Final/Class Review

Tuesday, April 28, 2015 2:18 PM

I'm assuming you know how to program in Java competently or have taken CSE 1223. One thing that always fucks me up, first though. Parameters in Java are pass by copy. A method does not alter primitive variables or immutable types when they are copied in as formal parameters from arguments. If a variable is mutable, this does happen, however. Also, refer to the midterm study guide on everything after midterm 1 and before midterm 2. Also, Java ROUNDS DOWN when doing int division.

Design By Contract:

- A *system* is any part of anything that you want to think as one whole unit.
- An *interface* is the boundary between a system and everything else.
- A *component* is a system that is used inside another system.
- A *client* is a role played by some agent, be it a person or something else, viewing a system from the outside as an indivisible unit.
- An *implementer* is a person viewing a system from the inside, as an assembly of components.
- *Information Hiding* is where one purposefully holds back of how some parts of a system work.
- *Abstraction* is a technique where you create a cover story for the hidden information. Think about how we learned about recursion as a valid example of abstraction.
- Design by Contract was created in the 1980s, and is the standard policy regarding the creation of software components.
- *Mathematical Models* are used in contracts. A mathematical type exists for every type of program variable in Java. Models also exist that aren't math. *Informal Models* are those models which are based on concepts from other situations.
- A method contract contains a *precondition,* which tells the implementer what the client will do when the method they are creating is called. It also contains a *postcondition*, which is the responsibility of the implementer to code and ensure is correct. Respectively, these are characterized in the documentation as a *requires and ensures clause*.
- If a precondition is true, then the method will terminate if the postcondition is true. If it isn't, then anything in the world could happen. Therefore, an implementer must ensure that the code in the body is correct, and a client must provide the correct information to validate the precondition.
- A Javadoc is placed before a method, and specifies the pre/postconditions of the method. It is specified by /** … */ comment structure. They generate an online API document automatically, as long as correct syntax is used.
- *Assert statements* allows programmers to check if a precondition has been met before an implementation is run. This is not used in final code, but in code development so that the implementer knows that their code is correct and the precondition is fucking up.

Static Methods:

- A *static method* is one that takes zero or more parameters, and returns a specific return type or nothing. If it returns nothing, its return type is dimmed as void.
- These can be called without a receiver, by just writing the name of the method with correct arguments.
- Written with the word static in the method header.
- Cannot use the *this* variable.

Instance Methods:

- An *instance method* is one that takes zero or more parameters, and returns a specific return type, or nothing. Static is not used in the method header.
- Are called with a *receiver,* which is an object variable created from a class that contains the instance method.
- Can use keyword *this*, which means that the value of the receiver is represented by the variable

this.
- Instance methods are generally considered better than static methods. This is because one cannot declare static methods in interfaces.

Trees:
- A *tree* is a recursive structure consisting of a parent node and zero or more child nodes.
- Leaf nodes are those at the bottom of the tree.
- Sibling nodes are those places next to each other on a tree, those that are the initial children of a relative parent node.
- The parent node is also called the root node.
- The *length* of a *path* (Parent node to leaf node) is the amount of nodes between the parent and leaf node inclusive.
- The *height* of a tree is the length of its longest path.
- The *size* of a tree is its total number of nodes.
- The true view of a tree is in its recursive structure. A true tree is made up of a root note and a string of zero or more subtrees of the root, each of which is another tree.

XML/XMLTree:
- XML is a document format used all over the web.
- An *element tag* looks like this: <cool> This is called *content* </cool>
- An empty element tag can look like this <cool></cool> or this <cool />
- *Attributes* are created like this <cool numOfFucksGiven = "0"> </cool>
- There can be many attributes created. The text in between the quotes is called a value.
- An XML document is made up of a top level element, and zero or more child elements, which can be XML subdocuments themselves. This means XML has a recursive structure.
- An XMLTree component allows one to create and navigate a tree whose structure mirrors that of an XML file. It does all the parsing for you.
- Each node in an XMLTree has two properties or invariants. The node will have a String object called a label that is a tag or some ordinary text. If the label is a tag, then it has zero or more attributes.
- All XMLTree methods are instance methods, which means you have to create an XMLTree object first. An XMLTree is immutable however.
- It is constructed like this: XMLTree myHumps = new XMLTree1(" Name of file containing tree");
- The following points will cover the methods.
- String label() returns the label of the root of the XML tree, or the specified node.
- Boolean isTag() returns whether the root of this is tag. Use it to check for tags. A good way to program recursively with XMLTrees is through this method. If a node is a tag, it might have children, but if it isn't, it definitely does not have children. Therefore, if a node is leaf, you have hit the bottom of a subtree, and you can move onto the next one.
- Boolean hasAttribute(String name) returns whether the root tag of this has an attribute called name. Make sure the node you call it on is a tag.
- String attributeValue(String name) returns the value of the attribute name of the root node this. This will crash the program if the attribute does not exist.
- Int numberOfChildren() returns the number of subtrees of the root of this. This must be a tag or the program will crash.
- XMLTree child(int k) returns the kth subtree of the root of this. Make sure it has children, it is a tag, and that you know that 0 returns the first subtree.

RSS:
- *RSS* is a text format used on the web for news/web feeds. It uses XML to represent info that is frequently updated, in summary form with links to the original content.
- RSS root nodes are actually the zeroth child of the root node, which contains the RSS version info, and these are called channel, and there is one in every RSS feed. The RSS feed must also contain a title tag, that contains title content, a link tag, which contains a url to the original article in the content, and a description tag, describing the website.
- There can be zero or more item elements within channel, each has at least a title or description

tag in it, and always have a link.

NATURAL NUMBER OVERVIEW:

- Nonnegative integers that have no upper bound like ints.
- You need to call methods to do operations.
- NaturalNumber extends from a class called Standard, which is the base natural number.
- Standard has methods that are part of most OSU CSE component families, where they can always be found in one place.
- Next is the natural number kernel, which has a minimal set of methods which are special.
- Then there is the interface NaturalNumber itself, which has a lot of the operator and convenient methods to use.
- NaturalNumbers can have constructors from ints, other NaturalNumbers of the same type, no constructor so it defaults to zero, and Strings!
- All methods for NaturalNumber are instance methods, so they are called like this.whatever(argument)
- You can do every single mathematical operation on a Natural Number you can do on an int, except for modulo.
- You can copy from other Natural Numbers, and also compare them to each other.
- You can divide and multiply these by 10, and pass a int with that method call to add that to the end of the number.
- You can check to see if they are zero, and clear them to zero.
- newInstance() actually creates a cleared object of the same type of the Natural Number you call it on.
- transferFrom() allows you to set the value of this from another object, and clears that object. It's way more efficient than copyFrom.

Specs:

void add(NaturalNumber n)

- adds n to this
- updates: this
- ensures: this = #this + n (#this is the value of this before the call)

void subtract(NaturalNumber n)

- Subtracts n from this
- Updates: this
- Requires: this >=n
- Ensures: this = #this - n

void multiply(NaturalNumber n)

- duh

void divide(NaturalNumber n)

- Updates: this
- Requires: n>0
- Ensures: #this = n*this + divide and 0<=divide<=n

This value divides this by n and returns the remainder

void power(int p)

- duh

void root(int r)

- Updates this to the r-th root of its incoming value

- Updates: this
- Requires: r>= 2

void copyFrom(Naturalnumber n)
- Copies n to this
- Replaces: this
- Ensures: this = n

Replaces means that a variable might be changed by a call to the method, but the new value is independent of the old one.

int compareTo(NaturalNumber n)
- Compares n to this, returning negative if this<n, 0 if this = n, positive if this>n
- Ensures: compareTo = [a negative number, zero, or a positive integer as this is less than, equal to, or greater than n].
  Its good to check if this is greater, less than, or equal to zero, and not to check for equality between any number

void multiplyBy10(int k)
- Multiplies this by 10 and adds k
- Updates this
- Requires 0 <= k <= 10
- Ensures: this = 10*this + k

int divideBy10()
- Divides this by 10 and returns the remainder
- Updates: this
- Ensures #this = 10 * this + divideBy10

boolean isZero()
- Reports whether this is zero

void clear()
- Resets this to an initial value
- Clears: this
- Ensures: this = 0

void newInstance()
- Creates a new instance of a naturalnumber from an existing object, of the same naturalnumber type

void transferFrom()
- Sets this to the incoming value of n

REFERENCES:

Java has two types: Reference and Primitive types. Primitive types are ints and shit like that. Reference type objects are way cooler. There are infinitely many reference types that are user defined that can be created. Think about it like this:

- Objects are fragmented memory islands firmly planted in your RAM until they are deleted, cleared, or garbage collected. You can't get to them directly, or fuck with 'em.

- Reference types in java aren't pointers, they're references. The value of a reference variable only refers to a specific object.
- Reference variables have **TWO** values, the memory address the object is located at, and the value of that object. We don't give a rats ass about the reference value of an object in this course, only that we remember the object it refers to.
- In pictures, a good way to depict this relationship is by drawing an arrow towards the object shape from the reference shape. The correct notation is an arrow.
- The equality assignment operator basically changes what your reference variable refers to. Two reference variables can refer to the same object. This is called **aliasing.**
- You can only alias variables when the reference type is mutable. Mutable types are reference variables that can change the value of this. Immutable types have none of these methods.
- You can pass reference variables like arguments baby.
- Arrays are reference types, you can't use == or .equals on them
- It's good to use the package java.util to compare arrays.
- A better alternative to arrays are Lists.
- Passing a mutable type as an argument creates a new formal parameter reference variable that refers to the same object in the calling program.
- Never call the same mutable reference type more than once in a method call.

INTERVAL HALVING ALGORITHM:

1. Set lowEnough and tooHigh to zero and n+1 respectively.
2. Find the difference and continue following steps while this is > 1.
3. Find the midpoint between these two points, this is the halving part.
4. Set the midpoint to the power of the root you're trying to find.
5. If your number is lower than this, set tooHigh to your midpoint, else, set your midpoint to lowEnough.
6. Return lowEnough.

MODULO AND CLOCK LOGIC:

- The value of a mod b or a modulo b where a and b are integers and b>0 is computed by doing clock arithmetic on a clock face with b positions.
- If a>0, the hand on the clock starts at 0 and moves |a| positions clockwise, else it moves the same number of positions counter clockwise.
- Modulo is not remainder! Many people say % is modulo, but it is in fact remainder. It works for positive numbers, but not for negative ones.

CONTRACTS:

- Nothing in OSU contracts are ever null, and all parameters refer to object values.
- There are four parameter modes which indicate how a method might change the value of an argument.
- The first is the standard default mode, **restores mode.** This makes sure that the parameter has the same value at the end of the method as it did at the start.
- The second is the **clears mode.** This resets the value of the parameter to how it would appear initialized as a no argument constructor.
- The third is the **replaces mode.** This means the method has a parameter that might be changed from its incoming value, but the method doesn't depend on this incoming value.
- The fourth is the **updates mode.** This means the method has a parameter that has a value that might be changed from its incoming value, and the methods behavior does depend on its

incoming value.
- Don't make null references, which is when your reference variable refers to nothing.

MATHEMATICAL STRING NOTATION:

- A mathematical string is a series of zero or more **entries** of literally any mathematical type, like an integer. An integer would be called the entry type, and the math type is a string of integers.
- Java Strings are not strings, they are actually mathematical strings with a character entry type.
- An empty string is denoted by: <>
- Any string can be described by listing its entries between < and >, with entries separated by commas like: <69, 420>
- The **concatenation** of strings s and t is denoted by s * t, and is when you add one string to a point in another.
- We say s is a **substring** of t if the entries of s appear consecutively in t
  <1,2> is a substring of <1,2,3,4,5>

- We say s is **prefix** of t if the entries of s appear consecutively at the beginning of t.

- Same for **suffix**, except it appears at the end of a string.

- The **length** of a string is how many entries are in the string.

- The substring of s starting at position I (inclusive) and ending at positon j (exclusive) is denoted by s[I,j)

  s=<'a','s','s','b'>;
  s[0,2) = as

- The **reverse** of a string s is the entries in s but in opposite order.

- The question whether strings s1 and s2 are **permuations**, whether they are simply reorderings of one another.
  Ex:
  perms(<1,2,3><2,3,1>)

- The **occurrence count** of an entry x in a string s is how many times x appears as an entry in s. It's denoted by count(s,x)

RECURSION:

- Recursion is the act of calling your own method, while providing smaller and smaller arguments so that it eventually reaches something called a **base case.**
- Another way of thinking about it is having a magical method, that when handed a smaller argument than before, and with all boundary test cases handled, will work.
- Working with recursion means you have to be confident your code will work in a lot of cases, a good way to work with this mirrors mathematical induction.
- Mathematical induction is a form of proof that proves a base case (typically of a natural number of value zero or one), and then proves an inductive case, which means the base case plus one.

- This is the same for your recursive methods, where you handle an initial base case and then handle the iterative cases that will run before your base case, it's kind of like backwards induction.
- A **size metric** is the size of the problem you're trying to solve, and the point of your logical argument is to make sure that it's growing smaller with each reverse induction.
- A good strategy is to consider each smallest program according to the specific size metric you're working with. (This is the creation of your base case.)
- After your smallest problem is handled, consider the next smallest problem according to that metric, write a solution to this problem, and convince yourself it works. Keep going on with this method, so on and so forth.

In Class Review:

In an XML tree, the nodes can only be tag nodes and content nodes. Content nodes cannot have children, they make a good base case. Another base case to check is to see if a tag node is empty. If you're working with tags, always start with .isTag() as your base case and then work from there.

Polymorphism means methods are called based on object type.

A static type is the interface associated with an object, the object type is the actual class it draws methods (overridden or not) from.

Object Oriented Programming:

- If a class implements an interface, then the implementer contains the code specified by the interface. This means that the class has to have method bodies for contracts specified in the interface.
- You just have to have them, but the program will still run if you don't implement them correctly.
- If a class extends another class, then the extender inherits all of the methods from the parent class. This means it can use all the methods in its own class body, and the ones from the class it extends.
- Interfaces cannot have constructors or static methods.
- You can also override methods in the parent class, using @Override which lets you provide new implementations for them.
- When a variable is declared using the name of an interface, ie NaturalNumber, that is its static type, and it's called an interface type. You should declare variables using interface types as the static types.
- When a variable is declared using the name of a class, it's called a class type. Keep the class type on the right side of the assignment operator, which is called your dynamic or object type.
- Java uses polymorphism, which means it decides which class to look at when executing method calls through the object type.

Static Methods vs Instance Methods

- A static method is called without a receiver. This means that they don't need to be assigned to a variable, and work with the arguments passed onto them.
- Instance methods are declared without the keyword static.
- They are called with a receiver, which is directly manipulated by the method itself.
- The receiver is treated like a variable called this. In the method, you work with this to alter the state of the receiver. It basically acts like passing the receiver as an argument.

Recursion On Trees:

- A tree is made up of a root node, and smaller subtrees of the root.
- The base case is usually when you reach a content node, as those nodes have no children and end

the recursive structure of a tree.
- It's also good to consider the base case of an empty tag when searching through a tree for something.
- Other than this info, it's just basic recursion.

Testing

- Testing is done in all software engineering. 30 to 40 percent of development time is spend on testing.
- The best practice for testing it to test individual units at a time, say one class, one method at a time. This is known as unit testing.
- Testing what happens when multiple components are put together into a larger system is known as integration testing.
- Testing a whole user end system is known as system testing.
- A unit being tested is known as the UUT, or unit under test.
- A program is correct if it does what it's supposed to do, and doesn't do what it's not supposed to do.
- To know what the program is supposed to do, you look at the contract, which specifies its intended behavior. There are many levels of contracts.
- A method body is correct if the actual functions of the method are within the allowed behaviors of the method.
- Testing is a technique for trying to refute the claim that a method body is correct for the method contract. The goal is to show that the method body doesn't correctly implement the contract.
- Design/coding are creative, testing is destructive. You need to be able to absolutely fuck up your method.
- A set of test cases is called a test plan! The best way to design a test plan is to test a couple of special cases to reveal flaws.
- The first type of case is a boundary case, where you choose smallest, largest, and special values based on the contract.
- Routine cases, which are cases that one would expect the code to run easily.
- Challenging cases, which are cases that one might find difficult to handle if they were writing the code themselves.

Queue:

- A *queue* is a string of entries that can be put in and pulled out of the string in a first in, first out order. Think of this object like a line in Roller Coaster Tycoon. The first person in line is the first person to get on the ride. This is referred to as FIFO, and "first" in this case is in regards to temporal order, not left to right order.
- A queue is a generic type, which means that the entries into the queue can be any type of variable. A int queue is initialized like this: Queue <Integer> qi = new Queue1L<>(); Notice that its Integer, not int. This is because Java needs generics to be reference types. Integer is a wrapper type, that allows a regular int to be regarded as a reference type.
- All queue methods are instance methods. The methods are as **follows:**
- Void enqueue(T x) adds x to the back (right end) of this.
- T dequeue() removes and returns the entry at the front (left end) of this.
- Int length() reports the length of this.
- T front() Returns the entry at the front of this.
- T replaceFront(T x) replaces the front of this with x, and returns the old front.
- Void append(Queue<T> q) Concatenates q to the end of this.
- Void flip() reverses this. This basically puts the front at the back, and does that for every pair. If the queue is an even number, the middle two numbers are switched, if its an odd number, the median stays put.
- Void sort(Comparator<T> order) sorts a queue. A comparator is an object that describes the order

a method should sort by. This won't be on the exam, it's just cool. PS: To sort anything alphabetically with strings, use the built in CASE_INSENSITIVE_ORDER included with all Java. It's a comparator that ignores case and sorts strings alphabetically.
- Void rotate(int distance) shifts each entry to the left distance times. There is a bunch of math that goes along with this, but just know that it shifts the queue to the left distance times and you'll be fine. Also, entering a negative number shifts it to the right.

Set Mathematics

- Sets have two important features. There are no duplicate elements, and there is no order among the elements.
- An empty set is described as { }.
- We say x is in s if a variable x is in a set s.
- The *union* of sets s and t is denoted by s union t. When two sets are in union, every variable that is in s and t are displayed ***once*****.**
- The intersection of sets s and t is a set consisting of the elements in both s and t, and is denoted by s intersection t.
- The difference of sets s and t is a set consisting of all the elements of s that are not in t, and is denoted by s\t. IE: {1,2,3,4} \ {3,2} = {1,4}.
- S is a subset of t if and only if every element of s is also in t. A proper subset, however, will not be identical to t.
- The size of a set is the number of elements in the set, and is denoted like |set|.

Set Component:

- A set variable is a set containing variables of type T. It is constructed like so: Set<Integer> si = new Set1l<>(); and this creates an empty set { }.
- Everything in a set is an instance method.
- Void add(T x) unions the variable x to this.
- T remove(T x) removes x from this, and returns it.
- T removeAny() removes and returns a random element from this.
- Boolean contains(T x) reports whether x is in this.
- Int size() reports the size of this.
- Void add(Set<T> s) adds this all elements of s that are not already in this to this, and removes those elements from s.
- Set<T> remove(Set<T> s) removes from this all elements of s that are also in this, leaving s unchanged and returns the removed elements.
- Iterating over a set is difficult. You have to transfer your existing set you want to iterate over to a temporary set, then, while the temp set's size is greater than zero, remove any element from the set, do what you want with it, and then add it back to your previous set.

Overloading/Overriding

- Overloading is when you create a method in an interface with the same name as another method, but give it different formal parameters.
- Overriding is when you provide a new implementation for a method in a child class in OO programming.

Sequence

- A sequence is just a set with order. The component allows you to directly manipulate strings through direct access by position, like an array.
- Constructor: Sequence<Integer> si = new Sequence1L<>();
- Void add(int pos, T x) adds x at position pos of this. Make sure pos is less than or equal to the length of this.
- T remove(int pos) removes and returns the entry at position pos of this.
- Int length() reports the length of this.
- T entry(int pos) reports the entry at position pos of this.
- T replaceEntry(int pos, T x) replaces the entry at position pos with x, and returns the old entry.
- Void append(Sequence<T> s) concatenates s to the end of this.

- Void flip() reverses this.
- Void insert(int pos, Sequence<T> s) inserts s at position pos of this and clears s.
- Void exract(int pos1, int pos2, Sequence<T> s) removes the substring of this starting at pos1 and ending at position pos2-1, and puts it in s.

Stack

- Stacks are like decks of cards. They allow you to edit strings of entries of any type in LIFO order. LIFO means last in, first out. It functions like a gun magazine, only being able to access the very beginning of the string (left side).
- Constructor: Stack<Integer> si = new Stack1L<>();
- Void push(T x) adds x at the top (left end) of this.
- T pop() removes and returns the entry at the top of this.
- Int length() reports the length of this.
- T top() returns the entry at the top of this.
- T replaceTop(T x) replaces the top of this with x, and returns the old top.
- Void flip() reverses this.

Map

- The map component allows you to create a set of keys that have certain values assigned to them. Imagine a map like a keyboard, with each button corresponding to a letter, but with the buttons being keys and the letter being arbitrary values.
- Constructor: Map<KeyType,ValueType> si = new Map1L<>();
- No two values have the same key assigned to them. This is described as a function property.
- Void add(K key, V Value) adds the pair(key,value) to this.
- These are special methods:
- Map.Pair<K,V> remove(K key) returns the key and value from this and removes it from this.
- Map.Pair<K,V> removeAny() removes and returns an arbitrary pair from this.
- V value(K key) reports the value associated with key in this.
- Boolean hasKey(K key) reports whether there is a pair in this that has Key key.
- Int size() reports the size of this.
- V replaceValue(K key, V value) replaces the value associated with key in this with V value, and returns the old value.
- Void combineWith(Map<K,V> m) combines m with this, but will only work if m does not contain any keys that this contains.
- Boolean sharesKeyWith(Map<K,V> m) reports whether this and m have any keys in common.

GUI

- A GUI is a graphical user interface, which means it has like buttons and shit that you can click a hellacious amount of times to make the program do something.
- A user manipulating a GUI part, or widget, is called an event.
- The widget that has been manipulated is called the subject.
- These widgets need objects that do something in response to the events for a particular subject, and these are called observers/listeners.
- One way to do this is called *polling*, and this is when the program asks every possible subject if an event has occurred over a small amount of time.
- Another way is to have a designated observer for every subject, and have the subject tell it when an event has occurred. This is called callback.
- Threads are a way to execute two strings of code simultaneously.

MVC

- Model View Controller is an industry standard way of coding GUIs.
- View is the section of the code that creates the actual GUI (Swing in this case) code, and sends callbacks to the controller when an event occurs.
- The Model is used to handle non GUI aspects of the code exclusively. In our case, it collects the data from the view and handles it by putting it into variables. Typically involves getter and setter methods.
- The Controller mediates between the View and Model, by getting information about variables

from the Model, and changing elements and how the program appears to the user in the View. In between these two components of the system that makes up MVC, the Controller can edit the variables got from Model so that they are appropriate based off of the type of event happening in View.

Invariants

- What a while loop is doing is described by its *loop invariant*. An *invariant* is a property that is true every time code reaches a certain point.
- Code that tests the loop condition should not update any variables appearing in the loop invariant.
- The loop does not terminate until and unless the loop condition is false. To show that a loop terminates, it is sufficient to provide a progress metric, which is a variable that decreases every time a loop occurs.
- A loop can be traced in one fucking step if you use a really strong loop invariant.

Queue<String> q = new Queue1L<>();

Queue is the static type and Queue1L is the dynamic type. You can have same static and dynamic type for a variable as well (Queue1L<Integer> q = new Queue1L<>())..

From <https://piazza.com/class/i4n2pjxop6i4vw?cid=660>