**Reviewing Test**

CS 1A Final Exam

|  |  |
| --- | --- |
| Started: | Dec 8, 2014 8:33 PM |
| Finished: | Dec 8, 2014 10:33 PM - auto |

 1 of 1

**CS 1A Final Exam     Total Grade: 34.02   (of possible 39.96 points)**

Finish this test before the due date, ***Tuesday, 12/9 11:59 PM***.  Once you begin, you will have 120 minutes to complete.   If you do not submit before that time your incomplete exam will be automatically submitted as is.

You can look at lectures or texts and even use your compiler, but you may not consult any other individuals for help.

Each question is worth 1.08 points, for an exam total of about 40 points.

Multiple choice questions with square check-boxes may have more than one correct answer.  Multiple choice questions with round radio-buttons have only one correct answer.

Any code fragments you are asked to analyze are assumed to be contained in a program that has all the necessary variables defined and/or assigned.

Your Exam has been accepted. Thank you.

(Any further attempts to resubmit this exam will result in an error.)

**Question 1 of 37     Score: 1.08   (of possible 1.08 points)**

Consider the following method definition:

int update( int userIn )

{

userIn++;

return userIn;

}

Which of the following client calls will always result in the variable **clientVal**being incremented by exactly 1?

|  |  |  |
| --- | --- | --- |
|  | A. | clientVal = update( y ); |
|  | B. | y = update( clientVal ); |
| Correct | C. | clientVal = update( clientVal ); |
|  | D. | update( clientVal ); |

 Answer Key: C

**Review**

 Check to review **before** finishing (will be flagged in Table of Contents)

**Feedback**

Since the client argument passed into the method sets the value of  the *local variable,***userIn***,* which is *distinct*from that client variable, the method statement**userIn++** *does not affect*the client argument directly.  It is only the***return value*** that affects the client value, and that happens only if the client variable "catches" the result as a functional return.  Since we are asking that the method increment the client variable **clientVal**, we have to both

1. pass **clientVal**to the method, and
2. capture the return into that same variable, **clientVal**

in order to get a reproducible "increment" out of the method.

**Question 2 of 37     Score: 0   (of possible 1.08 points)**

Consider the class as partially defined here:

//class PizzaOrder ------------------------------------

class PizzaOrder

{

// static public members

public static final int MAX\_TOPPINGS = 20;

// instance members

private String toppings[];

private int numToppings;

// constructor

public PizzaOrder()

{

numToppings = 0;

toppings = new String[MAX\_TOPPINGS];

}

// accessor tells # toppings on pizza, i.e., #toppings in array

public int getNumToppings() { return numToppings; }

// etc.

}

We want to write an instance method,**addTopping()** that will take a **String**parameter, **topping**, and add it to the **toppings[]** array at the next available location as determined by the int member **numToppings**.    Assume any **String** passed in is valid and no other method modifies the**toppings[]**array or **numToppings**.  
  
 The client would call it like so:

somePizzaOrder.addTopping( "onions" );

Which is a correct definition for **addTopping()** based on what you see, above?

|  |  |  |
| --- | --- | --- |
|  | A. | public void addTopping( String topping )     {        toppings[numToppings] = topping;     } |
|  | B. | public void addTopping( String topping )     {        toppings[numToppings++] = topping;     } |
|  | C. | public boolean addTopping( String topping )     {       numTopping++;       if ( numToppings >= MAX\_TOPPINGS )          return false;       toppings[numToppings] = topping;       return true;     } |
| Incorrect | D. | public boolean addTopping( String topping )     {       if ( numToppings >= MAX\_TOPPINGS || numToppings < 0 )          return false;       toppings[numToppings++] = topping;       return true;     } |
|  | E. | public boolean addTopping( String topping )     {       if ( numToppings > MAX\_TOPPINGS )          return false;       toppings[numToppings++] = topping;       return true;     } |
|  | F. | public boolean addTopping( String topping )     {       if ( numToppings >= MAX\_TOPPINGS )          return false;       toppings[numToppings++] = topping;  return true;     } |

 Answer Key: F

**Review**

 Check to review **before** finishing (will be flagged in Table of Contents)

**Feedback**

We have to make sure that **numTopping**is not**>= MAX\_TOPPING**, but besides that, other tests are not useful based on the information provided.  We must increment**numTopping**each time a topping is successfully added.  Incrementing **numToppings**should only happen if there is room to add a topping, otherwise the accessor will return an inaccurate # of toppings.

**Question 3 of 37     Score: 1.08   (of possible 1.08 points)**

A ***linear search algorithm*** is written (as in the modules, for example) which searches an array for some user-defined value, **clientData**.  If **clientData**is stored in the array, it returns its array position, and if not found, it returns -1 (again, just like in the modules).  Assume the array to be searched has 100 data elements in it.   (Check *all*that apply):

*[NOTE: due to common off-by-one interpretations when counting such things, if your predicted answer is within one (+1 or -1) of a posted option below, you can assume your prediction and the choice you are looking at are equivalent and check that option.]*

|  |  |  |
| --- | --- | --- |
| Correct | A. | It ***may*require*as many as 100***comparisons of **data**before it returns. |
| Correct | B. | It ***will never require more than 100***comparisons of **data**before it returns. |
| Correct | C. | It ***might***return to the client with an answer after ***only one*** comparison of **data.** |
|  | D. | It ***will always*** return with an answer ***in 10 or fewer***comparisons of **data.** |
|  | E. | It ***will always*** return with an answer ***in 50 or fewer***comparisons of **data.** |

 Answer Key: A,B,C

**Feedback**

The data may be in position 0, in which case it only requires one comparison (of data), so it may return after about one comparison.  At the other extreme, the data may be in position 99, or not in the array at all.  In those cases it will require about 100 comparisons (of data), but never more.  So 100 is the most it would require.

**Question 4 of 37     Score: 1.08   (of possible 1.08 points)**

Here is an outline for a working class **OldCellPhone**that has no errors*(you may see this****exact class****several times in this exam)*:

class OldCellPhone

{

   static public final int MIN\_CAP = 10;

   static public final int MAX\_CAP = 1000;

   static public final String DEFAULT\_DSCR = "(generic phone)";

   private String description;

   private int   memCapacity;

   private boolean   camera;

   private boolean   gps;

   public void setCamera( boolean hasCam ) { camera = hasCam; }

   public void setGps( boolean hasGps ) { gps = hasGps; }

   public OldCellPhone()

   {

      this(DEFAULT\_DSCR, MIN\_CAP, false, false);

   }

   public OldCellPhone(String dscr, int mem, boolean cam, boolean gp)

   {

// not shown

   }

   public String toString()

   {

// not shown

   }

   public boolean setMemCapacity(int mem)

   {

// not shown

   }

};

A programmer decides to add a method**addMemory()** whose job it will be to increase the private int **memCapacity** by an amount specified by the client as long as the new, increased memory capacity after increasing it is still **<= MAX\_CAP**.  Otherwise, it will not touch **memCapacity**.  
  
Check the true statements (there may be more than one correct answer):

|  |  |  |
| --- | --- | --- |
| Correct | A. | This should be an***instance method***. |
|  | B. | This should be a ***static method***. |
| Correct | C. | If correctly defined, a client could add 30 units of memory to the **OldCellPhone**object, **myCell**, by using the syntax:  myCell.addMemory( 30 ); |
|  | D. | If correctly defined, a client could add 30 units of memory to the **OldCellPhone**object, **myCell**, by using the syntax:  OldCellPhone.addMemory( 30 ); |
|  | E. | If correctly defined, a client could add 30 units of memory to the **OldCellPhone**object, **myCell**, by using the syntax:  OldCellPhone.addMemory( myCell(30) ); |

 Answer Key: A,C

**Feedback**

The method signature should be **boolan addMemory( int memToAdd )**and it should be a ***public instance method***, as it modifies private data and has a chance to detect an error (no room to add more memory) so must return a **boolean**.  The only way a client could modify the object **myCell**using this method would be to use**myCell.addMemory( 30 );**  Other attempts either have inadequate information or are unreasonable and invalid calls no matter how **addMemory ()** was designed.

**Question 5 of 37     Score: 0   (of possible 1.08 points)**

When an object reference is passed as a parameter to a method, modifying the members of that object from inside the method will not result in a change to that object as seen from the client.

Incorrect True

 False

 Answer Key: False

**Feedback**

In Java, objects are passed by reference; the actual object stays where it is (in the client memory space) when passed to a method, but the reference, sent to the method, allows the method to modify that object from inside that method.  Changes to the object members "persist" when control returns to the caller (client).

**Question 6 of 37     Score: 0   (of possible 1.08 points)**

Here is an outline for a working class **OldCellPhone**that has no errors*(you may see this****exact class****several times in this exam)*:

class OldCellPhone

{

   static public final int MIN\_CAP = 10;

   static public final int MAX\_CAP = 1000;

   static public final String DEFAULT\_DSCR = "(generic phone)";

   private String description;

   private int   memCapacity;

   private boolean   camera;

   private boolean   gps;

   public void setCamera( boolean hasCam ) { camera = hasCam; }

   public void setGps( boolean hasGps ) { gps = hasGps; }

   public OldCellPhone()

   {

      this(DEFAULT\_DSCR, MIN\_CAP, false, false);

   }

   public OldCellPhone(String dscr, int mem, boolean cam, boolean gp)

   {

// not shown

   }

   public String toString()

   {

// not shown

   }

   public boolean setMemCapacity(int mem)

   {

// not shown

   }

};

A programmer decides to make the static **DEFAULT\_DSCR** ***mutable***, removing its **final**status, as in:

static public String DEFAULT\_DSCR = "(generic phone)";

A ***mutator***is added for this member. Any **String**whose length is at least 2 characters will be acceptable as a new **DEFAULT\_DSCR**.   
  
Check the true statements (there may be more than one correct answer):

|  |  |  |
| --- | --- | --- |
| Incorrect | A. | The mutator for this member will be an***instance method***. |
| Missed Correct Answer | B. | The mutator for this member will be a ***static method***. |
|  | C. | If the mutator takes a **String**parameter, **newDefault**, a reasonable definition would be:  if ( newDefault.length() < 2 )  {  this.newDefault = DEFAULT\_DSCR;  return false;  }  this.newDefault = newDefault;  return true; |
| Correct | D. | If the mutator takes a **String**parameter, **newDefault**, a reasonable definition would be:  if ( newDefault.length() < 2 )  return false;  DEFAULT\_DSCR = newDefault;  return true; |
|  | E. | If the mutator takes a **String**parameter, **newDefault**, a reasonable definition would be:  if ( newDefault.length() > 0 )  return false;  DEFAULT\_DSCR = newDefault;  return true; |

 Answer Key: B,D

**Feedback**

It's a ***static member*** so its must must be a***static method***.  The only mutator that tests the parameter and makes a reasonable assignment to the member**DEFAULT\_DSCR** is the one code fragment marked.

**Question 7 of 37     Score: 1.08   (of possible 1.08 points)**

Here is an outline for a working class **OldCellPhone**that has no errors*(you may see this****exact class****several times in this exam)*:

class OldCellPhone

{

   static public final int MIN\_CAP = 10;

   static public final int MAX\_CAP = 1000;

   static public final String DEFAULT\_DSCR = "(generic phone)";

   private String description;

   private int   memCapacity;

   private boolean   camera;

   private boolean   gps;

   public void setCamera( boolean hasCam ) { camera = hasCam; }

   public void setGps( boolean hasGps ) { gps = hasGps; }

   public OldCellPhone()

   {

      this(DEFAULT\_DSCR, MIN\_CAP, false, false);

   }

   public OldCellPhone(String dscr, int mem, boolean cam, boolean gp)

   {

// not shown

   }

   public String toString()

   {

// not shown

   }

   public boolean setMemCapacity(int mem)

   {

// not shown

   }

};

A programmer decides to add a method **add20ToMemory()** whose job it will be to increase the private int **memCapacity** by **20**as long as the new, increased memory capacity, after adding 20, is still **<= MAX\_CAP.** Otherwise, it will not touch **memCapacity**.  
  
Check the true statements (there may be more than one correct answer):

|  |  |  |
| --- | --- | --- |
| Correct | A. | This should be an ***instance method***. |
|  | B. | This should be a ***static method***. |
| Correct | C. | If the programmer chose to make it an***instance method***, it would not need any parameters to do its job. |
|  | D. | If the program chose to make it a ***static method***, it would not need any parameters to do its job. |
| Correct | E. | The method should have a **boolean**return type. |
|  | F. | The method should be a **void**return type. |

 Answer Key: A,C,E

**Feedback**

The method signature should be**boolean add20ToMemory()** and it should be a***public instance method***, as it modifies private data and has a chance to detect an error (no room to add more memory) so must return a **boolean**.  If a programmer unwisely tried to make it a static method, then it would require an object parameter since there is no object whose **memCapacity**it would be able to increment by 20 without such a parameter.

**Question 8 of 37     Score: 1.08   (of possible 1.08 points)**

The statements

Card card1, card2, card3, myCard;

card1 = new Card();

card2 = card1;

will cause how many **Card**objects to be instantiated?  (only one correct choice):

|  |  |  |
| --- | --- | --- |
| Correct | A. | 1 |
|  | B. | 2 |
|  | C. | 3 |
|  | D. | 4 |
|  | E. | none |

 Answer Key: A

**Feedback**

There are four references but only one object instantiation.  Assigning one reference to another does not produce a new object.  So only one object is instantiated after this code.

**Question 9 of 37     Score: 1.08   (of possible 1.08 points)**

Here is an outline for a working class **OldCellPhone**that has no errors*(you may see this****exact class****several times in this exam)*:

class OldCellPhone

{

   static public final int MIN\_CAP = 10;

   static public final int MAX\_CAP = 1000;

   static public final String DEFAULT\_DSCR = "(generic phone)";

   private String description;

   private int   memCapacity;

   private boolean   camera;

   private boolean   gps;

   public void setCamera( boolean hasCam ) { camera = hasCam; }

   public void setGps( boolean hasGps ) { gps = hasGps; }

   public OldCellPhone()

   {

      this(DEFAULT\_DSCR, MIN\_CAP, false, false);

   }

   public OldCellPhone(String dscr, int mem, boolean cam, boolean gp)

   {

// not shown

   }

   public String toString()

   {

// not shown

   }

   public boolean setMemCapacity(int mem)

   {

// not shown

   }

};

Two of the methods have a return type of **void**.

Check the true statements (there may be more than one):

|  |  |  |
| --- | --- | --- |
|  | A. | This is a mistake, because if the client passes an illegal **boolean**argument down to either method, there is no way to report that fact back to the client via a return value (since their return types are **void**.) |
|  | B. | Even though their return types are void, either one of these method definitions can still return a **boolean**as a functional return to the client. |
| Correct | C. | These methods are mutators, i.e, their name and intent is to modify private members **camera** and/or **gps**. |
| Correct | D. | The **void**return types of these mutators does not pose a problem since it is impossible for a **boolean**passed-in to be illegal; we don't have to test, and we don't have to return a **boolean**response to the client, because if the client passes in a **boolean**, it will naturally be either **true**or **false**, so no test is needed and the return can always be **void**. |
| Correct | E. | The client could call either method by passing the literal **false**as its argument. |

 Answer Key: C,D,E

**Feedback**

A client passing a **boolean**to the method can only pass **true**or **false**.  There are no illegal values that can be passed so the void return type is compatible and appropriate for this mutator method and is no mistake.  As a void method, it cannot return any value, including **boolean**.

**Question 10 of 37     Score: 0   (of possible 1.08 points)**

**ClassA**has a method,**void methodA( ClassA aObj )** , which takes, as a parameter, an object reference of the same **ClassA**.

***There are no "sub-classes"*** (a topic from CS 1B we have not covered in CS 1A) ***involved***.  **ClassA**is the only class in the discussion and is not derived from any other user-defined classes.)

Check all  the true statements. (Check *all*that apply):

|  |  |  |
| --- | --- | --- |
| Missed Correct Answer | A. | **methodA()** can access private data of **aObj**directly, as in **aObj.somePrivateMember = something**, without the need for a public mutator or accessor. |
| Correct | B. | **methodA()** can access private data of its ***calling object*** (the ***this*** object) directly, as in **somePrivateMember = something**, without the need for a public mutator or accessor and without the need to dereference anything. |
| Incorrect | C. | If **methodA()** modifies a private instance member of its ***calling object*** (the ***this*** object), it will result in a simultaneous change of  the  corresponding private member in  the parameter object, **aObj**, even if **aObj**is a different object than the calling object. |
|  | D. | If **methodA()** modifies a private instance member of ***the parameter object,*aObj*,*** it will result in a simultaneous change of  the  corresponding private member in  the ***calling object*** (the ***this*** object), even if **aObj**is a different object than the calling object. |

 Answer Key: A,B

**Review**

 Check to review **before** finishing (will be flagged in Table of Contents)

**Feedback**

Member methods can access private data of objects of the same class.  This applies to both the ***calling object*** as well as any objects that it may receive ***as a parameter***.  In the case of the calling object, no dereference is necessary.  However, the calling object and the parameter object ***may be different***  -- and usually are: there is no need to pass a calling object as a parameter.  The problem tells you to consider that they are distinct, in which case changes to one do not affect changes in the other.

**Question 11 of 37     Score: 1.08   (of possible 1.08 points)**

Here is a method that returns a random **String**(in the form of ***symbolic constants*** defined elsewhere):

   public static String randString()

   {

      int testNum;

      testNum = (int)(Math.random() \* 5000);

      if ( testNum > 4600 )

         return BAR;

      else if ( testNum > 3000 )

         return CHERRIES;

      else if ( testNum > 1200 )

         return SPACE;

      else

         return SEVEN;

   }

If this method were called from a program many times, the following lists the **Strings**that we would expect it to return, ordered from the ***most frequently*** returned **Strings**(***top***of list) to***least frequently*** returned **Strings**(***bottom***of list),

(Check the one that fits this description.)

|  |  |  |
| --- | --- | --- |
| Correct | A. | SPACE CHERRIES SEVEN BAR |
|  | B. | SEVEN SPACE CHERRIES BAR |
|  | C. | BAR CHERRIES SPACE SEVEN |
|  | D. | SPACE CHERRIES BAR SEVEN |
|  | E. | BAR SEVEN CHERRIES SPACE |
|  | F. | SEVEN BAR CHERRIES SPACE |

 Answer Key: A

**Feedback**

BAR is selected in a range that spans 400  
CHERRIES is selected in a range that spans 1600  
SPACE is selected in a range that spans 1800  
SEVEN is selected in a range that spans 1200  
  
Therefore, when listed from the largest span to the smallest, we get the answer.

**Question 12 of 37     Score: 1.08   (of possible 1.08 points)**

Match each term with the phrase that best describes it.

|  | Choices - use a choice only once |
| --- | --- |
| A. | Local variables |
| B. | Formal parameters |
| C. | Static class variables |
| D. | Instance variables |
|  | Match each of the following to a choice |  |  |
| 1. | ... are defined inside a method and known (by their names) only within that method. | https://myetudes.org/ambrosia_library/icons/correct.png |  |
| 2. | ... are  member variables whose values are unique to each object of the class. | https://myetudes.org/ambrosia_library/icons/correct.png |  |
| 3. | ... are member variables whose values are shared by all object instances of the class. | https://myetudes.org/ambrosia_library/icons/correct.png |  |
| 4. | ... are types of local variables that get their values from the caller (client). | https://myetudes.org/ambrosia_library/icons/correct.png |  |

 Answer Key: 1 - A, 2 - D, 3 - C, 4 - B

**Feedback**

Local variables... are defined inside a method and known (by their names) only within that method.

Formal parameters... are types of local variables that get their values from the caller (client).

Static class variables... are member variables whose values are shared by all object instances of the class.

Instance variables... are  member variables whose values are unique to each object of the class.

**Question 13 of 37     Score: 1.08   (of possible 1.08 points)**

Here is a class and some static members:

class Student

{

static public final int SORT\_BY\_FIRST = 88;

static public final int SORT\_BY\_LAST = 98;

static public final int SORT\_BY\_POINTS = 108;

static private int sortKey = 88;

// ... etc.

};

Check the true statements (there may be more than one):

|  |  |  |
| --- | --- | --- |
| Correct | A. | The initial **sortKey**is**SORT\_BY\_FIRST** |
|  | B. | The initial **sortKey**is **SORT\_BY\_LAST** |
|  | C. | The initial **sortKey**is **SORT\_BY\_POINTS** |
|  | D. | The program will not run correctly because a literal is used to initialize **sortKey**. |
| Correct | E. | If a programmer changes the ***first three*** **final**initializations from **88, 98, 108** to**0, 1, 2**, but makes no other change, the program will have a ***logic error***. |
|  | F. | If a programmer changes the ***first three*** **final**initializations from **88, 98, 108** to**0, 1, 2**, but makes no other change, the program will have a ***compiler error***. |
| Correct | G. | Assuming everything else is correct, a program using the code above will run perfectly but is not making maximum use of the three symbolic constants defined. |

 Answer Key: A,E,G

**Feedback**

The program has no compiler errors and will run perfectly as is.  However, because the literal **88**is used to initialize **sortKey**, it is not capable of handling a change if the programmer modifies the definitions of the constants to 0, 1 or 2 and makes no other change.  This will cause a***logical error*** (**sortKey**will be initialized to **88**, but**88**no longer means anything), but no compiler error.  

**Question 14 of 37     Score: 1.08   (of possible 1.08 points)**

Consider this method definition:

static int someRecMethod( int n )

{

if ( n < 0 )

return -1;

return someRecMethod( n + 1 );

}

This method has the following problem -- or none (only one correct choice):

|  |  |  |
| --- | --- | --- |
|  | A. | It will always produce a runaway recursive call to itself resulting in a run-time error. |
|  | B. | It will always return the same number no matter what is passed in as an argument from the client. |
|  | C. | Nothing is wrong;  it has both an explicit case and a recursive case. |
| Correct | D. | It will sometimes return a -1 without error, and other times produce a runaway recursive call to itself resulting in a run-time error. |

 Answer Key: D

**Feedback**

The recursive call does not move the parameter it passes to itself closer to the explicit "escape" case of < 0.  Instead, if a positive number is passed to it, it creates a number even greater (n+1) before it makes the recursive call making the new argument even farther from the necessary explicit condition of (n < 0) needed to end the recursion.  If the client passes in a negative number, all is well, and it returns -1.  But if the client passes in a 0 or positive int, there will be an infinite recursive call, which will create a run-time error.

**Question 15 of 37     Score: 1.08   (of possible 1.08 points)**

A **stack**data structure, when implemented by a programmer as done in the modules as a LIFO data structure, has the following property (only one correct choice):

*[Assume there are several items remaining in the stack when considering these options.]*

|  |  |  |
| --- | --- | --- |
| Correct | A. | It allows the client to **pop()** the most recently **push()**ed item (the ***newest***item in the stack). |
|  | B. | It allows the client to **pop()** the earliest, i.e., first,  **push()**ed item (the ***oldest***item in the stack). |
|  | C. | It allows the client to **pop()** ***any***item off the stack, based on the parameter the client passes to **pop()**. |

 Answer Key: A

**Feedback**

Stacks, when  implemented as a LIFO -- as we did -- only allow the newest item **push()**ed to be returned by **pop()**.

**Question 16 of 37     Score: 1.08   (of possible 1.08 points)**

A **static**class method is nest called from a client using an object to dereference it, as in **someObject.classMethod()**.

 True   
Correct False

 Answer Key: False

**Review**

 Check to review **before** finishing (will be flagged in Table of Contents)

**Feedback**

Static class methods (and variables) can and should be accessed by dereferencing the ***class name***, not an object of the class.  No objects are needed.

**Question 17 of 37     Score: 1.08   (of possible 1.08 points)**

When an array is passed as a parameter to a method, modifying the elements of the array from inside the method will result in a change to those array elements as seen from the client, after the method call is complete.

Correct True

 False

 Answer Key: True

**Feedback**

An array name, when passed into a method, is merely a reference (pointer) to the elements in the array.  When using this reference to modify the array elements inside the method, as in**arrayName[k] = x**, the client array is also modified.

**Question 18 of 37     Score: 1.08   (of possible 1.08 points)**

Consider the code fragment (assumed to be in a program in which all variables are legally defined):

int num1;

double num2;

double answer;

// program gets num1 and num2 from user, and values received

// are always non-zero numbers between +1 and +100 (code not shown)

...

// compute precise quotient:

answer = (double) ( num1 / num2 );

After the assignment statement the variable **answer**, will hold the *most precise quotient possible*, accurate to several digits to the right of the decimal point  ...

|  |  |  |
| --- | --- | --- |
| Correct | A. | ...  *ALWAYS, regardless of what* values **num1**and **num2** hold (as long as they are in the stated range). |
|  | B. | ...  *NEVER, regardless of what* values **num1**and **num2** hold. |
|  | C. | ...  *SOMETIMES, i.e., for some, but not all*, values of **num1**and **num2.** |

 Answer Key: A

**Feedback**

If **num1**= 1  and **num2**= 3, then **answer**will be 0.33333... accurate to several decimal places.  This is true of ***any***two values for the numbers because the first quotient  **num1 / num2** is done using***double arithmetic***, *always*.  It is a binary (two-number) mixed-mode operation (one int and one double) which will promote the int to a double.

**Question 19 of 37     Score: 1.08   (of possible 1.08 points)**

In a statement that has multiple "else if" clauses, each "else if" should always be indented to the right of the one above it (i.e., *progressive indentation* should be used.)

 True   
Correct False

 Answer Key: False

**Feedback**

There are many times when all the "else if" clauses should be aligned with each other and not indented.  Examples have been given in the lesson on the if/else statement.

**Question 20 of 37     Score: 1.08   (of possible 1.08 points)**

At the ***end*of the third** (***3rd***) loop pass of the following loop:

for (k = 3, count = -1; k < 10; k++)

count++;

what is the value of count?

|  |  |  |
| --- | --- | --- |
|  | A. | 5 |
|  | B. | 3 |
| Correct | C. | 2 |
|  | D. | 9 |

 Answer Key: C

**Feedback**

This is not about the final value of **count**, but the value only part way through the loop, an important intermediate stage to be able to predict.

**Question 21 of 37     Score: 1.08   (of possible 1.08 points)**

A static class method might take one or more objects of its class as parameters -- or it might take none -- depending on the purpose of the method.

Correct True

 False

 Answer Key: True

**Feedback**

Static class methods may, or may not, take objects of the class as parameters, depending on their purpose.

**Question 22 of 37     Score: 1.08   (of possible 1.08 points)**

Consider the statement:

if ( (ans == 'Y' || errors < 5) && numTries < 10 ) // note uppercase 'Y'

count++;

Which combinations of values results in **count**being incremented after the statement is complete?

(Select *all*that apply.)

|  |  |  |
| --- | --- | --- |
|  | A. | ans = 'N'    errors = 3    numTries = 10 |
| Correct | B. | ans = 'y'     (lower case)    errors = 4    numTries = 5 |
| Correct | C. | ans = 'Y'  (upper case)    errors = 6    numTries = 5 |
|  | D. | ans = 'N'    errors = 5    numTries = 5 |
| Correct | E. | ans = 'Y'  (upper case)    errors = 100    numTries = -1 |

 Answer Key: B,C,E

**Question 23 of 37     Score: 1.08   (of possible 1.08 points)**

A mutator's job is to (check *all*that apply):

|  |  |  |
| --- | --- | --- |
| Correct | A. | protect private data. |
|  | B. | return the value of private data to the client. |
| Correct | C. | set values of private data as requested by the client, if the values are within acceptable bounds. |
|  | D. | print errors if bad data is passed. |
| Correct | E. | return some indication of a possible error to the client, if the client requests something of the mutator that it cannot do, based on the state of the object and/or argument passed into the mutator. |

 Answer Key: A,C,E

**Feedback**

Mutators are meant to protect private data by testing values passed in before setting the members.  They should never print anything, and do return something to the client indicating success or failure.   The error can be of any type, like a bad parameter or a full array, not allowing any further data to be added.

**Question 24 of 37     Score: 1.08   (of possible 1.08 points)**

The source code that a programmer writes which describes the exact details of that method's implementation is called the "***method invocation***".

 True   
Correct False

 Answer Key: False

**Feedback**

The code that contains the details about how a method accomplishes its goal is called the ***method definition***.  The method ***invocation***(or method ***call***) is used by a client or **main()** to get the method to run, but it gives us no information about the exact implementation of the method.

**Question 25 of 37     Score: 1.08   (of possible 1.08 points)**

Which are true about instance variables of a class (check *all*that apply):

|  |  |  |
| --- | --- | --- |
|  | A. | They should usually be declared **public**. |
| Correct | B. | Mutators should protect them by filtering bad parameter values before assigning those values to them. |
|  | C. | Loop counters and other helper variables should be declared as instance variables so that such counters and helpers don't have to be re-declared local to every member method, individually. |
| Correct | D. | Member instance methods of the same class can modify their values directly, without calling mutators. |
|  | E. | Member instance methods of the same class must use**this.** before their names in order to access them. |

 Answer Key: B,D

**Review**

 Check to review **before** finishing (will be flagged in Table of Contents)

**Feedback**

Instance variables should normally be private and protected by mutator methods.  Helper variables like loop counters that have no intrinsic meaning to the class, should not be declared as instance variables, but should be local to the methods in which they are used.  Instance methods do not need to use mutators or accessors to get to these private data since these methods automatically have direct access to them.  Instance methods do not need to use **this.*xxx*** in order to access them.

**Question 26 of 37     Score: 1.08   (of possible 1.08 points)**

Assuming **num1**is an **int** and **answer**is a **double**variable, which of the following expressions will always give an accurate, double-precision, answer?

(Check all combinations that give double precision accuracy - there will be more than one that should be checked.)

|  |  |  |
| --- | --- | --- |
| Correct | A. | answer = (double) ( (num1 + 1.) / 100. ); |
|  | B. | answer = (double) ( (num1 + 1) / 100 ); |
| Correct | C. | answer = ( ( (double)num1 + 1 ) / 100 ); |
| Correct | D. | answer = (num1 + 1) / 100. ; |
|  | E. | answer = (num1 + 1) / 100 ; |

 Answer Key: A,C,D

**Feedback**

If the quotient is computed using int arithmetic BEFORE casting to double, the fractional data is lost, so the answer will, in general, be wrong.

If either the numerator or the denominator is a double before the quotient is computed, then the answer will be promoted to double and no information lost.

Finally, in the case where either of the terms in the numerator sum is cast to double, then that sum will be promoted to double and the entire numerator will become double, which means the quotient will use double arithmetic.

**Question 27 of 37     Score: 0   (of possible 1.08 points)**

When a client method calls a second method (both in the same class), it is normally preferred to send***primitive data*** to the called method via ***class members*** rather than  by passing the data as arguments to its***formal parameter list***.

Incorrect True

 False

 Answer Key: False

**Feedback**

We avoid using "globals" or class variables to pass information, like primitives, between methods because of the large risk of error and exposure.  In Java, class variables are not intended for passing information between methods.  Rather, they represent the intrinsic, implementation-independent data that defines the class.

**Question 28 of 37     Score: 1.08   (of possible 1.08 points)**

Consider a***binary search algorithm*** as described in the modules.  Assume that the array to be searched is pre-sorted, so that sort is ***not***part of the search algorithm.

Searching an array using a***binary search*** (check *all*choices that apply):

|  |  |  |
| --- | --- | --- |
| Correct | A. | ... is usually faster for each search than a simple linear search. |
| Correct | B. | ... requires a pre-sorted array in order to work. |
|  | C. | ... is usually slower for each search than a simple linear search. |
|  | D. | ... requires less code and logic than a simple linear search. |

 Answer Key: A,B

**Feedback**

A binary search takes less time for each search than a simple linear search, but requires that the array be pre-sorted (an operation that we presumed to be done prior to the search, and was not part of the search, as stated in the question.) The algorithm is much more complex than a linear search, requiring a recursive method call.

**Question 29 of 37     Score: 1.08   (of possible 1.08 points)**

As soon as the statement

JFrame frmMyWindow = new JFrame("Transporter Room");

is executed, a **JFrame**is created and is displayed, making it instantly visible to the user.

 True   
Correct False

 Answer Key: False

**Feedback**

Creating a **JFrame**is very far from displaying it.  We must call

frmMyWindow.setVisible(true);

before the user will see anything. Usually there is a lot to do before this step.

**Question 30 of 37     Score: 1.08   (of possible 1.08 points)**

A ***binary search algorithm*** is written (as in the modules, for example) which searches a ***pre-sorted*** array for some user-defined value, **clientData**.  If **clientData**is stored in the array, it returns its array position, and if not found, it returns -1 (again, just like in the modules).  Assume the array to be searched has 100 data elements in it. (Check *all*that apply):

*[NOTE: due to common off-by-one interpretations when counting such things, if your predicted answer is within one (+1 or -1) of a posted option below, you can assume your prediction and the choice you are looking at are equivalent and check that option.]*

|  |  |  |
| --- | --- | --- |
|  | A. | It ***may*require*as many as 99*** comparisons of **data**before it returns. |
| Correct | B. | It ***might***return to the client with an answer after ***only one*** comparison of **data.** |
| Correct | C. | It ***will always*** return with an answer ***in 7 or fewer***comparisons of **data.** |
|  | D. | It ***will always*** return with an answer ***in 3 or fewer***comparisons of **data.** |

 Answer Key: B,C

**Feedback**

The data may be in middle position, position 49, in which case the algorithm only requires one comparison (of data), so it may return after about one comparison.  At the other extreme, the data may be in position 0 or 99, or not in the array at all.  In those cases it will require testing the middle element about seven times -- as the hint indicates by walking you through the seven tests --  but never more.  So seven is the most it would require (+/- 1 to account for off-by-one details).  It certainly could easily be more than three  (as our example in the modules shows.)

**Question 31 of 37     Score: 1.08   (of possible 1.08 points)**

Assume you have defined a **Dog**class that has a **String name** member and corresponding mutator and accessor.  Also, assume you***did not define*** a **toString()** method for your **Dog**class.

Two **Dog references**, **myDog**and **yourDog**, are declared, and one of them, **yourDog**,is used to instantiate a **Dog**object (code not shown).   Next, the following statement is executed:

myDog = yourDog;

so that both references manage (i.e., point to) the**same Dog object**.

If we change **myDog**'s name, by a mutator call:

myDog.setName("LiliKoi");

this will (choose all true statements):

|  |  |  |
| --- | --- | --- |
|  | A. | cause ***only***  System.out.println( myDog.getName() );  to display **LiliKoi**, but ***not***  System.out.println( yourDog.getName() ); |
|  | B. | cause ***only***  System.out.println( yourDog.getName() );  to display **LiliKoi**, but ***not***  System.out.println( myDog.getName() ); |
| Correct | C. | cause ***both***  System.out.println( myDog.getName() );  ***and***  System.out.println( yourDog.getName() );  to display **LiliKoi*.*** |
|  | D. | cause ***both***  System.out.println( myDog );  ***and***  System.out.println( yourDog );  to display **LiliKoi*.*** |

 Answer Key: C

**Feedback**

There is only one **Dog**object in this scenario, and when we change its **name**, the references which point to that **Dog** object will see the***same*name** (as well as the same data for any *other*members not discussed here).

Since no**toString()** method is defined for the class, the result of attempting to display **myDog**or **yourDog**, directly, will result is a cryptic object description that would not include the name "LiliKoi".

**Question 32 of 37     Score: 1.08   (of possible 1.08 points)**

**ClassB**and **ClassA**have no special relationship;  they are two separate classes both used by the same **main()** of a third, client, class (**Foothill**, perhaps).   **void methodB( ClassA aObj )** is a method of **classB**.  It takes a **ClassA**object as a parameter.  The client, **main()**, calls **methodB()**.

Here is an outline of the situation:

class ClassA

{

private char someMemb;

...

}

class ClassB

{

public void methodB( ClassA aObj )

...

}

// in client class ...

   // -------  main --------------

   public static void main(String[] args) throws Exception

   {

      ClassA myObjA = new ClassA();

      ClassB myObjB = new ClassB();

      myObjB.methodB ( myObjA );

      System.out.println( myObjA.get() );

   }

Check all the true statements (check *all*that apply):

|  |  |  |
| --- | --- | --- |
| Correct | A. | If **methodB()** modifies **aObj**'s **someMemb,**the client's **myObjA**'s **someMemb**will also be modified, accordingly, when the method returns. |
|  | B. | If **methodB()** modifies **aObj**'s **someMemb,**the client's **myObjA**'s **someMemb**will not be modified when the method returns. |
|  | C. | **methodB()** can change **aObj**'s **someMemb**value through direct assignment, as in **aObj.someMemb = something.** |
|  | D. | **methodB()** can read (access) **aObj**'s **someMemb**value directly without changing it, as in **something =** **aObj.someMemb.** |
| Correct | E. | **methodB()** can read from (access) and/or write to (change)  **aObj**'s **someMemb** indirectly through appropriate public **ClassA**methods, if **ClassA**offers such methods. |

 Answer Key: A,E

**Feedback**

Private data of **ClassA**is never accessible form a non-**ClassA** method like **methodB()**, directly, either for read or write.  However, mutators and accessors that **ClassA**offers, can always be used for this purpose.   Finally, because **methodB()** takes a reference as a parameter, the **methodB()*does modify*** the client object if and when**methodB()** makes a change though its local reference, **aObj**.  There is only one **ClassA**object in this scenario, and it doesn't matter if we access it through the client's**myObjA**, or the method's **aObj**reference.

**Question 33 of 37     Score: 1.08   (of possible 1.08 points)**

Here is an expression which contains the + operator in several places on the ***RHS***of an assignment statement (and we don't need to know anything about the shaded variable on the ***LHS***to answer the question):

someVar = **'1' + '2' + '3'** ;

In this expression ...

|  |  |  |
| --- | --- | --- |
| Correct | A. | ... the + operator represents addition. |
|  | B. | ... the + operator represents string concatenation. |

 Answer Key: A

**Feedback**

The operands are all **chars**, and **chars**are small **ints**, not **Strings**.    The + operator will add the ASCII values, not concatenate the individual digits.

**Question 34 of 37     Score: 1.08   (of possible 1.08 points)**

Here is a code fragment which appears in a valid **main()**:

int numGifts;

double costPerGift;

// numGifts gets a value somehow in code that is not shown. then ...

if ( numGifts < -10 )

costPerGift = 3.0;

else if ( numGifts < 8.5 )

costPerGift = 2.75;

else

costPerGift = 2.5;

For which values of **numGifts**will **costPerGift**be**2.75** at the end of the fragment?

(Check all boxes that apply - there will be more than one.)

|  |  |  |
| --- | --- | --- |
| Correct | A. | numGifts = -1 |
| Correct | B. | numGifts = 8 |
|  | C. | numGifts = 10 |
|  | D. | numGifts = 15 |
|  | E. | numGifts = 20 |
|  | F. | numGifts = 30 |

 Answer Key: A,B

**Question 35 of 37     Score: 0.54   (of possible 1.08 points)**

An ***array of 100 elements*** is to be sorted using the ***bubble sort*** in the modules (the one that tests the return value of **floatLargestToTop()** to see if it can return "early".)  Check all the true statements about the sort algorithm, i.e., the sort method and its support methods. (Check *all*that apply.)

|  |  |  |
| --- | --- | --- |
|  | A. | It will ***always***return (completely sorted) after 99 data comparisons. |
| Correct | B. | It will ***sometimes***return (completely sorted) after only 99 data comparisons. |
| Incorrect | C. | It will ***always***require at least one swap. |
| Correct | D. | It will always require at least 99 comparisons |

 Answer Key: B,D

**Feedback**

The **floatLargestToTop()** method will compare neighboring elements in a loop.  In the first call, the loop goes up to 99:  it compares array elements 0 with 1, then 1 with 2, then 2 with 3, etc, up to the last test of 98 with 99.  Therefore there will always be at ***least 99 comparisons***, no matter what.  If the array was already sorted, then no swaps will occur and the sorting method will return after only one call to**floatLargestToTop()**, thus ***99 comparisons and no swaps is possible***.  However, if the array is unsorted before the call (as is usually the case) it will require more calls to**floatLargestToTop()**, so 99 is a lower bound on the possible number of comparisons -- there could be many more.

**Question 36 of 37     Score: 1.08   (of possible 1.08 points)**

A **static**member of some **ClassX** (check *all*that apply):

|  |  |  |
| --- | --- | --- |
|  | A. | can, at any given time, store one value for **objA**and a ***different***value for **objB**, two different objects of **ClassX**. |
| Correct | B. | can be ***public***. |
| Correct | C. | can be modified from *any*client  (like**Foothill**'s **main()**)***even if the client has not instantiated an object***of **ClassX**.  (You can assume that **ClassX**has some means for clients to modify the static member in question.) |
| Correct | D. | can be ***private***. |
| Correct | E. | can have associated mutators and/or accessors. |

 Answer Key: B,C,D,E

**Feedback**

One does not need objects to get at static data.  A client can use the class name, not an object, as a technique to refer to the static member (if public) or a static accessor that indirectly modifies or accesses the static member.

**Question 37 of 37     Score: 1.08   (of possible 1.08 points)**

Within an instance method, a programmer might utilize the special syntax**this.someMember** to *reasonably*assist with which of the following challenges?  (Check *all*that apply):

|  |  |  |
| --- | --- | --- |
|  | A. | To get at private data of the calling object, since without **this.** before the member, that member's ***privacy prohibits direct access***. |
|  | B. | To get at private data of a ***parameter***that is passed to the method. |
| Correct | C. | To get at private data of the ***calling object*** (the ***this***object), if there is a***local variable that has the same name*** as the instance member in question. |
| Correct | D. | ***To enhance readability***, if the programmer wants to allow the reader to easily distinguish instance members in their methods from local variables of those methods.  While not required, prepending***this.*** to each member used in the method can only mean that those variables refer to the calling object's members. |

 Answer Key: C,D

**Feedback**

**this.** is of no help in getting around privacy or accessing a parameter's members.  There is also no problem based on privacy with getting at the calling object's members.  However, disambiguating members from local variables, either due to local variable "hiding" or readability is a reasonable use of the **this.** syntax.

 1 of 1