CS 3250 Notes******

4/3/19 - DESIGN PATTERNS

Hooks

- Can define methods that do nothing unless subclass overrides them.

Hollywood Principal

- Don’t call us, we’ll call you
- Low-level hooks into system, high-level calls at the appropriate time
- Java Arrays.sort calls compareTo()

- To prevent subclasses from changing the algorithm, make the template method final.
- Both the strategy and template patterns encapsulate algorithms
    - Strategy via composition
    - Template via inheritance
- Factory is a very specialized template
    - Returns result from subclass

Iterator Patterns

- Provides a way to access the elements of an aggregate object sub sequentially without exposing the underlying representation
- This places the task of traversal on the iterator object, not on the aggregate, which simplifies the implementation of the aggregate and places the responsibility where it should be.

Java

- Enumeration is the older that has been replaced by iterator
    - Iterator allows removal

Design Principle

- A class should have only one reason to change
    - Single-responsibility principle
- High Cohesion
    - All methods related to purpose
- In python we don’t want to use multiple elifs(similar to switch cases in Java). We want to use dictionaries or polymorphism.

Composite Pattern
  - Allows you to compose objects into tree structures to represent part/whole hierarchies
  - Composite allows clients to treat individual objects and compositions of objects uniformly
  - We can apply the same operations over both composites and individual objects
  - Can ignore differences between the two
  - Think recursion

April 17, 2019

State
- The combination of the value of all the variables in an object
- We use state machines all the time
  - NFA's, DFA's
- Automata
  - Combinational logic
  - FSM's
  - Pushdown
  - Turing Machines
- Vending Machines, elevator, locks, traffic lights, etc.
- FSM's limited to the amount of memory (states) it has

State Pattern
- Allows an object to alter its behavior when its internal state changes
- The object will appear to change its class
- Very similar to strategy pattern
  - Strategy an alternative to sub-classing as it uses composition
  - State is an alternative to having lots of conditionals

4/17/19: METHODS INSTEAD OF CONSTRUCTORS

Consider Static Factory Methods Instead of Constructors
- One advantage is they have names
  - Constructors do not and one has to differentiate via parameters
  - This can be confusing and lead to errors
- A class can have only one constructor with a given name
  - Don't change order of constructor parameters to differentiate
- Static factory methods don't have to create a new object
  - Constructors always do
  - Maybe there's an object already created that works
  - Helps with immutable classes and pre-constructed instances

- Singletons, flyweights, non-instantiable
- Can return a subtype
    - java.util.collections contains all static methods that work on many types
    - Polymorphic
    - addAll, binary search, disjoint, frequency, min, max, sort, shuffle, reserve
    - Type returned can be non-public
    - Can vary implementation
- Returned class need not exist at the time the class is written
    - Allows run-time specification
    - JDBC an example

Service-Provider Framework

- Service interface
- Provider registration
- Service access
- Disadvantages
    - Classes without public or protected constructors cannot be sub-classed
    - Not called out in Javadoc
- Popular Java static factory name
    - valueOf, getInstance, newInstance, get*Type*, …

Consider a Builder when Faced with Many Constructor Parameters

- If a class has many fields that need initializing, constructors have long list of parameters
    - Constructors often chained/telescoped
- Create empty instance and have many set()s
    - Problem: instance in inconsistent state
- Builder Pattern
- build() is a parameter-less static method
- Required parameters passed in to constructor
    - Optionals set()
    - Other languages have optional parameters instead

Enforce the Singleton Property with a Private Constructor or an Enum Type

- All static instances are executed before anything else in a class!!

Enforce Non-Instantiability with Private Constructor

- Just have a private no-args constructor
    - If have any no-arhs constructor, the default isn’t created
- Class cannot be sub-classed
    - Sub-classes would need to call constructor

- Might want to have constructor throw an AssertionError
    - Just to be safe

Avoid Creating Unnecessary Objects

- Use literals and valueOf()
- Prefer primitives to boxed primitives
- Be careful o unintended auto-boxing
- Try very hard to not manage memory
- Nulling object references should be very unusual
- Arrays are typically only useful for numerical calculations and so we should use lists or other data structures

Avoid Finalizers

- Unpredictable, often dangerous, generally unnecessary
- Unlike C++ destructors
    - These are called immediately
    - Java uses try/finally for these types of users
- One never knows when a finalizer is called
    - Part of garbage collection
    - Might not be called at all
- If you may need a finalizer, use try catch finalizer
- Don’t e.g. close files as there is a limited number of open files
- Finalizers are slow
- Finalizers are not chained
- If really need functionality, provide explicit termination method

When to Implement

- When logical equality(.equals) is different from simple object identity (==)
- This is the typical case as classes have state, kept by variables with values
- Tests for equivalence, not the same object
- When we need the class to be map keys or set elements

Must implement an equivalence relation

- Must be reflexive:x.equals(x) must return true
- Must be symmetric: x.equals(y) must be true if and only if y.equals(x)
- Must be transitive: if x.equals(y) is true and y.equals(z) is true, x.equals(z) must be true
- Must be consistent

Recipe

- Check for object == this
- Use instanced to check for correct type

- Cast argument to correct type
- Test == for all significant fields
    - except for float.compare, Double.compare, and Arrays.equal
- Also override hashCode
- Use @Override

Always Override hashCode when your Override equals
- When invoked on the same object, and the object hasn't changed to affect equals, always return the same integer
    - Does not have to be the same integer

Creating a dashcode
- Set Result = 17

Always Override toString
- Makes class much more pleasant to use
- When practical, toString should return all interesting information in object
- One has to choose the format returned
    - Good idea to create a constructor or static factory that takes string representation and creates object

Consider Implementing Comparable
- Similar to equals
    - But provides ordering information
    - is generic
    - Useful in e.g. Arrays.sort()
- Returns Comparison between two objects
    - -1 if first less than second
    - 0 if equal to

compareTo
- x.compareTo(y) == -y.compareTo(x)
- x.compareTo(y) > 0 and y.compareTo(z) > 0 then x.compareTo(z) > 0
- x.compareTo(y) == 0 -> x.compareTo(z) == y.compareTo(z) for all z
- x.compareTo(y) == 0 -> x.equals(y)

April 24, 2019

Minimize the Accessibility of Classes and Members

- “The single most important factor that distinguishes a well-designed module from a poorly-designed one is the degree to which the model hides its internal data and other implementation details from other modules.”
- Encapsulation
- Decouples modules allowing them to be developed, tested, optimized, used, understood, and modified in isolation
- Make each class or member as inaccessible as possible
- If used nowhere else, nest a class within the class that uses it
- Don’t make any variable/field/attribute public
    - At worst, make it package private
- Try to avoid protected too
    - Must always support
    - Exposes implementation detail to subclasses
    - Should be rare
- If a method overrides a superclass method, it must have the same access level
    - To not violate the Liskov inversion principle
- Implementing an interface requires all methods to be public
    - Implicit in implementing an interface
- Instance field should never be public
    - Limits typing
    - Limits invariants
    - Are not thread-safe
- Arrays are always mutable
    - Never have a public static final array field
    - Or an accessor that returns such beast
    - Be careful of IDE’s that create accessors automatically

In public classes, use Accessor Methods, not Public Fields

- Book still insists on using lame examples of sets instead of simply making fields public
    - With the ostensible argument that we can change internal representation
        - But we never do
        - And if we do, we break the preexisting

Minimize Mutability

- All information provided at construction
- Any changes result in new objects
    - Which in general isn’t true
- Don’t provide methods that modify an object’s state
    - Mutators
- Ensure class cannot be extended
    - Subclasses can’t change intent

- Make all fields final
- Make all fields private
- Ensure that the client cannot obtain references to mutable data
  - Don’t use client-provided reference
  - Don’t return direct object reference
  - Make defensive copies
- Immutable objects are simple
  - Always the same behavior
  - Never global data
- Immutable objects are thread-safe
  - Implicitly parallelizable
  - No synchronization needed
- Only possible downside is the need for an object for each value
  - But: objects are in general cheap
  - Are you sure it’s inefficient?
- “Classes should me immutable unless there is a very good reason to make them mutable.”
  - At the very least, from an external point of view
- If cannot be immutable, limit mutability as much as possible
  - Make every field final unless there is a complexing reason not to

Favor composition Over Inheritance

- GO4
- Inheritance violates encapsulation
  - Subclass depends on superclass’s implementation

Compose Instead

- AN insturmentedHashSet **has a** HashSet instead of extending a HashSet
  - And extends a Forwarding set

Inheritance

- Is-A relationship
- Is every instance of a subclass really an instance of the superclass?
  - If not, have a private instance of the referred-to class

Design and Document for Inheritance

- The *only* way to test a class designed for inheritance is to write subclasses
- Constructors must not call over-ridable methods
  - Directly or indirectly
  - A superclass constructor runs before a subclass constructor, so any subclass methods that are overridden will be called before constructor called

Prefer Interfaces to Abstract Classes

- Classes force inheritance
    - Java is single inheritance
- Existing classes can be easily changed t implement interface
- Interface are ideal for defining mixins
    - Loosely, a mixin is an additional type for a class
    - Useful for polymorphism
    - Know what methods are available to client, which in general define a type
- Can create skeletal implementation for each interface
    - Generally call Abstract*Interface* (Skeleton*Interface* might be better)
    - AbsrtactCollection, Map, List, Set
- Abstract classes do permit multiple implementations
    - Easier to evolve
    - If you want to add a method, can add and implement
    - Everything else still works
- Once an interface is released, much more difficult to change
    - Requires all dependent classes to implement new method

Use interfaces Only to Define Types

- ‘nuff said

Prefer Lists to Arrays

- Arrays are ovarian
    - If a sub is a subtype of super, sub[] is a subtype of super
- Generics are invariant
    - List<t1> is never a subtype of List<t2>

Arrays Versus Generics

- Arrays are reified
    - Their element types are enforced at runtime
- Generics aree implemented by type erasure
    - Types enforced at compile time and erase type at run time
- Cannot create arrays of generic types, parameterized types, or type parameters

Bounded Wildcards

- List<string> is not a subtype of List<Object>
- However, every object is a subtype of itself

When to use

- Use bounded wildcards in methods that have producer or consumer parameters
    - Maybe not a great idea anyway

- PECS: Producer/Extends, Consumer/Super
- Do not use wildcard return types
    - Client of class shouldn’t have to know about wildcards