Effective Java (chapter 2)

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 only have on constructor with a given name.

- Static Factory methods don't have to create a new object.
  - constructors always do
  - maybe theres an object created that already 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
  - (Polymorphism).
  - addAll, binarySearch, disjoint, frequency. min, max, Sort, Shuffle, Reverse, ...
  - Type returned can be no-public
  - can vary implementation.

- Returned class need not exist at the time the class is written.
  - Allows run-time specification
  - JDBC an example

↑

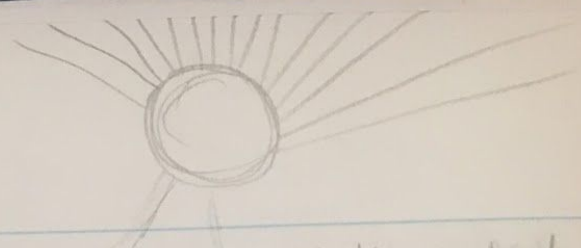
Service-Provider Framework

- Service Interface
- Provider registration
- Service access

- Disadvantages
  - classes without pubic or protected constructors cannot be sub-classed
  - Not called out in javadoc
- Popular Java static factory name
  - valueOf, getInstance, newInstance, getType

Consider a Builder when faced with many Constructor Parameters

- If a class has many fields that need initilizing, constructors have long list of parameters
- create empty instance and have may set()s
  - Problem: instance in inconsistant state
- Builder Pattern



- build() is a parameter-less static method
- Required parameters passed into constructor
  - optionals set()
  - other languages have optional parameters

Enforce the Singleton Property with a Private Constructor

Private no arg constructor (one way to do Singleton)

1.5: Enumeration ← set of Symbols/"name"

~~looks~~ a lot like a class.
*instead instance variables, we have symbols

Enforce non-Instantiability with Private Constructor

- Just have a private no-args constructor
  - If have any no-args constructor, the default isn't created.
- Class cannot be sub-classed
- Might want to have private no-args constructor throw an AssertionError
  - just be safe

Avoid creating Unnecessary Objects

- Use literals and valueOf()

- Prefer primitives to boxed primitives
- be careful of unintended auto-boxing

Avoid Finalizers

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

(Chapter 3)

OBEY the General Contract When Overriding Equals

- Sometimes, you don't need to
  - when all objects are unique, such as threads
  - When you don't need it, such as RNG
  - Superclass equals works well, such as sets, lists, and maps getting from AbstractList, etc.
  - Class is private or package private and

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 variables.

(==) comparing references (mem addr.)

.equals	==
checks if same state	checks if exactly the same

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:

- must be

- there is no way to extend an instantiable class and add a value while preserving the equals contract.
- you can safely add values to a subclass of an abstract class.

Consistency

- Do not not write an equals method that depends on unreliable resources.
- Java's URL equals relies on IP addr. comparison
  - What happens when not on network?
  - network addresses change?

- Check for object == this
- Use instanceof 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 you 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 from runtime to runtime.

large range → smaller range

HASH

- If two objects are equals, both hashCodes must be the same.

- If they are not equals, it is not required to product distinct hasCodes
  - If not, hash table performance can be affected
  - "return 42" is legal, but horrible

Creating a hashCode

- Set result = 17
- For all instance fields
  - If boolean, c = f ? 1 : 0
  - If byte, char, short, or int, c = (int) f
  - If long, c = (int) (f ^ (f >>
  - ⋮

Always Override toString

- makes class much more pleasant to use
- When pratical, toString should return all interesting information in object.
  - ↳ state of object
- One has to choose the format returned
  - Good idea to create a constructor or static factory that takes string representation and creates object
- Provides access to values in toString via getters

Override clone Judiciously

- creates and returns a copy of an object
  - x.clone() != x (two diff objects)
  - x.clone.getClass() == x.getClass() (they're the same class)

- x.clone().equals(x) (has cloned it, and should be the same comments)
- constructors are not called.

• Dark clone
• If you override the clone method in a non-final class, return an object obtained by invoking super.clone

Consider implementing Comparable

• Similar to equals
  - But provides ordering information
  - is generic
  - useful in e.g Arrays.sort()

• Returns comparison between two objects

CompareTo
• x.compareTo(y) = -y.compareTo(x)

Minimize the Accessibility of classes and Members

• Encapsulation
• Decouples modules allowing them to be developed, tested, optimized, used, understood, and modified in isolation.

- Make each class and it's members inaccessable as possible.
- If used nowhere else, nest a class within the class that uses it.
- Try to avoid Protected too
  - must always support
  - Exposes implementation detail to Subclasses
  - Shold be rare
- If a method overrides a superclass method, it must have the same access level.
  - to not violate the Liskov inversion Priciple
- Implemanting an interface requires all methods to be public
  - implicit in impaemanting an interface
- Instance Field Should never be Public
- limits the typing
- limits invariants
- Are not thread safe
- Arrays are always mutable
  - Never have a public static final array fild
  - or an accessor that returns such a beast
  - Be careful of IDEs that create accessors automatically.

In Public Classes, use Accessor Methods, not Public Fields

- Book still insists on using lame ex. of sets instead of simply making fields Public.
  - With the ostensible argument

⋮

Minimize Mutability

- All info provided at construction
- Any changes result in new objects
  - in general is 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/Private
- Ensure client cannot obtain references to mutable data
  - Don't use client-provided reference
  - Don't return direct object reference
  - make defensive copies

dictionarys > switch Statem

Compose Instead

- An InstrumentedHashSet has a HashSet instead

Design and Document for Inheritance

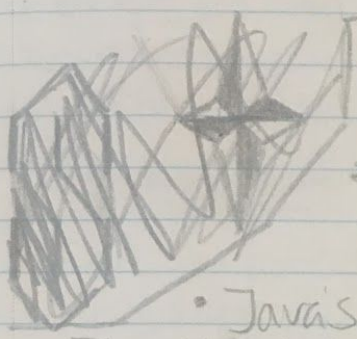
- The only way to test a class designed for inheritance is to write subclasses.
- Constructors must not call overridable methods
  - Directly or indirectly
  - A superclass constructor runs before

Prefer Interfaces to Abstract Classes

- Classes force inheritance
  - Java is single inheritance
- Existing classes can easily be changed to 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 define type.
- Abstract classes do permit multiple implementations
  - easier to evolve
  - if you want to add a method, can add and implement
  - Everything else still works

in Python) All variables are references that refer to objects.

Prefer class hierarchies to tagged classes



Effective Java 5 Generics

Typing

- Java's types system is very complex
- It adds various mechanisms to add "generics"
- Other languages simply have references to objects and duck typing

Generic Types

- Generic classes and interfaces are known as generic types
- Generic types define sets of parameterized types

~~List~~ List<String>

- Raw type is List

Prefer Lists to Arrays

- Arrays are covariant
  - If Sub is a subtype of Super, Sub[] is a subtype of Super[]
- Generics are invariant
- Arrays are reified
  - their element types are enforced at runtime
- Generics are implemented by type erasure
  - types enforced at compile time and erase type at run time

Banded Wildcards

- List<String> is not a subtype of List<Object>
- However, every object is a subtype of itself
- So, can have a

Java

- 'Java's enumerations are more powerful than other languages'
- Almost classes
  - Can't extend, but can implement an interface
- Export one instance of each enumeration via a public static final field
- Enums are fine
  - ~~One~~ only one instance

Annotations

- Prefer Annotations to Naming Patterns
  - JUnit a major example
- Consistently use @Override
  - makes sure you are actually overriding
  - especially for equals, toString, hashCode

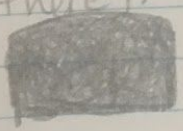
Effect Java Chp 7.

Methods

- most Parameters have restrictions on their validity
  - positive, non-null, not zero-lenght, etc.
- AKA preconditions
- Program defensively
- Catch problems as soon as Possible
- Fail fast
- Method may die — Check Parmetas for validity
  - or wase, work but in an unexpected way
- Throw an exception
  - An illegalArgumentException a good choice.

Assertions

- Optional in Java
  - must enable with -ea
  - or in first class (and doesn't enable them there):



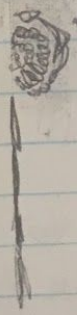
Factories

- Handles the details of object creation
  - Encapsulates the creation in a subclass
  - Decouples interface from creation
- Return a variety of types
- Client doesn't care which type
- add additional types
  if static can't subtype

- Don't have to be abstract
  - can have default and then call down if nessecary

The dependency Inversion Principle

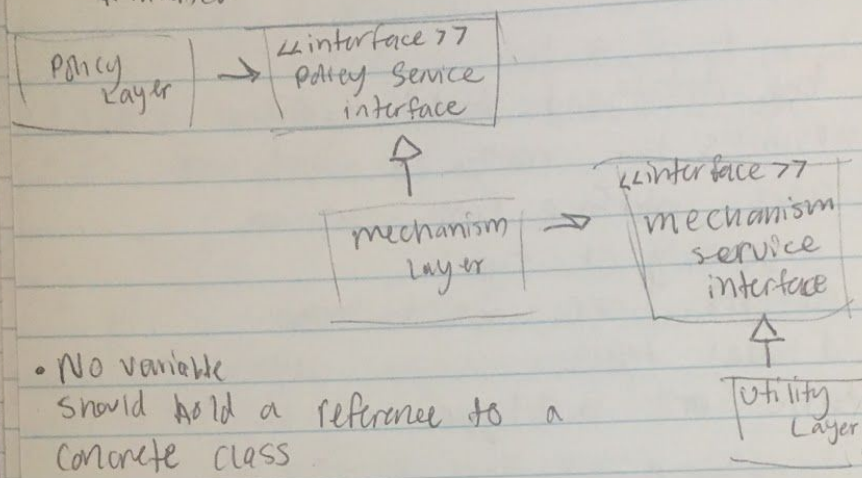
- Depend upon abstractions
- Don't depend upon concrete classes.
- High level modules shouldn't depend on low-level modules.
  - Both should depend on abstractions



Abstractions do not depend on the details but the details depend on the abstractions.

Policy Layer → mechanism Layer → utility Layer

* Inverted



- No variable should hold a reference to a concrete class
- No class should derive from a concrete class.
- No method should override an implemented of any of its base classes

Dependency [redacted] Injection

- A technique whereby one object (or static method) supplies the dependencies of another object.
- A dependency is an object that can be used (a service)
- An injection is the passing of a dependency to a dependent object (a client) that would use it.
- This fundamental requirement means that using values (services) produced within the class from new or static methods is prohibited.
- The client should accept values passed in from outside.
- This allows the client to make acquiring dependencies someone else's problem.

- The intent behind dependency injection is to decouple objects to the extent that no client code has

Inversion of control

- A design principle in which custom written portions of a computer program receive the flow of control from a generic framework.
- A software architecture with this design inverts control as compared to traditional procedural programming.
  - in traditional programming, the custom code that expresses the purpose of the program calls into reusable libraries to take care of generic tasks.
  - with inversion of control, it is the framework that calls into the custom, or task-specific, code.

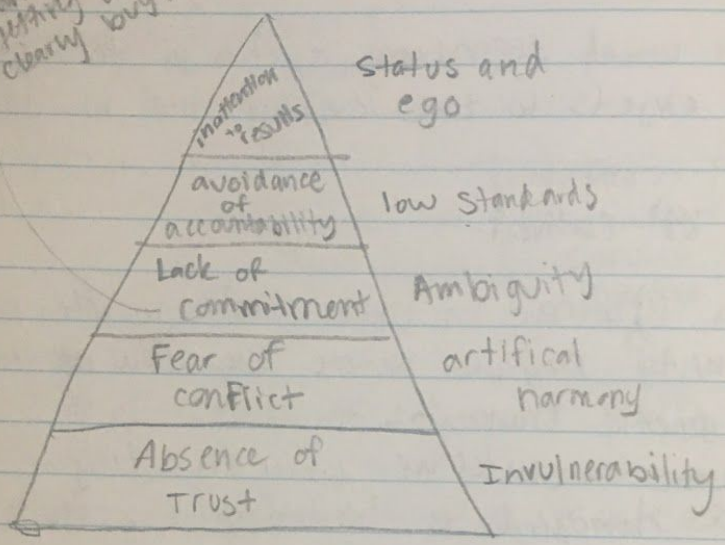
Abstract Factory Pattern

- Provides an interface for creating families

Factory
- creation through inheritence
- creates objects of a single type

Abstract Factory
- creation through composition
- Instantiated via new and passed
- creates families of related objects via factories

committing to a plan or a decision and getting everyone to clearly buy into it.



Chapter 5

Need only one of (Singleton):
- Thread pools, caches, dialog boxes, preferences, logging, device drivers, I/O.
- But might not need each time, so lazy initialization ← don't do something until its asked for

Pattern

Ensures a class has only one instance and provides global access to it.

Threading
- Synchronize getInstance
- Eagerly create
- Double-checked locking

Visibility and Synchronization
- Visibility assures that all threads read the same value of a variable
- synchronization make sure that only one thread can write to a variable.

Volatile
- Reads and writes happen to main memory - not from individual CPU caches
- Writes to a volitile also write all the thread-visible variables to main memory
- Reads from a volatile re-read all thread-visible variables to main memory

* Java Concurrency in Practic — Bryan Goetz

Atomic ← happens in a single instruction

- Any write to a volatile variable establishes a happens-before relationship with subsequent reads of that same variable
- this means that changes to a volatile variables are always visible to other threads.
- Reads and writes are atomic for reference variables and for most Primitive variables (except longs and doubles)
- Read and writes are atomic for all variables declared volatile (including long and duble variables)
- And there are AtomicInt, and AtomicLong, ...

Volatile Not Always Enough

- If there is a read/modify/write such as variable++.
- Must use synchronized keyword to remove race conditions.

Command Pattern
Chapter 6

Participants:

- Client is responsible for creating a concrete command and setting its receiver
- Invoker holds a command object and then calls execute()
- Command declares an interface
- the reciever knows how to perform the work needed

ADAPTER AND FACADE
Chapter 7

- converts the interface of a class into another interface the clients expect.
- Adapter let classes work together that couldn't because of incompatible interfaces.

Two Types of Adapters

- Object adapters use composition
- class adapters use inheritance

Facade

- Provides a unified interface to a set of interfaces in a subsystem.
- Facade defines a higher-level interface that makes the subsystem easier to use.

Difference

- Adapter alters an interface to make it usable
- facade makes a complicated interface easier to use.

Principle of Least Knowledge

- Talk to only immediate friends
- Decouples
- Law of Demeter — Don't talk to Friends of Friends
- Methods may talk to
  - their own object
  - Objects passed as parameters
  - Objects they instantiate
  - Instance variables

HOOKS

- can define concrete methods that do nothing unless subclass overides them.
- Use abstract when subclass must

Summary

- 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

Iterater and Composite

Iterator Pattern

- Provides a way to access the elements of an aggregate object sequentially without exposing the underlying representation.

- This places the task of traversal on the iterator object, not on the aggregate, which simplifies the aggregate interface and implementation, and places the responsibility where it's suppose to be.

Design Priciple

- A class should only have one reason to change.
  - single-responsibility Principle
- High cohesion (All methods related to Purpose)

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.

What is a Thread?

A thread is a single sequential flow of control within a program.

Sequential Programs → each has a beginning, an execution sequence, and an end. At any given time during the runtime of the program, there is a single point of execution.

- Hello World
- Sort list of names
- Compute list of Prime numbers

However, a thread itself is not a program. It cannot run on its own, but runs within a program.

Multiple Threads: the use of multiple threads in a single program all run at the same time and perform different tasks.

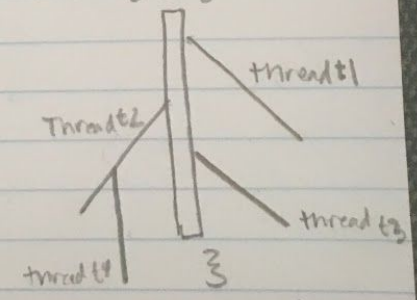
* helps creating multiple independent path of execution within a program which can run in parallel.

Per Thread (each thread of execution has the following components:

- Program Counter
- Stack
- Native Stack
- Stack Restrictions
- Frame
- Local Variable Array
- Operand Stack
- Dynamic Linking

main Thread Starts

Public static void main (String[] arg) {



main thread Ends

Program Counter (PC)

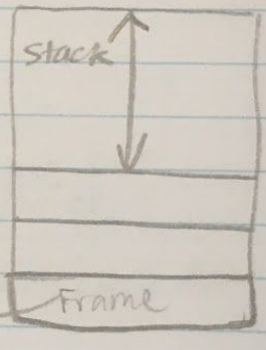
The JVM uses the PC to keep track of where its executing opcodes. The PC will be pointing at a memory address in the Method Area.
All CPUs have a PC, which is incremented after each opcode and therefore hold the address of the next instruction to be executed.

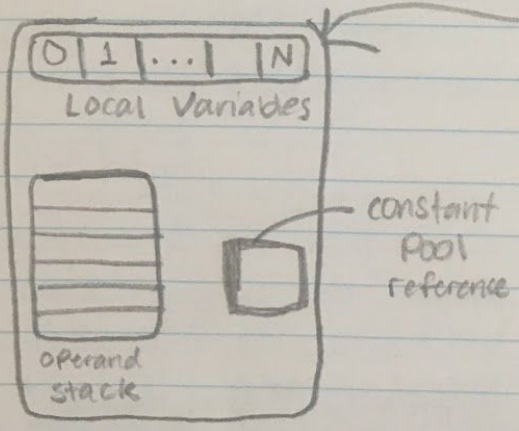
Java Code to Byte Code

Basic Programming Concepts

- A JVM is a stack-based machine.
- Each Thread has a JVM stack which stores frames.
  - Every time a method is invoked a frame is created.

A FRAME consists of:
- Operand Stack
- Local Variable Array (LVA)
- Constant Pool reference





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.

Enum in Python

- Enumerations are created using classes.
- Enums have names and values associated with them.

Tuple = ("ww", "ww", "ww")
- ordered
- unchangable

List = ["ww", "ww", "ww"]
- ordered
- changable

Dict = {"ww": 1
"ww": 2}
- unordered
- changable
- Indexed (keys/values)

[To Read a Java class file and parse the headers means:

1. java = ClassFile() } reading in
2. java.get_majc() etc. } Parsing]