CSE 1325 OBJECT-ORIENTED PROGRAMMING

Exam #2 «---» 1 1 001 «---» Practice #1

Vocabulary

Write the word or phrase from the Word List below to the left of the definition that it best matches. Each word or phrase is used at most once, but some will not be used. {10 at 2 points each}

1 Inheritance

2 Code

3 Object-Oriented Programming (OOP)

4 Iterator

5 Process

6 Attribute

7 Class

8 Global

9 Stack

10 Package

Multiple Choice

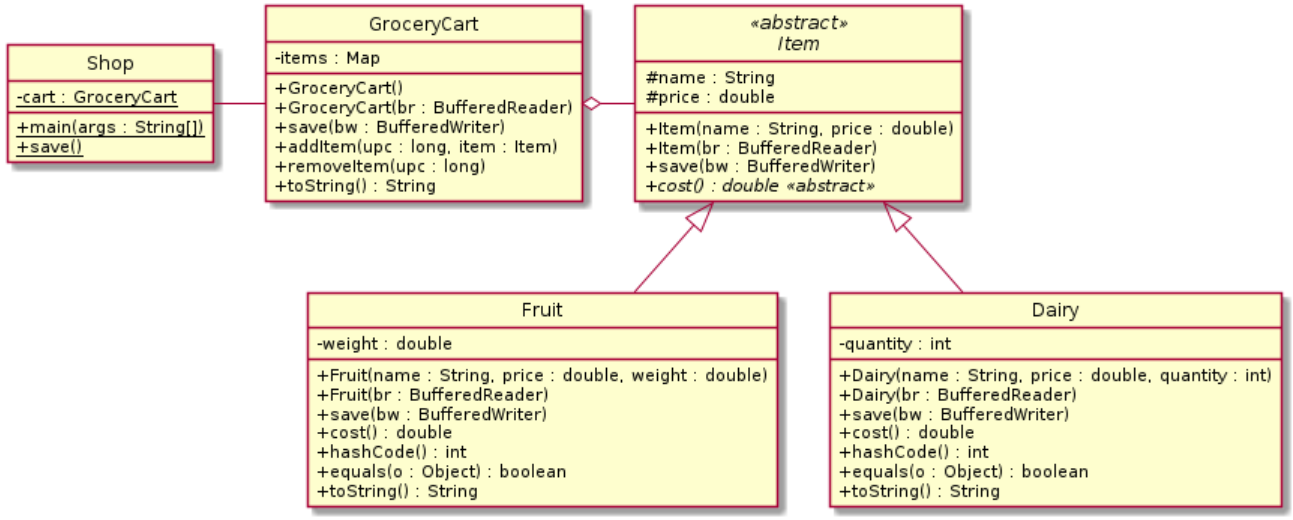
Read the full question and every possible answer. Choose the one best answer for each question and write the corresponding letter in the blank next to the number. {15 at 2 points each}

1 A	6 D	11 A
2 A	7 D	12 B
3 D	8 B	13 C
4 B	9 B	14 D
5 C	10 D	15 A

Free Response

- Question 1 is a larger set of code covering much of our second section concepts and skills.

1. {polymorphism, collections, iterators, file I/O} Consider the following class diagram representing grocery shopping for food. The `GroceryCart` class stores and removes `Item` objects in `Map items` using the `addItem` and `removeItem` methods, respectively. The diagram shows two subclasses of `Item`, `Fruit` (measured by weight) and `Dairy` (measured by quantity). All classes can save and restore themselves. You will NOT write much of this application.



a. {2 point} Given existing code `Item item = new Dairy ( ... );` or `Item item = new Fruit ( ... );` (depending on exam) (with parameters you should NOT write in place of `...`), write code showing how you would get the cost of the Dairy or Fruit object (depending on exam) from variable `item`. (Do NOT write the `Item item` code.)

- Not exactly a *trick* question, but certainly a simple one for those who understand polymorphism!
- 0 "Won't compile" or any exception
- 2 for each code element (`item`, `.`, `cost`, and `();`)

```
item.cost();
```

b. {4 points} Write `Dairy.equals(Object o)` or `Fruit..equals(Object o)` (depending on exam). All 3 fields (including inherited fields) are significant.

- VERIFY the class coded matches the question, and notify me if they don't match.
- Most students will likely use the primary solutions below, but no deduction if they use something like the alternate. Omitting the check for null incurs no penalty.
- ½ point for `@Override`
- ½ point for `public`
- ½ point for `boolean equals`
- ½ point for `(Object o)` (or any other parameter name)
- ½ point for is it me?
- ½ point for is it my type?
- ½ point for cast to my type.
- ½ point for return equality of fields.

```
  @Override
  public boolean equals(Object o) { // Dairy
      if(o == this) return true;
      if(o == null || this.getClass() != o.getClass()) return false; // alternate
// OR if(o == null || !(o instanceof Dairy)) return false;
      Dairy d = (Dairy) o;
      return name.equals(d.name)
          && price == d.price
          && quantity == d.quantity;
  }

  @Override
  public boolean equals(Object o) { // Fruit
      if(o == this) return true;
      if(o == null || this.getClass() != o.getClass()) return false; // alternate
// OR if(o == null || !(o instanceof Fruit)) return false;
      Fruit d = (Fruit) o;
      return name.equals(d.name)
          && price == d.price
          && weight == d.weight;
  }
```

c. {3 points} Write `Dairy.hashCode()` or `Fruit.hashCode()` (depending on exam). All 3 fields (including inherited fields) are significant.

- VERIFY the class coded matches the question, and notify me if they don't match.
- Most students will likely use the primary solutions below, but no deduction if they use something like the alternative. (The key on the alternatives is that the numbers at the 7 and 31 positions MUST be prime.)
- ½ point for @Override
- 1 point for public int hashCode()
- 1½ point for the body

```java
@Override
public int hashCode() {                          // Dairy
    return Objects.hash(name, price, quantity);
}

@Override
public int hashCode() {                          // Dairy (alternate)
    int hash = 7;
    hash = 31*hash + name.hashCode();
    hash = 31*hash + (int) (100 * price);
    hash = 31*hash + quantity;
    return hash;
}

@Override
public int hashCode() {                         // Fruit
    return Objects.hash(name, price, weight);
}

@Override
public int hashCode() {                          // Fruit (alternate)
    int hash = 7;
    hash = 31*hash + name.hashCode();
    hash = 31*hash + (int) (100 * price);
    hash = 31*hash + (int) (100 * weight);
    return hash;
}
```

d. {3 points} Write the `addItem` method in `GroceryCart` that adds its `upc` and `item` parameters to the `items` Map.

- ½ point for `public void addItem`
- ½ point for `(long upc, Item item)`
- 1 point for `items.put`
- ½ point for `(upc,`
- ½ point for `item);`

```
public void addItem(long upc, Item item) {
     items.put(upc, item);
 }
```

e. {3 points} Write the `removeItem` method in `GroceryCart` that removes the pair from the `items` Map where the key matches the `upc` parameter.

- ½ point for `public void removeItem`
- ½ point for `(long upc) {`
- ½ point for `return`
- 1 point for `items.remove`
- ½ point for `upc`

```
public void removeItem(long upc) {
    return items.remove(upc);
}
```

f. {3 points} Write the `save` method for Dairy / Fruit (depending on exam) class. Write all inherited fields to `bw` without referencing them, then write the field defined in Dairy / Fruit and do NOT catch the *checked* exception `IOException`.

- VERIFY the class coded matches the question, and notify me if they don't match.
- ½ for `public void save`
- ½ for `(BufferedWriter bw)`
- ½ for `throws IOException`
- ½ for `super.save(bw)`
- ½ for `bw.write`
- ½ for `"" + quantity + '\n'`

```
public void save(BufferedWriter bw) throws IOException { // Dairy
    super.save(bw);
    bw.write("" + quantity + '\n');
}

public void save(BufferedWriter bw) throws IOException {// Fruit
    super.save(bw);
    bw.write("" + weight + '\n');
}
```

g. {3 points} Write the Dairy or Fruit (depending on exam) constructor with the `BufferedReader` parameter. Construct a new Dairy or Fruit (depending on exam) by restoring the inherited fields first, then the field defined in Dairy or Fruit (depending on exam) and do NOT catch the *checked* exception `IOException`.

- VERIFY the class coded matches the question, and notify me if they don't match.
- ½ for `public Dairy` or `public Fruit`
- ½ for `(BufferedReader br)`
- ½ for `throws IOException`
- ½ for `super(br)`
- ½ for `this.quantity` or `this.weight`
- ½ for `Integer.parseInt(br.readLine()) or Double.parseDouble(br.readLine())`

```java
public Dairy(BufferedReader br) throws IOException { // Dairy
    super(br);
    this.quantity = Integer.parseInt(br.readLine()); // Dairy
}

public Fruit(BufferedReader br) throws IOException { // Fruit
    super(br);
    this.weight = Double.parseDouble(br.readLine()); // Fruit
}
```

h. {6 points} Write `GroceryCart .toString()` , which formats the UPC codes (the key in Map `items`) and Item objects (the value in Map `items`) into a receipt that looks similar to this.

- Instance a `StringBuilder` object with the "Receipt" header.
- Obtain the set of keys from Map `items` .
- Obtain an iterator from the set of keys. (You will receive no credit unless you use an iterator.)
- While additional keys remain, obtain the next key and use it to obtain the associated item, then append both to the `StringBuilder` object.
- Return the String version of the `StringBuilder` object.

- This code may vary a bit. Always give credit for correct code, even if it is very different from the suggested solution. Ask or test if you're unsure.
- ½ point for `@Override`
- ½ for `public String toString()`
- ½ for `StringBuilder` declaration (constructor parameter may vary - accept any *short* String there, but concatenating everything in the parameter is -3).
- 1 for obtaining the keys. (This line may be combined with the next, omitting the keys variable, or a different variable of the same type may be used, without penalty. The `<Long>` is optional, although <> is NOT correct. Accept long with a lower-case el but mark with upper-case L, as primitives can't be used for generic types.)
- ½ for obtaining the iterator. (May be combined with previous line. The `<Long>` is REQUIRED here. Iterator name may vary. Same deal on long vs Long.)
- 1 for `while(it.hasNext())` or equivalent for loop (½ for `while`, ½ for `hasNext()`).
- ½ for `long upc = it.next()`. NO credit if they use it.next() TWICE in the following statement, as this actually advances to the next element of keys!)
- ½ for `sb.append`
- ½ for `items.get(upc)`
- ½ for any reasonably formatted string.
- ½ for `return sb.toString()`

```
@Override
public String toString() {
    StringBuilder sb = new StringBuilder("        Receipt\n\n");
    Set<Long> keys = items.keySet(); // Set keys = is also OK, but NOT Set<>
    Iterator<Long> it = keys.iterator(); // MUST be Iterator<Long>
    while(it.hasNext()) {
        long upc = it.next();
        sb.append(String.format("%15d", upc)
                + items.get(upc) + '\n');
    }
    return sb.toString();
}
```

i. {3 points} Write the main method in class `Shop`. Instance a new `GroceryCart` referenced by variable `cart`. Using the `Menu` and `MenuItem` classes discussed in lecture, add just 1 `MenuItem` instance: "Save" that calls method `save()`. Then use your `Menu` instance to dispatch the selection of "Save" as if by the user.

- ½ for `public static void main(String[] args)` and `cart = new GroceryCart();`
- ½ for `Menu menu = new Menu();`
- ½ for `menu.addMenuItem(`
- ½ for `new MenuItem("Save",`
- ½ for `() -> save()` (the lambda)
- ½ for `menu.run(0);`

```
public static void main(String[] args) {
    cart = new GroceryCart();
    Menu menu = new Menu();
    menu.addMenuItem(new MenuItem("Save", () -> save());
    menu.run(0);
}
```

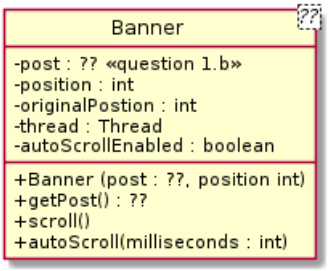
j. {3 points} In class Shop, write ONLY the `save()` method. Use a try-with-resources to open file "cart.txt" and tell the `cart` instance to save itself. If an `IOException` is thrown, print it to the console's error stream.

- 0 for `public static void save()` (ungraded even if omitted)
- 1 for `try (` (the parenthesis signifying try-with-resources is the key here)
- 1 for `BufferedWriter bw = new BufferedWriter(new FileWriter("list.txt")))`
- ½ for `cart.save(bw);`
- ½ for `catch(IOException e) {System.err.println` ... (what is printed is ungraded).

```
public static void save() {
    try (BufferedWriter bw = new BufferedWriter(new FileWriter("cart.txt"))) {
        cart.save(bw);
    } catch(IOException e) {
        System.err.println("Failed to save cart: " + e);
    }
}
```

2. {generics, threads} Consider the class diagram.

- GRADING NOTE: The class may be labeled Scroller, Marquee, or Banner. The field may be labeled message, value, or post. It is important to catch any students using names different from their exam, as that is strong evidence of academic dishonesty.



Generic class Scroller (or Marquee or Banner, depending on exam) scrolls the toString representation of field message (or value or post, depending on exam) across the terminal. The constructor accepts an object of its one generic type which is stored in the message / value / post field. The `scroll()` method prints the number spaces defined by `position` followed by message / value / post on first call, with one less space on each subsequent call, giving the effect of the text scrolling right to left on the terminal. Method `autoScroll(int milliseconds)` scrolls the object in a separate thread, one character every `milliseconds` until `autoScrollEnabled` becomes false, while the main thread continues executing.

a. {2 points} Write the one-line declaration for the generic class. You may use any generic variable you please and you may declare you are implementing any interfaces you believe you need.

- VERIFY that both Scroller / Marquee / Banner and message / value / post (these are *independently* selected) match the question, and notify me if not.
- We expect an `implements Runnable` if they will write a `void run()` method for their thread in part f. Otherwise, no `implements` are expected, but no deductions as long as their syntax is correct, e.g., `implements Runnable, Comparable {` would be fine.

- 1 point for the <E> *in the correct location*. -½ if in the wrong location. Any other single capital letter in place of E is fine (often T is used).
- 1 point for the rest of the code.

```
public class Scroller <E> { // or <E> implements Runnable {

public class Marquee <E> { // or <E> implements Runnable {

public class Banner <E> { // or <E> implements Runnable {
```

b. {2 points} Write the one-line definition of the private field of the generic type.

- VERIFY that both Scroller / Marquee / Banner and message / value / post (these are *independently* selected) match the question, and notify me if not.
- 1 point for the E. This MUST match the letter in the <E> in the class declaration, otherwise notify me.
- 1 point for the rest of the code.

```
private E message;

private E value;

private E post;
```

c. {3 points} Write the constructor for class Scroller / Marquee / Banner (depending on exam) that accepts a message / value / post (depending on exam) parameter of the generic type and a `position` parameter of type int. Assign the parameters to fields of the same name. Also assign `position` to field `originalPosition`.

- VERIFY that both Scroller / Marquee / Banner and message / value / post (these are *independently* selected) match the question, and notify me if not.
- 1 point for the `E`. This MUST match the letter in the <E> in the class declaration, otherwise notify me.
- ½ for the (rest of the) constructor declaration
- ½ for `this.value = message;`
- ½ for `this.position = position;`
- ½ for `this.originalPosition = position;`

```
public Scroller /* or Marquee or Banner */
        (E message /* or value or post */,
         int position) {
    this.value = message;       // or value or post
    this.position = position;
    this.originalPosition = position;
}
```

d. {2 points} Write the `getMessage()` (or `getValue()` or `getPost()`, depending on exam) getter, which simply returns field `message` (or `value` or `post`, depending on exam) as its native type (that is, you may NOT just return type Object).

- VERIFY that both `getMessage` / `getValue` / `getPost` AND `message` / `value` / `post` match the question, and notify me if not.
- 1 point for the E. This MUST match the letter in the <E> in the class declaration, otherwise notify me.
- ½ point for the rest of the declaration
- ½ point for the return statement.

```
public E getMessage() { // or getValue() or getPost()
    return message;     // or    value   or    post
}
```

e. {2 points} Write Java code demonstrating how you would ensure that only one thread could execute method `scroll()` at a time. You may write the method declaration (do NOT write the entire method!) OR demonstrate how to call the method from within a thread while avoiding thread interference.

```
// EITHER fixing the method declaration

public synchronized void scroll()

// OR calling the method inside a mutex

// name and type may vary
private static Object mutex = new Object();
// Name of mutex should match what was declared.
sychronized(mutex) {scroll();}
```

f. {6 points} Write method `autoScroll(int milliseconds)`.

- First, if field `thread` is not null, set `autoScrollEnabled` to false and wait for the Thread referenced by `thread` to exit.
- Next, if parameter `milliseconds` is less than 0, return. (Thus, calling `autoScroll` with a negative parameter just terminates autoscrolling.)
- Finally, set `autoScrollEnabled` to true and run a new thread. The thread body loops while `autoScrollEnabled` is true, calling method `scroll()` and then pausing for `milliseconds` milliseconds each iteration. You are permitted to write a separate method for the thread body, or you may use an anonymous class or a lambda for the thread body as you please. **Do** handle *checked* exception `InterruptedException` here in any (valid) way you please.

- 2 points for first bullet:
    - 1 point for `thread.join()`
    - ½ for `try` / `catch(InterruptedException e`
    - ½ for the rest
- ½ for second bullet
- 3½ for third bullet:
    - 1 point for `thread = new Thread`
    - ½ for `scroll()` (also deduct if this is inconsistent with part e)
    - 1 point for `Thread.sleep(milliseconds)`
    - ½ for `try` / `catch(InterruptedException e)`
    - ½ for the rest.

```
public void autoScroll(int milliseconds) {

    // First bullet
    if(thread != null) {
        autoScrollEnabled = false;
        try {
            thread.join();
        } catch(InterruptedException e) {
            System.err.println("#### thread join interrupted!");
        }
    }

    // Second bullet
    if(milliseconds < 1)  return;

    // Third bullet
    autoScrollEnabled = true;
    thread = new Thread(() -> {
        while(autoScrollEnabled) {
            // If part e. isn't "public synchronized void scroll()"
            // the next line must be "synchronized(this) {scroll();}"
            scroll();
            try {
                Thread.sleep(milliseconds);
            } catch(InterruptedException e) {
                System.err.println("#### thread sleep interrupted!");
            }
        }
    });
    thread.start();
}
```

```
// --- OR alternately, with "implements runnable" in the class declaration

        // Third bullet
        autoScrollEnabled = true; // OR could be right after "private void run() {"
        thread = new Thread(this);
        thread.start();
    }

    @Override
    private void run() {
        while(autoScrollEnabled) {
            // If part e. isn't "public synchronized void scroll()"
            // the next line must be "synchronized(this) {scroll();}"
            scroll();
            try {
                Thread.sleep(milliseconds);
            } catch(InterruptedException e) {
                System.err.println("#### thread sleep interrupted!");
            }
        }
    }
```

Bonus

Bonus {+4 points} Given `class Animal`, `class Dog extends Animal`, `Animal animal`, and `Dog dog`, give an example of the following:

Upcast (if this might fail, set the variable to null if it does):

- +1 for assigning dog to animal
- +½ for NOT trying to handle an error that can't occur

```
animal = dog;
```

Downcast (if this might fail, set the variable to null if it does):

- +1 for assigning animal to dog
- +1 for detecting an invalid cast
- ½ for assigning null

```
// with either option, dog = null could be moved to the start of the sequence instead

// one option
if(animal instanceOf Dog) dog = animal;
else dog = null;
```

```
// another option
try {
    dog = animal;
} catch(ClassCastException e) {
    dog = null;
}
```