Monday, February 18, 2019 4:38 PM

Product Backlog
-Comprises an ordered list of requirements that a scrum team maintains for a product
-Consists of features, bug fixes, non-functional requirements, etc.
-Items added to a backlog are commonly written in story format
-Visible to everyone but may only be changed with consent of product owner
-Often, but not always, states in story points using the rounded Fibonacci scale

Sprint Backlog
-List of work the dev team must address during next sprint
-Derived by the scrum team progressively selecting product backlog items in priority order from top until they feel they have enough work to fill sprint
-Dev team should keep in mind its past performance assessing capacity
-Property of dev team
-Once committed, no additional work can be added to the sprint backlog except by the team

--Typical Additions--

-Sprint burn-down chart is a publicly displayed chart showing remaining work in the spring backlog
-Definition of done: the exit criteria to determine whether a produce backlog item is complete

--Velocity--

-Total effort a team is capable of in a spring
-The number is derived by evaluating the work (typically in user story points) completed in the last sprint
-The collection of historical velocity is a guideline for assisting the team in understand how much work they can likely achieve in a future sprint

-Sprints should not be planned more than 3 months in advanced

Spikes
-A time-boxed period used to research a concept or create a simple prototype

Three Pillars of Scrum
-Transparency
    -Letting customers, management, dev team, etc., what is going on
-Inspection (Introspection)
    -Look at what is being done and figure out how it can be done better
-Adaptation
    -Be prepared to change

Five Values
-Commitment: Team members individually commit to achieving their team goals, each and every sprint
-Courage: Team members know they have the courage to work through conflict and challenges together so that they can do the right thing.
-Focus: Team members focus exclusively on their team goals and the sprint backlog; there should be no work done other than through their backlog.
-Openness: Team members and their stakeholders agree to be transparent about their work and any challenges they face.
-Respect: Team members respect each other to be technically capable and to work with good intent.

Important Stuff
-Practices self-control
-Accepts responsibility for behavior
-Resolves conflict appropriately
-Exhibits positive attitude toward learning
-Participates in class discussions
-Engaged listener
-Follows written and oral directions
-Completes work in a timely manner
-Works independently
-Responsible for belongings and property
-Asks for help when needed
-Produces, neat, accurate, quality work
-Positive work ethic
-Completes timely, accurate homework
-Uses good organizational skills
-Demonstrates technology skills
-Uses library media to enhance learning

Side read: *Grit* - Angela Duckworth

Sprint 3 user stories:
-Implement all of the integer operations

Intro to Software Engineering

What's the Problem?
-Software is everywhere
-It's mostly crap
-Costs all of us an incredible amount of time and therefore money
-Pretty much everything depends on it
-Software engineering is the study of an application of engineering to the design, development, and maintenance of software
-Margaret Hamilton -- Credited with coining the term "software engineering" during creation of Apollo 11 software

-In java all of opcodes come from .class file
-All operands go onto the stack
-Integers go to the run time stack
    --Ex: 1 and 2 only are on RT stack, iadd command pushes pop+pop (2 + 1) => pushes (3)
    --RT stack now contains 3 only

No Silver Bullet - Brooks, Fred P. (1986) "No Silver Bullet - Essence and Accident in Software Engineering"
-"there is no single development, in either technology or management technique, which by itself promises even one order of magnitude improvement… "
-Tenfold difference between an ordinary designer and a great one
-difference between accidental complexity and essential complexity

Terminology

Program requirements
-Statements that define and quantify what the program needs to do
-The work requirements isn't used in the same way as elsewhere
-Software requirements tend to be negotiable
Functional requirements
-What a program needs to do
-Tend to be yes/no
Non-functional requirements
-The manner in which the functional requirements need to be achieved
-Performance, usability, maintainability
-Tend to be on a scale
Design constraints
-Statements that constrain the ways in which the software can be designed and implemented
-Platform, language, DB, web app, GUI, etc.
Testing

- Many types
  - Acceptance
  - Unit: method
    - Arguably the most important
    - Many modern SE methods use "test first"
      - Write a test
      - Make sure it fails
      - Write just enough code to make it work
    - If you don't know how to test it, how can you write it?
    - Correct frame of mind
    - Test-Driven Development
  - Integration: class
  - System

Implementation
- SE tends to focus on requirements, design, and processes
- A bad implementation will ruin everything else

Guideline: Naming
- Use good, descriptive names

Design Patterns
- Software design pattern is a generally reusable solution to a commonly-occurring problem within a given context in software design
- It is not a finished design that can be transformed directly into source or machine code
- It is a description or template for how to solve a problem that can be used in many different situations
- Design patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system
- object-oriented design patterns typically show relationships and interactions between classes or objects

Why talk about design patterns?
- Creates a shared vocabulary
  - developers can interact in richer terms
- Keeps thinking/designing at the abstract (pattern) level
  - creates better (more flexible, reusable, etc) design

Pattern categories:
- Creational Patterns
- Behavioral Patterns
- Structural Patterns

Documentation
- Pattern name and classification
- intent
- AKA
- Motivation (Forces): Scenario and context in which pattern is used
- Applicability: Situations in which pattern is usable
- Structure: Graphical representation of the pattern
- Class diagrams and interaction diagrams
- Participants: listing of classes and objects used in the pattern
- Collaboration:
- Consequences: results, side effects and trade offs
- Implementation: the solution of the pattern
- Sample Code: illustration of how pattern can be used
- Known uses
- Related Patterns

4 parts of object oriented programming
- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

(A, E, I, O+1)

Design patterns != design principles

First design principle
- Identify the aspects of your application that vary and separate them from what stays the same
- You can alter or extend without affecting the other parts
- Basis of almost every design pattern
- Also creates more easily reusable objects
- Objects delegate behavior to other objects

Second Design Principle
- Program to an interface, not an implementation
  - Not necessarily a Java interface
  - Program to a supertype
  - Can then better use polymorphism
  - Can more easily change implementation

Third Design Principle
- Favor composition over inheritance
- Favor "has-a" relationships over "is-a" relationships
- Inheritance limits reusability

Strategy Pattern
- Defines a family of algorithms, encapsulates each one, and makes them interchangeable
- Strategy lets the algorithm vary independently from the clients that use it

Observer Pattern
- key part in the familiar model-view-controller (MVC) architectural pattern

Languages
- Java has Observer, Observable, etc.
- JavaScript has addEventListener etc.

Design Principle
- Strive for loosely-coupled designs between objects that interact

-Objects have very little information about each other
-No shared state
Decoupling
-The observer pattern helps decoupling

Warnings
-Don't depend on order of evaluation of notifications
-Java Observable is a class
    -not an interface
    -must inherit
    -setChanged() is protected

Open/Close Principle
-Designs should be open for extension, but closed for modification
Decorators
-Have the same super type of the objects they decorate
-Can have one or more decorators
-Can pass wrapped object anywhere original could be passed
-Adds behavior before and/or after delegating object
-Objects can be decorated at runtime
Inheritance
-Using inheritance to get type matching but not behavior
    -Java requires this, other languages don't

The Factory Pattern
Factories
-Handles the detail of object creation
    -Encapsulates the creation in a subclass
    -Decouples interface from creation
-Can return a variety of types
-Client doesn't care which type
-Can add additional types easily
-If static, can't subtype to extend
From Wikipedia:
…the factory method pattern is a creational pattern that uses factory methods to deal with the problem of creating objects without having to specify the exact class of the object that will be created

The Dependency Inversion Principle
-Depend upon abstractions
-Do not depend upon concrete classes.
Wikipedia:
--High-level modules should not depend on low-level modules
    --both should depend on abstractions
--Abstractions should not depend on details
    --Details should depend on abstractions
--By dictating that both high-level and low-level objects must depend on the same abstraction this design principle inverts the way some people may think about object-oriented programming
Therefore
-No variable should hold a reference to a concrete class
-No class should derive from a concrete class
-No method should override an implemented method of any of its base classes
Dependency 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 the 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 to be changed simply because an object depends on needs to be changed to a different one
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 of related or dependent objects without specifying their concrete classes
Abstract vs Non

-Factory								-Abstract Factory

-Creation through					-Creation through

Midterm:
-Class notes
-5 Dysfunctions
-Ch 1-4 Headfirst design
-Git and version control
-Dealing with complexity
    -Git/version control
    -Design patterns
-Software Engineering
    -Test Driven Development
    -Waterfall model - good for small projects
                                -bad for large projects
    -Agile/Scrum
        -Events
        -Roles
        -Scrum guide
-Interfacing, Inheritance
-OOP big 4:
    -Abstraction
    -Encapsulation
    -Inheritance
    -Polymorphism
-composition vs inheritance
-Observer pattern
-Decorator pattern
    -Open/close principle
-Factory pattern (not abstract)

-Interview question: compare and contrast frameworks and libraries

inheritance					composition
-Creates objects of a single type					-Instantiated via new and passed
					-Creates families of related objects

Singletons
-Need only One of
  -Thread pools, caches, dialog boxes, preferences, logging, pdevice drivers, I/O
  -But: might not need each time, so lazy initialization
    -lazy initialization: don't do something until 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-guaranteed by "volatile" keyword
  -Synchronization makes sure that only one thread can write to a variable
  -These are two different things
-Volatile
  -Reads and writes happen to main memory
    -Not from individual 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 that same variable
  -This means that changes to a volatile variable is always visible to other threads
  -What's more, it also means that when a thread reads a volatile variable, it sees not just the latest change to the volatile, but also the side effects of the code that led up to the change
  -Reads and writes are atomic for reference variables and for most primitive variables (except long and double)
  -Reads and writes are atomic for all variables declared volatile (including long and double variables)
  -And there are AtomicInt, AtomicLong, ….
-Volatile Not Always Enough
  -If there is a read/modify/write such as variable++
  -Must use synchronize keyword
Command Pattern
-A behavioral design pattern in which an object is used to encapsulate all information needed to perform an action or trigger an event at a later time
-Participants
  -Client is responsible for creating a concrete command and setting its receiver
  -Invoker holds a command object and at some point calls its execute() method
  -Command declares an interface that has at least an execute() method
  -A concrete command implements execute() and may call multiple methods in its receiver
Adapter and Façade
-The Adapter Pattern
  -Converts the interface of a class into another interface the clients expect
  -Adapter let s 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
  -Talk to only immediate friends
  -Decouples
  -Law of Demeter
  -Methods may talk to
    -Their own object
    -Objects passed as parameters
    -Objects they instantiate
    -Instance variables
Template Pattern
-Behavior design pattern that defines the program skeleton of an algorithm in an operation, deferring some steps to subclasses
-
-Defines the skeleton of an algorithm in a method, deferring some steps to a subclass
-
-Ex hot beverage

Java keyword: volatile
-changes often, all read/write must go to main memory every time it is called

Books: Java Concurrency in Practice, Educated

Getting a Job
-Networking (including classmates)
-Linked In
  -Use for networking
  -Join MSU Denver CS group
  -Requires rely on it heavily
-Meetup
-Maybe Twitter
  -Actual contributions, not just likes and re-tweets
  -Follow others
-Join ACM
-Denver Devs (Slack)
--STOP--
-Gaming
  -Gaming friends won't get you jobs
  -Go out and do things IRL
-FB and non-job related social media
-Watching sports
-Wasting time
  -Time is all we have
-Everyone is an expert at something
  -Choose
--START--
-Learning about field
  -ACM, IEEE
-Being relatable
  -Be able to interact with the non-geek
-Being nice
-Branching out
  -New languages etc.
-Article: "How to Get Any Job You Want"
  -It turns out that people applying for a job care more about their own credentials than the people who are hiring them.
  -Companies can usually get over a hundred applications for a single position
    -Filters
    -Have to get through HR

some steps to subclasses
-
-Defines the skeleton of an algorithm in a method, deferring some steps to a subclass
-
-Ex hot beverage
-For both coffee and tea
-Boil water
-Use hot water to extract
-pour into cup
-Add condiments
-prepareRecipe is a template method
-All steps present, some handled in base class, some in subclass
-Hooks
-Can define concrete methods that do nothing unless subclass overrides them
-Use abstract when subclass must implement, hooks when optional
Hollywood Principle
-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()
Summary
-To prevent subclass 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

Iterations 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 should be
Design Principle
-A class should have only 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
Part/Whole
-Animals, Mammals, cats, …
-Respiration, locomation, etc.
-Objects in a scene
-Texture, placement, etc.

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 number generators
-Superclass equals works well, such as sets, list, and maps getting from AbstractList, etc.
-Class is private or package private and you know you don't need it
-Consider implementing with an assertation
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 symmetrical: x.equals(y) must be true if and only if y.equals(x)
-Must be transitive: if.equals(y) is true and y.equals(z) is true, then x.equals(z) must be true
-Must be consistent: multiple calls to x.equals(y) must always return the same value
-For any non-null reference, x.equals(null) must return false
So?
-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
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
-Provide access to values in toString via getters

-Companies can usually get over a hundred applications for a single position
-Filters
-Have to get through HR
-Do the job before you get the job
-Do research on the company and the position
-What tech do they use?
-Whom have they bought and been bought by?
-Their strengths and weakness and those of their competitors
-It's OK if you're a few years below the minimum experience level, but not TOO far below
-It's OK if your education level is a little below the required amount, but again, not too much below
-Make sure that you can actually DO the job
-Market Yourself
-Print up some cards
-Include
-Non-school email address
-Twitter, Link-In, Github
-Github
-School work
-Gists
-Blog
-Create a presentation or two
-Write a paper or two
Interviewing
-Many Kinds
-Typically start with phone screen
-Possibly followed by an online quiz
-May be half- or full-day
-May be interviewed by group
-Or individuals
-Or mix
-Typically have to eat with them
-Learn some table manners
Overall
-Arrive at least 15 minutes early, but enter 8-10 minutes early
-Bring a notepad and make notes if necessary
-Bring extra resumes
-Follow up!
-The same day
-Say thanks, you know how much work it is, if there is anything else you can provide to help
Personal
-Dress well
-Eye contact
-Body language
-Don't cross your arms or legs
-Lean in
-Avoid your nervous habits
-No scents
What You Want
-You're interviewing them as much as they you
-You don't want to work for just any place
-Unless you can grit your teeth for a year and build your resume
-Maybe check out glassdoor.com
-Working at some places can be worse than not working in the field for a time
-Trailing edge technology that traps you
-A place you can't excel in
-Churn and burn
What to Ask
-Why do you work here?
-What are the working conditions?
-What is 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
-Imperative vs functional
-http://www.nerdparadise.com/tech/interview
-What have you've done?

-One has to choose the format returned
  -Good idea to create a constructor or static factory that takes string representation and creates object
-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
  -You must ensure that it does not harm original object and properly establishes invariants
-The clone architecture is incompatible with normal use of final fields referring to mutable objects

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 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
-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 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 examples of sets instead of simply making fields public
  -With ostensible argument that we can change internal representation
    -But we never do
    -And if we do, we break the preexisting API contract
  -Less harmful if immutable

Minimize Mutability

-All information provided at construction
-Any changes result in new objects
  -Which is in general true
-Don't provide methods that modify an objects state
  -Mutators
-Ensure class cannot be extended
  -Subclasses can't change intent
-Make all fields final
-Make all fields private
-Ensure 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 be 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 compelling reason not to

Favor Composition Over Inheritance

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

Design and Document for Inheritance

  -O of algorithms and choosing
  -Imperative vs functional
  -http://www.nerdparadise.com/tech/interview
-What have you've done?
  -Make sure you are ready to talk about what is on your resume
  -They are looking for specifics
  -Give a concrete example
  -If you don't have an example, say so
  -STAR
    -Situation
    -Task
    -Action
    -Resolve
-Language and frameworks
  -When would you use an anonymous inner class in Java?
  -What are the four types of anonymous inner classes in Java?
  -What's a lambda? (an unnamed function)
    -Know your versions
  -What's a closure?

OO

-What are the major features of OO?
  -Abstraction, encapsulation, inheritance, polymorphism
-What is an interface?
-What is a virtual method?
-What is multiple inheritance?
-Deep vs shallow copies?
-Dependency inversion/inversion of control?
  -Frameworks vs Library?
-Design patterns?

SOLID

-Single-responsibility principle
  -Classes do one thing
-Open-closed principle
-Liskov substitution principle
-Interface segregation principle
-Dependency Inversion Principle

OS

-Concurrency
  -Threads vs processor

Networks

-IP v4/6 addressing
-TCP vs UDP
-Client/server vs P2P
  -P2P a single program is both a client and server

Software Engineering

-Git
-Agile Manifesto
-TDD
-DRY, YAGNI
-CI/CD

Web

-MVC - Model View Controller
-ORM -Object Relational Mapper
-Ajax

Questions

-How many usable addresses in a class C network? 254
-Strengths and weaknesses?
-Why do you want to work here?

Tough Questions

-They typically will ask a few very difficult questions
  -To show off
  -To see how you reason
-How many trees are in the CONUS
-How far away is the moon

Verbal Hints

-Don't say "I feel like"
  -No one cares how you feel
  -"I think" or "I believe"
-Try to avoid "like"
  -Unless you're making a simile
  -Also "um", "you know", and "got"
-Don't use "literally"
  -Unless you almost always talk figuratively
-Irregardless is not a word
-Don't say methodology
-The word data is plural

-GO4
-Inheritance violates encapsulation
    -Subclass depends on superclass's implementation
Design and Document for Inheritance
-The *only* way to test a class designed for inheritance is to write a subclass
-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 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 in general define a type
-Can create skeletal implementation for each interface
    -Generally call Abstrace*Interface* (Skeleton*Interface* might be better)
    -AbstractCollection, Map, List, Set
-Abstract classes do permit multiple implementations
    -Easier to evolve
    -If you want to add a method, you 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
Prefer Class Hierarchies to Tagged Classes
-Verbose, error-prone,
-
Use Function Objects to Represent Strategies
-Java didn't have method references or lambda
-
Favor Static Method Classes Over Non-static
-A nested class is defined within another class
-Only serves the enclosing class
-Four kinds
    -Static
    -Non-static
    -Anonymous
    -Local
-Last three are called inner classes
-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

Generics
Typing
-Java's type 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 paramterized 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 run time
-Cannot create arrays of generic types, parameterized types or type parameters
When To Use Wildcard Type
-Use bounded wildcards in methods that have producer or consumer parameters
    -Maybe not a great idea anyway

    -Unless you almost always talk figuratively
-Irregardless is not a word
-Don't say methodology
-The word data is plural
    -data "are"
-Don't make jokes
-Be terse and don't BS
Guerrilla Interviewing
-Looking for smart people who get things done
-Introduction
-Question about recent project candidate worked on
-Easy programming question
-Pointer/Recursion question
-Do you have any questions?
Some Typical
-Please tell me about yourself.
-What makes you interested in this position?
-What do you know about our organization?
-What do you consider your greatest strengths?
-What would former coworkers/professors/supervisors say about you if we called them as a reference?
-Why did you choose this field?
-How did your college experience prepare you for a career in this field?
-Describe the work environment that makes you thrive

-PECS: Producer/Extends, Consumers/Super
-Do not use wildcard return types
    -Client of class shouldn't have to know about wildcards
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 final
    -Only one instance
Annotations
-Prefer Annotations to Naming Patterns
    -Junit a major example
-Vonsistently use @Override
    -Makes sure you are actually overriding
    -Especially for equals, toString, HashCode
Use Marker Interfaces to Define Types
-A marker interface is one with no methods
-Serializable is an example
    -Indicates object can be written via ObjectOutputStream
Check Parameters for Validity
-Most parameters have restrictions on their validity
    -Positive, non-null, not zero-length, etc.
-AKA preconditions
-Program defensively
-Catch problems as soon as possible
-Fail fast
Check Parameters for Validity
-Method may die
    -Or worse, 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):

```
static {
    ClassLoader.getSystemClassLoader().setDefaultAssertionStatus(true);
}
```

-Import for maintaining object consistency
    -Values stored for later use must be good
    -Checks in constructors important
-Check before doing any calculation
    -Unless calculation does the checks for you
Make Defensive Copies
-Assume the worst fo your class's clients
    -They will modify your invariants
Arrays
-Non-zero-length arrays are always mutable
-Return a copy
Design Method signatures Carefully
-Good names
-Short parameter lists
    -If all identical, maybe varags
Write Good JavaDoc Documentation
-For all your visible items

Tor
-The Onion Router
-Developed in the mid 1990s by United States Naval Research Laboratory employees with the purpose of protecting US Intelligence communications online
-A special Firefox browser and a sophisticated anonymizing network
Silk Road
-In October 2013, the FBI shut down the website and arrested Ross William Ulbricht under charges of being the site's pseudonymous found "Dread Pirate Roberts"
-Ulbricht was convicted of eight charges related to Sil Road in the US Federal Court in Manhattan and was sentenced to life in prison without possibility of parole
Silk Road 2
-Silk Road 2 came online, run by former administrators of Sil Road
-Shut down
Alphabay
-Was shut down after a law enforcement action as a part of Operation Bayonet
-The alleged founder, Alexandre Cazes, a Canadian citizen was found dead in his cell in Thailand
Bitcoin
-These transactions are verified by network nodes through the use of cryptography and recorded in a public distributed ledger called a blockchain
-Bitcoin was invented by an unknown person or group of people under the name Satoshi Nakamoto and released as open-source software in 2009

Blockchain
-A blockchain, originally block chain, is a growing list of records, called blocks, which are linked using cryptography
-Each block contains a cryptographic hash of the previous block, a timestamp, and transaction data
-An open, distributed ledger that can record transactions between two parties efficiently and in a verifiable and permanent way
Well, Maybe Not
-Open or public
-Distributed
-Parts might not be efficient if proof of work is required
Distribution
-Multiple nodes hold entire chain
-All changes require consensus by majority
Blockchain != Bitcoin
-Blockchain was invented by Satoshi Nakamoto in 2008 to serve as the public transation ledger of the cryptocurrency bitcoin
-The invention of the blockchain for bitcoin made it the first digital currency to solve the double-spending problem
Block time
-The average time it takes to generate a new block
-~10 minutes in bitcoin
Forks
-Changes for new features, bug fixes,, responses to hacking
-Hard forks
    -When the old rules are overridden by new rules
    -Old software can't validate new blocks
-Soft forks
    -Backwards compatible
Proof of Work
-Some amount of resource (e.g. CPU cycles) that shows an investment was made by the creator
    -"Difficult" to create
-"Easy" to verify work was done
Hashcash Proof Of Work
-The sender prepares a header and appends a counter value initialized to a random number
-It then computes the 160-bit SHA-1 hash of the header.
-If the first 20 bits (i.e. the 5 most significant hex digits) of the hash are all zeros, then this is an acceptable header.
-If not, then the sender increments the counter and tries the hash again
-Out of 2^160 possible hash values, there are 2^140 hash values that satisfy this criterion
-Thus the chance of randomly selecting a header that will have 20 zeros at the beginning of the hash is 1 in 2^20
-The number of times that the sender needs to try to get a valid hash value is modeled by geometric distribution
-Hence the sender will on average have to try 2^20 values to find a valid header

-*The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information* -George A Miller
-The average human can hold in working memory 7 +/- 2 objects
--Why?--
-Brain optimized
-Size of sports teams
-Digits in a phone number
-Complexity of a computer program
---Live variables, nesting, cyclomatic complexity, etc.

Informationisbeautiful.net/visualizations/millions-lines-of-code