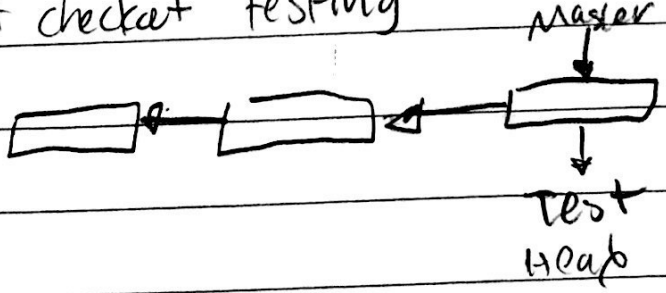
Jenkins

VCS: Version Control System
- keeps track of changes (edits) of files
- files are checked in and checked out from a repository
- Everytime you have something running, check it in (commit)
Everytime before you make a significant change check it in

```
$ Git init .
$ git add loremol
$ git commit -local -m 'first commit'
    git commit -m " "
    git add - File Name
    git push
Coverage run --omit = *test* -m unittest
```

git branch Name of Branch — Create a new branch
git checkout testing — changes head to diff branch

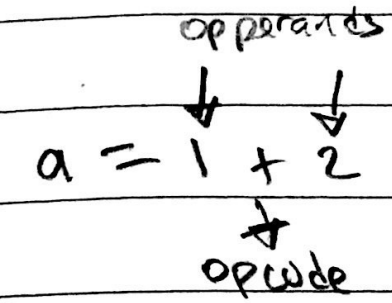


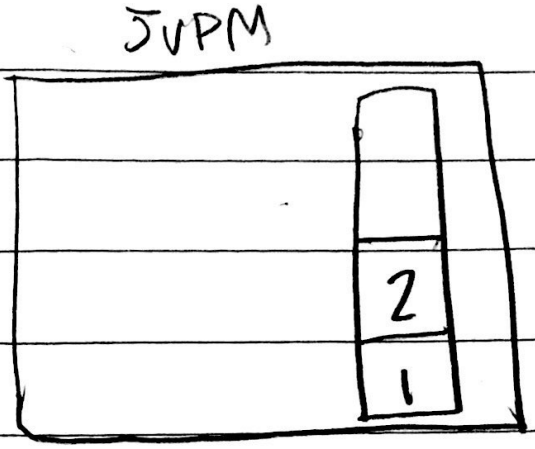
git branch -D BranchName - Removes branch
git diff
git Merge
git log
git stash
git pull

git checkout master go back to original branch

- Class
  Header
  Methods
  Opcodes

0x60 ⇒ Integer add





Push

Iadd ⇒ (Pop + Pop)

push 2 + 1

push (3)

Need Only One of

- 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 assigned)

Threading

- Synchronize getInstance
- Eagerly create
- Double-checked Locking

Visibility and Synchronization

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

Volatile

- Read and Writes to Main Memory - Not from indiv. CPU caches
- Writes to a volatile also write all the thread-visible variables to main memory
- Reads from a volatile re-read all thread-visible variables from main memory.

Atomic

Any write to a volatile variable establishes a happens-before relationship with subsequent reads of the same variable

This means that changes to a volatile variables are always visible to other threads

What's more, it also means that when a thread reads a volatile variable, it sees the latest changes are the effects

Reads and Writes are atomic for all variables declared volatile (including long and double variables

And there are AtomicInt and AtomicLong

Volatile is not always Enough

- If there is a read/modify/write such as variable++

The Adapter Pattern

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

Two Types

Object adapters use composition
Class adapters use inheritance

Façade

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

Difference

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

Principle of least knowledge

- object should only talk to immediate friends
- Decouples - Law of Demeter
- Methods may talk to - Their own object
  - Objects passed as parameters
  - Objects it instantiate
  - [illegible]

Hooks

- Can define concrete methods that do nothing unless subclass overide them
- Use abstract when subclass must implement

Hollywood Principle

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

Summary

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

Iterators and

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 the aggregate, which simplifies the aggregate interface and implementation, 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 relate to purpose

Composite Pattern

- Allows you to compose objects into 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 dif. between the two • Think recursion

Getting a Job

- Networking
- Maintaining your GPA is the #1 thing you can do.
- Impressing People in Class
- LinkedIn
-
- Meet Up (DJUG) denver java users group [Meetup]
- Denverdevs.org

Stop

- Gaming
- Stop Social Media
- Stop Watching Sportsball
- Stop wasting Time

Start

- Learn about your field
- Being Relatable - be able to interact with non-geek
- Being nice
- Branch Out - New languages

How to get any Job you want?

Do the job before you get the job

- Do research on the company and Position
- What tech do they focus?
- Whom have may taught and been taught by
- Strengths and Weaknesses and those of their competitors

Print UP some Cards
- Include: non-school email address
  - twitter, LinkedIn, github URLs

Github
- Your school work
- Gists

• Blog

Market Yourself

• Create a presentation or two
• Write a paper or two

Interviews

• Typically start with ~~for~~ phonescreen [fizbuzz]
• Followed by an online quiz
• Maybe half or full-day Whiteboard test
• Learn table manners
• Arrive at least 15 minutes early but enter 8-10 minutes early
• Bring a Notepad and make notes if necessary
• Bring Extra resumes
• Followup - Send Email - Say thanks, you know how much work it is, if there is anything else you can provide to help

Personal

- Dress Well.
- Eye Contact

Don't Cross arms and legs - Lean in - Avoid nervous habits
- Don't Wear Axe

- You're interviewing them as much as they want you
  - Don't want to work at just any place
- Working at some places is worse than not working in the field for a time.
  - training edge term that traps you
  - Place you can't excel in
  - Churn and Burn

What to Ask

Why do you work here?

What are the working conditions?

What SE lifecycle is used?

What is the tool chain?

- HR - how are the benefits
  - health, vacation, retirement?

Types of Questions

- Computer Science - Algorithms and data structures → O of algorithms and choosing
- What you've done → Imperative vs functional → Make sure you did what is on your resume are ready to talk about it → looking for specifics → give concrete example - Don't have one, say so.
- Language Frameworks

Whats a closure? **

Software Engineering

- Git
- Agile Manifesto

Web

MVC • ORM • Ajax

Questions

What is 2^{64}

Strengths and Weaknesses

Why do you want to work here.

Verbal Hints

Dont say "I feel like" - No one cares how you feel

-Use "I think" or "I believe"

Avoid using "like" "um" "You know", "got" "literally"

Irregardless is not a word.

Dont say methodology

The word data is plural - "The data are"

- Don't make jokes
- Don't lie and dont BS
-

- You can do arithmetic on pointers, not on references
- Go in with some questions. "What's your software dev life cycle"

"What are you looking for in a candidate"

"Why do you work here"

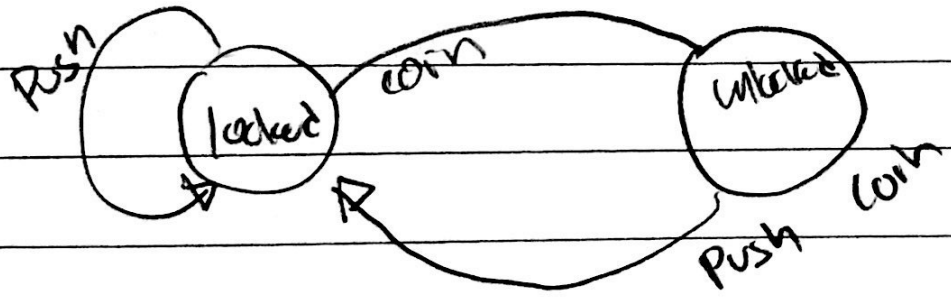
- Why did you choose this field How did college experience prepare you

State

- The combination of the value of all the variable in an object
- Use to all the time - NFA's, DFA's
- Automata
  - combinatorial logic
  - Push down
  - Turing machines
  - FSMs

- Vending machines, elevator, locks, traffic lights etc.
- FSMs limited to the amount of memory (states) it has.

Turnstile



State pattern

- Allows an object to alter its behavior when its internal state changes
- The object will appear to chang class

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 can lead to errors
- A class can have only one constructor with a given name.
  - Don't change order of
- Static factory methods don't have to create a new object.
  - Constructors always do
  - Maybe there is an object already created that works
  - Helps with immutable classes and preconstructed instances
- Singletons, Flyweights, non instantiable.
- Can return a subtype
  - java.util.collections contains all static methods that work on many types.
  - Polymorphic
  - addAll, binarySearch, disjunct, frequency, min, max sort, shuffle, reverse,
  - Type returned can be non-public
  - Can vary implementation
- Returned classes 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

Consider Static Factory Methods instead of Constructors

- Disadvantages
  - Classes without public or protected constructors can't be subclassed
  - 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 field that need initializing, constructors

- Create empty instance and have many setX()s
  - Problem: instance in inconsistent state.
- Builder Pattern
- Build() is a parameter-less static method
- Required parameters passed into constructor

Enforce Non-Instantiability With Private Constructor

- Just have a private no-args constructor.
  - If have any no-args constructor, default isn't created
- Class cannot be subclassed
  - Subclasses would need to call constructor.
- Might want to have constructor throw an Assertion Error.
  - Just to be safe

Avoid Creating Unnecessary Objects

- Use literals and valueOf()

So - Prefer primitives to boxed primitives
  - Careful of unintended auto boxing

Try very hard to not manage memory

Nulling object references should be very unusual

Avoid Finalizers

- Unpredictable, often dangerous, generally unnecessary
- Unlike C++ destructors
  - These are called immediately
  - Java uses try/finally for these type of cases
- One never knows when a finalizer is called
  - Part of garbage collection
  - Might not be called at all

Avoid Finalizers

Don't close files as there is a limited number of open files

- Finalizers are slow
- Finalizers are not chained

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 random # generators
  - Class is private or package private and you know you'd don't need it
    - Consider implementing with an assertion
  - Superclass equals works well, such as sets,

When to Implement

* When logical equality (.equals) is dif. 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

Consistency

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

Recipe

- 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.
- If two objects are equals, both hashCodes must be the same.
- If they are not equals, it is not required to product distinct hashcodes
  - If not, hash table performance can be affected
  - "return 42" is legal, but horrible.

Creating a hashCode

- Set result = 17
- For all the fields - if boolean, c = f ? 1 : 0
  - if byte, char, short, or int, c = (int) f
  - if long, c = (int)(f ^ (f >>> 32))
  - if float, c = Float.floatToIntBits(f)
  - if double, c = (int)(Double.doubleToLongBits(f) ^ (Double.doubleToLongBits

Update Result = 31 * result + c

Exclude any redundant Fields

- Which you shouldn't have anyway

- Ignore fields ignored by equals.

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 o
- Provide access to values in toString via getters.

Override Clone Judiciously

- Creates and returns a copy of an object
  - x.clone() != x
  - x.clone().getClass() == x.getclass()
  - x.clone().equals(x)
  - Constructors are not called.
- If you override the clone method in a non-final class, return an object obtained by invoking super.clone
- If class implements Cloneable, must have properly functioning public clone method.

Clone Elements Too

- The clone method is effectively another constructor

Consider Implementing Comparable

- similar to equals
  - But provides ordering information
  - Is generic
  - Is useful in Arrays.sort()
- Returns comparison between two objects
  - -1 if
  - 0

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)

Minimize the Accessibility of Classes and Members

"The single most important factor that distinguishes

Make as Few thing public as possible

- Try to avoid protected too
  - Must always support
  - Should be rare
  - Exposes implementation detail to subclasses.
- Arrays are always mutable
  - Never have a public static final array field
  - 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

Research @Override

Prefer Interfaces to Abstract Classes

- Classes force inheritance
  - Java is single inheritance
- Existing Classes can be easily changed to implement interface
- Interface are ideal for defining mixins
  - Loosely a mixin is an additional type for a Class
  - Useful polymorphism
  - known what methods are available to client, which in general define a type.
- Can make skeletal implementations for each interface
  - Generally call AbstractInterface
  - AbstractCollection, Map, List, Set

* Use Interfaces Only to Define Types

Prefer Class Hierarchies to tagged Classes

- Verbose, error prone

Favor Static Member Classes over Nonstatic

- For example, in a linked list, nodes do not need to refer to head, tail, etc. from list class.
- No need for node to contain all the data in list, so it can be a static member class
- One way to think of this is that the static member class could be a separate class, but the code reads better with it inside

Anonymous Classes

- Have no names
- Not a member of enclosing class
- Declared and instantiated at the same place.
- Permitted wherever expressions are allowed
- Cannot have static members
- Useful for creating function objects on the fly.

Effective Java 05

Generics

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

Generic Types
- Generic classes and interfaces are know as generic types
- Generic types define sets of parameterized types
  - 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
  - List<t1> is never a subtype of List<t2>

Arrays versus Generics
- 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 runtime.
- Cannot create arrays of generic types, parameterized types or type parameters.

Bounded Wildcards

- List<String> is not a subtype of List<Object>
- However, every ob

When to use

- Use bounded wildcards on methods that have produces or consumer parameters
  - Maybe not a great idea anyway.

PECS: Producer Extends, Consumer Super

- Do not use wild card return types
  - Client of class shouldn't have to know about wildcards