember that in Java, anytime you divide an integer by another integer, you get *integer division* where the result is always an integer. This is why answer choice **A** will not work. If one of the two numbers, like `sum`, is a real number (`double`) variable, then you will get real number division.

Second, answer choice **C** will not work because the loop starts on index `1`. This means it does not add the very first number in the list (the one at index `0`) so the mean is not accurate.

While answer choice **B** uses an *enhanced* `for` loop and answer choice **D** uses a *traditional* `for` loop, both properly add all of the numbers in the array and then use real number division to properly compute the mean.

UNIT 7
ArrayList

16. (A)

This one can easily trick you into thinking the answer is choice **E**. It starts by copying the contents of the `list1` array to the `list2 ArrayList` in a manner similar to what is done in the past few questions. After that, it seems like every element in the `list2 ArrayList` is removed; however, this is not what happens.

Right before the second `for` loop begins, `list2` stores `[11, 22, 33, 44, 55, 66, 77]`, which means `list2.size()` equals `7`. After that, if we trace through what happens to `list2` with each iteration of the second for loop, we get:

k	Removed item at index **k**	**list2**	**list2.size()**
`0`	`11`	`[22, 33, 44, 55, 66, 77]`	`6`
`1`	`33`	`[22, 44, 55, 66, 77]`	`5`
`2`	`55`	`[22, 44, 66, 77]`	`4`
`3`	`77`	`[22, 44, 66]`	`3`

After the first iteration, the first item – the one at index `0` – is removed. This is expected; however, in the second iteration, when the second item – the one at index `1` – is removed, it is the `33`, not the `22` that gets deleted. This is because `22` is no longer at index 1. It is now at index 0 because the 11 was deleted.

Also, even though the `ArrayList` had 7 items and the value of `list2.size()` was `7` before the second for loop began, the second for loop will <u>not</u> repeat `seven` times. This is because as we keep removing items from `list2`, its `size` keeps decreasing and after only four iterations the statement `k < list2.size()` becomes `false`.

UNIT 8
2D Array

08. (C)

In this question, you need to ignore the arbitrary names given to the "rows". They are misleading because `row1` is actually at index `0`; `row2` is actually at index `1` and is `row3` is actually at index `2`.

Remember that a 2D `ArrayList` is really a 1D `ArrayList` of 1D `ArrayLists`. To access the `500`, we first need to access the "row" at index `1`. Within that row, we need to access the item at index `1`.

In answer choice **C**, the first `.get(1)` retrieves the proper row and the second `.get(1)` retrieves the proper item within that row.

UNIT 9
Inheritance

05. (A)

This is similar to the first question in that it reemphasizes the point that anytime an object of a subclass is called; the constructor of the superclass is called first. The difference now is we have *Multi-Level Inheritance*. In this code segment `Class3 extends Class2` and `Class2 extends Class1`.

When a `Class1` object is constructed, we simply get the `Class1` constructor. When a `Class2` object is constructed, get the `Class2` constructor followed by the `Class1` constructor. When a `Class3` object is constructed, get the `Class3` constructor followed by the `Class2` constructor followed by the `Class1` constructor.

UNIT 10
Recursion

09. (E)

The `mystery` method will recursively traverse the array from the first element all the way to the second to last element. Each of these elements, `list[q]`, is then compared to the next element, `list[q+1]`.

If the current element, `list[q]`, is greater than the next element, `list[q+1]`, the two elements are swapped. The effect of this traversing and swapping will change the position of some of the array elements.

One particular element, the one with the largest value, will be swapped all of the way to the end of the array.

Free-Response Samples

Question 1. (Methods and Control Structures)

Part (a).

```
/** Precondition:  n1 and n2 are positive integers.
 *  Postcondition: returns the GCF of n1 and n2.
 */
public static int getGCF (int n1, int n2)

/** Precondition:  n1 and n2 are positive integers.
 *  Postcondition: returns the GCF of n1 and n2.
 */
private int getGCF(int n1, int n2)
{
   int rem = 1;
   int gcf = 1;
   while (rem != 0)
   {
      rem = n1 % n2;
      if (rem == 0)
      {
         gcf = n2;
      }
      else
      {
         n1 = n2;
         n2 = rem;
      }
   }
   return gcf;
 }
```

Part (b).

```
/** Precondition: n1 and n2 are positive integers.
 *                getGCF is available to compute the LCM of n1 and n2.
 * Postcondition: getLCM returns the LCM of n1 and n2.
 */

public static int getLCM(int n1, int n2)
{
   int gcf = getGCF(n1,n2);
   int lcm = n1 / gcf * n2;
   return lcm;
```

Question 2. (Class)

Part (a).

```
/** Precondition:  fract1 and fract2 are objects with attributes initialized.
 *  Postcondition: the results of adding fract1 to fract is stored in "this".
 */

public void add(Fraction fract1, Fraction fract2)
{
   int tempNum = fract1.num * fract2.den + fract2.num * fract1.den;
   int tempDen = fract1.den * fract2.den;
   int gcf = getGCF(tempNum,tempDen);
   this.num = tempNum / gcf;
   this.den = tempDen / gcf;
}
```

Part (b).

```
/** Precondition:  fract1 and fract2 are objects with attributes initialized.
 *  Postcondition: the results of adding fract1 to fract is stored in "this".
 */

public String toString()
{
   String fraction = num + "/" + den;
   return fraction;
}
```

Question 3. (Array/ArrayList)

Part (a).

```
/** Precondition:  list is a non-empty static array of random integers
 *  Postcondition: returns the mean of the numbers in list.
 */
public static double getMean(int[] list)
{
   int sum = 0;
   for (int k = 0; k < list.length; k++)
         sum += list[k];
      double mean = (double) sum / list.length;
      return mean;
}
```

Part(b).

```
/** Precondition:  list is a non-empty static array of random integers
 *  Postcondition: returns the Standard Deviation of the numbers in list.
 */
public static double getStdDev(int[] list)
{
   double mean = getMean(list);
   int n = list.length;
   int[] squaredDiffs = new int[n];
   double diffSum = 0.0;
   for (int k = 0; k < n; k++)
   {
      double diff = list[k] - mean;
      double sqDiff = diff * diff;
      diffSum += sqDiff;
   }
   double stdDev = Math.sqrt(diffSum / (n-1));
   return stdDev;
}
```

Question 4.

```
/** Precondition:  m1 and m2 are 2D static arrays with unknown int values.
 *                The number of rows of m1 are equal to the number of columns
 *                of m2, which allows matrix multiplication.
 *  Postcondition: returns a product matrix that uses the proper mathematical
 *                rules of matrix multiplication.
 */
public static int[][] multiply(int[][] m1, int[][] m2)
{
   int rows = m1.length;
   int cols = m2[0].length;
   int[][] product = new int[rows][cols];
   for(int r = 0; r < rows; r++)
   {
      for(int c = 0; c < cols; c++)
      {
         int sum = 0;
         for(int k = 0; k < m1[0].length; k++)
         {
            sum = sum + m1[r][k] * m2[k][c];
         }
         product[r][c] = sum;
      }
   }
   return product;
}
```